

UNIVERSITÀ DEGLI STUDI DI NAPOLI "PARTHENOPE"
DIPARTIMENTO DI SCIENZE E TECNOLOGIE
CORSO DI LAUREA IN INFORMATICA



ELABORATO DI LAUREA

Realizzazione di un'app Android per la gestione di code di sportelli universitari

RELATORE

Ch.mo Prof. Antonino Staiano

CANDIDATO

**Vittorio Triassi
Mat. 0124001048**

ANNO ACCADEMICO 2017-2018

Indice

1	Introduzione	1
1.1	App: definizione e tipologie	1
1.2	Applicazioni mobili vs piattaforme web	2
1.3	Il problema delle code	2
2	Strumenti per lo sviluppo	4
2.1	Panoramica sull'infrastruttura	4
2.2	Amazon Web Services - EC2	6
2.2.1	Amazon Machine Image (AMI)	6
2.2.2	Scelta del tipo dell'istanza	6
2.2.3	Lancio dell'istanza	7
2.2.4	Connessione all'istanza	9
2.2.5	Ottenimento credenziali	10
2.3	Parse Server	13
2.3.1	Installazione e dipendenze	13
2.3.2	Inizializzazione di Parse	13
2.3.3	Accesso alla Parse Dashboard	14
3	Progettazione ed implementazione	17
3.1	Raccolta dei requisiti	17
3.1.1	Use Case Diagram	18
3.2	Architettura dell'applicazione	19
3.2.1	Struttura del database	20
3.2.2	Classi Parse Dashboard	21
3.2.3	Gestione degli utenti	22
3.3	UML Class Diagram	24
3.3.1	Client	24
3.3.2	Gestore code	26
3.4	Funzioni gestore code	27
3.4.1	App Class	27
3.4.2	MainActivity Class	27
3.4.3	Services Class	28
3.4.4	Queue Class	28
3.5	Funzioni client	31
3.5.1	MainActivity Class	31

3.5.2	SignUp Class	32
3.5.3	AfterSignUp Class	32
3.5.4	LogIn Class	32
3.5.5	AfterLogIn Class	33
3.5.6	UserLocation Class	33
3.5.7	CompanyServices Class	38
3.5.8	Queue Class	39
3.5.9	Notifications Class	43
4	Scenari d'uso dell'app	44
4.1	Mockup client	44
4.2	Mockup gestore code	54
5	Conclusioni	58
	Appendice A Gestore code	62
	Appendice B Client	68
	Bibliografia	85

Abstract

La tesi in esame è incentrata sullo sviluppo di un prototipo che permetta la gestione di code che si vengono a creare dinanzi agli sportelli universitari. Tali sportelli sono generalmente le segreterie didattiche o studenti dell'Ateneo. L'obiettivo è quello di realizzare una piattaforma che dia la possibilità agli studenti di riservarsi un posto in coda presso lo sportello di loro interesse in maniera digitale, ed allo stesso tempo, si vuole offrire un applicativo che consenta agli sportellisti di tali segreterie di avanzare correttamente la numerazione delle code e di gestirne la relativa apertura e chiusura. Unico vincolo che si pone sulla logica di funzionamento dell'intero sistema, è quello di doversi localizzare con il proprio dispositivo entro un raggio prefissato dal servizio di interesse, pena l'impossibilità di visualizzare lo sportello sull'app e di mettersi in coda.

Capitolo 1

Introduzione

La crescita degli ultimi anni che ha interessato il mercato degli smartphone ha comportato un forte cambiamento soprattutto nelle abitudini dei propri utenti. Si è passato dal dover consultare necessariamente dispositivi fissi, al poter fruire in completa mobilità di contenuti di vario genere. Ad arricchire ulteriormente l'esperienza utente ci ha pensato il mercato delle App. Ma cos'è realmente un'app?

1.1 App: definizione e tipologie

Il termine app è la versione abbreviata di applicazione mobile. Questa è un'applicazione software dedicata ai dispositivi di tipo mobile, quali smartphone, tablet e smartwatch. Un'app è generalmente pensata per una determinata funzione ed è a tutti gli effetti un software applicativo che risiede sul dispositivo in nostro possesso. Ne consegue che risulta essere più leggera e performante di una generica applicazione compatibile per sistemi Desktop, per l'ovvia limitatezza di risorse hardware a disposizione che si ha su un dispositivo mobile.

Generalmente le app si suddividono in due categorie: app native e web app. L'evidente differenza risiede nel fatto che con le app native, l'applicazione viene fisicamente installata sul dispositivo e deve essere compatibile con il sistema operativo del device. Nel caso di una web app invece si crea un collegamento verso un applicativo remoto e si accede tramite il browser del dispositivo utilizzato.

Appare chiaro che con una web app non si incide sull'hardware dal momento che le elaborazioni vengono eseguite dai server del servizio che si è intenti ad utilizzare. In ogni caso, con una web app è richiesta una costante connessione internet e per questo motivo le prestazioni dipenderanno anche dalla velocità di connessione. Discorso diverso invece per una app nativa che a meno di funzioni dove è richiesto necessariamente l'accesso alla rete, ne diviene possibile l'utilizzo anche in assenza di connessione.

Esistono infine altre due tipologie di applicazioni che prendono il nome di app ibride ed universali. Le prime si propongono come soluzioni che impiegano sia componenti nativi che tecnologie web atte alla gestione di interfaccia e logica di funzionamento. Le tecnologie web vengono utilizzate tramite un componente che prende il nome di WebView. Quelle universali invece si piazzano a metà strada tra applicazioni mobili ed applicazioni tradizionali, comprendono le caratteristiche di ambedue ed offrono la possibilità di essere installate sia su apparati mobili che fissi. [15]

1.2 Applicazioni mobili vs piattaforme web

Si è parzialmente accennato al cambiamento nelle abitudini degli utenti. Cambiamento dovuto all'esplosione di dispositivi che permettono di consumare servizi e di ottenere informazioni in totale mobilità. Per inquadrare meglio il fenomeno però, sarebbe opportuno fare delle riflessioni più approfondite.

Degli studi mostrano chiaramente come gli utenti preferiscano di gran lunga le applicazioni presenti sugli smartphone e tablet rispetto agli stessi servizi offerti tramite apposite pagine web. E spesso la prima domanda che un utente si pone quando scopre un nuovo servizio è se ne esiste la rispettiva app. I siti web non sono funzionali ed interattivi così come lo sono le app sui dispositivi mobili. I siti web sono stati progettati per un'esperienza da consumare tramite mouse e tastiera. D'altro canto ci sono le app, pensate per display touch screen e spesso ottimizzate per un utilizzo ad una mano.

Possedere un'app per una compagnia equivale alla creazione di un canale di comunicazione diretto con i propri clienti. Questo avviene perché gli utenti, una volta installata l'app sul proprio dispositivo, non dovranno in futuro preoccuparsi di reperire aggiornamenti o quant'altro in prima persona, dal momento che sarà la compagnia stessa a provvedere con la notifica di tutto ciò che sarà necessario aggiornare.

1.3 Il problema delle code

Il tema che si vuole prendere in esame in questa tesi va a circoscrivere la fetta di app che mira a risolvere il problema della gestione delle code. Questo perché, spesso non ci si rende realmente conto di quanto tempo si passi ogni anno in attesa del proprio turno in code di vario genere.

Semplici esempi di tempo prezioso della nostra giornata speso in attesa di qualcosa o di qualcuno, sono ogniqualvolta ci si reca al bar per prendere un caffè (stimati 7 minuti di tempo medio di attesa) oppure ogni volta in cui si va al ristorante e si finisce con l'attendere una media di 15 minuti. O in coda dal proprio medico, una media di 32 minuti. Per non parlare di chi si sposta ogni

giorno con un mezzo di trasporto. Si stima che in un anno si perdano dalle 38 alle 50 ore imbottigliati nel traffico. [14]

Questi appena elencati sono ovviamente dei dati statistici da contestualizzare alle proprie abitudini, al proprio lavoro ed al proprio stile di vita. Il dato di fatto è che ci sono momenti lungo l’arco delle nostre giornate che non investiamo in maniera produttiva poiché impegnati a conservare il nostro turno in coda.

A questo punto si potrebbe erroneamente pensare di dover rinunciare ad alcune abitudini per evitare di perder troppo tempo, che sia al ristorante, in banca o altrove.

Molteplici app al giorno d’oggi provano a risolvere o quantomeno ad ammortizzare i tempi medi di attesa in code. Citiamo ad esempio Skiplino [12], che si propone come sistema universale di gestione code. Oppure Queue Mobile [6] che offre più applicativi a seconda del fatto che l’app venga adottata per un ristorante, una banca, una gestione di eventi etc. Dunque, appare evidente che prototipi in tal senso sono già ampiamente diffusi. Ciò che li rende eterogenei tra loro è il meccanismo con il quale è concepito il servizio offerto all’utente finale. È su tale questione che è incentrata la dissertazione della presente tesi.

Ci sono numerosi accorgimenti che se presi in considerazione, migliorano notevolmente l’esperienza utente finale quando si ha l’esigenza di mettersi in coda presso un’attività commerciale o un ente, che prevede un rapporto con il cliente finale tramite un front-office.

Ad esempio, qualora fosse possibile monitorare l’andamento della coda pur non essendo fisicamente presente sul posto bensì in un raggio di 500 m da essa, si permetterebbe alla persona che necessita di fruire del servizio, di fare altro, a patto di non allontanarsi del tutto da un perimetro circoscritto attorno all’attività interessata ad erogare tale servizio. Non di poco conto anche l’aspetto per il quale si eviterebbero inutili congestioni di spazi chiusi.

Raccolte delle particolari specifiche che vanno a regolare il funzionamento generale del servizio che si vuole offrire, appare possibile l’opportunità che si dà all’utente finale di migliorare la propria “esperienza” qualora necessitasse di inserirsi all’interno di una coda.

Il prototipo realizzato e che si propone come argomento di tale tesi mira ad essere uno strumento di ausilio alle segreterie di Ateneo che si interfacciano con gli studenti tramite front-office.

Capitolo 2

Strumenti per lo sviluppo

In tale sezione verranno inizialmente presentati e successivamente mostrati più in dettaglio gli strumenti che hanno reso possibile la realizzazione del prototipo sviluppato.

2.1 Panoramica sull'infrastruttura

Sebbene la logica di funzionamento dell'applicazione sia vincolata dagli orari di apertura e chiusura fisica degli sportelli universitari, si vuole verosimilmente avere un servizio che sia sempre raggiungibile, anche in ottica di manutenzione e di gestione da parte dello staff responsabile. È necessario dunque adottare una soluzione che permetta un accesso alla piattaforma senza disservizi.

Per soddisfare tale necessità si è fatto uso di una IaaS. Acronimo che sta per Infrastructure as a Service e che rappresenta la possibilità di avere un'infrastruttura di calcolo subito disponibile senza preoccuparsi di sostenere spese di gestione e mantenimento di hardware che non sempre rispecchia le esigenze del servizio che si sta ospitando.

Uno dei grandi vantaggi derivante dall'adozione di una IaaS è l'opportunità di scalare verticalmente all'occorrenza l'intera infrastruttura, pagando soltanto per le risorse che effettivamente vengono usate. Vantaggio non da poco dal momento che spesso per quanto accurate possano essere le previsioni di traffico utenti e di spazio richiesto per ospitare dei servizi una volta lanciato un progetto, nasce la necessità di apportare modifiche all'infrastruttura di base. Nel caso di infrastrutture non facilmente scalabili, questo comporta ingenti cambiamenti e conseguentemente modifiche onerose in termini economici.

L'IaaS adottata in questo contesto è la Amazon Elastic Compute Cloud meglio nota come Amazon EC2.

L'EC2 risulta essere un servizio Web che consente di ottenere e configurare le proprie risorse senza preoccuparsi della manutenzione fisica dell'infrastruttura

noleggiata. Il piano utilizzato è stato quello gratuito di AWS[11] che include 750 ore mensili di istanze t2.micro. Durante la fase di test iniziali tale vincolo sulle ore a disposizione è stato fortemente vincolante dal momento che in caso di istanze multiple mandate in esecuzione contemporaneamente, il tetto massimo delle 750 ore veniva consumato da ogni singola istanza in stato di running. In ogni caso, qualora si stesse utilizzando tale piano AWS, e si mantenesse una sola istanza attiva 24/24h, questa risulta sufficiente per mantenere il servizio attivo per un intero mese con il piano gratuito senza dover ricorrere ad un piano più capiente per il quale è richiesto un adeguamento.

Una volta assicurata l'infrastruttura che andrà ad ospitare i dati della nostra applicazione, è necessario provvedere a definire il meccanismo di memorizzazione di dati quali sono le credenziali di accesso ed i dati veri e propri che descriveranno il comportamento dell'app.

Il modello adottato è un'estensione del BaaS, acronimo che sta per backend as a service. Tale estensione prende il nome di MBaaS[16] ovvero mobile backend as a service la quale è in grado di fornire agli sviluppatori di web app o app native un modo per collegare le loro applicazioni ad un backend cloud storage ed API (Application Programming Interface) esposte da applicazioni backend, garantendo allo stesso tempo funzioni quali la gestione degli utenti, sistema di notifiche push ed altre integrazioni di vario genere. L'MBaaS scelto prende il nome di Parse Server.[7]

Parse Server è una versione open-source del backend Parse sviluppato originariamente da Facebook. Questo offre la possibilità di creare e gestire la modellazione dei propri dati in maniera semplificata grazie alle proprie API.

Affinché Parse Server venga eseguito correttamente, è sufficiente farlo girare su una qualsiasi infrastruttura che supporti Node.js.

Le motivazioni che hanno portato alla scelta di tale framework risiedono nel fatto che Parse Server[8] non è dipendente dal backend di Parse ed inoltre, questo usa MongoDB direttamente senza che dipenda dal database hostato su Parse. La scelta di MongoDB verrà poi motivata successivamente. Si può inoltre effettuare un processo di migrazione di app esistente sulla propria infrastruttura senza troppe complicazioni.

Un ruolo chiave hanno giocato le API per la gestione delle Geo queries. Query effettuate restituendo oggetti di tipo ParseGeoPoint che hanno permesso l'elaborazione delle posizioni localizzate degli utenti in maniera meno laboriosa.

Come già annunciato precedentemente, si è preferito un modello NoSQL ad uno relazionale dal momento che la struttura più "libera" di MongoDB si sposava meglio con la tipologia dei dati utilizzati.

2.2 Amazon Web Services - EC2

Nella seguente sezione si andrà ad illustrare la procedura necessaria per la creazione e la configurazione dell'istanza AWS presentata nel paragrafo precedente.

Affinché si possa usufruire di un'istanza dell'Elastic Compute Cloud (EC2) è innanzitutto necessario disporre di un regolare account Amazon Web Services. Nonostante verrà utilizzato un piano gratuito per il lancio dell'istanza, sarà richiesto di registrare una carta di credito all'atto della creazione dell'account. Una volta regolarizzato il proprio account, ci si reca sulla EC2 Dashboard che racchiuderà tutte le principali operazioni messe a disposizione da AWS per tale servizio.

Si può procedere adesso con la creazione dell'istanza che per comodità divideremo in passi.

2.2.1 Amazon Machine Image (AMI)

Una Amazon Machine Image è un template che contiene le configurazioni software richieste per lanciare la propria istanza. Può essere scelta una AMI fornita da AWS, dalla community o dal Marketplace di AWS. Nel nostro caso, si procede con una ricerca sul Marketplace digitando e selezionando “Parse” (Figura 2.1).

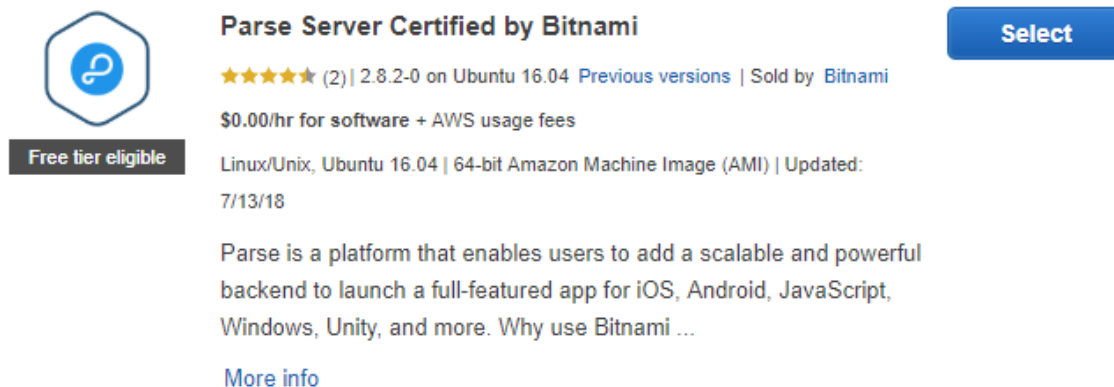


Figura 2.1: Parse Server AMI

2.2.2 Scelta del tipo dell'istanza

Amazon EC2 fornisce una vasta scelta di tipi di istanze ottimizzate per adattarsi a differenti casi d'uso. Le istanze sono dei server virtuali che possono eseguire applicazioni. Queste dispongono di varie combinazioni di CPU, memoria, spazio di archiviazione e capacità di rete. Tutto ciò garantisce una grande flessibilità nella scelta della quantità di risorse di cui si necessita per il proprio progetto.

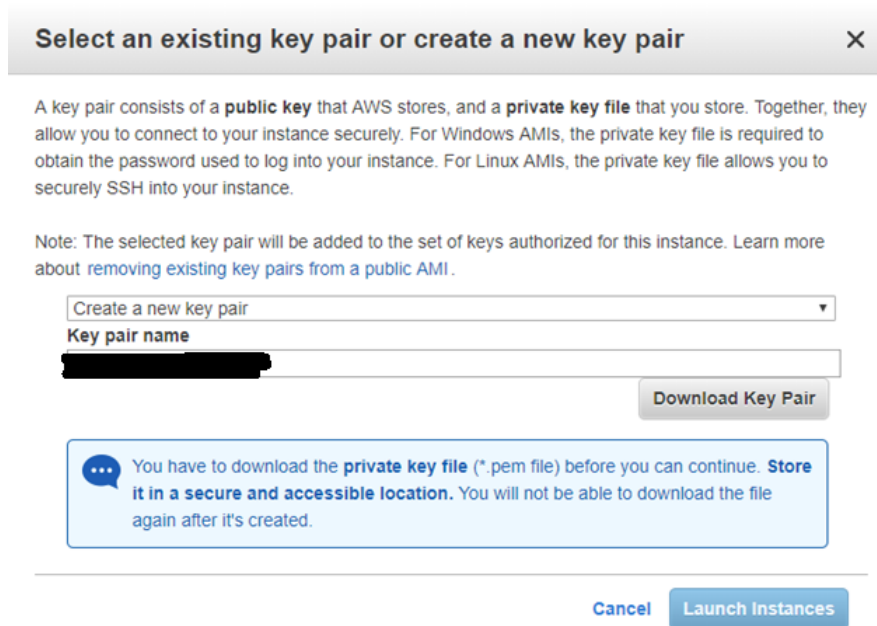
L'istanza scelta per l'esecuzione della nostra applicazione risulta composta dalle seguenti specifiche: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only) Figura 2.2.

	Family ▾	Type ▾	vCPUs ⓘ ▾	Memory (GiB) ▾	Instance Storage (GB) ⓘ ▾	EBS-Optimized Available ⓘ ▾	Network Performance ⓘ ▾	IPv6 Support ⓘ ▾
<input type="checkbox"/>	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
<input checked="" type="checkbox"/>	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes
<input type="checkbox"/>	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
<input type="checkbox"/>	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate	Yes

Figura 2.2: Istanza t2.micro

2.2.3 Lancio dell'istanza

Il terzo ed ultimo passaggio richiede la creazione di una key pair che permette il lancio dell'istanza. Una key pair consiste in una chiave pubblica che AWS salva e di un file che rappresenta la chiave privata che si preoccupa di salvare il proprietario dell'istanza. Queste due chiavi, combinate, permettono la connessione all'istanza in maniera sicura. Per le AMIs che si appoggiano su Linux (come nel nostro caso), il file che memorizza la chiave privata permette l'accesso all'istanza tramite SSH. Per AMIs basate su Windows invece, tale file è richiesto per ottenere la password usata per l'autenticazione. (Figura 2.3)



Select an existing key pair or create a new key pair ✕

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about [removing existing key pairs from a public AMI](#).

Create a new key pair ▼

Key pair name
[REDACTED]

Download Key Pair

... You have to download the **private key file** (*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created.

Cancel Launch Instances

Figura 2.3: Creazione di una nuova key pair

Una volta scelto un nome da assegnare al file key pair, si esegue un click su “Download Key Pair” e verrà generato un file con estensione .pem che servirà in seguito per connettere la nostra app all’istanza tramite il linguaggio Java[1].

Se tutto è stato eseguito in maniera corretta, si avrà l’istanza t2.micro in stato di running e sulla EC2 Dashboard risulterà uno scenario simile a quello mostrato in Figura 2.4. Notare che la nostra istanza risulta attiva a Francoforte. Questo perché è possibile scegliere anche la zona nella quale tenere acceso il nostro server. Essendo il nostro servizio verosimilmente erogato in Italia, si è preferito un server in Europa Centrale.



Resources	
You are using the following Amazon EC2 resources in the EU Central (Frankfurt) region:	
1 Running Instances	0 Elastic IPs
0 Dedicated Hosts	0 Snapshots
1 Volumes	0 Load Balancers
1 Key Pairs	2 Security Groups
0 Placement Groups	

Figura 2.4: Riepilogo istanze

2.2.4 Connessione all'istanza

Una volta terminata la configurazione, è possibile procedere con la procedura di connessione all'istanza. Fondamentalmente sono previste due tipologie di connessione: tramite SSH o con un client Java SSH direttamente dal browser. Verrà preferita la prima tipologia dal momento che la seconda non è più supportata dalla maggior parte dei web browser (Figura 2.5).

Connect To Your Instance

☒ I would like to connect with A standalone SSH client

☐ A Java SSH Client directly from my browser (Java required)

To access your instance:

1. Open an SSH client. (find out how to [connect using PuTTY](#))
2. Locate your private key file [REDACTED]. The wizard automatically detects the key you used to launch the instance.
3. Your key must not be publicly viewable for SSH to work. Use this command if needed:

```
chmod 400 [REDACTED]
```
4. Connect to your instance using its Public DNS:
[REDACTED]

Example:

```
ssh -i [REDACTED]
```

Please note that in most cases the username above will be correct, however please ensure that you read your AMI usage instructions to ensure that the AMI owner has not changed the default AMI username.

If you need any assistance connecting to your instance, please see our [connection documentation](#).

Close

Figura 2.5: Connessione all'istanza

Per far ciò si necessita di un software: PuTTY. Ancor prima di utilizzare PuTTY sarà necessario utilizzare “PuTTY Key Generator” il quale ci consentirà di convertire il file della chiave privata con estensione .pem ottenuto al Passo 3 in un file .ppk. Una volta generato tale file sarà possibile connettersi all’istanza tramite connessione SSH usando PuTTY inserendo le credenziali presenti nella figura precedente, sebbene oscurate per motivi di privacy. Bisogna tener presente che ogniqualvolta si dovesse stoppare l’istanza per poi riavviarla, AWS provvederà ad assegnare un nuovo indirizzo IP pubblico.

2.2.5 Ottenimento credenziali

Per poter connettere la propria applicazione è necessario effettuare l’accesso su PuTTY e salvare alcuni parametri presenti nel file server.js. Tali parametri verranno appositamente memorizzati sulla nostra applicazione tramite l’ausilio del linguaggio Java (Figura 2.6).

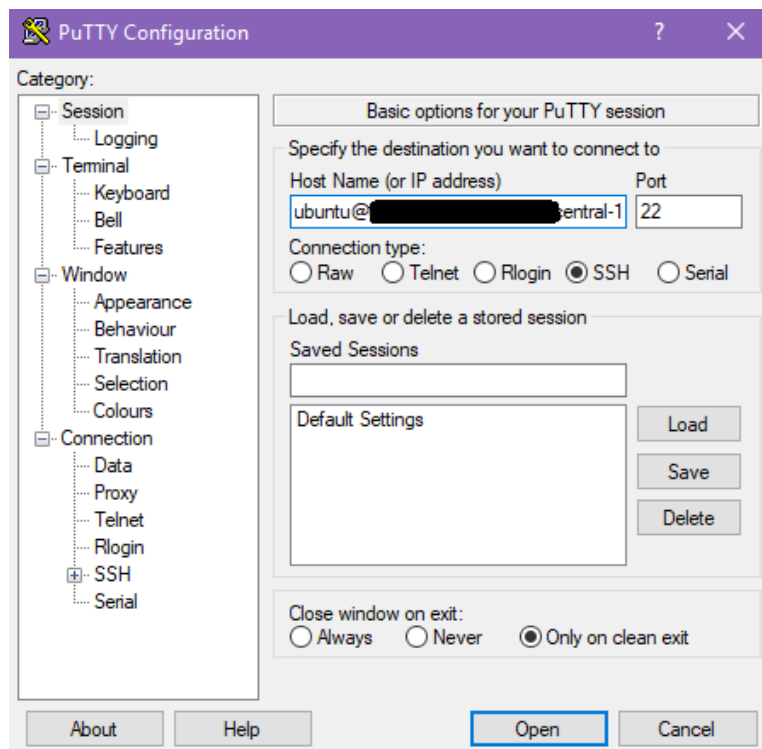
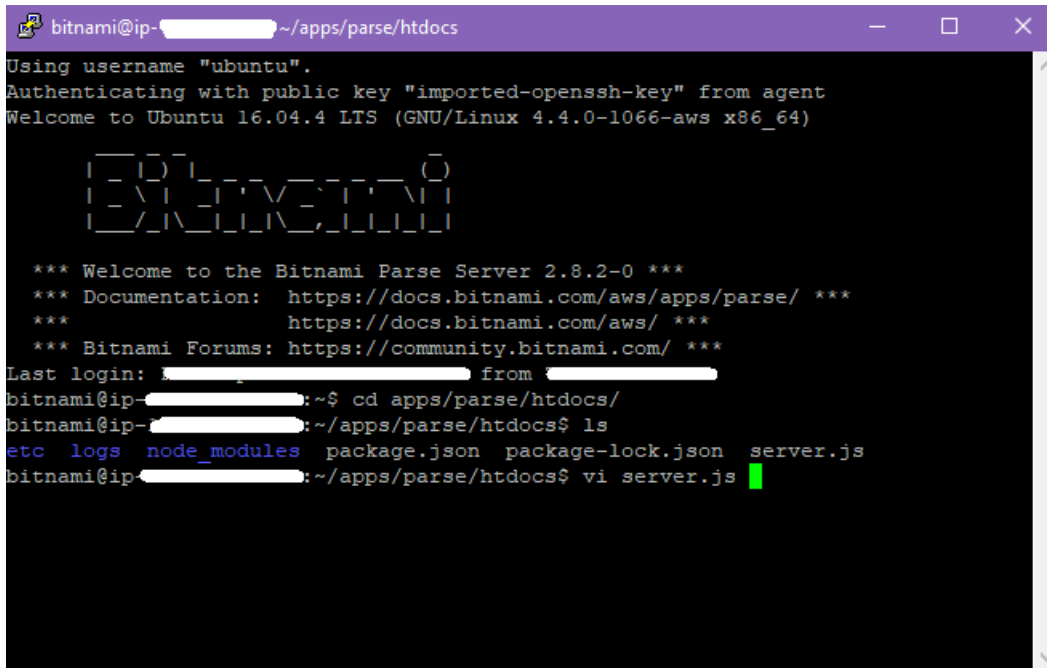


Figura 2.6: Connessione all’istanza tramite PuTTY

Una volta effettuato l'accesso su PuTTY, si avrà una schermata come quella mostrata in Figura 2.7.



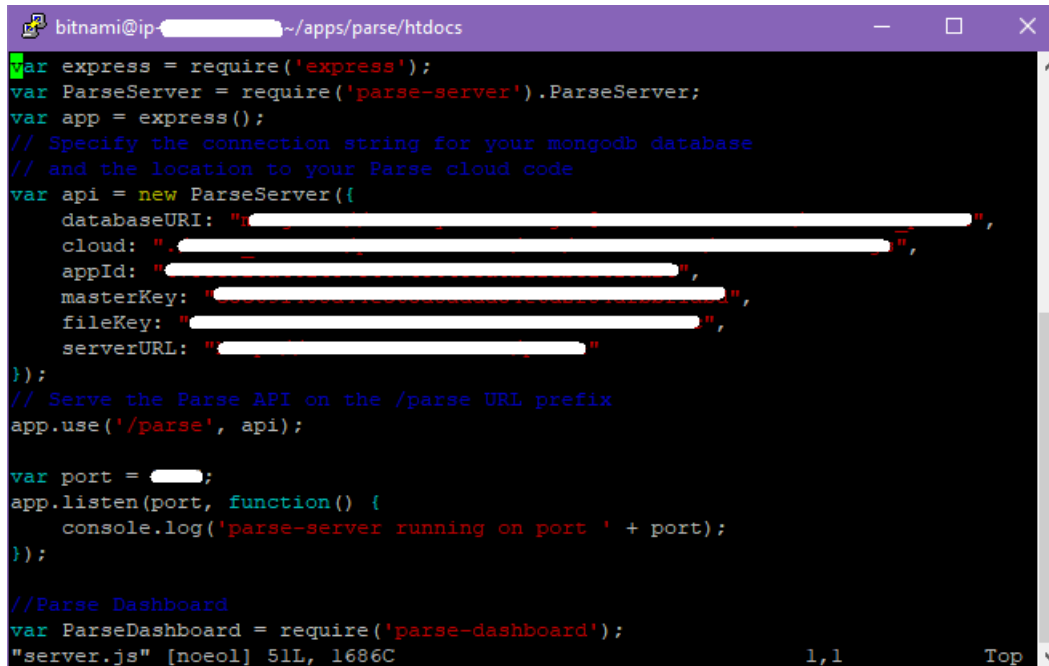
```
bitnami@ip-... ~/apps/parse/htdocs
Using username "ubuntu".
Authenticating with public key "imported-openssh-key" from agent
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-1066-aws x86_64)

  _ _ _ _ _
 | | | | |
 | | | | |
 | | | | |

*** Welcome to the Bitnami Parse Server 2.8.2-0 ***
*** Documentation: https://docs.bitnami.com/aws/apps/parse/ ***
*** https://docs.bitnami.com/aws/ ***
*** Bitnami Forums: https://community.bitnami.com/ ***
Last login: ... from ...
bitnami@ip-...:~$ cd apps/parse/htdocs/
bitnami@ip-...:~/apps/parse/htdocs$ ls
etc logs node_modules package.json package-lock.json server.js
bitnami@ip-...:~/apps/parse/htdocs$ vi server.js
```

Figura 2.7: Bitnami Parse Server

Ciò che ci interessa è posizionarci sotto la cartella `htdocs`, raggiungibile tramite il comando di change directory `cd/apps/parse/htdocs`. Ci serviamo dell'editor “vi” e digitiamo `vi server.js` (Figura 2.8).



```
bitnami@ip-... ~/apps/parse/htdocs
var express = require('express');
var ParseServer = require('parse-server').ParseServer;
var app = express();
// Specify the connection string for your mongodb database
// and the location to your Parse cloud code
var api = new ParseServer({
  databaseURI: "mongodb://...",
  cloud: "cloud/main.js",
  appId: "...",
  masterKey: "...",
  fileKey: "...",
  serverURL: "http://localhost:1337"
});
// Serve the Parse API on the /parse URL prefix
app.use('/parse', api);

var port = 1337;
app.listen(port, function() {
  console.log('parse-server running on port ' + port);
});

//Parse Dashboard
var ParseDashboard = require('parse-dashboard');
"server.js" [noeol] 51L, 1686C 1,1 Top
```

Figura 2.8: Server.js

I parametri `appId`, `masterKey` e `serverURL` verranno memorizzati all'interno di un metodo richiamato da un oggetto Parse in particolare presente nella nostra applicazione Android. Tale parte verrà discussa successivamente. Si conclude così la parte inerente alla creazione, configurazione e lancio dell'istanza che ci permetterà di interagire con la Dashboard di MongoDB.

2.3 Parse Server

La scelta di adottare Parse Server è dipesa prevalentemente dal fatto di essere supportato tramite la sua ampia community. Android[10], il linguaggio al quale ci siamo affidati per la realizzazione dell'applicazione, è fornito di un'ampia documentazione.

Qui di seguito vengono mostrati i passi per includere Parse Server all'interno della nostra app.

2.3.1 Installazione e dipendenze

Per poter utilizzare Parse Server per Android è necessario recarsi sul canale GitHub della community di Parse e scaricare il repository Parse-SDK-Android.[2]

Affinché si possano usare tutte le funzioni correttamente bisogna aggiungere nel file build.gradle (quello principale) le opportune dipendenze (Figura 2.9).

```
allprojects {  
    repositories {  
        ...  
        maven { url "https://jitpack.io" }  
    }  
}
```

Figura 2.9: Dipendenze build.gradle

Subito dopo, come appare in Figura 2.10, bisogna aggiungere al file build.gradle (del progetto stavolta):

```
dependencies {  
    implementation "com.github.parse-community.Parse-SDK-  
Android:parse:latest.version.here"  
}
```

Figura 2.10: Dipendenze build.gradle

La dicitura latest.version.here è stata rimpiazzata dall'ultima versione disponibile che risulta essere la 1.18.4.

2.3.2 Inizializzazione di Parse

Per poter interagire dalla nostra app con il server sul quale è montato Parse Server è necessario passare per Android Studio, IDE di sviluppo di Android.

Una volta creato un nuovo progetto, si procede con la creazione di una Java Class che per comodità chiamiamo App. Tale classe estenderà la classe Application e nel suo metodo onCreate() andrà ad implementare il metodo initialize di Parse come mostrato in Figura 2.11.

```
import com.parse.Parse;
import android.app.Application;

public class App extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        Parse.initialize(new Parse.Configuration.Builder(this)
            .applicationId("YOUR_APP_ID")
            // if defined
            .clientKey("YOUR_CLIENT_KEY")
            .server("http://localhost:1337/parse/")
            .build()
        );
    }
}
```

Figura 2.11: App class

I valori passati ai campi applicationId, clientKey e server saranno rimpiazzati da quelli memorizzati nel file server.js della precedente sezione.

Per evitare di incorrere in errori di compilazione è necessario registrare nell'AndroidManifest.xml la classe App con la seguente sintassi di Figura 2.12:

```
<application
    android:name=".App"
    ...>
</application>
```

Figura 2.12: AndroidManifest.xml

2.3.3 Accesso alla Parse Dashboard

Un aspetto molto comodo di Parse Server è la possibilità di poter gestire il nostro database tramite una Dashboard che risponde all'indirizzo sul quale è attiva la nostra istanza AWS.

Supponendo che l'indirizzo IP della nostra istanza sia 52.28.191.143, ci reheremo all'indirizzo: <http://52.28.191.143/apps> e verrà caricata la schermata di login d'accesso alla nostra Dashboard. Generalmente l'username di default è user (Figura 2.13).

Access your Dashboard

Username	user
Password

Figura 2.13: Parse Dashboard login

Mentre la password può essere ricavata dal pannello dell'istanza AWS cliccando con il tasto destro nella zona indicata in Figura 2.14 ed accedendo alla sezione Get System Log. Scorrendo in fondo ai log, si vedrà un riquadro con la password in chiaro. Tale password sarà la password di login sulla Dashboard di Parse.

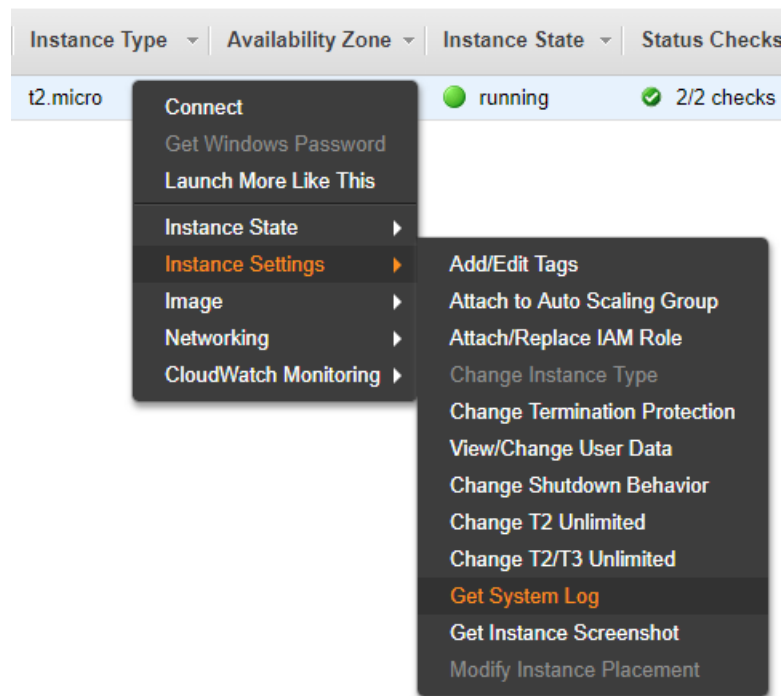


Figura 2.14: Ottenimento credenziali Dashboard

La Dashboard appare così come mostrato in Figura 2.15. La struttura che si vede sulla sinistra è quella adottata per la nostra applicazione.

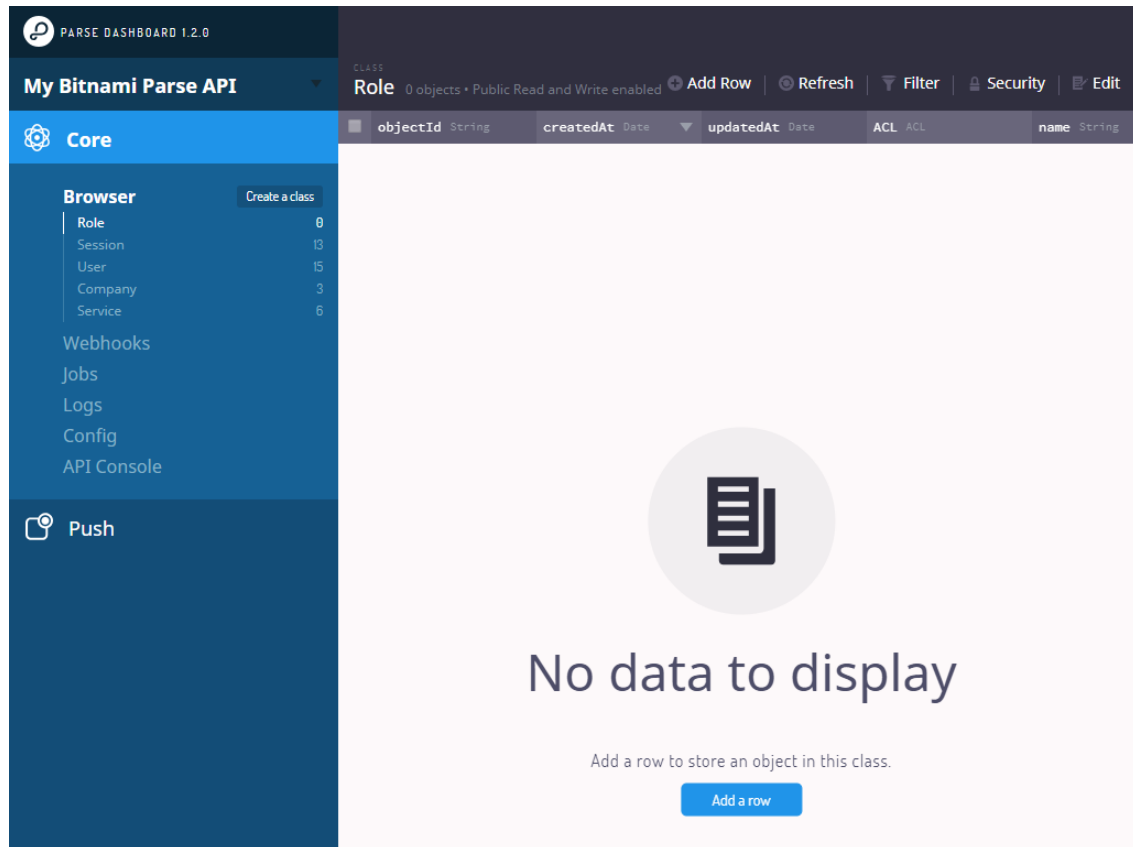


Figura 2.15: Parse Dashboard

Ora la Dashboard è accessibile ed il progetto Android è pronto per interfacciarsi con il database.

Capitolo 3

Progettazione ed implementazione

In tale sezione si andrà a presentare una sintesi dei requisiti che mira a descrivere il servizio che si vuole offrire e la sua logica di funzionamento. Inoltre verranno spiegati in dettaglio gli accorgimenti adottati in termini di progettazione, di architettura e di implementazione dell'applicazione.

3.1 Raccolta dei requisiti

Si vuole realizzare un applicativo che permetta agli studenti universitari di accedere ai servizi del proprio Ateneo quali Segreteria Didattica, Studenti, etc. di riservare un turno in coda sebbene tali studenti non siano fisicamente presenti dinanzi al front-office che eroga i servizi sopracitati. Il sistema dovrà tener conto del fatto che bisogna dotare l'Ateneo di apposito software in modo tale da rendere possibile il corretto funzionamento dell'intera piattaforma. Affinché gli studenti possano effettivamente mettersi in coda per ottenere informazioni dinanzi ad uno sportello universitario, sarà necessario che lo studente sia localizzato entro un determinato raggio dichiarato dall'Ateneo. Ciò comporterà che qualora l'utente dell'app si dovesse localizzare tramite il suo dispositivo in un'area che non soddisfa il vincolo, non riuscirà a visualizzare la sede dell'Ateneo con la rispettiva lista di sportelli presso i quali intendeva mettersi in coda. Questo vincolo è stato necessario imporlo altrimenti si sarebbe permesso a studenti non fisicamente presenti nei pressi della sede universitaria di riservarsi un turno in coda. Aggiungiamo che tale vincolo è attualmente previsto soltanto “in ingresso” nell'area in cui sono presenti i servizi e non anche “in uscita”. Ciò vuol dire che una volta entrati in coda, ci si può allontanare dal posto senza perdere l'accesso. Le entità da prevedere all'interno del sistema sono fondamentalmente due: la prima risulta essere quella del responsabile del servizio che si preoccupa di aprire lo sportello e di avanzare la numerazione tramite il software di gestione della coda; la seconda invece, viene rappresentata dagli studenti, i quali dovranno disporre di un'applicazione installata sul loro dispositivo tramite la quale, consentendo il rilevamento della posizione, riusciranno ad ottenere l'accesso ai

servizi di Ateneo. Sulla questione del vincolo imposto sul range entro il quale bisogna trovarsi, se ne discuterà ampiamente nel capitolo conclusivo.

3.1.1 Use Case Diagram

In Figura 3.1 viene mostrato un diagramma dei casi d'uso che mira a rappresentare come interagiscono tra loro le entità del sistema e a quali operazioni è concesso accedere.

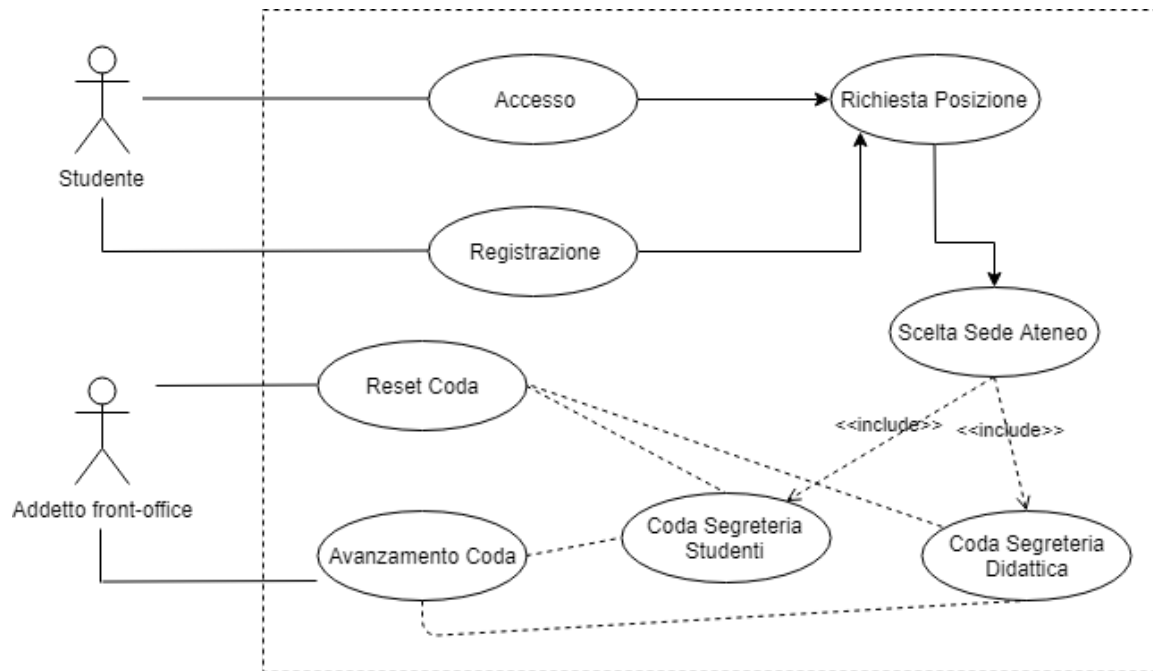


Figura 3.1: Use Case Diagram

3.2 Architettura dell'applicazione

Il modello architetturale della seguente applicazione, come è stato parzialmente anticipato nella sintesi dei requisiti, risulta essere basato su due entità: un gestore universale delle code ed i client rappresentati dagli studenti. Quello che si propone è un modello molto semplificato della realtà, dal momento che non disponendo di hardware a sufficienza, è stato più semplice compattare la gestione di tutte le code in un unico applicativo. Teoricamente ci sarebbe dovuto essere un applicativo per ogni singola sede universitaria. Essendo quello proposto, un prototipo, tale soluzione è stata sufficiente ai fini della dimostrazione del funzionamento della piattaforma. Andando a schematizzare tramite un diagramma ciò appena detto, avremo lo scenario di Figura 3.2.

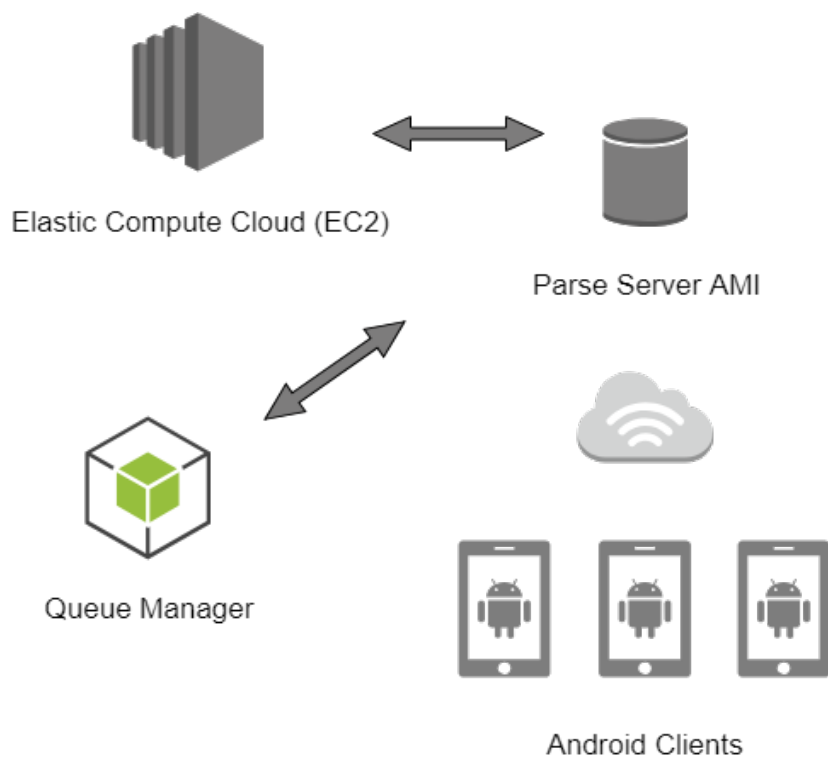


Figura 3.2: Architettura del sistema

Risulta dunque una parte infrastrutturale composta dall'Elastic Compute Cloud di AWS vista in dettaglio in precedenza, la quale ospita Parse Server. Il gestore delle code si interfaccia con il database e si occupa dell'apertura, della chiusura e dell'avanzamento della numerazione delle code. I client Android invece sono i dispositivi di cui sono dotati gli studenti che si interfacciano con il servizio.

3.2.1 Struttura del database

Per quanto concerne la struttura del database, non è stato adottato un meccanismo di persistenza dei dati relazionale. Questo perché la filosofia del salvataggio dei dati in Parse è molto document-store oriented. Questo dipende ovviamente dagli ingenti investimenti che sono stati fatti da Parse su MongoDB. Sebbene Parse non sia stato ideato per nient'altro se non per MongoDB, è possibile far girare anche PostgreSQL. Ovviamente ci sono degli inconvenienti. Bisogna aggiungere delle primary key in maniera casuale all'interno delle proprie tabelle, si esegue una ALTER TABLE ma alla fine si ha un ORM (Object Relational Mapping) con la possibilità di eseguire delle live-query a tutti gli effetti.

Ai fini della nostra applicazione, si è preferito utilizzare MongoDB sfruttando la documentazione di Parse Server a riguardo. Ma qualora si fosse voluto, si sarebbe potuto cambiare il valore di databaseURI all'interno del file server.js visto in precedenza nel quale al posto di avere: mongodb:// si sarebbe avuto postgres://. Questo perché, quando databaseURI inizia con mongodb:// viene automaticamente utilizzato MongoStorageAdapter (come nel nostro caso). Quando invece comincia con postgres://, viene richiamato PostgresStorageAdapter.[3]

Come detto in apertura di paragrafo, i dati in Parse vengono salvati tramite un document database. Tale tipologia di memorizzazione è propria dei database NoSQL. Tali database si dividono in più categorie e sono: key-valued stores, column family stores, document databases e graph databased. Non verranno spiegati tutti in dettaglio ma proviamo a dare una definizione sommaria per ognuna di queste categorie.

Nei key-valued stores è permesso memorizzare e recuperare molto velocemente coppie di chiave e valore. Questa tipologia di database è quella più facile da implementare grazie alla loro semplice struttura e sono anche i più performanti.

Nei column family stores i dati sono ottimizzati per la ricerca su colonne. Sono solitamente impiegati nella gestione di grandi quantità di dati distribuiti su più server. Uno dei database column family è Cassandra.

I document databases gestiscono i dati in un formato documentale semi strutturato. Qui, viene data la possibilità di avere una struttura diversa per ogni record. Questo tipo di database è consigliato quando sorge il bisogno di memorizzare dati con strutture variabili. Inoltre sono comodi per una gestione di dati geo spaziali.

Infine, i graph databases sono database realizzati a grafi nei quali i nodi e gli archi vengono utilizzati per memorizzare informazioni. Un classico impiego di questi database è con i social network.[9]

3.2.2 Classi Parse Dashboard

Le classi realizzate sulla Parse Dashboard sono fondamentalmente tre: User, Company e Service. Le altre sono gestite automaticamente da Parse e non ci occorre menzionarle. Difatti, la struttura richiesta per realizzare la piattaforma, non è particolarmente complessa al livello di astrazione preso da noi in considerazione. L'unica "relazione" vera e propria sussiste con i record inseriti nella classe Service che dipendono da una sede universitaria ben precisa che fa capo alla classe Company. Per esprimere tale collegamento si è fatto uso di un Pointer.

L'attributo objectId è univoco ed è automaticamente gestito come String da Parse Server e viene assegnato ad ogni creazione di un nuovo record.

ACL che letteralmente sta per Access Control List va a regolare i meccanismi che determinano l'accesso o meno da parte degli utenti a particolari risorse. Nel nostro caso sono impostati su Public Read + Write.

updatedAt e createdAt sono di tipo Date ed anche questi vengono gestiti automaticamente da Parse Server qualora si dovessero effettuare modifiche sul record o lo si stesse creando per la prima volta.

Gli attributi sulla classe Company: COMPANY_NAME e COMPANY_POSITION rappresentano rispettivamente il nome della sede universitaria e le relative coordinate espresse in latitudine e longitudine tramite un oggetto GeoPoint.

Gli attributi presenti sulla classe Service: SERVICE_NAME, CURRENT_NUMBER e MY_NUMBER rappresentano il nome dello sportello, il numero corrente della coda di tale sportello e l'ultimo numero rilasciato per tale coda.

Andando a schematizzare le tre entità appena elencate, avremo un diagramma come mostrato in Figura 3.3.

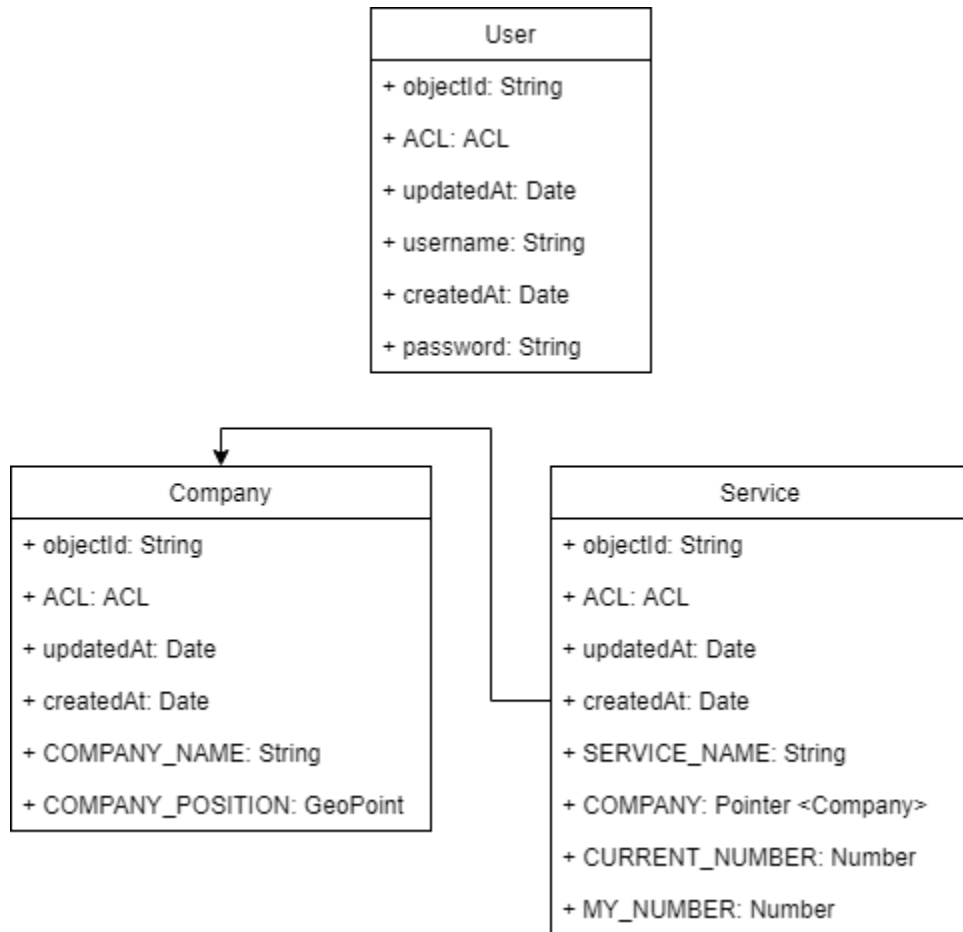


Figura 3.3: Classi Parse Dashboard

3.2.3 Gestione degli utenti

La gestione degli utenti è un aspetto importante da tenere a mente dal momento che si entra a contatto con dati sensibili e bisogna saper bene come trattarli. Uno dei vantaggi dell'aver adottato Parse Server risiede anche nella gestione delle password degli utenti. Essendo la nostra applicazione un prototipo che mira ad illustrare solo il carattere dell'idea generale e non disponendo dell'accesso a matricole e password degli studenti, è stato necessario introdurre una registrazione. Qualora gli atenei adottassero una piattaforma come quella da noi proposta, sarebbe loro cura garantirne l'accesso tramite le credenziali che gli studenti utilizzano solitamente per effettuare operazioni confinate all'ambiente istituzionale. Nel nostro caso invece, è richiesta una breve e semplice registrazione che richiede un nome utente ed una password. Come detto in apertura di paragrafo, Parse ci garantisce che le password vengano gestite in maniera sicura. Questo perché sono memorizzate nel database come hash e sono calcolate ad una via. L'hash della password viene tenuta lontana dalle SDK del client per motivi di sicurezza.

La classe che ci permette una facile gestione degli utenti prende il nome di ParseUser la quale è una sottoclasse di ParseObject. La classe ParseUser possiede diverse proprietà che la distinguono da ParseObject come ad esempio: username (obbligatorio), password (obbligatorio solo in fase di registrazione) ed email (opzionale).

3.3 UML Class Diagram

Qui di seguito è mostrato il diagramma delle classi che compongono il sistema realizzato. Sono inizialmente mostrate le classi che rappresentano la parte client della piattaforma. Successivamente, quelle che costituiscono il gestore delle code.

3.3.1 Client

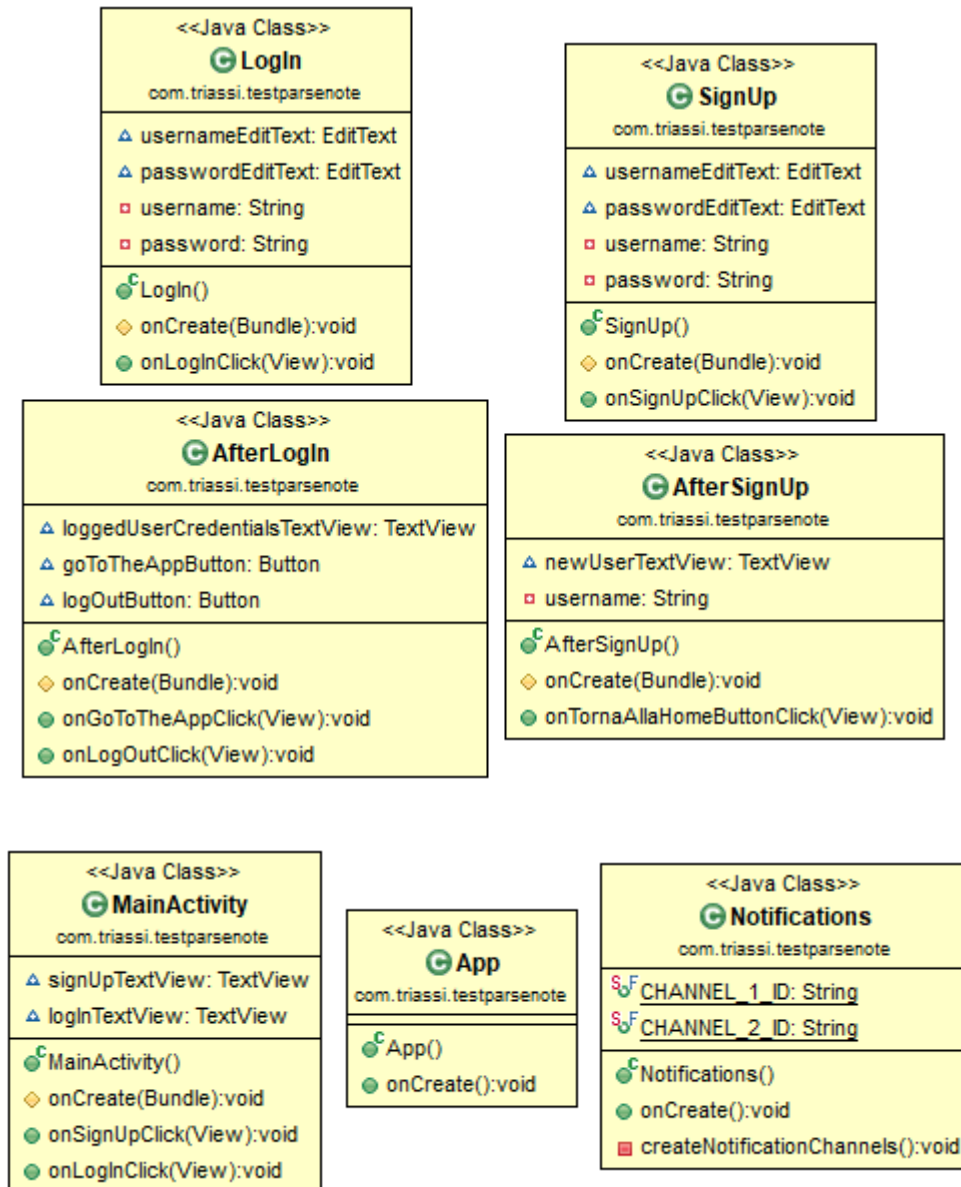


Figura 3.4: Client UML Class Diagram



Figura 3.5: Client UML Class Diagram

3.3.2 Gestore code

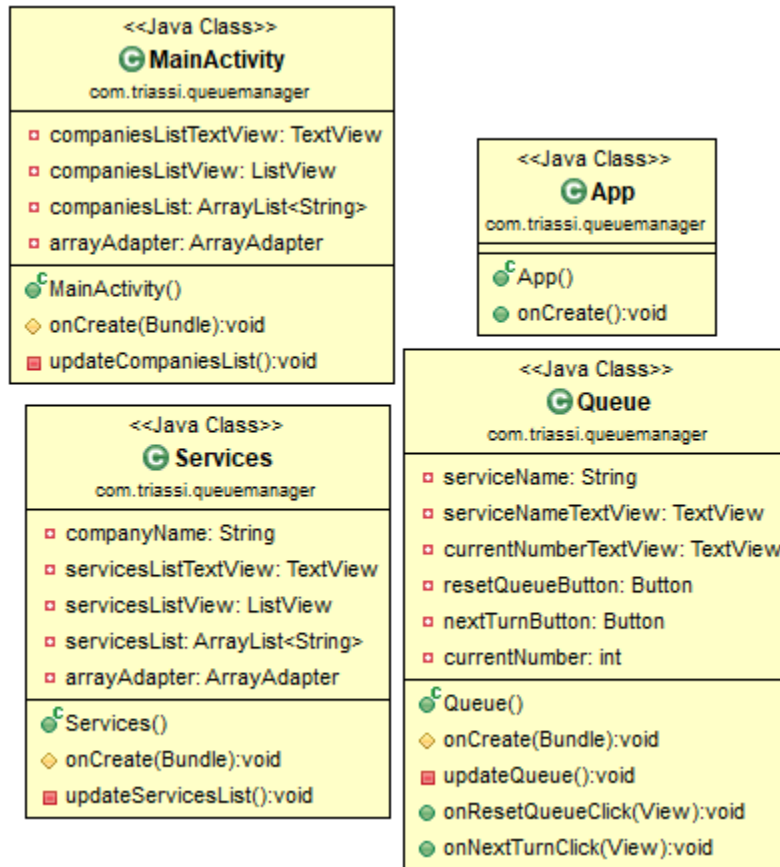


Figura 3.6: Queue Manager UML Class Diagram

3.4 Funzioni gestore code

Nella seguente sezione verrà discussa l'implementazione avvenuta tramite il linguaggio Java delle funzioni realizzate per l'applicativo in esame. Tale argomentazione verrà suddivisa in due parti. La prima riguarderà l'applicativo dato in gestione agli addetti dei front-office. La seconda invece, consisterà nell'applicativo offerto agli studenti.

Partiamo con il descrivere le classi fondamentali realizzate per il gestore delle code. Per le classi più significative, viene allegato anche il codice.

3.4.1 App Class

La classe App è di particolare importanza per l'intero sistema dal momento che abilita l'applicativo alla connessione con il database ospitato dall'istanza AWS. I parametri che vanno a caratterizzare tale connessione sono già stati descritti in precedenza ma per completezza li riportiamo anche qui. Come evidente da codice, si invoca il metodo initialize della classe Parse. In input a tale metodo passiamo i valori di applicationId, clientKey e server recuperati tramite il file server.js. Per una questione di sicurezza, tali parametri verranno oscurati.

```
1 public class App extends Application
2 {
3     @Override
4     public void onCreate()
5     {
6         super.onCreate();
7
8         Parse.enableLocalDatastore(this);
9
10        Parse.initialize(new Parse.Configuration.Builder(this)
11            .applicationId("applicationIdHere")
12            .clientKey("clientKeyHere")
13            .server("http://ipServerHere:portServerHere/parse/")
14            .build()
15        );
16    }
17 }
```

3.4.2 MainActivity Class

La classe che va a richiamare l'activity principale del gestore code è la MainActivity. In tale activity sono fondamentalmente implementate due funzioni. Quella di aggiornamento delle sedi a disposizione sul backend Parse e quella di gestione del tap su un item della ListView che funge da ponte verso l'activity che gestisce i servizi disponibili presso ogni sede.

Per quanto riguarda la lista delle sedi mostrate appena l'applicazione viene lanciata, si è fatto uso di un oggetto ListView. Tale ListView viene popolata

eseguendo un'interrogazione al database sulla classe Company. Ogni riga della ListView viene memorizzata in un ArrayList di stringhe. Affinché gli aggiornamenti sul backend fossero visibili in real-time sull'app, si è fatto uso di un ArrayAdapter al quale è stato agganciato l'ArrayList dichiarato. Una volta popolata la ListView è bastato richiamare il metodo `notifyDataSetChanged` sull'arrayAdapter per ottenere dati sincronizzati in tempo reale con il database.

Per implementare un comportamento di “ascolto” sulla ListView, in modo tale da catturare eventuali tap da parte dell'utente, è stato necessario richiamare il metodo `setOnClickListener`, il quale tramite opportuna gestione degli indici della ListView, restituisce la riga sulla quale si è eseguito un tap. Si provvede a salvare il nome della sede su cui si è fatto il tap e si rimanda l'utente ad una nuova activity. Per passare il nome della sede tra le due activity si è utilizzato il metodo `putExtra` al quale si sono passati due parametri: una stringa che identificasse l'attributo da recuperare nella nuova intent e la stringa contenente l'effettivo nome della sede. Per richiamare l'activity invece basta invocare il metodo `startActivity` passando come parametro l'oggetto della classe Intent istanziato poco prima.

3.4.3 Services Class

La classe Services rappresenta l'activity preposta alla visualizzazione dei servizi offerti da ogni singola sede universitaria. Anche in questo caso viene gestito tutto tramite l'ausilio di una ListView che viene popolata tramite un ArrayList di stringhe ed un arrayAdapter che permette una consistenza dei dati memorizzati sul database. Ciò che non è stato ancora illustrato è il processo di ottenimento degli attributi valorizzati nell'activity di provenienza. Nel metodo `onCreate` si dichiara un oggetto della classe Intent e si provvede a richiamare `getIntent`. Adesso è possibile ricavare il nome della sede sulla quale si è eseguito il tap nell'activity di provenienza tramite l'invocazione del metodo `getStringExtra` sull'oggetto intent passando come argomento la stringa che identifica l'attributo da voler recuperare.

3.4.4 Queue Class

Per quanto concerne la realizzazione della classe Queue, andremo ad isolare le singole funzioni in modo tale da commentarle più in dettaglio.

La parte iniziale della classe comprende la dichiarazione dei vari elementi grafici che compaiono nel frontend quali Button e TextView. Nel metodo `onCreate` eseguiamo il binding di tali elementi e recuperiamo il servizio sul quale si è eseguito il tap. Successivamente viene richiamata la funzione di aggiornamento dello stato della coda.


```

1 public class Queue extends AppCompatActivity {
2
3     private String serviceName;
4     private TextView serviceNameTextView;
5     private TextView currentNumberTextView;
6     private Button resetQueueButton;
7     private Button nextTurnButton;
8     private int currentNumber;
9
10    @Override
11    protected void onCreate(Bundle savedInstanceState) {
12        super.onCreate(savedInstanceState);
13        setContentView(R.layout.activity_queue);
14
15        serviceNameTextView = findViewById(R.id.serviceNameTextView);
16        currentNumberTextView = findViewById(R.id.currentNumberTextView);
17        resetQueueButton = findViewById(R.id.resetQueueButton);
18        nextTurnButton = findViewById(R.id.nextTurnButton);
19
20        Intent intent = getIntent();
21        serviceName = intent.getStringExtra("service_tapped");
22        serviceNameTextView.setText(serviceName);
23
24        updateQueue();
25    }

```

Vediamo adesso in dettaglio l'implementazione della funzione `updateQueue`. Tale funzione fa uso di un `Handler` e di un `Runnable` dal momento che nasce l'esigenza di notificare in tempo reale un aggiornamento del valore del numero corrente della coda sul frontend. Tale `Runnable` viene eseguito ogni 1000ms. La funzione esegue una interrogazione sulla classe `Service` e si posiziona sul record con nome del servizio uguale a quello sul quale si è eseguito il tap. Il risultato di tale interrogazione viene recuperato tramite una `List<ParseObject>` che conterrà un solo record. Su tale record andremo a recuperare il valore di `CURRENT_NUMBER` che rappresenta il numero corrente della coda selezionata. Qualora tale risorsa dovesse subire un cambiamento, tramite il `Runnable` verrà prontamente notificato il nuovo valore sul frontend.

```

1 private void updateQueue()
2 {
3     final Handler handler = new Handler();
4     Runnable runnable = new Runnable() {
5         @Override
6         public void run()
7         {
8             ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
9             query.whereEqualTo("SERVICE_NAME", serviceName);
10            query.findInBackground(new FindCallback<ParseObject>()
11            {
12                public void done(List<ParseObject> services, ParseException e)
13                {
14                    if (e == null)
15                    {
16                        if(services.size() > 0)
17                        {
18                            for (ParseObject service : services)

```

```

19         {
20             currentNumber = service.getInt("CURRENT_NUMBER");
21             currentNumberTextView.setText(String.valueOf(currentNumber));
22         }
23     }
24 }
25 }
26 });
27
28
29     handler.postDelayed(this, 1000);
30 }
31 };
32 handler.postDelayed(runnable, 1000);
33 }

```

Una funzione messa a disposizione dello sportellista è quella di reset della coda. Tale funzione può essere impiegata per due motivi: chiusura della coda o semplice reset della numerazione. Per quanto riguarda tale operazione, si esegue una semplice interrogazione sulla classe Service e si filtra per SERVICE_NAME. Ricordiamo che tutti i valori assunti da SERVICE_NAME sono univoci quindi ogni interrogazione di questo tipo restituirà sempre e soltanto un record. Una volta eseguita tale interrogazione si effettua una operazione di ALTER TABLE andando a modificare i valori degli attributi CURRENT_NUMBER che viene settato a -1 e MY_NUMBER portato a 0. Praticamente il valore -1 assume il significato di coda chiusa.

```

1 public void onResetQueueClick(View view)
2 {
3     ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
4     query.whereEqualTo("SERVICE_NAME", serviceName);
5     query.findInBackground(new FindCallback<ParseObject>()
6     {
7         public void done(List<ParseObject> services, ParseException e)
8         {
9             if (e == null)
10             {
11                 if(services.size() > 0)
12                 {
13                     for (ParseObject service : services)
14                     {
15                         service.put("CURRENT_NUMBER", -1);
16                         service.put("MY_NUMBER", 0);
17                         service.saveInBackground();
18
19                         currentNumberTextView.setText(String.valueOf(service.getInt("
20 CURRENT_NUMBER"))));
21                     }
22                 }
23             }
24         });
25 }

```

Un'altra funzione messa a disposizione dello sportellista è quella dell'avanzamento della numerazione della coda. Molto semplicemente si procede in modo analogo a quanto visto per la funzione precedente con la differenza che si va

ad incrementare il valore del campo CURRENT_NUMBER di un'unità. Tale operazione va a rappresentare il momento in cui si intende ricevere uno studente dinanzi allo sportello avanzando la numerazione del servizio di proprio interesse.

```
1 public void onNextTurnClick(View view)
2 {
3     ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
4     query.whereEqualTo("SERVICE_NAME", serviceName);
5     query.findInBackground(new FindCallback<ParseObject>()
6     {
7         public void done(List<ParseObject> services, ParseException e)
8         {
9             if (e == null)
10             {
11                 if(services.size() > 0)
12                 {
13                     for (ParseObject service : services)
14                     {
15                         service.increment("CURRENT_NUMBER");
16                         service.saveInBackground();
17                         currentNumberTextView.setText(String.valueOf(service.getInt("
18 CURRENT_NUMBER"))));
19                     }
20                 }
21             }
22         });
23     }
```

3.5 Funzioni client

In tale sezione viene conclusa la trattazione inerente all'implementazione delle feature dell'app. Vediamo le funzioni più importanti realizzate per la parte che riguarda gli studenti.

3.5.1 MainActivity Class

L'activity principale dell'applicativo utilizzato dagli studenti risulterà essere composta da una schermata di introduzione al servizio con la possibilità di essere reindirizzati ad una finestra di accesso o di registrazione. La prima volta che si lancia l'app occorre effettuare una veloce registrazione che richiede un nome utente ed una password senza particolari vincoli sulla lunghezza e/o composizione di quest'ultima. L'activity in esame prevede l'implementazione di due funzioni: onSignUpClick e onLogInClick. Entrambi i metodi vengono invocati allo scatenarsi dell'evento onClick sui due oggetti grafici TextView dichiarati in cima alla classe. Molto semplicemente, in tali metodi, si reindirizza l'utente alla activity adibita ad effettuare l'accesso o registrazione che si voglia. Nel metodo onCreate sono presenti dei controlli che verranno discussi all'atto della presentazione del sistema di funzionamento della coda.

3.5.2 SignUp Class

Presentiamo adesso la classe che incorpora l'activity che descrive la fase di registrazione al servizio che intendiamo offrire. Tale registrazione è stata implementata in maniera molto semplice facendo uso della classe ParseUser messa a disposizione dalle API di Parse Server. Il metodo che si occupa di gestire tale operazione è onSignUpClick e viene invocato una volta compilate correttamente le EditText che incapsulano il nome utente e la password del futuro utente della piattaforma. La procedura di registrazione avviene istanziando un oggetto della classe ParseUser e successivamente invocando i metodi setUsername e setPassword su tale oggetto. Per completare la registrazione infine, si invoca il metodo signUpInBackground sull'oggetto istanziato. Tale metodo gestisce tramite una callback, l'esito dell'operazione. Se non viene generata alcuna eccezione ParseException, allora si notifica tramite un Toast che l'operazione di registrazione è andata a buon fine e si reindirizza l'utente ad una nuova activity, altrimenti si notifica il fallimento dell'iscrizione e si rimane sulla stessa activity.

3.5.3 AfterSignUp Class

In caso di registrazione effettuata con successo, si viene reindirizzati all'activity AfterSignUp. Tale activity ha la funzione di schermata di transizione tra la fase di iscrizione alla piattaforma e l'effettivo accesso ad essa. L'unico metodo da menzionare è onTornaAllaHomeButtonClick il quale riporta l'utente all'activity MainActivity.

3.5.4 LogIn Class

La parte che si occupa della gestione del processo di accesso all'applicazione è implementata nella classe LogIn. Anche in questo caso, le API di Parse Server ci vengono in soccorso. Il metodo onLogInClick viene invocato allo scatenarsi dell'evento onClick su un oggetto della classe Button nel frontend presupponendo che i campi nome utente e password siano valorizzati. L'operazione di login viene implementata tramite il metodo logInBackground invocato sulla classe ParseUser. A tale metodo vengono passati in input username, password ed una LogInCallback che restituisce un ParseUser ed una ParseException qualora rilevata. Tale callback implementa il metodo done di cui facciamo Override. Se user è diverso dal valore null allora vuol dire che il login è stato effettuato con successo e in tal caso richiamiamo un Toast sul frontend con conseguente reindirizzamento ad una nuova activity. Altrimenti notificiamo il fallimento. Alleghiamo per chiarezza la funzione di logIn.

```
1 public void onLogInClick(View view)
2 {
3     username = usernameEditText.getText().toString();
4     password = passwordEditText.getText().toString();
```

```

5
6     Log.i("Username content", username);
7
8     ParseUser.logInInBackground(username, password, new LogInCallback() {
9         @Override
10         public void done(ParseUser user, ParseException e) {
11             if(user != null)
12                 {
13                     Log.i("LogIn Info", "The user is logged in.");
14                     Toast.makeText(getApplicationContext(), "Accesso effettuato con successo!",
15                         Toast.LENGTH_SHORT).show();
16                     Intent intent = new Intent(getApplicationContext(), AfterLogIn.class);
17                     intent.putExtra("username", username);
18                     startActivity(intent);
19                     finish();
20                 }
21             else
22                 {
23                     Toast.makeText(getApplicationContext(), "I dati inseriti non sono corretti!",
24                         Toast.LENGTH_SHORT).show();
25                     Log.i("LogIn Info", "The user is NOT logged in.");
26                 }
27         }
28     });
29 }

```

3.5.5 AfterLogIn Class

Dopo aver eseguito il login con successo si viene reindirizzati all'activity gestita dalla classe AfterLogIn. Tale classe funge da activity di transizione tra la fase di login sulla piattaforma e l'accesso vero e proprio ai servizi. AfterLogIn si compone di due metodi: onGoToTheAppClick ed onLogOutClick. Entrambi i metodi vengono invocati allo scatenarsi dell'evento onClick su due oggetti della classe Button. In onGoToTheAppClick si reindirizza l'utente all'activity che gestisce il rilevamento della sua posizione tramite l'ausilio del GPS provider. In onLogOutClick si procede con la fase di disconnessione dello studente con conseguente reindirizzamento alla MainActivity.

3.5.6 UserLocation Class

La classe più rappresentativa dell'intero sistema risulta essere sicuramente UserLocation. Questo è dovuto dal fatto che in tale classe viene implementato un sistema di localizzazione della posizione dell'utente. Funzione cardine per il funzionamento della piattaforma. Uno dei moduli di cui si fa più uso nelle app è proprio quello della geolocalizzazione. In Android c'è ampia documentazione a riguardo e per questo motivo vale la pena spendere qualche parola in più a riguardo. Ottenere la posizione di un utente risulta abbastanza facile se si fa affidamento ad un paio di metodi delle API messe a disposizione. Il reale problema sorge quando si intendono ricevere posizioni accurate e si mira ad ottimizzare l'utilizzo della batteria. Per risolvere queste questioni è necessario affidarsi ad una specifica API che prende il nome di Fused Location API, la

quale combina i segnali intercettati tramite GPS con quelli delle celle Wi-Fi e del proprio gestore. Affinché si possa eseguire una richiesta di ricezione della posizione del device di nostro interesse, è necessario che i relativi permessi siano attivi. Parliamo nello specifico di `ACCESS_COARSE_LOCATION` che si limita a dare una posizione approssimativa con un margine di errore nell'ordine di un paio di chilometri. E di `ACCESS_FINE_LOCATION` che restituisce una posizione precisa nell'ordine di pochi metri. Per ricevere aggiornamenti sulla posizione, ci sono diversi parametri da fissare in modo tale da avere il risultato da noi voluto. Il metodo `getLastLocation` restituisce l'ultima posizione rilevata. Quando la posizione non è disponibile, restituisce `null`. Per quanto concerne le impostazioni necessarie per poter autorizzare il device ad utilizzare il GPS od il Wi-Fi, è possibile richiedere esplicitamente i permessi facendo mostrare un popup di conferma di questi ultimi così come mostrato in Figura 3.7 [13].

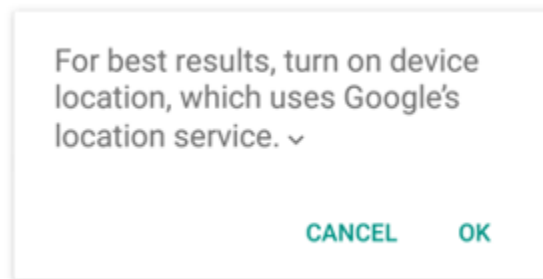


Figura 3.7: Permessi Google's location service

I parametri da valorizzare di cui si è anticipato in precedenza risultano essere: l'intervallo di aggiornamento della posizione che viene espresso in millisecondi; il limite superiore di tale intervallo (sempre espresso in millisecondi); la priorità, che va a determinare la precisione con cui la posizione verrà recuperata. Più questa è alta, più si tende a vedersi restituita una posizione accurata. La priorità può assumere i valori di BALANCED, HIGH, LOW o NO_POWER. È importante menzionare che nel file AndroidManifest.xml va aggiunto il permesso per utilizzare il GPS. Autorizzazione che viene data aggiungendo tale sintassi:

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

Occorre adesso aggiungere le dipendenze necessarie nel build.gradle (Module: app). Per una questione di chiarezza, mostriamo l'intero file posizionato sotto la cartella Gradle Scripts. Le dipendenze aggiunte sono quelle relative al play-services-location, ai permessi dexter e al binding di ButterKnife.

```
1 apply plugin: 'com.android.application'
2
3 android {
4     compileSdkVersion 27
5     defaultConfig {
6         applicationId "com.triassi.testparsenote"
7         minSdkVersion 21
8         targetSdkVersion 27
9         versionCode 1
10        versionName "1.0"
11        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
12    }
13    buildTypes {
14        release {
15            minifyEnabled false
16            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
17        }
18    }
19 }
20
21 dependencies {
22     implementation fileTree(dir: 'libs', include: ['*.jar'])
23     implementation 'com.android.support:appcompat-v7:27.1.1'
24     implementation 'com.android.support.constraint:constraint-layout:1.1.2'
25     implementation 'com.google.android.gms:play-services-maps:15.0.1'
26
27     implementation "com.google.android.gms:play-services-location:15.0.1"
28     // dexter runtime permissions
29     implementation 'com.karumi:dexter:4.2.0'
30     // ButterKnife view binding
31     implementation 'com.jakewharton:butterknife:8.8.1'
32     annotationProcessor 'com.jakewharton:butterknife-compiler:8.8.1'
33     implementation 'com.android.support:design:27.1.0'
34
35     testImplementation 'junit:junit:4.12'
36     androidTestImplementation 'com.android.support.test:runner:1.0.2'
37     androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
38     implementation "com.github.parse-community.Parse-SDK-Android:parse:1.18.4"
39 }
```

Fatta questa panoramica generale sulle API che verranno impiegate, adesso possiamo alla spiegazione della parte implementativa della classe.

Tra i parametri più significativi da descrivere c'è `RADIUS_AREA_IN_KILOMETERS` che è definito come costante e va a rappresentare il raggio massimo espresso in chilometri dalla posizione localizzata dell'utente per il quale si scansionano sedi universitarie disponibili. In fase di testing, per comodità è stato settato un raggio di 40km. In realtà verrà settato nell'ordine di un paio di chilometri.

Dopodiché menzioniamo la `ListView` che incapsula tramite un `ArrayList` di stringhe, le singole sedi appena scansionate. Come già spiegato per altre funzioni implementate, si fa uso di un `ArrayAdapter` che ci garantisce la consistenza dei dati memorizzati sul database.

```
1 public class UserLocation extends AppCompatActivity {
2     private static final String TAG = UserLocation.class.getSimpleName();
3     public static final int RADIUS_AREA_IN_KILOMETERS = 40;
4     private String mLastUpdateTime;
5     private static final long UPDATE_INTERVAL_IN_MILLISECONDS = 10000;
6     private static final long FASTEST_UPDATE_INTERVAL_IN_MILLISECONDS = 5000;
7     private static final int REQUEST_CHECK_SETTINGS = 100;
8     private FusedLocationProviderClient mFusedLocationClient;
9     private SettingsClient mSettingsClient;
10    private LocationRequest mLocationRequest;
11    private LocationSettingsRequest mLocationSettingsRequest;
12    private LocationCallback mLocationCallback;
13    public static Location mCurrentLocation;
14    private Boolean mRequestingLocationUpdates;
15    ListView companyListView;
16    ArrayList<String> companies = new ArrayList<String>();
17    ArrayAdapter arrayAdapter;
```

Vediamo adesso la parte riguardante il metodo `onCreate`. Metodo che ricordiamo essere il primo ad essere invocato appena l'activity viene lanciata. In tale metodo c'è una prima parte di inizializzazione che avviene tramite il richiamo della funzione `init` (funzione che verrà discussa in seguito). Dopodiché si fa uso di un `PermissionListener` il quale implementa dei comportamenti a seconda dell'esito ricevuto dalla richiesta dei permessi.

Infine, si richiama `setOnItemClickListener` sull'oggetto `ListView` popolato precedentemente con le sedi nelle vicinanze. Tale listener ha la funzione di catturare il nome della sede sulla quale si è eseguito il tap. Si viene dunque reindirizzati all'activity che mostra gli sportelli aperti per la sede di preferenza.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_user_location);
5     ButterKnife.bind(this);
6     init();
7     restoreValuesFromBundle(savedInstanceState);
```



```

8     Dexter.withActivity(this)
9         .withPermission(Manifest.permission.ACCESS_FINE_LOCATION)
10        .withListener(new PermissionListener() {
11            @Override
12            public void onPermissionGranted(PermissionGrantedResponse response) {
13                mRequestingLocationUpdates = true;
14                startLocationUpdates();
15            }
16            @Override
17            public void onPermissionDenied(PermissionDeniedResponse response) {
18                if (response.isPermanentlyDenied()) {
19                    openSettings();
20                }
21            }
22            @Override
23            public void onPermissionRationaleShouldBeShown(PermissionRequest permission,
24                PermissionToken token) {
25                token.continuePermissionRequest();
26            }
27        }).check();
28    setTitle("Nearby Companies");
29    companyListView = findViewById(R.id.companyListView);
30    arrayAdapter = new ArrayAdapter(this, android.R.layout.simple_list_item_1, companies);
31    companyListView.setAdapter(arrayAdapter);
32    companyListView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
33        @Override
34        public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
35            if (companies.size() > position && mCurrentLocation != null)
36            {
37                String companyName = companies.get(position);
38                int iEnd = companyName.indexOf(" -");
39                String substringCompanyName = null;
40                if (iEnd != -1)
41                {
42                    substringCompanyName = companyName.substring(0, iEnd);
43                }
44
45                Intent intent = new Intent(getApplicationContext(), CompanyServices.class);
46                intent.putExtra("company_tapped", substringCompanyName);
47                startActivity(intent);
48            }
49        });
50    }

```

La funzione che si occupa del popolamento della ListView è fillCompanyListView e adesso la vedremo in dettaglio. Una volta ottenuta la posizione dell'utente, si controlla che questa non sia null. A questo punto si istanzia un oggetto della classe ParseGeoPoint nel quale memorizziamo le coordinate del device dello studente fornendo latitudine e longitudine. Ora si esegue una interrogazione su Parse Server sfruttando le API per dati geospaziali. L'interrogazione effettuata risulta essere limitata a tutti i record presenti nella classe Company tali da essere distanti RADIUS_AREA_IN_KILOMETERS dall'oggetto myPosition che rappresenta la posizione dell'utente. Quindi si esegue una scansione di tutte le sedi universitarie rilevate in un preciso raggio con centro le coordinate salvate in myPosition. Le sedi che rientrano in tale area verranno aggiunte alla ListView mostrata sul frontend dell'activity. Se non vengono rilevate sedi, si visualizzerà un messaggio che determinerà la presenza di zero sedi.

```

1 private void fillCompanyListView()
2 {
3     if(mCurrentLocation != null)
4     {
5         final ParseGeoPoint myPosition = new ParseGeoPoint(mCurrentLocation.getLatitude(),
6 mCurrentLocation.getLongitude());
7         ParseQuery<ParseObject> query = ParseQuery.getQuery("Company");
8         query.whereWithinKilometers("COMPANY_POSITION", myPosition, RADIUS_AREA_IN_KILOMETERS);
9
10        query.setLimit(8);
11        query.findInBackground(new FindCallback<ParseObject>() {
12            @Override
13            public void done(List<ParseObject> objects, ParseException e) {
14                if(e == null)
15                {
16                    companies.clear();
17                    if(objects.size() > 0)
18                    {
19                        for(ParseObject object : objects)
20                        {
21                            String companyName = (String) object.get("COMPANY_NAME");
22                            Double distanceInKilometers = myPosition.distanceInKilometersTo((
23 ParseGeoPoint) object.get("COMPANY_POSITION"));
24                            Double distanceRounded = (double) Math.round(distanceInKilometers *
25 10) / 10;
26                            companies.add(companyName + " - " + distanceRounded.toString() + "
27 km");
28                        }
29                    }
30                }
31                else
32                {
33                    companies.add("No active companies nearby!");
34                }
35                arrayAdapter.notifyDataSetChanged();
36            }
37        });
38    }
39 }

```

3.5.7 CompanyServices Class

L'argomentazione per tale classe verrà suddivisa in due parti. Si partirà dalla descrizione del metodo onCreate che viene invocato appena l'activity va in esecuzione, dopodiché si andrà ad illustrare l'implementazione della funzione updateServicesList.

Come descritto in altre activity viste in precedenza, anche qui si fa uso di una ListView che viene opportunamente popolata tramite l'utilizzo di un ArrayList di stringhe aggiornate grazie ad un ArrayAdapter. Di fondamentale importanza è il listener attivato su ogni riga della ListView. Difatti, tramite l'invocazione del metodo setOnItemClickListener si memorizza il servizio presso il quale lo studente intende mettersi in coda e lo si inoltra all'activity che andrà a gestire tale coda.

In updateServicesList si provvede a popolare la ListView che mostra la li-

sta degli sportelli che possiede ogni sede. Si va inizialmente alla ricerca del record che abbia COMPANY_NAME uguale a companyName (nome della sede recuperato dall'activity di provenienza). E per tale sede se ne memorizzano i servizi associati nell'ArrayList, precedentemente abbinato alla ListView tramite l'ArrayAdapter.

3.5.8 Queue Class

La classe che va ad implementare l'effettiva gestione della coda è Queue. La struttura di tale classe verrà divisa in più parti per comprenderne meglio l'intero funzionamento. Nel metodo onCreate, dopo aver fatto il binding con gli oggetti grafici presenti sul frontend si prosegue con un controllo sulla variabile booleana isAlreadyInQueue inizialmente settata a false. Tale variabile viene opportunamente modificata a true se lo studente risulta essere già inserito in una coda. Se dunque lo studente non è già in coda, si recupera il servizio su cui è stato eseguito il tap nell'activity precedente e si richiamano due funzioni: checkIfQueueHasBeenReset e checkTurn. L'implementazione di tali funzioni verrà descritta a breve. Se invece l'utente risulta essere già inserito in una coda, tramite la classe SharedPreferences si recupereranno le informazioni che definiscono il suo turno.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_queue);
5
6     serviceNameTextView = findViewById(R.id.serviceNameTextView);
7     myNumberTextView = findViewById(R.id.myNumberTextView);
8     currentNumberTextView = findViewById(R.id.currentNumberTextView);
9
10    notificationManager = NotificationManagerCompat.from(this);
11
12    if (!isAlreadyInQueue) {
13        Intent intent = getIntent();
14        serviceName = intent.getStringExtra("service_tapped");
15        checkIfQueueHasBeenReset();
16        checkTurn();
17    }
18
19    serviceNameTextView.setText(serviceName);
20    myNumberTextView.setText(String.valueOf(myNumber));
21 }
```

Affinché l'utente conservi il proprio turno anche qualora dovesse chiudere definitivamente l'applicazione, è necessario predisporre l'activity ad una fase di ripristino automatico delle informazioni che definiscono i dettagli della posizione in coda. Per fare ciò è necessario richiamare la funzione checkTurn nel metodo onResume che verrà invocato non appena l'app torna nuovamente in foreground.

```
1 @Override
2 protected void onResume() {
```

```

3     super.onResume();
4
5     sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
6
7     checkTurn();
8
9     serviceNameTextView.setText(sharedPrefs.getString("serviceNameLabel", ""));
10    myNumberTextView.setText(String.valueOf(sharedPrefs.getInt("myTurn", 0)));
11 }

```

Per garantire una costante consistenza delle informazioni mostrate sul display del dispositivo, è necessario controllare se la coda per la quale intendiamo ricevere il numero è chiusa o meno. Difatti, nel caso in cui dovesse esser chiusa si provvederà al richiamare la procedura di logout dell'utente. Se invece la coda è aperta, si richiama la funzione di ottenimento del numero.

```

1 private void checkIfQueueHasBeenReset() {
2     sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
3     ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
4
5     query.whereEqualTo("SERVICE_NAME", serviceName);
6     query.findInBackground(new FindCallback<ParseObject>() {
7         public void done(List<ParseObject> services, ParseException e) {
8             if (e == null) {
9                 if (services.size() > 0) {
10                     for (ParseObject service : services) {
11                         myNumber = service.getInt("MY_NUMBER");
12                         currentNumber = service.getInt("CURRENT_NUMBER");
13                         Log.i("SERVICE NAME LABEL", serviceName);
14                         Log.i("CURRENT NUMBER", String.valueOf(currentNumber));
15                         Log.i("MY NUMBER", String.valueOf(myNumber));
16                         if (currentNumber == -1 && myNumber == 0) /* If the queue is closed */ {
17
18                             onLogoutClick(view);
19                         } else /* If the queue is open */
20                             gettingNumber();
21                     }
22                 }
23             }
24         }
25     });
26 }

```

Il numero viene rilasciato tramite un meccanismo molto semplice. Si esegue una interrogazione sul database sul servizio presso il quale si ha intenzione di mettersi in coda e si controlla il valore dell'attributo MY_NUMBER. Tale valore lo si incrementa di una unità e lo si assegna al client che lo sta richiedendo. Ovviamente, viene conseguentemente aggiornato anche lo stato del booleano isAlreadyInQueue a true per garantire il funzionamento del controllo eseguito nel metodo onCreate visto prima.

```

1 private void gettingNumber()
2 {
3     ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
4     query.whereEqualTo("SERVICE_NAME", serviceName);
5     query.findInBackground(new FindCallback<ParseObject>()

```

```

6      {
7          public void done(List<ParseObject> services, ParseException e)
8          {
9              if (e == null)
10             {
11                 if(services.size() > 0)
12                 {
13                     for (ParseObject service : services)
14                     {
15                         service.increment("MY_NUMBER");
16                         service.saveInBackground();
17
18                         myNumber = service.getInt("MY_NUMBER");
19
20                         sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
21                         editor = sharedPrefs.edit();
22                         editor.putInt("myTurn", myNumber);
23                         isAlreadyInQueue = true;
24                         editor.putBoolean("isAlreadyInQueue", isAlreadyInQueue);
25                         editor.putString("serviceNameLabel", serviceName);
26                         editor.apply();
27
28                         serviceNameTextView.setText(service.getString("SERVICE_NAME"));
29
30                         myNumberTextView.setText(String.valueOf(myNumber));
31
32                     }
33                 }
34             }
35         }
36     });
37 }

```

Una volta ottenuto il numero, si è pensato di non permettere all'utente di utilizzare il tasto back. Si è provveduto dunque a disabilitarlo tramite riscrittura del metodo `onBackPressed`. Dal momento che all'invocazione di tale metodo non deve accadere nulla, il corpo di quest'ultimo risulta vuoto. Se l'utente ha intenzione di terminare il suo turno in coda, è necessario eseguire un tap sul bottone Esci presente sul frontend di tale activity.

Qualora l'utente decidesse di abbandonare la coda, cliccando su Esci, verrà invocato il metodo `onLogOutClick` che molto banalmente procede con la disconnessione del Parse User autenticato, aggiorna lo stato di `isAlreadyInQueue` che adesso diventa false e si reindirizza alla MainActivity.

```

1 public void onLogOutClick(View view)
2 {
3     ParseUser.logOut();
4
5     sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
6     editor = sharedPrefs.edit();
7     isAlreadyInQueue = false;
8     editor.putBoolean("isAlreadyInQueue", isAlreadyInQueue);
9     editor.apply();
10
11     Intent intent = new Intent(getApplicationContext(), MainActivity.class);
12     startActivity(intent);
13     finish();

```

14 }

Una funzione molto significativa è `checkTurn`, la quale garantisce l'aggiornarsi in tempo reale del turno sul frontend dell'app. Tale funzione è stata realizzata tramite l'ausilio della classe `Handler` a cui è associato un `Runnable` richiamato ogni 1000 millisecondi. La procedura implementata si compone dapprima di una interrogazione al database per ottenere il numero corrente, dopodiché si confronta tale numero con quello assegnato all'utente. Se i due numeri risultano essere gli stessi, è il turno dello studente e viene prontamente notificato sul frontend. In caso contrario, si visualizza l'avanzare della numerazione, in attesa che scatti il proprio turno. Nel momento in cui è il turno dell'utente, si provvede a richiamare una notifica push sul dispositivo tramite la creazione di un `CHANNEL` per le notifiche. Tale aspetto verrà presentato nella prossima ed ultima classe che riguarda tale capitolo.

```
1 private void checkTurn()
2 {
3     sharedPreferences = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
4
5     final Handler handler = new Handler();
6     Runnable runnable = new Runnable() {
7         @Override
8         public void run()
9         {
10             ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
11             query.whereEqualTo("SERVICE_NAME", sharedPreferences.getString("serviceNameLabel", ""));
12             query.findInBackground(new FindCallback<ParseObject>()
13             {
14                 public void done(List<ParseObject> services, ParseException e)
15                 {
16                     if (e == null)
17                     {
18                         if(services.size() > 0)
19                         {
20                             for (ParseObject service : services)
21                             {
22                                 currentNumber = service.getInt("CURRENT_NUMBER");
23
24                                 if(currentNumber == sharedPreferences.getInt("myTurn", 0))
25                                 {
26                                     currentNumberTextView.setText("E' il tuo turno!");
27
28                                     /* Notification popup here */
29                                     Notification notification = new NotificationCompat.Builder(
30                                         getApplicationContext(), CHANNEL_1_ID)
31                                         .setSmallIcon(R.drawable.ic_done)
32                                         .setContentTitle(sharedPreferences.getString("
33                                             serviceNameLabel", ""))
34                                         .setContentText("E' il tuo turno!")
35                                         .setPriority(NotificationCompat.PRIORITY_HIGH)
36                                         .setCategory(NotificationCompat.CATEGORY_MESSAGE)
37                                         .setOngoing(false)
38                                         .build();
39
40                                     notificationManager.notify(1, notification);
41                                 }
42                                 else if(currentNumber == sharedPreferences.getInt("myTurn", 0) + 1)
43                                 {
```

```

43         handler.removeCallbacksAndMessages(null);
44         onLogoutClick(view);
45         finish();
46     }
47     else
48     {
49         currentNumberTextView.setText("Il numero corrente e' " +
String.valueOf(currentNumber));
50     }
51 }
52 }
53 }
54 }
55 });
56
57     handler.postDelayed(this, 1000);
58 }
59 };
60
61     handler.postDelayed(runnable, 1000);
62 }

```

3.5.9 Notifications Class

Dal momento che l'utente non potrà costantemente controllare se il suo turno è scattato o meno, è stato pensato di implementare un semplice sistema di notifica push. Tale sistema avviene tramite la creazione di due canali di notifiche. Il primo ha una priorità settata ad HIGH, il secondo invece a LOW. All'interno della nostra app verrà fatto uso solo del CHANNEL_1_ID. La parte riguardante le notifiche, sebbene possa sembrare semplice da realizzare, prevede numerose complicazioni. Molte delle quali dovute anche alla versione del firmware in uso nel dispositivo nel quale si intende notificarle. Nel nostro caso, nella funzione createNotificationChannels, andiamo a controllare la versione dell'SDK in uso. Se è uguale o superiore ad Oreo, creiamo il canale tramite la classe NotificationChannel. Infine richiamiamo createNotificationChannel su un oggetto Notification Manager.

Così si conclude la trattazione riguardante la parte di progettazione ed implementazione della nostra applicazione. In seguito si discuteranno i casi d'uso con scenari reali.

Capitolo 4

Scenari d'uso dell'app

All'interno del seguente capitolo si mostrerà il funzionamento della piattaforma tramite dei semplici mockup. Un mockup fondamentalmente è una prototipazione digitale che ci permette tramite opportune rappresentazioni, di comprendere meglio la logica di funzionamento della nostra interfaccia utente.

Come già è stato anticipato nei capitoli precedenti, il nostro sistema si compone di due applicativi: quello destinato agli studenti ed il gestore universale delle code.

4.1 Mockup client

Qui di seguito partiamo col mostrare il funzionamento dell'app offerta agli studenti.

La schermata con la quale si apre l'applicazione risulta essere quella mostrata in Figura 4.1. Come è evidente, è stata realizzata una interfaccia utente davvero molto semplice. Questo è dovuto dal fatto che non si vuole distogliere l'attenzione dell'utente facendolo perdere in menù e sottomenù vari.

In primo piano appare anche il logo dell'Università degli Studi di Napoli "Parthenope". La scelta è facilmente ricaduta su tale Ateneo, poichè Ateneo di appartenenza. Ciò non esula altri atenei dal poter adottare tale sistema per la gestione dei loro servizi.

Le operazioni che possono essere scelte una volta aperta l'app sono due: registrazione o accesso al servizio.



Figura 4.1: MainActivity

Qualora si scegliesse di procedere con la registrazione sul portale (operazione obbligata almeno la prima volta che si lancia l'app), l'utente verrà reindirizzato all'activity di signUp. Nella procedura di registrazione sono richiesti nome utente e password. Una volta compilati correttamente tali campi, si giunge ad una schermata nella quale viene confermata l'avvenuta registrazione (Figura 4.2).

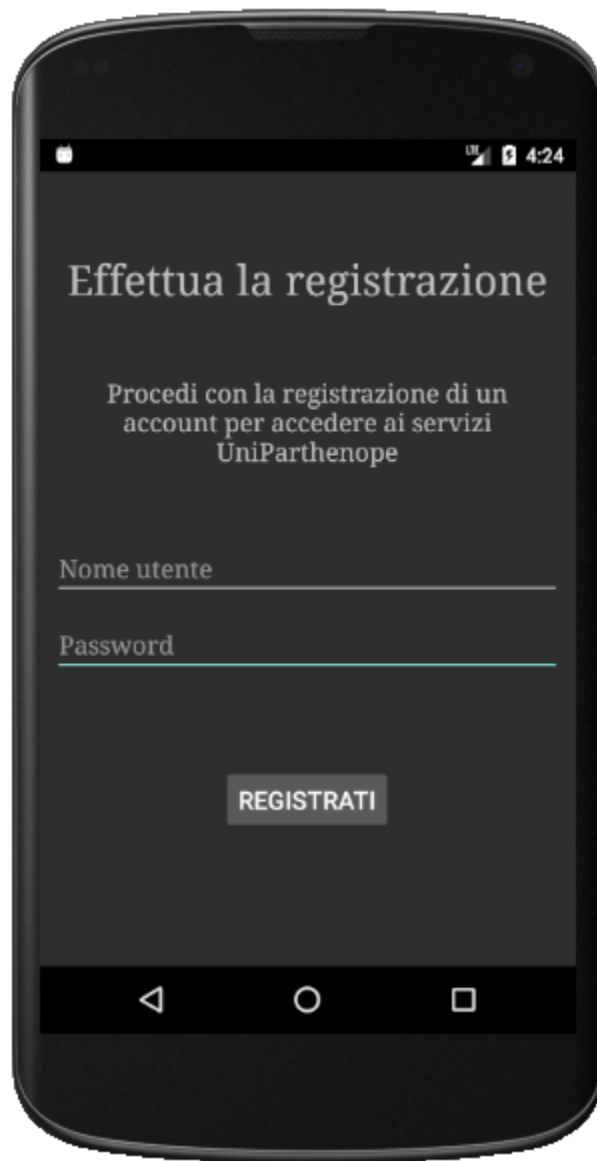


Figura 4.2: SignUp

Qualora si fosse scelto l'accesso all'app, è richiesto di compilare i due campi mostrati in figura con le credenziali scelte all'atto della registrazione dell'utente. Anche in questo caso, una volta compilati i campi e fatto tap sul bottone di accesso, si viene reindirizzati ad un'altra activity. Tale activity confermerà l'avvenuto accesso sul portale e permetterà all'utente di accedere alla procedura di localizzazione. Procedura che ricordiamo essere obbligatoria pena l'impossibilità di visualizzare le sedi presenti nelle vicinanze dello studente (Figura 4.3).

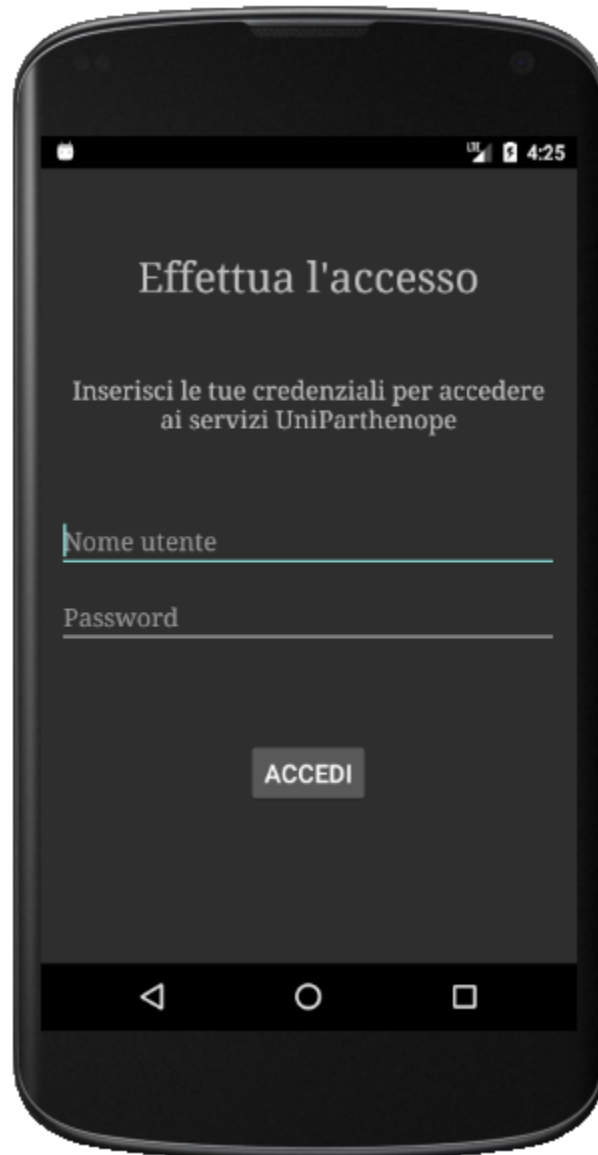


Figura 4.3: LogIn

In Figura 4.4 è mostrato il mockup una volta effettuato l'accesso sull'app. In tale schermata è possibile eseguire il logOut con conseguente disconnessione dalla piattaforma oppure tramite un tap sul bottone “Vai ai servizi” si viene reindirizzati all'activity adibita alla richiesta dei permessi per il GPS con rilevamento della posizione del device in uso.



Figura 4.4: LogIn Successful

Una volta forniti gli adeguati permessi al sistema per rilevare le coordinate del device, verrà mostrato un elenco di tutte le sedi universitarie che sono presenti entro un raggio (prefissato dal sistema) dal dispositivo in uso dallo studente. Come evidente in Figura 4.5, il nostro studente si è localizzato nei pressi della sede del Centro Direzionale dell'Università degli Studi di Napoli "Parthenope".

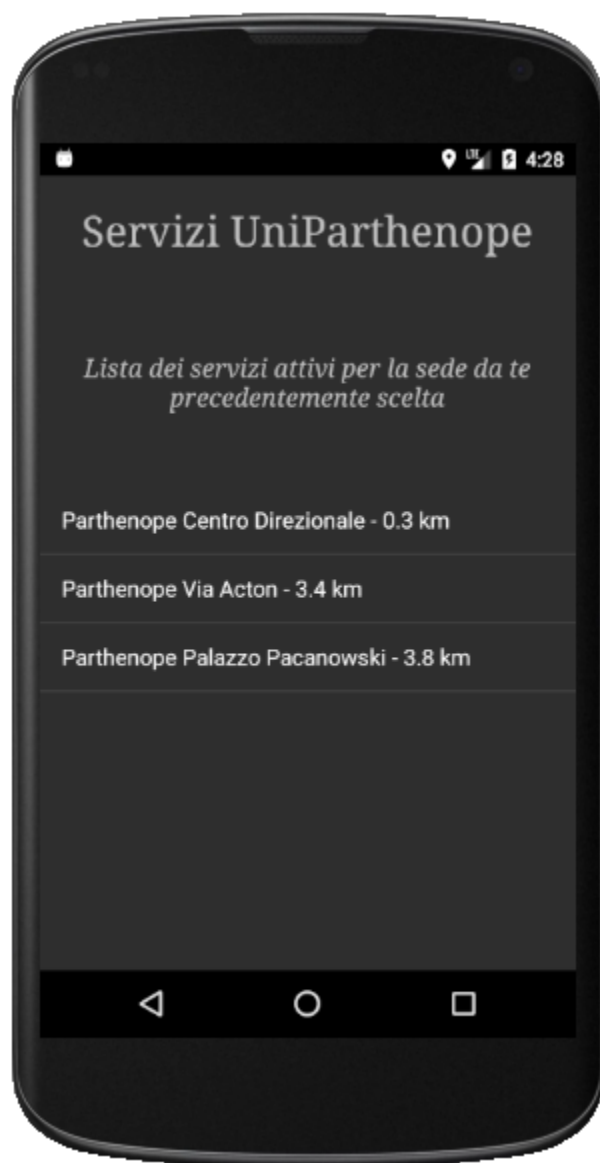


Figura 4.5: List of companies

Per tale sede risultano attivi due sportelli: la segreteria studenti e quella didattica. Ciò vuol dire che lo studente, una volta localizzatosi nei pressi della sede di suo interesse, ed in seguito alla visualizzazione di tali segreterie, può decidere di mettersi in coda pur non essendo fisicamente presente dinanzi allo sportello ma nelle sue vicinanze. Gli stessi sportelli (segreteria studenti e didattica) risultano essere attivi anche per le altre due sedi di Via Acton e di Palazzo Pacanowski. Nei mockup da noi mostrati, l'utente, nonostante si sia localizzato al Centro Direzionale, riesce a vedere anche le altre due sedi appena menzionate. Questo perché ai fini di testing del prototipo realizzato, si è pensato di utilizzare un raggio più alto. Un valore ideale sarebbe quello di 1-2 km (Figura 4.6).



Figura 4.6: List of services

Una volta scelto uno dei due sportelli, il sistema genera automaticamente un numero e lo assegna al client che lo sta richiedendo. Nella Figura 4.7, il nostro client è in coda per la Segreteria Studenti presso la sede del Centro Direzionale. In tale activity è presente un bottone Esci per lasciare la coda, il numero corrente che si sta servendo ed il numero assegnato dal sistema allo studente.



Figura 4.7: Getting number

Non appena scattato, il turno dell'utente viene notificato tramite una notifica push in alto a sinistra e allo stesso tempo viene segnalato sul front end dell'activity mostrata in figura tramite il messaggio: "È il tuo turno!". Una volta che la numerazione della coda prosegue, il nostro client viene disconnesso dal momento che risulterà essere già servito (Figura 4.8).



Figura 4.8: Your turn

Di seguito è riportata la notifica che si riceve una volta scattato il turno dell'utente (Figura 4.9).



Figura 4.9: Notification

4.2 Mockup gestore code

Concludiamo la trattazione del seguente capitolo illustrando il funzionamento dell'applicativo messo a disposizione dei responsabili dei front-office. Come già anticipato nei capitoli precedenti, si è provveduto a realizzare un "telecomando universale" che permette l'avanzamento di tutte le code registrate sul backend. Praticamente invece, ogni sportello dovrebbe disporre di un applicativo indipendente dagli altri. Anche qui si parte col descrivere la schermata iniziale che risulta essere davvero molto semplice. La MainActivity si presenta tramite una lista di sedi che è possibile gestire. Nell'esempio che illustreremo, verrà mostrato il funzionamento della coda Segreteria Didattica presso la sede del Centro Direzionale.

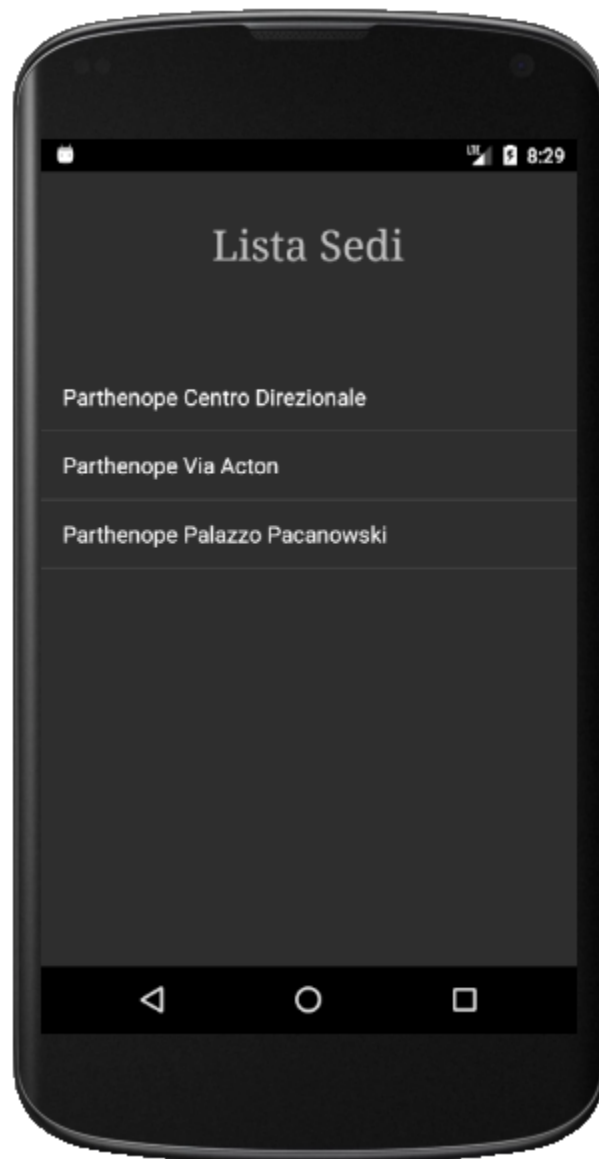


Figura 4.10: MainActivity

Una volta eseguito un tap su tale sede, vengono mostrate le code sulle quali lo sportellista può agire (Figura 4.11). Nel nostro caso, si intende simulare il lavoro svolto da un responsabile presso lo sportello della Segreteria Didattica.

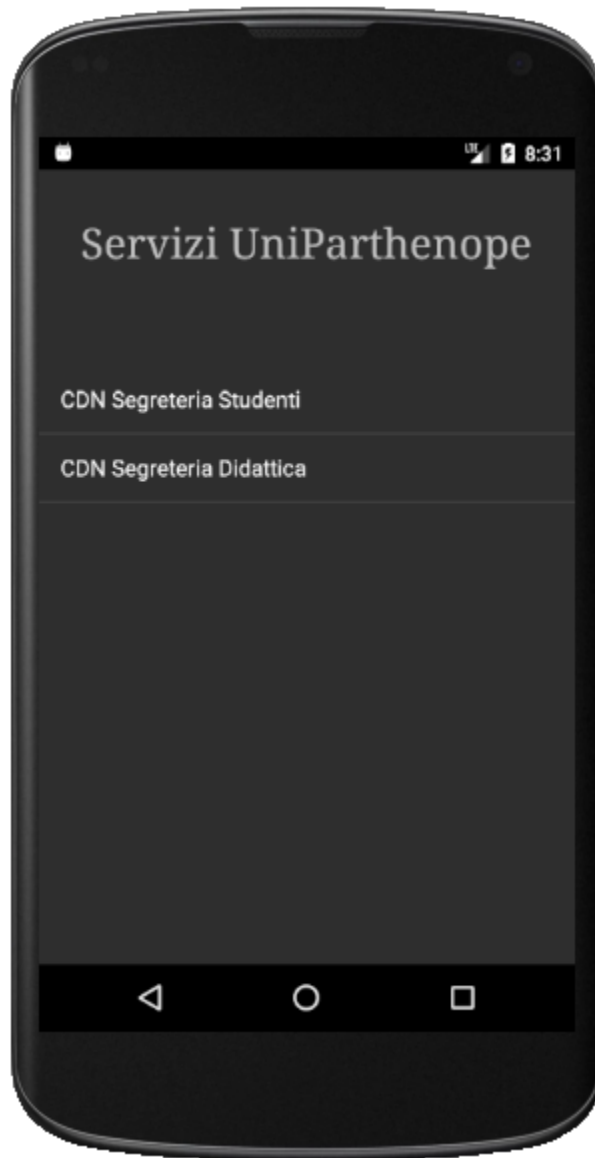


Figura 4.11: List of companies

Adesso andiamo a mostrare due scenari che possono presentarsi. Il primo risulta essere quando la coda è "aperta". In questa circostanza, il numero corrente per lo sportellista sarà "0" ed i client che proveranno a richiedere il numero, ne otterranno uno e verranno successivamente serviti dal sistema una volta scattato il loro turno. Tale scenario è quello che appare nella Figura 4.12.



Figura 4.12: Queue Open

L'altro scenario si presenta quando lo sportello è chiuso (Figura 4.13). In tale circostanza il numero corrente per l'addetto al front-office risulta essere "-1" e qualora dei client richiedessero un numero al sistema per tale coda, verrebbero disconnessi automaticamente. Per garantire l'erogazione dei numeri ai client è necessario avanzare la numerazione tramite un tap su "Prossimo turno". Così facendo il numero corrente passa a "0" e tutto funzionerà come visto alla pagina precedente.

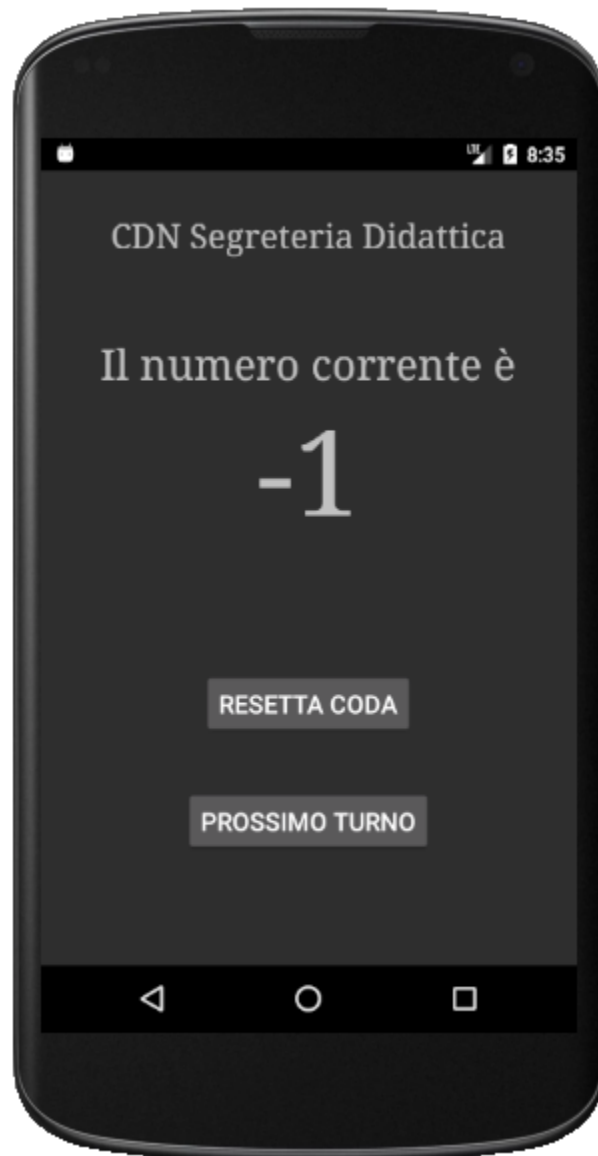


Figura 4.13: Queue Closed

Capitolo 5

Conclusioni

Nel seguente capitolo si conclude la dissertazione di tale tesi. Si andrà a fare un riepilogo del lavoro svolto, si discuteranno i vantaggi e gli svantaggi derivanti dal servizio proposto ed infine si presenteranno delle idee per sviluppi futuri.

Partiamo col dire che sebbene il prototipo proposto sia completamente funzionante, è necessaria l'adozione di adeguato hardware da parte dell'Ateneo che vuole utilizzarlo. La soluzione da noi illustrata ha fatto uso, per semplicità, di due device Android. Uno che simulasse il client utilizzato dallo studente ed un altro preposto all'avanzamento ed alla gestione delle code.

Nulla vieta la possibilità di migrare il gestore delle code su una web app in modo tale da evitare agli addetti ai front-office di dover regolare l'andamento della coda necessariamente tramite uno smartphone o un tablet. In tale evenienza, bisognerebbe riscrivere il servizio di gestione code in un linguaggio differente affinché possa essere controllato dal personale tramite una dashboard più versatile. Gli addetti agli sportelli, così facendo, avrebbero un accesso direttamente da browser.

I vantaggi principali derivanti dall'adozione di una digitalizzazione dei servizi di Ateneo si possono tradurre sicuramente in una gestione più ordinata ed efficiente dei servizi offerti agli studenti. Allo stesso tempo, dietro un'adeguata progettazione, si potrebbero raccogliere numerose informazioni a fini statistici e mirati al miglioramento dei servizi. Ad esempio, monitorare gli orari di maggiore affluenza degli studenti per garantire più personale in tali fasce orarie e riallocando il personale inutilizzato in fasce meno affollate. Per gli studenti invece, appare chiara l'opportunità di potersi riservare un turno in coda presso lo sportello di interesse nonostante si stia continuando a studiare in biblioteca o aula studio, evitando così di attendere per il proprio turno o per l'apertura di tali sportelli. Allo stesso tempo si contribuisce alla riduzione della congestione di spazi chiusi quali possono essere corridoi o ambienti adibiti all'attesa.

Tra gli svantaggi annoveriamo l'obbligo dell'uso del GPS affinché l'utente

venga localizzato correttamente. Tale scelta è dipesa dal fatto che il servizio realizzato deve rispondere con una estrema precisione sulla posizione dell'utente per capire se risultano presenti sedi universitarie in uno specifico range dalle sue coordinate. Sfortunatamente, ci si è resi conto della poca accuratezza rilevata tramite celle Wi-Fi o appartenenti al gestore telefonico del device, arrivando ad apprezzare errori anche nell'ordine di qualche km. Errori non accettabili per un servizio come quello proposto che sebbene in fase di testing abbia fatto uso di range più elevati, è pensato per mantenersi in un range complessivo di massimo 2km. Appare chiaro che impiegando costantemente il GPS, si ha un side-effect sulla durata della batteria. Per ovviare a tale problema non si è fatto uso delle semplici funzioni offerte da Android per rilevare la posizione del dispositivo ma, come visto nel capitolo 3, si è ricorso alle Fused Location API che evitano un polling costante, ed al contrario, spalmano tali richieste su un intervallo compreso tra 5 e 10 secondi.

Come già anticipato nelle specifiche dei requisiti, la nostra app richiede che ci si localizzi nelle vicinanze delle sedi a cui si è interessati affinché si possa ottenere accesso alle code. Tale vincolo è posto soltanto in ingresso nell'area e non in uscita. Tale accorgimento è stato realizzato tramite l'ausilio di query geospaziali tra la posizione dell'utente e le coordinate delle sedi (opportunamente registrate sul backend) tramite connessione Wi-Fi o connettività offerta dal gestore telefonico.

Un'alternativa a tale tipologia di comunicazione è l'impiego dei "beacon" [4]. I beacon risultano essere dei piccoli trasmettitori che possono comunicare con dispositivi dotati di Bluetooth. Tale approccio si è evitato dal momento che si sarebbe dovuto adottare dell'hardware apposito e, considerando che ogni beacon ha un raggio di azione di qualche decina di metri, l'Ateneo interessato avrebbe dovuto far fronte ad un acquisto di svariati beacon. Tale soluzione può essere più facilmente presa in considerazione per ambienti di piccole dimensioni nei quali si vogliono simulare comportamenti particolari come ad esempio suggeriscono alcune applicazioni di domotica.

Per quanto riguarda la gestione della coda in sé, vale la pena fare un paio di riflessioni. Nel nostro modello, si provvede ad un semplice incremento di contatori con relativo sistema di notifiche dei turni. La questione, se inquadrata da un punto di vista più matematico e probabilistico, risulta essere decisamente più articolata.

Un aspetto fondamentale da considerare è il processo di arrivo: come arrivano gli studenti (se arrivano singolarmente o in gruppo) e come vengono distribuiti gli arrivi nel tempo. Ovvero, qual è la distribuzione di probabilità nel tempo tra gli arrivi successivi, se esiste una popolazione limitata di studenti o (effettivamente) un numero infinito.

Il processo di arrivo più semplice è quello in cui abbiamo arrivi completamente

regolari (cioè lo stesso intervallo di tempo costante tra gli arrivi successivi).

Una buona approssimazione di tale fenomeno è descritta tramite il modello di Poisson nel quale si ipotizza che gli arrivi avvengano in maniera casuale. In un flusso di Poisson gli studenti successivi arrivano dopo intervalli che sono distribuiti in maniera esponenziale in modo indipendente. Il flusso di Poisson è importante in quanto è un modello matematico conveniente per l'analisi di molti sistemi di code nel mondo reale ed è descritto da un singolo parametro: il tasso di arrivo medio [5].

Numerosi sono i possibili sviluppi futuri per tale prototipo. Il primo tra tutti risulta essere l'idea di volerlo estendere a circuito. Un circuito che sfruttando la geolocalizzazione dell'utente, permetta la scansione di attività commerciali, ospedali, banche, ristoranti etc. nei paraggi. Ovviamente affinché tale circuito possa esistere, dovrebbero essere stipulati accordi e convenzioni tra enti e privati. Tra i benefici derivanti da un meccanismo del genere, menzioniamo i costi e sprechi evitati per l'erogazione di ticket cartacei destinati ai clienti; si ridurrebbero i tempi morti; e ci sarebbe inoltre un conseguente aumento della produttività potendo destinare il tempo che si sarebbe speso in coda, facendo altro. Un altro accorgimento che migliorerebbe ampiamente l'esperienza utente finale, deriva dall'integrazione di tecniche di machine learning ed analisi dei dati per i quali si dovrebbe predisporre una precisa progettazione in modo tale da convertire i dati prodotti dagli utenti in informazioni utili per il miglioramento dei servizi offerti.

Appendice

Appendice A

Gestore code

```
1 package com.triassi.queuemanager;
2
3 import android.app.Application;
4
5 import com.parse.Parse;
6
7 public class App extends Application
8 {
9     @Override
10    public void onCreate()
11    {
12        super.onCreate();
13
14        Parse.enableLocalDatastore(this);
15
16        Parse.initialize(new Parse.Configuration.Builder(this)
17            .applicationId("applicationIdHere")
18            .clientId("clientIdHere")
19            .server("http://ipServerHere:portServerHere/parse/")
20            .build()
21        );
22    }
23 }
```

```
1 package com.triassi.queuemanager;
2
3 import android.content.Intent;
4 import android.support.v7.app.AppCompatActivity;
5 import android.os.Bundle;
6 import android.util.Log;
7 import android.view.View;
8 import android.widget.AdapterView;
9 import android.widget.AdapterView.OnItemClickListener;
10 import android.widget.ListView;
11 import android.widget.TextView;
12
13 import com.parse.FindCallback;
14 import com.parse.ParseException;
15 import com.parse.ParseObject;
16 import com.parse.ParseQuery;
17
18 import java.util.ArrayList;
19 import java.util.List;
20
21 public class MainActivity extends AppCompatActivity {
22 }
```

```

23     private TextView companiesListTextView;
24     private ListView companiesListView;
25     private ArrayList<String> companiesList = new ArrayList<String>();
26     private ArrayAdapter arrayAdapter;
27
28     @Override
29     protected void onCreate(Bundle savedInstanceState) {
30         super.onCreate(savedInstanceState);
31         setContentView(R.layout.activity_main);
32
33         companiesListTextView = findViewById(R.id.companiesListTextView);
34         companiesListView = findViewById(R.id.companiesListView);
35         arrayAdapter = new ArrayAdapter<this, android.R.layout.simple_list_item_1,
companiesList>;
36         companiesListView.setAdapter(arrayAdapter);
37
38         updateCompaniesList();
39
40         companiesListView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
41             @Override
42             public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
43                 if (companiesList.size() > position)
44                 {
45                     String companyName = companiesList.get(position);
46
47                     Intent intent = new Intent(getApplicationContext(), Services.class);
48                     intent.putExtra("company_tapped", companyName);
49                     startActivity(intent);
50                 }
51             }
52         });
53     }
54
55     private void updateCompaniesList()
56     {
57         ParseQuery<ParseObject> query = ParseQuery.getQuery("Company");
58         query.findInBackground(new FindCallback<ParseObject>() {
59             public void done(List<ParseObject> companies, ParseException e) {
60                 if (e == null)
61                 {
62                     if (companies.size() > 0)
63                     {
64                         for (ParseObject company : companies)
65                         {
66                             Log.i("COMPANY_NAME", company.getString("COMPANY_NAME"));
67                             companiesList.add(company.getString("COMPANY_NAME"));
68                         }
69                     }
70                     arrayAdapter.notifyDataSetChanged();
71                 }
72             }
73         });
74     }
75 }

```

```

1 package com.triassi.queuemanager;
2
3 import android.content.Intent;
4 import android.os.Handler;
5 import android.support.v7.app.AppCompatActivity;
6 import android.os.Bundle;
7 import android.view.View;
8 import android.widget.Button;
9 import android.widget.TextView;
10

```

```

11 import com.parse.FindCallback;
12 import com.parse.ParseException;
13 import com.parse.ParseObject;
14 import com.parse.ParseQuery;
15
16 import java.util.List;
17
18 public class Queue extends AppCompatActivity {
19
20     private String serviceName;
21     private TextView serviceNameTextView;
22     private TextView currentNumberTextView;
23     private Button resetQueueButton;
24     private Button nextTurnButton;
25     private int currentNumber;
26
27     @Override
28     protected void onCreate(Bundle savedInstanceState) {
29         super.onCreate(savedInstanceState);
30         setContentView(R.layout.activity_queue);
31
32         serviceNameTextView = findViewById(R.id.serviceNameTextView);
33         currentNumberTextView = findViewById(R.id.currentNumberTextView);
34         resetQueueButton = findViewById(R.id.resetQueueButton);
35         nextTurnButton = findViewById(R.id.nextTurnButton);
36
37         Intent intent = getIntent();
38         serviceName = intent.getStringExtra("service_tapped");
39         serviceNameTextView.setText(serviceName);
40
41         updateQueue();
42     }
43
44     private void updateQueue()
45     {
46         final Handler handler = new Handler();
47         Runnable runnable = new Runnable() {
48             @Override
49             public void run()
50             {
51                 ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
52                 query.whereEqualTo("SERVICE_NAME", serviceName);
53                 query.findInBackground(new FindCallback<ParseObject>()
54                 {
55                     public void done(List<ParseObject> services, ParseException e)
56                     {
57                         if (e == null)
58                         {
59                             if(services.size() > 0)
60                             {
61                                 for (ParseObject service : services)
62                                 {
63                                     currentNumber = service.getInt("CURRENT_NUMBER");
64                                     currentNumberTextView.setText(String.valueOf(currentNumber))
65                                 }
66                             }
67                         }
68                     }
69                 });
70
71                 handler.postDelayed(this, 1000);
72             }
73         };
74     }
75

```

```

76     handler.postDelayed(runnable, 1000);
77 }
78
79 public void onResetQueueClick(View view)
80 {
81     ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
82     query.whereEqualTo("SERVICE_NAME", serviceName);
83     query.findInBackground(new FindCallback<ParseObject>()
84     {
85         public void done(List<ParseObject> services, ParseException e)
86         {
87             if (e == null)
88             {
89                 if(services.size() > 0)
90                 {
91                     for (ParseObject service : services)
92                     {
93                         service.put("CURRENT_NUMBER", -1);
94                         service.put("MY_NUMBER", 0);
95                         service.saveInBackground();
96
97                         currentNumberTextView.setText(String.valueOf(service.getInt("
CURRENT_NUMBER"))));
98                     }
99                 }
100             }
101         }
102     });
103 }
104
105 public void onNextTurnClick(View view)
106 {
107     ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
108     query.whereEqualTo("SERVICE_NAME", serviceName);
109     query.findInBackground(new FindCallback<ParseObject>()
110     {
111         public void done(List<ParseObject> services, ParseException e)
112         {
113             if (e == null)
114             {
115                 if(services.size() > 0)
116                 {
117                     for (ParseObject service : services)
118                     {
119                         service.increment("CURRENT_NUMBER");
120                         service.saveInBackground();
121                         currentNumberTextView.setText(String.valueOf(service.getInt("
CURRENT_NUMBER"))));
122                     }
123                 }
124             }
125         }
126     });
127 }
128 }

```

```

1 package com.triassi.queuemanager;
2
3 import android.content.Intent;
4 import android.support.v7.app.AppCompatActivity;
5 import android.os.Bundle;
6 import android.util.Log;
7 import android.view.View;
8 import android.widget.AdapterView;
9 import android.widget.ArrayAdapter;

```

```

10 import android.widget.ListView;
11 import android.widget.TextView;
12
13 import com.parse.FindCallback;
14 import com.parse.ParseException;
15 import com.parse.ParseObject;
16 import com.parse.ParseQuery;
17
18 import java.util.ArrayList;
19 import java.util.List;
20
21 public class Services extends AppCompatActivity {
22
23     private String companyName;
24     private TextView servicesListTextView;
25     private ListView servicesListView;
26     private ArrayList<String> servicesList = new ArrayList<String>();
27     private ArrayAdapter arrayAdapter;
28
29     @Override
30     protected void onCreate(Bundle savedInstanceState) {
31         super.onCreate(savedInstanceState);
32         setContentView(R.layout.activity_services);
33
34         servicesListTextView = findViewById(R.id.servicesListTextView);
35         servicesListView = findViewById(R.id.servicesListView);
36         arrayAdapter = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1,
37         servicesList);
38         servicesListView.setAdapter(arrayAdapter);
39
40         Intent intent = getIntent();
41         companyName = intent.getStringExtra("company_tapped");
42
43         updateServicesList();
44
45         servicesListView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
46             @Override
47             public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
48                 if (servicesList.size() > position)
49                 {
50                     String serviceName = servicesList.get(position);
51
52                     Intent intent = new Intent(getApplicationContext(), Queue.class);
53                     intent.putExtra("service_tapped", serviceName);
54                     startActivity(intent);
55                 }
56             });
57     }
58
59     private void updateServicesList()
60     {
61         ParseQuery<ParseObject> query = ParseQuery.getQuery("Company");
62         query.whereEqualTo("COMPANY_NAME", companyName);
63         query.findInBackground(new FindCallback<ParseObject>() {
64             @Override
65             public void done(List<ParseObject> objects, ParseException e) {
66                 if (e == null)
67                 {
68                     if (objects.size() > 0)
69                     {
70                         for (final ParseObject object : objects)
71                         {
72                             ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
73                             query.whereEqualTo("COMPANY", object);
74                             query.findInBackground(new FindCallback<ParseObject>() {

```

```

75         @Override
76         public void done(List<ParseObject> services, ParseException e) {
77
78             if(e == null)
79             {
80                 if(services.size() > 0)
81                 {
82                     for (ParseObject service : services)
83                     {
84                         Log.i("SERVICE INFO", service.getString("
SERVICE_NAME"));
85
86                         String serviceName = service.getString("
SERVICE_NAME");
87
88                         servicesList.add(serviceName);
89                     }
90                 }
91                 arrayAdapter.notifyDataSetChanged();
92             }
93         }
94     }
95 }
96
97
98 }

```

Appendice B

Client

```
1 package com.triassi.testparsenote;
2
3 import android.app.Application;
4
5 import com.parse.Parse;
6
7 public class App extends Application {
8     public void onCreate() {
9         super.onCreate();
10
11         Parse.enableLocalDatastore(this);
12
13         Parse.initialize(new Parse.Configuration.Builder(this)
14             .applicationId("applicationIdHere")
15             .clientId("clientIdHere")
16             .server("http://ipServerHere:portServerHere/parse/")
17             .build()
18         );
19     }
20 }

1 package com.triassi.testparsenote;
2
3 import android.content.Context;
4 import android.content.Intent;
5 import android.content.SharedPreferences;
6 import android.support.v7.app.AppCompatActivity;
7 import android.os.Bundle;
8 import android.view.View;
9 import android.widget.TextView;
10
11 import com.parse.ParseUser;
12
13 public class MainActivity extends AppCompatActivity {
14
15     TextView signUpTextView;
16     TextView loginTextView;
17
18     @Override
19     protected void onCreate(Bundle savedInstanceState) {
20         super.onCreate(savedInstanceState);
21         setContentView(R.layout.activity_main);
22
23         signUpTextView = (TextView) findViewById(R.id.signUpTextView);
24         loginTextView = (TextView) findViewById(R.id.loginTextView);
25     }
26 }
```



```

26     ParseUser currentUser = ParseUser.getCurrentUser();
27
28     SharedPreferences sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
29     boolean isAlreadyInQueue = sharedPrefs.getBoolean("isAlreadyInQueue", false);
30
31
32     if(currentUser != null && !isAlreadyInQueue)
33     {
34         Intent intent = new Intent(getApplicationContext(), AfterLogIn.class);
35         intent.putExtra("username", currentUser.getUsername());
36         startActivity(intent);
37     }
38     else if(currentUser != null && isAlreadyInQueue)
39     {
40         Intent intent = new Intent(getApplicationContext(), Queue.class);
41         startActivity(intent);
42     }
43 }
44
45 public void onSignUpClick(View view)
46 {
47     Intent intent = new Intent(this, SignUp.class);
48     startActivity(intent);
49 }
50
51 public void onLogInClick(View view)
52 {
53     Intent intent = new Intent(this, LogIn.class);
54     startActivity(intent);
55 }
56 }

```

```

1 package com.triassi.testparsenote;
2
3 import android.content.Intent;
4 import android.support.v7.app.AppCompatActivity;
5 import android.os.Bundle;
6 import android.util.Log;
7 import android.view.View;
8 import android.widget.EditText;
9 import android.widget.Toast;
10
11 import com.parse.LogInCallback;
12 import com.parse.ParseException;
13 import com.parse.ParseUser;
14
15 public class LogIn extends AppCompatActivity {
16
17     EditText usernameEditText;
18     EditText passwordEditText;
19     private String username;
20     private String password;
21
22     @Override
23     protected void onCreate(Bundle savedInstanceState) {
24         super.onCreate(savedInstanceState);
25         setContentView(R.layout.activity_log_in);
26
27         usernameEditText = findViewById(R.id.usernameEditText);
28         passwordEditText = findViewById(R.id.passwordEditText);
29     }
30
31     public void onLogInClick(View view)
32     {
33         username = usernameEditText.getText().toString();

```

```

34     password = passwordEditText.getText().toString();
35
36     Log.i("Username content", username);
37
38     ParseUser.logInInBackground(username, password, new LogInCallback() {
39         @Override
40         public void done(ParseUser user, ParseException e) {
41             if(user != null)
42             {
43                 Log.i("LogIn Info", "The user is logged in.");
44                 Toast.makeText(getApplicationContext(), "Accesso effettuato con successo!",
45                     Toast.LENGTH_SHORT).show();
46                 Intent intent = new Intent(getApplicationContext(), AfterLogIn.class);
47                 intent.putExtra("username", username);
48                 startActivity(intent);
49                 finish();
50             }
51             else
52             {
53                 Toast.makeText(getApplicationContext(), "I dati inseriti non sono corretti!",
54                     Toast.LENGTH_SHORT).show();
55                 Log.i("LogIn Info", "The user is NOT logged in.");
56             }
57         }
58     });
59 }

```

```

1  package com.triassi.testparsenote;
2
3  import android.content.Intent;
4  import android.support.v7.app.AppCompatActivity;
5  import android.os.Bundle;
6  import android.view.View;
7  import android.widget.Button;
8  import android.widget.TextView;
9
10 import com.parse.ParseUser;
11
12 public class AfterLogIn extends AppCompatActivity {
13
14     TextView loggedUserCredentialsTextView;
15     Button goToTheAppButton;
16     Button logOutButton;
17
18     @Override
19     protected void onCreate(Bundle savedInstanceState) {
20         super.onCreate(savedInstanceState);
21         setContentView(R.layout.activity_after_log_in);
22
23         loggedUserCredentialsTextView = findViewById(R.id.loggedUserCredentialsTextView);
24         goToTheAppButton = findViewById(R.id.goToTheAppButton);
25         logOutButton = findViewById(R.id.logOutButton);
26
27         Intent intent = getIntent();
28         String username = intent.getStringExtra("username");
29
30         loggedUserCredentialsTextView.setText(username);
31     }
32
33     public void onGoToTheAppClick(View view)
34     {
35         Intent intent = new Intent(getApplicationContext(), UserLocation.class);
36         startActivity(intent);
37         finish();
38     }
39 }

```

```

38     }
39
40     public void onLogOutClick(View view)
41     {
42         ParseUser.logOut();
43         Intent intent = new Intent(getApplicationContext(), MainActivity.class);
44         startActivity(intent);
45         finish();
46     }
47 }

1 package com.triassi.testparsenote;
2
3 import android.content.Intent;
4 import android.support.v7.app.AppCompatActivity;
5 import android.os.Bundle;
6 import android.util.Log;
7 import android.view.View;
8 import android.widget.EditText;
9 import android.widget.Toast;
10
11 import com.parse.ParseException;
12 import com.parse.ParseUser;
13 import com.parse.SignUpCallback;
14
15 public class SignUp extends AppCompatActivity {
16
17     EditText usernameEditText;
18     EditText passwordEditText;
19     private String username;
20     private String password;
21
22     @Override
23     protected void onCreate(Bundle savedInstanceState) {
24         super.onCreate(savedInstanceState);
25         setContentView(R.layout.activity_sign_up);
26
27         usernameEditText = findViewById(R.id.usernameEditText);
28         passwordEditText = findViewById(R.id.passwordEditText);
29     }
30
31     public void onSignUpClick(View view)
32     {
33         username = usernameEditText.getText().toString();
34         password = passwordEditText.getText().toString();
35
36         Log.i("Username content", username);
37         Log.i("Password content", password);
38
39         ParseUser user = new ParseUser();
40         user.setUsername(username);
41         user.setPassword(password);
42
43         user.signUpInBackground(new SignUpCallback() {
44             @Override
45             public void done(ParseException e) {
46                 if(e == null)
47                 {
48                     Log.i("Success Info", "Let's use the app now");
49                     Toast.makeText(getApplicationContext(), "Registrazione effettuata con
50 successo!", Toast.LENGTH_SHORT).show();
51                     Intent intent = new Intent(getApplicationContext(), AfterSignUp.class);
52                     intent.putExtra("username", username);
53                     intent.putExtra("password", password);
54                     startActivity(intent);

```

```

54         }
55         else
56         {
57             Toast.makeText(getApplicationContext(), "Registrazione fallita!", Toast.
LENGTH_SHORT).show();
58             Log.i("Fail Info", "SignUp failed!");
59         }
60     }
61 });
62 }
63 }

```

```

1 package com.triassi.testparsenote;
2
3 import android.content.Intent;
4 import android.support.v7.app.AppCompatActivity;
5 import android.os.Bundle;
6 import android.view.View;
7 import android.widget.TextView;
8
9 public class AfterSignUp extends AppCompatActivity {
10
11     TextView newUserTextView;
12     private String username;
13
14     @Override
15     protected void onCreate(Bundle savedInstanceState) {
16         super.onCreate(savedInstanceState);
17         setContentView(R.layout.activity_after_sign_up);
18
19         newUserTextView = findViewById(R.id.newUserTextView);
20
21         Intent intent = getIntent();
22
23         username = intent.getStringExtra("username");
24
25         newUserTextView.setText(username);
26     }
27
28     public void onTornaAllaHomeButtonClick(View view)
29     {
30         Intent intent = new Intent(getApplicationContext(), MainActivity.class);
31         startActivity(intent);
32     }
33 }

```

```

1 package com.triassi.testparsenote;
2
3 import android.Manifest;
4 import android.annotation.SuppressLint;
5 import android.app.Activity;
6 import android.content.Intent;
7 import android.content.IntentSender;
8 import android.content.pm.PackageManager;
9 import android.location.Location;
10 import android.net.Uri;
11 import android.os.Bundle;
12 import android.os.Looper;
13 import android.provider.Settings;
14 import android.support.annotation.NonNull;
15 import android.support.v4.app.ActivityCompat;
16 import android.support.v7.app.AppCompatActivity;
17 import android.util.Log;

```

```

18 import android.view.View;
19 import android.widget.AdapterView;
20 import android.widget.AdapterView;
21 import android.widget.ListView;
22 import android.widget.Toast;
23
24 import com.google.android.gms.common.api.ApiException;
25 import com.google.android.gms.common.api.ResolvableApiException;
26 import com.google.android.gms.location.FusedLocationProviderClient;
27 import com.google.android.gms.location.LocationCallback;
28 import com.google.android.gms.location.LocationRequest;
29 import com.google.android.gms.location.LocationResult;
30 import com.google.android.gms.location.LocationServices;
31 import com.google.android.gms.location.LocationSettingsRequest;
32 import com.google.android.gms.location.LocationSettingsResponse;
33 import com.google.android.gms.location.LocationSettingsStatusCodes;
34 import com.google.android.gms.location.SettingsClient;
35 import com.google.android.gms.tasks.OnFailureListener;
36 import com.google.android.gms.tasks.OnSuccessListener;
37 import com.karumi.dexter.Dexter;
38 import com.karumi.dexter.PermissionToken;
39 import com.karumi.dexter.listener.PermissionDeniedResponse;
40 import com.karumi.dexter.listener.PermissionGrantedResponse;
41 import com.karumi.dexter.listener.PermissionRequest;
42 import com.karumi.dexter.listener.single.PermissionListener;
43 import com.parse.FindCallback;
44 import com.parse.ParseException;
45 import com.parse.ParseGeoPoint;
46 import com.parse.ParseObject;
47 import com.parse.ParseQuery;
48
49 import java.text.DateFormat;
50 import java.util.ArrayList;
51 import java.util.Date;
52 import java.util.List;
53
54 import butterknife.ButterKnife;
55
56 public class UserLocation extends AppCompatActivity {
57
58     private static final String TAG = UserLocation.class.getSimpleName();
59
60     public static final int RADIUS_AREA_IN_KILOMETERS = 40;
61
62     // location last updated time
63     private String mLastUpdateTime;
64
65     // location updates interval - 10sec
66     private static final long UPDATE_INTERVAL_IN_MILLISECONDS = 10000;
67
68     // fastest updates interval - 5 sec
69     // location updates will be received if another app is requesting the locations
70     // than your app can handle
71     private static final long FASTEST_UPDATE_INTERVAL_IN_MILLISECONDS = 5000;
72
73     private static final int REQUEST_CHECK_SETTINGS = 100;
74
75     // bunch of location related apis
76     private FusedLocationProviderClient mFusedLocationClient;
77     private SettingsClient mSettingsClient;
78     private LocationRequest mLocationRequest;
79     private LocationSettingsRequest mLocationSettingsRequest;
80     private LocationCallback mLocationCallback;
81     public static Location mCurrentLocation;
82
83     // boolean flag to toggle the ui

```

```

84     private Boolean mRequestingLocationUpdates;
85
86     ListView companyListView;
87     ArrayList<String> companies = new ArrayList<String>();
88     ArrayAdapter arrayAdapter;
89
90     @Override
91     protected void onCreate(Bundle savedInstanceState) {
92         super.onCreate(savedInstanceState);
93         setContentView(R.layout.activity_user_location);
94
95         ButterKnife.bind(this);
96
97         // initialize the necessary libraries
98         init();
99
100        // restore the values from saved instance state
101        restoreValuesFromBundle(savedInstanceState);
102
103        // Requesting ACCESS_FINE_LOCATION using Dexter library
104        Dexter.withActivity(this)
105            .withPermission(Manifest.permission.ACCESS_FINE_LOCATION)
106            .withListener(new PermissionListener() {
107                @Override
108                public void onPermissionGranted(PermissionGrantedResponse response) {
109                    mRequestingLocationUpdates = true;
110                    startLocationUpdates();
111                }
112
113                @Override
114                public void onPermissionDenied(PermissionDeniedResponse response) {
115                    if (response.isPermanentlyDenied()) {
116                        // open device settings when the permission is
117                        // denied permanently
118                        openSettings();
119                    }
120                }
121
122                @Override
123                public void onPermissionRationaleShouldBeShown(PermissionRequest permission
124                    , PermissionToken token) {
125                    token.continuePermissionRequest();
126                }
127            }).check();
128
129        setTitle("Nearby Companies");
130        companyListView = findViewById(R.id.companyListView);
131        arrayAdapter = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, companies);
132        companyListView.setAdapter(arrayAdapter);
133
134        companyListView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
135            @Override
136            public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
137                if (companies.size() > position && mCurrentLocation != null)
138                {
139                    String companyName = companies.get(position);
140                    int iEnd = companyName.indexOf(" -");
141                    String substringCompanyName = null;
142                    if (iEnd != -1)
143                    {
144                        substringCompanyName = companyName.substring(0, iEnd);
145                    }
146                    Log.i("INFO COMPANY NAME", substringCompanyName);
147
148                    Intent intent = new Intent(getApplicationContext(), CompanyServices.class);
149                    intent.putExtra("company_tapped", substringCompanyName);

```

```

149         startActivity(intent);
150     }
151 }
152 });
153
154 }
155
156 private void fillCompanyListView()
157 {
158     if(mCurrentLocation != null)
159     {
160         final ParseGeoPoint myPosition = new ParseGeoPoint(mCurrentLocation.getLatitude(),
161 mCurrentLocation.getLongitude());
162         ParseQuery<ParseObject> query = ParseQuery.getQuery("Company");
163         query.whereWithinKilometers("COMPANY_POSITION", myPosition,
164 RADIUS_AREA_IN_KILOMETERS);
165         query.setLimit(8);
166         query.findInBackground(new FindCallback<ParseObject>() {
167             @Override
168             public void done(List<ParseObject> objects, ParseException e) {
169                 if(e == null)
170                 {
171                     companies.clear();
172                     if(objects.size() > 0)
173                     {
174                         for(ParseObject object : objects)
175                         {
176                             String companyName = (String) object.get("COMPANY_NAME");
177                             Double distanceInKilometers = myPosition.distanceInKilometersTo
178 ((ParseGeoPoint) object.get("COMPANY_POSITION"));
179                             Double distanceRounded = (double) Math.round(
180 distanceInKilometers * 10) / 10;
181                             companies.add(companyName + " - " + distanceRounded.toString() +
182 " km");
183                         }
184                     }
185                     else
186                     {
187                         companies.add("No active companies nearby!");
188                     }
189                     arrayAdapter.notifyDataSetChanged();
190                 }
191             }
192         });
193     }
194 }
195
196 private void init() {
197     mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this);
198     mSettingsClient = LocationServices.getSettingsClient(this);
199
200     mLocationCallback = new LocationCallback() {
201         @Override
202         public void onLocationResult(LocationResult locationResult) {
203             super.onLocationResult(locationResult);
204             // location is received
205             mCurrentLocation = locationResult.getLastLocation();
206             mLastUpdateTime = DateFormat.getTimeInstance().format(new Date());
207
208             fillCompanyListView();
209         }
210     };
211
212     mRequestingLocationUpdates = false;
213
214     mLocationRequest = new LocationRequest();

```

```

210         mLocationRequest.setInterval(UPDATE_INTERVAL_IN_MILLISECONDS);
211         mLocationRequest.setFastestInterval(FATEST_UPDATE_INTERVAL_IN_MILLISECONDS);
212         mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
213
214         LocationSettingsRequest.Builder builder = new LocationSettingsRequest.Builder();
215         builder.addLocationRequest(mLocationRequest);
216         mLocationSettingsRequest = builder.build();
217     }
218
219     /**
220      * Restoring values from saved instance state
221      */
222     private void restoreValuesFromBundle(Bundle savedInstanceState) {
223         if (savedInstanceState != null) {
224             if (savedInstanceState.containsKey("is_requesting_updates")) {
225                 mRequestingLocationUpdates = savedInstanceState.getBoolean("
is_requesting_updates");
226             }
227
228             if (savedInstanceState.containsKey("last_known_location")) {
229                 mCurrentLocation = savedInstanceState.getParcelable("last_known_location");
230             }
231
232             if (savedInstanceState.containsKey("last_updated_on")) {
233                 mLastUpdateTime = savedInstanceState.getString("last_updated_on");
234             }
235         }
236     }
237
238     @Override
239     public void onSaveInstanceState(Bundle outState) {
240         super.onSaveInstanceState(outState);
241         outState.putBoolean("is_requesting_updates", mRequestingLocationUpdates);
242         outState.putParcelable("last_known_location", mCurrentLocation);
243         outState.putString("last_updated_on", mLastUpdateTime);
244     }
245
246     /**
247      * Starting location updates
248      * Check whether location settings are satisfied and then
249      * location updates will be requested
250      */
251     private void startLocationUpdates() {
252         mSettingsClient
253             .checkLocationSettings(mLocationSettingsRequest)
254             .addOnSuccessListener(this, new OnSuccessListener<LocationSettingsResponse>() {
255                 @SuppressWarnings("MissingPermission")
256                 @Override
257                 public void onSuccess(LocationSettingsResponse locationSettingsResponse) {
258                     Log.i(TAG, "All location settings are satisfied.");
259
260                     Toast.makeText(getApplicationContext(), "Rilevamento posizione in corso
...", Toast.LENGTH_SHORT).show();
261
262                     //noinspection MissingPermission
263                     mFusedLocationClient.requestLocationUpdates(mLocationRequest,
264                         mLocationCallback, Looper.myLooper());
265
266                     //updateLocationUI();
267                 }
268             })
269             .addOnFailureListener(this, new OnFailureListener() {
270                 @Override
271                 public void onFailure(@NonNull Exception e) {
272                     int statusCode = ((ApiException) e).getStatusCode();
273

```



```

274         switch (statusCode) {
275             case LocationSettingsStatusCodes.RESOLUTION_REQUIRED:
276                 Log.i(TAG, "Location settings are not satisfied. Attempting to
upgrade " +
277                     "location settings ");
278                 try {
279                     // Show the dialog by calling startResolutionForResult(),
and check the
280                     // result in onActivityResult().
281                     ResolvableApiException rae = (ResolvableApiException) e;
282                     rae.startResolutionForResult(UserLocation.this,
REQUEST_CHECK_SETTINGS);
283                 } catch (IntentSender.SendIntentException sie) {
284                     Log.i(TAG, "PendingIntent unable to execute request.");
285                 }
286                 break;
287             case LocationSettingsStatusCodes.SETTINGS_CHANGE_UNAVAILABLE:
288                 String errorMessage = "Location settings are inadequate, and
cannot be " +
289                     "fixed here. Fix in Settings.";
290                 Log.e(TAG, errorMessage);
291
292                 Toast.makeText(UserLocation.this, errorMessage, Toast.
LENGTH_LONG).show();
293             }
294
295             //updateLocationUI();
296         }
297     });
298 }
299
300 @Override
301 protected void onActivityResult(int requestCode, int resultCode, Intent data) {
302     switch (requestCode) {
303         // Check for the integer request code originally supplied to
startResolutionForResult().
304         case REQUEST_CHECK_SETTINGS:
305             switch (resultCode) {
306                 case Activity.RESULT_OK:
307                     Log.e(TAG, "User agreed to make required location settings changes.");
308                     // Nothing to do. startLocationupdates() gets called in onResume again.
309                     break;
310                 case Activity.RESULT_CANCELED:
311                     Log.e(TAG, "User chose not to make required location settings changes."
);
312                     mRequestingLocationUpdates = false;
313                     break;
314             }
315             break;
316         }
317     }
318
319     private void openSettings() {
320         Intent intent = new Intent();
321         intent.setAction(
322             Settings.ACTION_APPLICATION_DETAILS_SETTINGS);
323         Uri uri = Uri.fromParts("package",
324             BuildConfig.APPLICATION_ID, null);
325         intent.setData(uri);
326         intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
327         startActivity(intent);
328     }
329
330 @Override
331 public void onResume() {
332     super.onResume();

```

```

333
334     // Resuming location updates depending on button state and
335     // allowed permissions
336     if (mRequestingLocationUpdates && checkPermissions()) {
337         startLocationUpdates();
338     }
339
340     //updateLocationUI();
341 }
342
343 private boolean checkPermissions() {
344     int permissionState = ActivityCompat.checkSelfPermission(this,
345         Manifest.permission.ACCESS_FINE_LOCATION);
346     return permissionState == PackageManager.PERMISSION_GRANTED;
347 }
348 }

```



```

1 package com.triassi.testparsenote;
2
3 import android.content.Intent;
4 import android.support.v7.app.AppCompatActivity;
5 import android.os.Bundle;
6 import android.util.Log;
7 import android.view.View;
8 import android.widget.AdapterView;
9 import android.widget.AdapterView.OnItemClickListener;
10 import android.widget.ArrayAdapter;
11 import android.widget.ListView;
12 import android.widget.TextView;
13
14 import com.parse.FindCallback;
15 import com.parse.ParseException;
16 import com.parse.ParseObject;
17 import com.parse.ParseQuery;
18
19 import java.util.ArrayList;
20 import java.util.List;
21
22 public class CompanyServices extends AppCompatActivity {
23
24     private TextView companyNameTextView;
25     private String companyName;
26     ListView serviceListView;
27     ArrayList<String> servicesList = new ArrayList<String>();
28     ArrayAdapter arrayAdapter;
29
30     @Override
31     protected void onCreate(Bundle savedInstanceState) {
32         super.onCreate(savedInstanceState);
33         setContentView(R.layout.activity_company_services);
34
35         companyNameTextView = findViewById(R.id.companyNameTextView);
36         serviceListView = findViewById(R.id.serviceListView);
37
38         arrayAdapter = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1,
39             servicesList);
40         serviceListView.setAdapter(arrayAdapter);
41
42         Intent intent = getIntent();
43         companyName = intent.getStringExtra("company_tapped");
44         companyNameTextView.setText(companyName);
45
46         updateServicesList();
47
48         serviceListView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
49             @Override

```

```

48         public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
49             if(servicesList.size() > position)
50             {
51                 String serviceName = servicesList.get(position);
52
53                 Intent intent = new Intent(getApplicationContext(), Queue.class);
54                 intent.putExtra("service_tapped", serviceName);
55                 startActivity(intent);
56                 finish();
57             }
58         }
59     });
60 }
61
62 private void updateServicesList()
63 {
64     ParseQuery<ParseObject> query = ParseQuery.getQuery("Company");
65     query.whereEqualTo("COMPANY_NAME", companyName);
66     query.findInBackground(new FindCallback<ParseObject>() {
67         @Override
68         public void done(List<ParseObject> objects, ParseException e) {
69             if(e == null)
70             {
71                 if(objects.size() > 0)
72                 {
73                     for (final ParseObject object : objects)
74                     {
75                         ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
76                         query.whereEqualTo("COMPANY", object);
77                         query.findInBackground(new FindCallback<ParseObject>() {
78                             @Override
79                             public void done(List<ParseObject> services, ParseException e) {
80
81                                 if(e == null)
82                                 {
83                                     if(services.size() > 0)
84                                     {
85                                         for (ParseObject service : services)
86                                         {
87                                             Log.i("SERVICE INFO", service.getString("
88 SERVICE_NAME"));
89                                             String serviceName = service.getString("
90 SERVICE_NAME");
91                                             servicesList.add(serviceName);
92                                         }
93                                     }
94                                     arrayAdapter.notifyDataSetChanged();
95                                 }
96                             }
97                         });
98                     }
99                 }
100             }
101 }

```

```

1 package com.triassi.testparsenote;
2
3 import android.app.Notification;
4 import android.content.Context;
5 import android.content.Intent;
6 import android.content.SharedPreferences;
7 import android.os.Handler;

```

```

8 import android.support.v4.app.NotificationCompat;
9 import android.support.v4.app.NotificationManagerCompat;
10 import android.support.v7.app.AppCompatActivity;
11 import android.os.Bundle;
12 import android.util.Log;
13 import android.view.View;
14 import android.widget.TextView;
15
16 import com.parse.FindCallback;
17 import com.parse.ParseException;
18 import com.parse.ParseObject;
19 import com.parse.ParseQuery;
20 import com.parse.ParseUser;
21
22 import java.util.List;
23
24 import static com.triassi.testparsenote.Notifications.CHANNEL_1_ID;
25
26 public class Queue extends AppCompatActivity {
27
28     private View view;
29     private String serviceName;
30     private TextView serviceNameTextView;
31     private TextView myNumberTextView;
32     private TextView currentNumberTextView;
33     private int myNumber;
34     private int currentNumber;
35     private boolean isAlreadyInQueue = false;
36
37     private NotificationManagerCompat notificationManager;
38
39     SharedPreferences sharedPrefs;
40     SharedPreferences.Editor editor;
41
42     @Override
43     protected void onCreate(Bundle savedInstanceState) {
44         super.onCreate(savedInstanceState);
45         setContentView(R.layout.activity_queue);
46
47         serviceNameTextView = findViewById(R.id.serviceNameTextView);
48         myNumberTextView = findViewById(R.id.myNumberTextView);
49         currentNumberTextView = findViewById(R.id.currentNumberTextView);
50
51         notificationManager = NotificationManagerCompat.from(this);
52
53         if (!isAlreadyInQueue) {
54             Intent intent = getIntent();
55             serviceName = intent.getStringExtra("service_tapped");
56             checkIfQueueHasBeenReset();
57             checkTurn();
58         }
59
60         serviceNameTextView.setText(serviceName);
61         myNumberTextView.setText(String.valueOf(myNumber));
62     }
63
64     @Override
65     protected void onResume() {
66         super.onResume();
67
68         sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
69
70         checkTurn();
71
72         serviceNameTextView.setText(sharedPrefs.getString("serviceNameLabel", ""));
73         myNumberTextView.setText(String.valueOf(sharedPrefs.getInt("myTurn", 0)));

```

```

74     }
75
76     private void checkIfQueueHasBeenReset() {
77         sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
78
79         ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
80
81         query.whereEqualTo("SERVICE_NAME", serviceName);
82         query.findInBackground(new FindCallback<ParseObject>() {
83             public void done(List<ParseObject> services, ParseException e) {
84                 if (e == null) {
85                     if (services.size() > 0) {
86                         for (ParseObject service : services) {
87                             myNumber = service.getInt("MY_NUMBER");
88                             currentNumber = service.getInt("CURRENT_NUMBER");
89                             Log.i("SERVICE NAME LABEL", serviceName);
90                             Log.i("CURRENT NUMBER", String.valueOf(currentNumber));
91                             Log.i("MY NUMBER", String.valueOf(myNumber));
92                             if (currentNumber == -1 && myNumber == 0) /* If the queue is closed
93
94                                     onLogOutClick(view);
95                                 } else /* If the queue is open */ {
96                                     gettingNumber();
97                                 }
98                             }
99                         }
100                     }
101                 });
102             }
103
104     private void checkTurn()
105     {
106         sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
107
108         final Handler handler = new Handler();
109         Runnable runnable = new Runnable() {
110             @Override
111             public void run()
112             {
113                 ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
114                 query.whereEqualTo("SERVICE_NAME", sharedPrefs.getString("serviceNameLabel", ""
115
116                 query.findInBackground(new FindCallback<ParseObject>()
117                 {
118                     public void done(List<ParseObject> services, ParseException e)
119                     {
120                         if (e == null)
121                         {
122                             if(services.size() > 0)
123                             {
124                                 for (ParseObject service : services)
125                                 {
126                                     currentNumber = service.getInt("CURRENT_NUMBER");
127
128                                     if(currentNumber == sharedPrefs.getInt("myTurn", 0))
129                                     {
130                                         currentNumberTextView.setText("E' il tuo turno!");
131
132                                         /* Notification popup here */
133                                         Notification notification = new NotificationCompat.
134                                         Builder(getApplicationContext(), CHANNEL_1_ID)
135                                             .setSmallIcon(R.drawable.ic_done)
136                                             .setContentTitle(sharedPrefs.getString("
137
138                                             serviceNameLabel", ""))
139                                             .setContentText("E' il tuo turno!")

```

```

136         .setPriority(NotificationCompat.PRIORITY_HIGH)
137         .setCategory(NotificationCompat.CATEGORY_MESSAGE)
138     )
139     .setOngoing(false)
140     .build();
141     notificationManager.notify(1, notification);
142 }
143 else if(currentNumber == sharedPrefs.getInt("myTurn", 0) +
144 1)
145 {
146     handler.removeCallbacksAndMessages(null);
147     onLogOutClick(view);
148     finish();
149 }
150 else
151 {
152     currentNumberTextView.setText("Il numero corrente e' " +
String.valueOf(currentNumber));
153     }
154 }
155 }
156 }
157 }
158 });
159
160     handler.postDelayed(this, 1000);
161 }
162 };
163
164     handler.postDelayed(runnable, 1000);
165 }
166
167 private void gettingNumber()
168 {
169     ParseQuery<ParseObject> query = ParseQuery.getQuery("Service");
170     query.whereEqualTo("SERVICE_NAME", serviceName);
171     query.findInBackground(new FindCallback<ParseObject>()
172     {
173         public void done(List<ParseObject> services, ParseException e)
174         {
175             if (e == null)
176             {
177                 if(services.size() > 0)
178                 {
179                     for (ParseObject service : services)
180                     {
181                         service.increment("MY_NUMBER");
182                         service.saveInBackground();
183
184                         myNumber = service.getInt("MY_NUMBER");
185
186                         sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE)
187
188                         ;
189                         editor = sharedPrefs.edit();
190                         editor.putInt("myTurn", myNumber);
191                         isAlreadyInQueue = true;
192                         editor.putBoolean("isAlreadyInQueue", isAlreadyInQueue);
193                         editor.putString("serviceNameLabel", serviceName);
194                         editor.apply();
195
196                         serviceNameTextView.setText(service.getString("SERVICE_NAME"));
197
198                         myNumberTextView.setText(String.valueOf(myNumber));
199

```

```

198         }
199     }
200 }
201 }
202 });
203 }
204
205 @Override
206 public void onBackPressed()
207 {
208
209 }
210
211 public void onLogOutClick(View view)
212 {
213     ParseUser.logOut();
214
215     sharedPrefs = getSharedPreferences("myPrefs", Context.MODE_PRIVATE);
216     editor = sharedPrefs.edit();
217     isAlreadyInQueue = false;
218     editor.putBoolean("isAlreadyInQueue", isAlreadyInQueue);
219     editor.apply();
220
221     Intent intent = new Intent(getApplicationContext(), MainActivity.class);
222     startActivity(intent);
223     finish();
224 }
225 }

```

```

1 package com.triassi.testparsenote;
2
3 import android.app.Application;
4 import android.app.NotificationChannel;
5 import android.app.NotificationManager;
6 import android.os.Build;
7
8 public class Notifications extends Application
9 {
10     public static final String CHANNEL_1_ID = "channel1";
11     public static final String CHANNEL_2_ID = "channel2";
12
13
14     @Override
15     public void onCreate()
16     {
17         super.onCreate();
18
19         createNotificationChannels();
20     }
21
22     private void createNotificationChannels()
23     {
24         if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O)
25         {
26             NotificationChannel channel1 = new NotificationChannel(
27                 CHANNEL_1_ID,
28                 "Channel 1",
29                 NotificationManager.IMPORTANCE_HIGH
30             );
31             channel1.setDescription("This is Channel 1");
32
33             NotificationChannel channel2 = new NotificationChannel(
34                 CHANNEL_2_ID,
35                 "Channel 2",
36                 NotificationManager.IMPORTANCE_LOW

```

```
37         );
38         channel2.setDescription("This is Channel 2");
39
40         NotificationManager manager = getSystemService(NotificationManager.class);
41         manager.createNotificationChannel(channel1);
42         manager.createNotificationChannel(channel2);
43     }
44 }
45 }
```


Bibliografia

- [1] Developer Android. *Documentation Android Developers*. <https://developer.android.com/docs/>.
- [2] GitHub. *Parse-compatible API server module for Node/Express*. <https://github.com/parse-community/parse-server>.
- [3] Arnav Gupta. *Parse in 2017—remembering the forgotten, using PostgreSQL, and some benchmarks*. <https://medium.com/coding-blocks/parse-in-2017-remembering-the-forgotten-using-postgresql-and-some-benchmarks-7d35e17b7698>, 2017.
- [4] Kontakt.io. *Infographic: What are Beacons and What Do They Do?* <https://kontakt.io/blog/infographic-beacons/>.
- [5] Brunel University London. *Queuing theory*. <http://people.brunel.ac.uk/~mastjjb/jeb/or/queue.html>.
- [6] Queue Mobile. *Queue Management Systems — Virtual SMS Queuing System*. <http://queuemobile.com/index.html>.
- [7] Parse. *Parse Server*. <https://parseplatform.org/>.
- [8] Parse. *Parse Server Guide*. <https://docs.parseplatform.org/parse-server/guide/>.
- [9] Rackone. *Big Data con i Database NoSQL: un'introduzione pratica*. <https://www.rackone.it/big-data-con-database-nosql-unintroduzione-pratica/>, 2015.
- [10] Parse Server. *Android Developers Guide*. <https://docs.parseplatform.org/android/guide/>.
- [11] Amazon Web Services. *Amazon EC2*. <https://aws.amazon.com/it/ec2/>.
- [12] Skiplino. *Skiplino - Mobile Queueing App*. <https://skiplino.com/customer/>.
- [13] Ravi Tamada. *Android Location API using Google Play Services*. <https://www.androidhive.info/2015/02/android-location-api-using-google-play-services/>, 2018.

- [14] The Telegraph. *Britons spend six months queuing*.
<https://www.telegraph.co.uk/news/newstopics/howaboutthat/5052956/Britons-spent-six-months-queuing.html>, 2009.
- [15] Wikipedia. *Applicazione mobile*. https://it.wikipedia.org/wiki/Applicazione_mobile.
- [16] Wikipedia. *Backend as a service*. https://it.wikipedia.org/wiki/Backend_as_a_service.