



Emergency system I'm in danger



Supervised by

Dr/ Ehab Rushdy

Project Team

- Elsayed Ragaey Mohamed
- Mahmoud Mohamed Abdullah
- Mohamed Mamdouh Mohamed
- Abeer Lotfy Ali
- Maryain Gerges Shehata

Supervised by

Dr/ Ehab Rushdy

Eng/ mahmoud elmahdy

Acknowledgment

It has been our great pleasure to work on this project, we enjoyed the INFORMATION TECHNOLOGY field very much to be done with that project it was hard work but we are finding that it also quite fun because we are working under a very good supervisor.

As students of "FACULTY OF COMPUTER AND INFORMATION",

We would like to owe our first appreciation and thanks to " DR/ EHAB RUSHDY " for his care, supervision and for directing us to carry out this project.

We would like to thank the supervisors and the university staff, we are grateful for all the help and support they provided to guide us through this project. However, they were many more individuals who have lent a hand, we show gratitude to those who contributed to present our project in the most appropriate manner.

Abstract

Everyday there are a lot of deaths because of wrong Diagnosis especially if the patient is Unconscious. In accidents the most probably case is the injured is unconscious and if the rescuers want to know any information or contact any known body for him they found his phone is locked.

It will be an emergency system on Android devices doesn't need to unlock phone to contact with mobile owner emergency list by widget on lock screen.

The new system must include the following:

- Ability to locate and send the location through Emergency SMS.
- Ability to display on the mobile screen the medical history of the injured.
- Ability to incorporate automated routing and emergency notifications based on location.

And it's benefit is:

- Avoid and assist is diagnose.
- Automated connection with emergency agents and contacts (based on location).

Contents

CHAPTER 1 ANDRIOD

1.1 INTRODUCTION

1.2 CREATING SIMPLE VALUES

1.3 CUSTOMIZE RESOURCE VALUES

1.4 A CLOSER LOOK AT ANDROID ACTIVITIES

1.5 CREATING USER INTERFACE

1.6 LOCATION-BASED SERVICES

CHAPTER 2 PARSE.COM

2.1 GETTING START

2.2 OBJECTS

2.3 Queries

2.4 Users

2.5 Sessions

2.6 File system

2.7 Relations

2.8 Data

2.9 Security

2.10 User Interface

2.11 Cloud Code

2.12 Performance

CHAPTER 3 PROBLEM & SOLUTION

CHAPTER 4 PROJECT IMPLEMENTATION

CHAPTER 5 CONCLUSION

ANDROID



1.1 Introduction

In the days before Twitter and Facebook, when Google was still a twinkle in its founders' eyes and dinosaurs roamed the earth, mobile phones were just that portable phones small enough to fit inside a briefcase, featuring batteries that could last up to several hours. They did however offer the freedom to make calls without being physically connected to a landline.

Increasingly small, stylish, and powerful mobile phones are now as ubiquitous as they are indispensable.

Hardware advancements have made mobiles smaller and more efficient while including an increasing number of peripherals.

After first getting cameras and media players, mobiles now include GPS systems, accelerometers, and touch screens. While these hardware innovations should prove fertile ground for software development, the applications available for mobile phones have generally lagged behind the hardware.

Android sits alongside a new wave of mobile operating systems designed for increasingly powerful mobile hardware. Windows Mobile, the Apple iPhone, and the Palm Pre now provide a richer, simplified development environment for mobile applications.

However, unlike Android, they're built on proprietary operating systems that in some cases prioritize native applications over those created by third parties, restrict communication among applications and native phone data, and restrict or control the distribution of third-party apps to their platforms.

Android offers new possibilities for mobile applications by offering an open development environment built on an open-source Linux kernel. Hardware access is available to all applications through a series of API libraries, and application interaction, while carefully controlled, is fully supported.

In Android, all applications have equal standing. Third-party and native Android applications are written with the same APIs and are executed on the same run time. Users can remove and replace any native application with a third-party developer alternative; even the dialer and home screens can be replaced.

1.2 Creating Simple Values

Strings

Externalizing your strings helps maintain consistency within your application and makes it much easier to create localized versions.

String resources are specified with the `<string>` tag, as shown in the following XML snippet.

```
<string name="stop_message">Stop.</string>
```

Android supports simple text styling, so you can use the HTML tags ``, `<i>`, and `<u>` to apply bold, italics, or underlining respectively to parts of your text strings, as shown in the following example:

```
<string name="stop_message"><b>Stop.</b></string>
```

You can use resource strings as input parameters for the `String.format` method. However, `String.format` does not support the text styling described above. To apply styling to a format string you have to escape the HTML tags when creating your resource, as shown in the following.

```
<string name="stop_message">&lt;b>Stop&lt;/b>.<br>%1$s</string>
```

Within your code, use the `Html.fromHtml` method to convert this back into a styled character sequence.

```
String rString =  
getString(R.string.stop_message);
```

```
String fString = String.format(rString,  
"Collaborate and listen.");
```

```
CalrSequence styledString =
```

```
Html.fromHtml(fString);
```

Colors

Use the `<color>`

tag to define a new color resource. Specify the color value using a `#` symbol followed

by the (optional) alpha channel, then the red, green, and blue values using one or two hexadecimal numbers with any of the following notations:

- #RGB
- #RRGGBB
- #ARGB
- #AARRGGBB

The following example shows how to specify a fully opaque blue and a partially transparent green.

```
<color name="opaque_blue">#00F</color>
```

```
<color name="transparent_green">#7700FF00</color>
```

Dimensions

Dimensions are most commonly referenced within style and layout resources. They're useful for creating layout constants such as borders and font heights.

To specify a dimension resource use the `<dimen>` tag, specifying the dimension value, followed by an identifier describing the scale of your dimension:

- `px` (screen pixels)
- `in` (physical inches)

- `pt` (physical points)
- `mm` (physical millimeters)
- `dp` (density-independent pixels relative to a 160-dpi screen)
- `sp` (scale-independent pixels)

These alternatives let you define a dimension not only in absolute terms, but also using relative scales that account for different screen resolutions and densities to simplify scaling on different hardware.

The following XML snippet shows how to specify dimension values for a large font size and a standard border:

```
<dimen name="standard_border">5dp</dimen>
<dimen name="large_font_size">16sp</dimen>
```

Styles and Themes

Style resources let your applications maintain a consistent look and feel by enabling you to specify the attribute values used by Views. The most common use of themes and styles is to store the colors and fonts for an application.

You can easily change the appearance of your application by simply specifying a different style as the theme in your project manifest.

To create a style use a `<style>` tag that includes a `name` attribute and contains one or more `item` tags.

Each `item` tag should include a `name` attribute used to specify the attribute (such as font size or color) being defined. The tag itself should then contain the value, as shown in the following skeleton code.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<resources>

<style name="StyleName">

<item name="attributeName">value</item>

</style>

</resources>
```

Styles support inheritance using the `parent` attribute on the `<style>` tag, making it easy to create simple variations.

The following example shows two styles that can also be used as a theme: a base style that sets several text properties and a second style that modifies the first to specify a smaller font.

```
<?xml version="1.0" encoding="utf-8"?>

<resources>

<style name="BaseText">

<item name="android:textSize">14sp</item>

<item name="android:textColor">#111</item>

</style>

<style name="SmallText"parent="BaseText">

<item name="android:textSize">8sp</item>

</style>

</resources>
```

Drawables

Drawable resources include bitmaps and NinePatch (stretchable PNG) images. They also include complex composite Drawables, such as `LevelListDrawables` and `StateListDrawables` that can be defined in XML.

All Drawables are stored as individual files in the `res/drawable` folder. The resource identifier for a Drawable resource is the lowercase file name without an extension.

Layouts

Layout resources let you decouple your presentation layer by designing user interface layouts in XML rather than constructing them in code.

The most common use of a layout is for defining the user interface for an Activity. Once defined in XML, the layout is “inflated” within an Activity using `setContentView` , usually within the `onCreate` method. You can also reference layouts from within other layout resources, such as layouts for each row in a List View.

Using layouts to create your screens is best-practice UI design in Android. The decoupling of the layout from the code lets you create optimized layouts for

different hardware configurations, such as varying screen sizes, orientation, or the presence of keyboards and touchscreens.

1.3 customize resource values

➤ Mobile Country Code and Mobile Network Code (MCC/MNC) The country, and optionally the network, associated with the SIM currently used in the device. The MCC is specified by `mcc` followed by the three-digit country code. You can optionally add the MNC using `mnc` and the two- or three-digit network code (e.g., `mcc234-mnc20` or `mcc310`).

You can find a list of MCC/MNC codes on Wikipedia at http://en.wikipedia.org/wiki/Mobile_Network_Code

➤ Language and Region Language specified by the lowercase two-letter ISO 639-1 language code, followed optionally by a region specified by a lowercase `r` followed by the uppercase two-letter ISO 3166-1-alpha-2 language code (e.g., `en` , `en-rUS`, or `en-rGB`).

➤ Screen Size One of `Small`(smaller than HVGA), `medium`(at least HVGA and smaller than VGA), or `large`(VGA or larger).

➤ Screen Width/Length Specify `Long` or `notlong` for resources designed specifically for wide screen (e.g., WVGA is `long` ,QVGA is `notlong`).

➤ Screen Orientation One of `Port` (portrait), `land`(landscape), or `Square`(square).

➤ Screen Pixel Density Pixel density in dots per inch (dpi). Best practice is to use `Ldpi` , `Mdpi` , Or `hdpi` to specify low (120 dpi), medium (160 dpi), or high (240 dpi) pixel density respectively. You can specify `nodpi` for bitmap resources you don't want scaled to support an exact screen density. Unlike with other resource types Android does not require an exact match to select a resource. When selecting the appropriate folder it will choose the nearest match to the device's pixel density and scale the resulting Drawables accordingly.

➤ Touchscreen Type One of `Notouch` , `Stylus` , or `finger`.

➤ **Keyboard Availability** One of `Keysexposed` , `Keyshidden` ,or `keysoft`.

➤ **Keyboard Input Type** One of `Nokeys` ,`Qwerty` ,or `12key`.

➤ **UI Navigation Type** One of `Nonav` ,`Dpad` ,`Trackball` ,or `wheel`.

You can specify multiple qualifiers for any resource type, separating each qualifier with a hyphen. Any combination is supported; however, they must be used in the order given in the preceding list, and no more than one value can be used per qualifier.

The following example shows valid and invalid directory names for alternative Drawable resources.

➤ **Valid:**

`drawable-en-rUS`

`drawable-en-keyshidden`

`drawable-long-land-notouch-nokeys`

➤ **Invalid:**

`drawable-rUS-en` (out of order)

`drawable-rUS-rUK` (multiple values for a single qualifier)

Android retrieves a resource at run time, it will find the best match from the available alternatives. Starting with a list of all the folders in which the required value exists, it will select the one with the greatest number of matching qualifiers. If two folders are an equal match, the tiebreaker will be based on the order of the matched qualifiers in the preceding list.

4.1 A CLOSER LOOK AT ANDROID ACTIVITIES

To create user interface screens you extend the `Activity` class, using Views to provide the UI and allow user interaction.

Each Activity represents a screen (similar to a Form) that an application can present to its users. The more complicated your application, the more screens you are likely to need.

Create a new Activity for every screen you want to display. Typically this includes at least a primary interface screen that handles the main UI functionality of your application. This primary interface is often supported by secondary Activities for entering information, providing different perspectives on

your data, and supporting additional functionality. To move between screens start a new Activity (or return from one).

Most Activities are designed to occupy the entire display, but you can also create

A Closer Look at Android Activities

Creating an Activity

Extend `Activity` to create a new Activity class. Within this new class you must define the user interface and implement your functionality. The basic skeleton code for a new Activity is

```

package com.paad.myapplication;

import android.app.Activity;
import android.os.Bundle;

public class MyActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}

```

The base Activity class presents an empty screen that encapsulates the window display handling. An empty Activity isn't particularly useful, so the first thing you'll want to do is create the user interface with Views and layouts.

Views are the user interface controls that display data and provide user interaction. Android provides several layout classes, called View Groups , that can contain multiple Views to help you design your user interfaces., examining what's available, how to use them, and how to create your own Views and layouts.To assign a user interface to an Activity, call `setContentView` from the `onCreate` method of your Activity.

In this first snippet, an instance of `TextView` is used as the Activity's user interface:

```

@Override

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    TextView textView = new TextView(this);

```

```
setContentView(textView);  
}
```

Usually you'll want to use a more complex UI design. You can create a layout in code using layout View Groups, or you can use the standard Android convention of passing a resource ID for alayout defined in an external resource, as shown in the following snippet:

```
@Override  
  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
}
```

In order to use an Activity in your application you need to register it in the manifest. Add new `<activity>` tags within the `<application>` node of the manifest; the `<activity>` tag includes attributes for metadata such as the label, icon, required permissions, and themes used by the Activity. An Activity without a corresponding `<activity>` tag can't be displayed

```
<activity android:label="@string/app_name"  
    android:name=".MyActivity">  
  
</activity>
```

Within the `<activity>` tag you can add `<intent-filter>` nodes that specify the Intents your Activity will listen for and react to. Each Intent Filter defines one or more actions and categories that your Activity supports. Intents and Intent Filters , but it's worth noting that for an Activity to be available from the main application launcher it must include an Intent Filter listening for the `MAIN` action and the `LAUNCHER` category

```
<activity android:label="@string/app_name"
android:name=".MyActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN"
/>

<category
android:name="android.intent.category.LAUNCHER"
/>

</intent-filter>

</activity>
```

The Activity Life Cycle

A good understanding of the Activity life cycle is vital to ensure that your application provides a seam-less user experience and properly manages its resources.

As explained earlier, Android applications do not control their own process lifetimes; the Android runtime manages the process of each application, and by extension that of each Activity within it.

While the run time handles the termination and management of an Activity's process, the Activity's state helps determine the priority of its parent application. The application priority, in turn, influences the likelihood that the run time will terminate it and the Activities running within it

Activity Stacks

The state of each Activity is determined by its position on the Activity stack, a last-in–first-out collection of all the currently running Activities. When a new Activity starts, the current foreground screen is moved to the top of the stack. If the user navigates back using the Back button, or the foreground

Activity is closed, the next Activity on the stack moves up and becomes active.

Activity States

As Activities are created and destroyed they move in and out of the stack. As they do so, they transition through four possible states:

➤ **Active** When an Activity is at the top of the stack it is the visible, focused, foreground Activity that is receiving user input. Android will attempt to keep it alive at all costs, killing Activities further down the stack as needed, to ensure that it has the resources it needs. When another Activity becomes active, this one will be paused.

➤ **Paused** In some cases your Activity will be visible but will not have focus; at this point it's paused. This state is reached if a transparent or non-full-screen Activity is active in front of it. When paused, an Activity is treated as if it were active; however, it doesn't receive user input events. In extreme cases Android will kill a paused Activity to recover resources for the active Activity. When an Activity becomes totally obscured, it is stopped.

➤ **Stopped** When an Activity isn't visible, it "stops." The Activity will remain in memory, retaining all state information; however, it is now a candidate for termination when the system requires memory elsewhere. When an Activity is stopped it's important to save data and the current UI state. Once an Activity has exited or closed, it becomes inactive.

➤ **Inactive** After an Activity has been killed, and before it's been launched, it's inactive. Inactive Activities have been removed from the Activity stack and need to be restarted before they can be displayed and used. State transitions are nondeterministic and are handled entirely by the Android memory manager. Android will start by closing applications that contain inactive Activities, followed by those that are stopped. In extreme cases it will remove those that are paused

5.1 creating user interface

FUNDAMENTAL ANDROID UI DESIGN

User interface (UI) design, user experience (UX), human computer interaction (HCI), and usability are huge topics that aren't covered in great depth in this book. Nonetheless, it's important that you get them right when creating your user interfaces.

Android introduces some new terminology for familiar programming metaphors that will be explored in detail in the following sections:

➤ **Views**

Views are the base class for all visual interface elements (commonly known as controls or widgets).

All UI controls, including the layout classes, are derived from

View.

➤ View Groups

View Groups are extensions of the View class that can contain multiple child Views. Extend the `ViewGroup` class to create compound controls made up of interconnected child Views. The `ViewGroup` class is also extended to provide the layout managers that help you lay out controls within your Activities.

➤ Activities

Activities, described in detail in the previous chapter, represent the window, or screen, being displayed. Activities are the Android equivalent of Forms. To display a user interface you assign a View (usually a layout) to an Activity.

Android provides several common UI controls, widgets, and layout managers.

For most graphical applications it's likely that you'll need to extend and modify these standard Views or create composite or entirely new Views to provide your own user experience.

INTRODUCING LAYOUTS

Layout managers (more generally just called layout) are extensions of the `ViewGroup` class used to position child controls for your UI. Layouts

can be nested, letting you create arbitrarily complex interfaces using a combination of layouts.

The Android SDK includes some simple layouts to help you construct your UI. It's up to you to select the right combination of layouts to make your interface easy to understand and use.

The following list includes some of the more versatile layout classes available:

➤ `FrameLayout`

The simplest of the Layout Managers, the Frame Layout simply pins each child view to the top left corner. Adding multiple children stacks each new child on top of the one before, with each new View obscuring the last.

➤ `LinearLayout` A Linear Layout aligns each child View in either a vertical or a horizontal

line. A vertical layout has a column of Views, while a horizontal layout has a row of Views. The Linear Layout manager enables you to specify a "weight" for each child View that controls the relative size of each within the available space.

➤ `RelativeLayout`

The most flexible of the native layouts, the Relative Layout lets you define the positions of each child View relative to the others and to the screen boundaries.

➤ `TableLayout`

The Table Layout lets you lay out Views using a grid of rows and columns.

Tables can span multiple rows and columns, and columns can be set to shrink or grow.

➤ Gallery

A Gallery Layout displays a single row of items in a horizontally scrolling list.

The Android documentation describes the features and properties of each layout class in detail, so rather than repeat it here, I'll refer you to [http://developer.android.com/guide/topics/ui/](http://developer.android.com/guide/topics/ui/layout-objects.html)

[layout-objects.html](http://developer.android.com/guide/topics/ui/layout-objects.html) Later in this chapter you'll also learn how to create compound controls (widgets made up of several interconnected Views) by extending these layout classes.

Optimizing Layouts

Inflating layouts into your Activities is an expensive process. Each additional nested layout and View can have a dramatic impact on the performance and seamlessness of your applications.

In general, it's good practice to keep your layouts as simple as possible, but also to avoid needing to inflate an entirely new layout for small changes to an existing one. The following points include some best practice guidelines for creating efficient layouts. Note that they are not exhaustive.

➤ Avoid unnecessary nesting: Don't put one layout within another unless it is necessary. A Linear Layout within a Frame Layout, both of which are set to `FILL_PARENT`, does nothing but add extra time to inflate. Look for redundant layouts, particularly if you've been making significant changes to an existing layout.

➤ Avoid using too many Views: Each additional View in a layout takes time and resources to inflate. A layout shouldn't ever include more than 80 Views or the time taken to inflate it becomes significant.

➤ Avoid deep nesting: As layouts can be arbitrarily nested, it's easy to create complex, deeply nested hierarchies. While there is no hard limit, it's good practice to restrict nesting to fewer than 10 levels.

It's important that you optimize your layout hierarchies to reduce inefficiencies and eliminate unnecessary nesting. To assist you, the Android SDK includes the `Layoutopt` command line tool. Call `calllayoutopt`, passing in the name of the layout resource (or a resource folder) to have your layouts analyzed and to receive recommendations for fixes and improvements.

Menus

Create menu resources to further decouple your presentation layer by designing your menu layouts in XML rather than constructing them in code. Menu resources can be used to define both Activity and context menus within your applications, and provide the same options you would have when constructing your menus in code. Once defined in XML, a menu is "inflated" within your application via the `Inflate`

method of the `MenuInflater` Service, usually within the `onOptionsItemSelected` method..

Each menu definition is stored in a separate file, each containing a single menu, in the `res/menu` folder.

The file name then becomes the resource identifier. Using XML to define your menus is best-practice design in Android.

```
<?xml version="1.0" encoding="utf-8"?>

<menu
xmlns:android="http://schemas.android.com/apk/res
/android">

<item android:id="@+id/menu_refresh"
android:title="Refresh" />

<item android:id="@+id/menu_settings"
android:title="Settings" />

</menu>
```

CREATING AND USING MENUS

Menus offer a way to expose application functions without sacrificing valuable screen space. Each Activity can specify its own menu that's displayed when the device's menu button is pressed.

Android also supports context menus that can be assigned to any View. Context menus are normally triggered when a user holds the middle D-

pad button,depresses the trackball, or long-presses the touch screen for around three seconds when the View has focus.

Activity and context menus support submenus, checkboxes, radio buttons, shortcut keys, and icons.

Menu Item Options

Android supports most of the traditional Menu Item options you're probably familiar with, including

icons, shortcuts, checkboxes, and radio buttons, as listed here:

➤ Checkboxes and radio buttons

Checkboxes and radio buttons on Menu Items are visible in

expanded menus and submenus, as shown in Figure 4-9. To set a Menu Item as a checkbox, use the `setCheckable` method. The state of that checkbox is controlled via `setChecked`.

.

A radio button group is a group of items displaying circular buttons, in which only one item can be selected at any given time. Checking one of these items will automatically uncheck any checked item in the same group.

To create a radio button group, assign the same group identifier to each item and then call `Menu.setGroupCheckable`, passing in that group identifier and setting the exclusive parameter to `true`.

Checkboxes are not visible in the icon menu, so Menu Items that feature checkboxes should be reserved for submenus and items that appear only in the expanded menu. The following code snippet shows how to add a checkbox and a group of three radio buttons.

```
// Create a new check box item.

menu.add(0,          CHECKBOX_ITEM,          Menu.NONE,
"CheckBox").setCheckable(true);

// Create a radio button group.

menu.add(RB_GROUP,    RADIOBUTTON_1,    Menu.NONE,
"Radiobutton 1");

menu.add(RB_GROUP,    RADIOBUTTON_2,    Menu.NONE,
"Radiobutton 2");

menu.add(RB_GROUP,    RADIOBUTTON_3,    Menu.NONE,
"Radiobutton 3").setChecked(true);

menu.setGroupCheckable(RB_GROUP, true, true);
```

➤ Shortcut keys

You can specify a keyboard shortcut for a Menu Item using the

`setShortcut` method. Each call to `setShortcut`

requires two shortcut keys, one for use with the numeric keypad and a second to support a full keyboard. Neither key is case-sensitive.

```
// Add a shortcut to this menu item, '0' if using
the numeric keypad

// or 'b' if using the full keyboard.
```

```
menuItem.setShortcut('0', 'b');
```

➤ Condensed titles

The icon menu does not display shortcuts or checkboxes, so it's often necessary to modify its display text to indicate its state. The `setTitleCondensed` method lets you specify text to be displayed only in the icon menu.

```
menuItem.setTitleCondensed("Short Title");
```

➤ Icons

The icon property is a Drawable resource identifier for an icon to be used in the Menu Item. Icons are displayed only in the icon menu; they are not visible in the extended menu or submenus. You can specify any Drawable resource as a menu icon, though by convention menu icons are generally grayscale and use an embossed style.

```
menuItem.setIcon(R.drawable.menu_item_icon);
```

➤ Menu item click listener

An event handler that will execute when the Menu Item is selected.

For efficiency, the use of such an event handler is discouraged; instead, Menu Item selections should be handled by the `onOptionsItemSelected` handler.

```
menuItem.setOnMenuItemClickListener(new
OnMenuItemClickListener() {

public boolean onMenuItemClick(MenuItem
_menuItem) {

[ ... execute click handling, return true if
handled ... ]

return true;

}

});
```

➤Intents

An Intent assigned to a Menu Item is triggered when the clicking of a Menu Item isn't handled by either a MenuItemClickListener or the Activity's onOptionsItemSelected handler. When the Intent is triggered Android will execute startActivity, passing in the specified Intent.

```
menuItem.setIntent(new Intent(this,
MyOtherActivity.click.
```

LOCATION-BASED SERVICES

Location-based services is an umbrella term used to describe the different technologies used to find a device's current location. The two main LBS elements are:

- Location Manager Provides hooks to the location-based services
- Location Providers Each of these represents a different location-finding technology used to determine the device's current location

Using the Location Manager, you can:

- Obtain your current location
- Track movement
- Set proximity alerts for detecting movement into and out of a specified area
- Find available Location Providers

CONFIGURING THE EMULATOR TO TEST LOCATION-BASED SERVICES

Location-based services are dependent on device hardware to find the current location. When you are developing and testing with the emulator your hardware is virtualized, and you're likely to stay in pretty much the same location.

To compensate, Android includes hooks that let you emulate Location Providers for testing location-based applications. In this section you'll learn how to mock the position of the supported GPS provider.

UPDATING LOCATIONS IN EMULATOR LOCATION PROVIDERS

Use the Location Controls available from the DDMS perspective in Eclipse to push location changes directly into the emulator's GPS Location Provider.

Using the Manual tab you can specify particular latitude/longitude pairs. Alternatively, the KML and GPX tabs let you load KML (Keyhole MarkupLanguage) and GPX (GPS Exchange Format) files, respectively. Once these are loaded you can jump to particular waypoints (locations) or play back each location sequentially.

Most GPS systems record track-files using GPX, while KML is used extensively

online to define geographic information. You can handwrite your own KML file or

generate one by using Google Earth to find directions between two locations.

All location changes applied using the DDMS Location Controls will be applied to the GPS receiver, which must be enabled and active.

Note that the GPS values returned by `getLastKnownLocation` will not change unless at least one application has requested location updates.

SELECTING A LOCATION PROVIDER

Depending on the device, there may be several technologies that Android can use to determine the current location. Each technology, or Location Provider, will offer different capabilities, including differences in power consumption, monetary cost, accuracy, and the ability to determine altitude, speed, or heading information. To get an

instance of a specific provider, call `getProvider`, passing in the name:

```
String providerName =
LocationManager.GPS_PROVIDER;

LocationProvider gpsProvider;

gpsProvider =
locationManager.getProvider(providerName);
```

This is generally useful only for determining the abilities of a particular provider. Most Location Manager methods require only a provider name to perform location based services.

Finding the Available Providers

The `LocationManager` class includes static string constants that return the provider name for the two most common Location Providers:

- `LocationManager.GPS_PROVIDER`
- `LocationManager.NETWORK_PROVIDER`

To get a list of names for all the providers available on the device, call `getProviders`, using a Boolean

to indicate if you want all, or only the enabled, providers to be returned:

```
boolean enabledOnly = true;

List<String>
providers=locationManager.getProviders(enabledOnly);
```

FINDING YOUR LOCATION

The purpose of location-based services is to find the physical location of the device.

Access to the location-based services is handled by the Location Manager system Service. To access the Location Manager, request an instance of the `LOCATION_SERVICE` using the `getSystemService` method, as shown in the following snippet:

```
String serviceString = Context.LOCATION_SERVICE;

LocationManager locationManager;

locationManager =
(LocationManager) getSystemService(serviceString);
```

Before you can use the Location Manager you need to add one or more `uses-permission` tags to your manifest to support access to the LBS hardware.

The following snippet shows the Fine and coarse permissions. An application that has been granted fine permission will have coarse permission granted implicitly.

```
<usespermissionandroid:name="android.permission.ACCESS_FINE_LOCATION">

<usespermissionandroid:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

The GPS provider requires fine permission, while the Network (Cell ID/Wi-Fi)

provider requires only coarse.

You can find the last location fix determined by a particular Location Provider using the `getLastKnownLocation` method, passing in the name of the Location Provider. The following example finds the last location fix taken by the GPS provider:

```
String provider = LocationManager.GPS_PROVIDER;
```

```
Location  
location=locationManager.getLastKnownLocation(provider);
```

The `Location` object returned includes all the position information available from the provider that supplied it. This can include latitude, longitude, bearing, altitude, speed, and the time the location fix was taken. All these properties are available via `Get` methods on the `Location` object. In some instances additional details will be included in the extras `Bundle`.

Getting Your Maps API Key

In order to use a Map View in your application you must first obtain an API key from the Android developer web site at <http://code.google.com/android/maps-api-signup.html>

.

Without an API key the Map View will not download the tiles used to display the map.

To obtain a key you need to specify the MD5 fingerprint of the certificate used to sign your application. Generally, you will sign your application using two certificates — a default debug certificate and a production certificate. The following sections explain how to obtain the MD5 fingerprint of each signing certificate used for your application.

Getting Your Development/Debugging MD5 Fingerprint

If you are using Eclipse with the ADT plug-in to debug your applications, they will be signed with the default debug certificate. To view map tiles while debugging you will need to obtain a Maps API key registered via the MD5 fingerprint of the debug certificate.

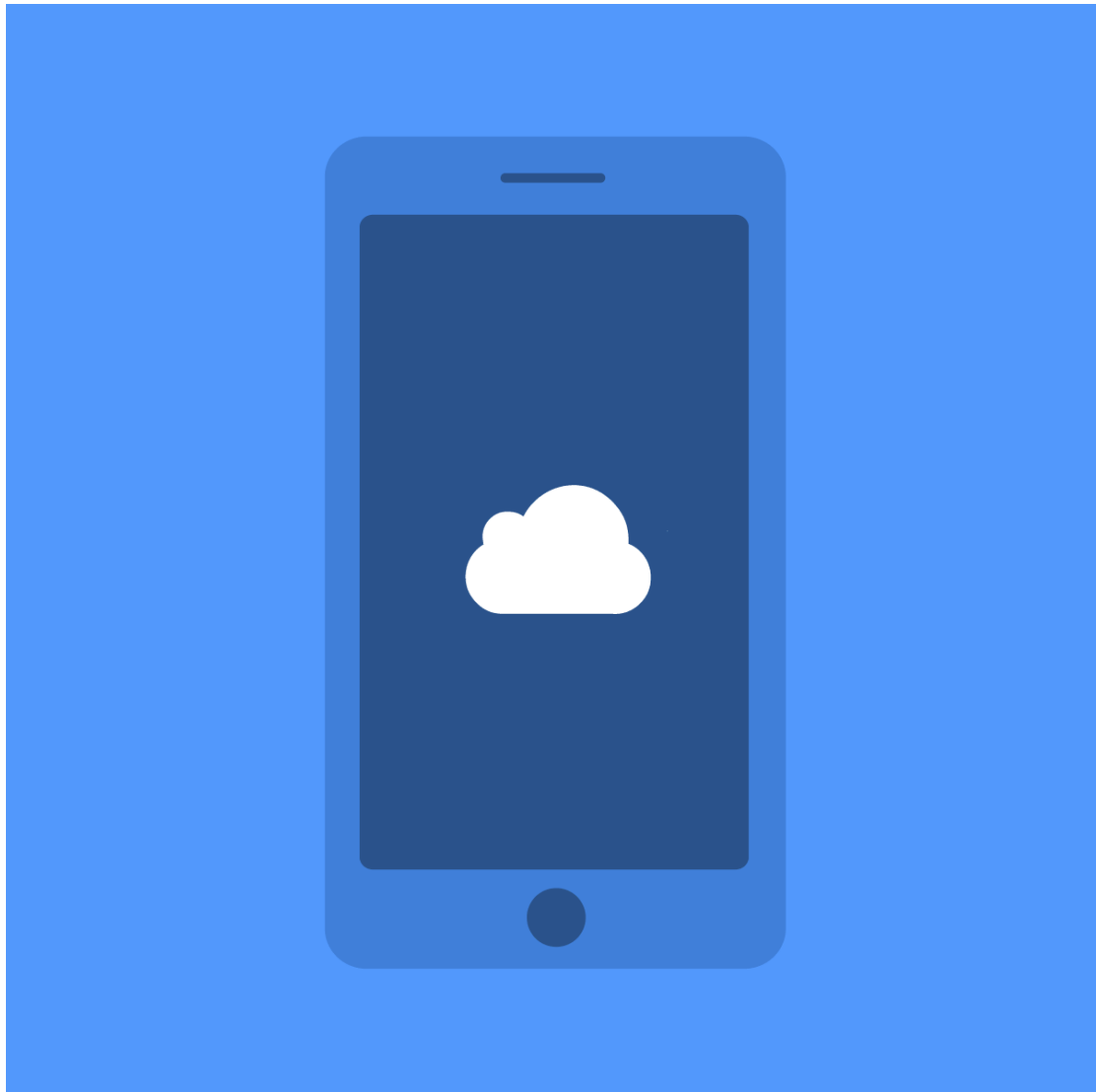
You can find the location of your keystore in the Default Debug Keystore textbox after selecting **Windows** ⇨ **Preferences** ⇨ **Android** ⇨ **build**. Typically the debug keystore is stored in the following platform-specific locations:

➤ Windows Vista\users\<username>\.android\debug.keystore

➤ Windows XP\Documents and Settings\<username>\.android\debug.keystore

➤ Linux or Mac ~/.android/debug.keystore





Getting Started

The Parse platform provides a complete backend solution for your mobile application. Our goal is to totally eliminate the need for writing server code or maintaining servers. If you're familiar with web frameworks like Ruby on Rails, we've taken many of the same principles and applied them to our platform. In particular, our SDK is ready to use out of the box with minimal configuration on your part.

On Parse, you create an App for each of your mobile applications. Each App has its own application id and client key that you apply to your SDK install. Your account on Parse can accommodate multiple Apps. This is useful even if you have one application, since you can deploy different versions for test and production.

Objects

The ParseObject

Storing data on Parse is built around the `ParseObject`. Each `ParseObject` contains key-value pairs of JSON-compatible data. This data is schemaless, which means that you don't need to specify ahead of time what keys exist on each `ParseObject`. You simply set whatever key-value pairs you want, and our backend will store it. For example, let's say you're tracking high scores for a game. A single `ParseObject` could contain:

```
score: 1337 , playerName: "Sean Plott" , cheatMode: false
```

Keys must be alphanumeric strings. Values can be strings, numbers, booleans, or even arrays and objects - anything that can be JSON-encoded.

Each `ParseObject` has a class name that you can use to distinguish different sorts of data. For example, we could call the high score object a `GameScore`. We recommend that you `NameYourClassesLikeThis` and `nameYourKeysLikeThis`, just to keep your code looking pretty.

Saving Objects

Let's say you want to save the `GameScore` described above to the Parse Cloud. The interface is similar to a `Map`, plus the `saveInBackground` method:

```
ParseObject gameScore = new ParseObject("GameScore");
```

```
gameScore.put("score", 1337);
```

```
gameScore.put("playerName", "Sean Plott");
```

```
gameScore.put("cheatMode", false);
```

```
gameScore.saveInBackground();
```

After this code runs, you will probably be wondering if anything really happened. To make sure the data was saved, you can look at the Data Browser in your app on Parse. You should see something like this:

```
objectId: "xWMyZ4YEGZ", score: 1337, playerName: "Sean Plott",  
cheatMode: false,  
createdAt:"2011-06-10T18:33:42Z", updatedAt:"2011-06-10T18:33:42Z"
```

Retrieving Objects from the Local Datastore

Storing an object is only useful if you can get it back out. To get the data for a specific object, you can use a `ParseQuery` just like you would while on the network, but using the `fromLocalDatastore` method to tell it where to get the data.

```
ParseQuery<ParseObject> query =
```

```
ParseQuery.getQuery("GameScore");
```

```
query.fromLocalDatastore();
```

```

query.getInBackground ( "xWMyZ4YEGZ" , new
    GetCallback<ParseObject> () {

        public void done(ParseObject object, ParseException e) {

            if (e == null ) {

                // object will be your game score

            } else {

                // something went wrong

            }

        }

    }

});

```

If you already have an instance of the object, you can instead use the `fetchFromLocalDatastoreInBackground` method.

```

ParseObject object =
    ParseObject.createWithoutData ( "GameScore" , "xWMyZ4YEGZ" );

object .fetchFromLocalDatastoreInBackground ( new
    GetCallback<ParseObject> () {

        public void done(ParseObject object, ParseException e) {

            if (e == null ) {


```

```
// object will be your game score
```

```
    } else {
```

```
// something went wrong
```

```
    }
```

```
    }
```

```
});
```

Relational Data

Objects can have relationships with other objects. To model this behavior, any `ParseObject` can be used as a value in other `ParseObject`s. Internally, the Parse framework will store the referred-to object in just one place, to maintain consistency.

For example, each `Comment` in a blogging app might correspond to one `Post`. To create a new `Post` with a single `Comment`, you could write:

```
// Create the post
```

```
ParseObject myPost = new ParseObject( "Post" );
```

```
myPost.put( "title", "I'm Hungry" );
```

```
myPost.put( "content", "Where should we go for lunch?" );
```

```
// Create the comment
```

```
ParseObject myComment = new ParseObject( "Comment" );
```

```
myComment.put( "content", "Let's do Sushirrito." );
```

```
// Add a relation between the Post and Comment
```

```
myComment.put ( "parent" , myPost );
```

```
// This will save both myPost and myComment
```

```
myComment.saveInBackground();
```

You can also link objects using just their `objectId`s like so:

```
// Add a relation between the Post with objectId "1zEcyElZ80" and the  
comment
```

```
myComment.put ("parent", ParseObject.createWithoutData("Post",  
"1zEcyElZ80"));
```

SUBCLASSING PARSEOBJECT

To create a `ParseObject` subclass:

1. Declare a subclass which extends `ParseObject`.
2. Add a `@ParseClassName` annotation. Its value should be the string you would pass into the `ParseObject` constructor, and makes all future class name references unnecessary.
3. Ensure that your subclass has a public default (i.e. zero-argument) constructor. You must not modify any `ParseObject` fields in this constructor.
4. Call `ParseObject.registerSubclass(YourClass.class)` in your `Application` constructor before calling `Parse.initialize()`. The following code successfully implements and registers the `Armor` subclass of `ParseObject`:

```
5. // Armor.java  
6. import com.parse.ParseObject;  
7. import com.parse.ParseClassName;  
8.
```

```

9. @ParseClassName("Armor")
10.     public class Armor extends ParseObject {
11.     }
12.
13.     // App.java
14.     import com.parse.Parse;
15.     import android.app.Application;
16.
17.     public class App extends Application {
18.         @Override
19.         public void onCreate() {
20.             super.onCreate();
21.
22.             ParseObject.registerSubclass(Armor.class);
23.             Parse.initialize(this, PARSE_APPLICATION_ID,
24.                 PARSE_CLIENT_KEY);
25.         }
26.     }

```

Queries

We've already seen how a `ParseQuery` with `getInBackground` can retrieve a single `ParseObject` from Parse. There are many other ways to retrieve data with `ParseQuery` - you can retrieve many objects at once, put conditions on the objects you wish to retrieve, cache queries automatically to avoid writing that code yourself, and more.

Basic Queries

In many cases, `getInBackground` isn't powerful enough to specify which objects you want to retrieve. The `ParseQuery` offers different ways to retrieve a list of objects rather than just a single object.

The general pattern is to create a `ParseQuery`, put conditions on it, and then retrieve a `List` of matching `ParseObjects` using the `findInBackground` method with a `FindCallback`. For example, to retrieve scores with a particular `playerName`, use the `whereEqualTo` method to constrain the value for a key:

```

ParseQuery<ParseObject> query = ParseQuery.getQuery("GameScore");

query.whereEqualTo("playerName", "Dan Stemkoski");

query.findInBackground(new FindCallback<ParseObject>() {

    public void done(List<ParseObject> scoreList, ParseException e) {

        if (e == null) {

            Log.d("score", "Retrieved " + scoreList.size() + "
scores");

        } else {

            Log.d("score", "Error: " + e.getMessage());

        }

    }

});

```

`findInBackground` works similarly to `getInBackground` in that it assures the network request is done on a background thread, and runs its callback in the main thread.

Query Constraints

There are several ways to put constraints on the objects found by a `ParseQuery`. You can filter out objects with a particular key-value pair with `whereNotEqualTo`:

```
query.whereNotEqualTo("playerName", "Michael Yabuti");
```

You can give multiple constraints, and objects will only be in the results if they match all of the constraints. In other words, it's like an AND of constraints.

```
query.whereNotEqualTo("playerName", "Michael Yabuti");
```

```
query.whereGreaterThan( "playerAge" , 18 );
```

you can limit the number of results with `setLimit`. By default, results are limited to 100, but anything from 1 to 1000 is a valid limit:

```
query.setLimit( 10 ); // limit to at most 10 results
```

Queries on Array Values

If a key contains an array value, you can search for objects where the key's array value contains 2 by:

```
// Find objects where the array in arrayKey contains the number 2.
```

```
query.whereEqualTo( "arrayKey" , 2 );
```

You can also search for objects where the key's array value contains each of the values 2, 3, and 4 with the following:

```
// Find objects where the array in arrayKey contains all of the numbers 2, 3, and 4.
```

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

```
numbers.add(2);
```

```
numbers.add(3);
```

```
numbers.add(4);
```

```
query.whereContainsAll( "arrayKey", numbers);
```


Queries on String Values

Use `whereStartsWith` to restrict to string values that start with a particular string. Similar to a MySQL LIKE operator, this is indexed so it is efficient for large datasets:

```
// Finds barbecue sauces that start with 'Big Daddy's'.

ParseQuery<ParseObject> query = ParseQuery.getQuery("BarbecueSauce");

query.whereStartsWith("name", "Big Daddy's");
```

The above example will match any `BarbecueSauce` objects where the value in the "name" String key starts with "Big Daddy's". For example, both "Big Daddy's" and "Big Daddy's BBQ" will match, but "big daddy's" or "BBQ Sauce: Big Daddy's" will not.

Queries that have regular expression constraints are very expensive. Refer to the [Performance Guide](#) for more details.

Relational Queries

There are several ways to issue queries for relational data. If you want to retrieve objects where a field matches a particular `ParseObject`, you can use `whereEqualTo` just like for other data types. For example, if each `Comment` has a `Post` object in its `post` field, you can fetch comments for a particular `Post`:

```
// Assume ParseObject myPost was previously created.

ParseQuery<ParseObject> query = ParseQuery.getQuery("Comment");

query.whereEqualTo("post", myPost);

query.findInBackground(new FindCallback<ParseObject>() {

    public void done(List<ParseObject> commentList, ParseException e) {

        // commentList now has the comments for myPost

    }

})
```

```
});
```

If you want to retrieve objects where a field contains a `ParseObject` that matches a different query, you can use `whereMatchesQuery`. Note that the default limit of 100 and maximum limit of 1000 apply to the inner query as well, so with large data sets you may need to construct queries carefully to get the desired behavior. In order to find comments for posts containing images, you can do:

```
ParseQuery<ParseObject> innerQuery =
```

```
ParseQuery.getQuery( "Post" );
```

```
innerQuery.whereExists( "image" );
```

```
ParseQuery<ParseObject> query =
```

```
ParseQuery.getQuery( "Comment" );
```

```
query.whereMatchesQuery( "post" , innerQuery);
```

```
query.findInBackground( new FindCallback<ParseObject>() {
```

```
public void done(List<ParseObject> commentList, ParseException e) {
```

```
// comments now contains the comments for posts with images.
```

```
}});
```

Caching Queries

It's often useful to cache the result of a query on a device. This lets you show data when the user's device is offline, or when the app has just started and network requests have not yet had time to complete. The easiest way to do this is with the local datastore. When you pin objects, you can attach a label to the pin, which lets you manage a group of objects together. For example, to cache the results of the query above, you can call `pinAllInBackground` and give it a label.

```
final String TOP_SCORES_LABEL = "topScores" ;
```

```
// Query for the latest objects from Parse.
```

```
query.findInBackground( new FindCallback<ParseObject>() {
```

```
    public void done(final List<ParseObject> scoreList, ParseException e) {
```

```
        if (e != null) {
```

```
            // There was an error or the network wasn't available.
```

```
            return ;
```

```
        }
```

```
// Release any objects previously pinned for this query.
```

```
ParseObject.unpinAllInBackground(TOP_SCORES_LABEL, scoreList,
```

```
new DeleteCallback() {
```

```
    public void done(ParseException e) {
```

```

        if (e != null) {

            // There was some error.

            return ;

        }

```

```

        // Add the latest results for this query to the cache.

        ParseObject.pinAllInBackground(TOP_SCORES_LABEL,
scoreList);

    }

    });

}

});

```

Parse provides several different cache policies:

- **IGNORE_CACHE**: The query does not load from the cache or save results to the cache. **IGNORE_CACHE** is the default cache policy.
- **CACHE_ONLY**: The query only loads from the cache, ignoring the network. If there are no cached results, that causes a **ParseException**.
- **NETWORK_ONLY**: The query does not load from the cache, but it will save results to the cache.

- `CACHE_ELSE_NETWORK`: The query first tries to load from the cache, but if that fails, it loads results from the network. If neither cache nor network succeed, there is a `ParseException`.
- `NETWORK_ELSE_CACHE`: The query first tries to load from the network, but if that fails, it loads results from the cache. If neither network nor cache succeed, there is a `ParseException`.
- `CACHE_THEN_NETWORK`: The query first loads from the cache, then loads from the network. In this case, the `FindCallback` will actually be called twice - first with the cached results, then with the network results. This cache policy can only be used asynchronously with `findInBackground`.

If you need to control the cache's behavior, you can use methods provided in `ParseQuery` to interact with the cache. You can do the following operations on the cache:

- Check to see if there is a cached result for the query with:

```
boolean isInCache = query.hasCachedResult();
```

- Remove any cached results for a query with:

```
query.clearCachedResult();
```

- Remove cached results for all queries with:

```
ParseQuery.clearAllCachedResults();
```

Query caching also works with `ParseQuery` helpers including `getFirst()` and `getInBackground()`.

Counting Objects

Caveat: Count queries are rate limited to a maximum of 160 requests per minute. They can also return inaccurate results for classes with

more than 1,000 objects. Thus, it is preferable to architect your application to avoid this sort of count operation (by using counters, for example.)

If you just need to count how many objects match a query, but you do not need to retrieve all the objects that match, you can use `count` instead of `find`. For example, to count how many games have been played by a particular player:

```
ParseQuery<ParseObject> query =
    ParseQuery.getQuery( "GameScore" );

query.whereEqualTo( "playerName" , "Sean Plott" );

query.countInBackground( new CountCallback() {

    public void done(int count, ParseException e) {

        if (e == null) {

            // The count request succeeded. Log the count

            Log.d( "score" , "Sean has played " + count + " games" );

        } else {

            // The request failed

        }

    }

} );
```

If you want to block the calling thread, you can also use the synchronous `query.count()` method.

Compound Queries

If you want to find objects that match one of several queries, you can use `ParseQuery.or` method to construct a query that is an or of the queries passed in. For instance if you want to find players who either have a lot of wins or a few wins, you can do:

```
ParseQuery<ParseObject> lotsOfWins =
```

```
ParseQuery.getQuery( "Player" );
```

```
lotsOfWins.whereGreaterThan( 150 );
```

```
ParseQuery<ParseObject> fewWins =
```

```
ParseQuery.getQuery( "Player" );
```

```
fewWins.whereLessThan( 5 );
```

```
List <ParseQuery<ParseObject>> queries = new
```

```
ArrayList<ParseQuery<ParseObject>>();
```

```
queries.add(lotsOfWins);
```

```
queries.add(fewWins);
```

```

ParseQuery<ParseObject> mainQuery = ParseQuery.or (queries);

mainQuery.findInBackground(new FindCallback<ParseObject>() {

    public void done(List <ParseObject> results, ParseException
e) {

    // results has the list of players that win a lot or haven't won much.

    }

});

```

You can add additional constraints to the newly created `ParseQuery` that act as an 'and' operator.

Note that we do not, however, support `GeoPoint` or non-filtering constraints

(e.g. `whereNear`, `withinGeoBox`, `setLimit`, `skip`, `orderBy...`, `include`) in the subqueries of the compound query.

Users

At the core of many apps, there is a notion of user accounts that lets users access their information in a secure manner. We provide a specialized user class called `ParseUser` that automatically handles much of the functionality required for user account management.

With this class, you'll be able to add user account functionality in your app.

`ParseUser` is a subclass of the `ParseObject`, and has all the same features, such as flexible schema, automatic persistence, and a key value interface. All the methods that are on `ParseObject` also exist

in `ParseUser`. The difference is that `ParseUser` has some special additions specific to user accounts.

Properties

`ParseUser` has several properties that set it apart from `ParseObject`:

- username: The username for the user (required).
- password: The password for the user (required on signup).
- email: The email address for the user (optional).

We'll go through each of these in detail as we run through the various use cases for users. Keep in mind that if you set `username` and `email` using the setters, you do not need to set it using the `put` method.

Signing Up

The first thing your app will do is probably ask the user to sign up. The following code illustrates a typical sign up:

```
ParseUser user = new ParseUser();
```

```
user.setUsername("my name");
```

```
user.setPassword("my pass");
```

```
user.setEmail("email@example.com");
```

// other fields can be set just like with ParseObject

```
user.put("phone", "650-253-0000");
```

```

user.signUpInBackground( new SignUpCallback() {

    public void done(ParseException e) {

        if (e == null) {

            // Hooray! Let them use the app now.

        } else {

            // Sign up didn't succeed. Look at the ParseException
            // to figure out what went wrong

        }

    }

});

```

This call will asynchronously create a new user in your Parse App. Before it does this, it checks to make sure that both the username and email are unique. Also, it securely hashes the password in the cloud using bcrypt. We never store passwords in plaintext, nor will we ever transmit passwords back to the client in plaintext.

Note that we used the `signUpInBackground` method, not the `saveInBackground` method. New `ParseUsers` should always be created using the `signUpInBackground` (or `signUp`) method. Subsequent updates to a user can be done by calling `save`.

The `signUpInBackground` method comes in various flavors, with the ability to pass back errors, and also synchronous versions. As usual, we highly recommend using the asynchronous versions when

possible, so as not to block the UI in your app. You can read more about these specific methods in our [API docs](#).

If a signup isn't successful, you should read the error object that is returned. The most likely case is that the username or email has already been taken by another user. You should clearly communicate this to your users, and ask them try a different username.

You are free to use an email address as the username. Simply ask your users to enter their email, but fill it in the username property — `ParseUser` will work as normal. We'll go over how this is handled in the reset password section.

Logging In

Of course, after you allow users to sign up, you need be able to let them log in to their account in the future. To do this, you can use the class method `loginInBackground`.

```
ParseUser.loginInBackground( "Jerry" , "showmethemoney" , new  
  
    LoginCallback() {  
  
        public void done(ParseUser user, ParseException e) {  
  
            if (user != null ) {  
  
                // Hooray! The user is logged in.  
  
            } else {  
  
                // Signup failed. Look at the ParseException to see what happened.  
  
            }  
  
        }  
    }  
}
```

```
});
```

Verifying Emails

Enabling email verification in an application's settings allows the application to reserve part of its experience for users with confirmed email addresses. Email verification adds the `emailVerified` key to the `ParseUser` object. When a `ParseUser`'s `email` is set or modified, `emailVerified` is set to `false`. Parse then emails the user a link which will set `emailVerified` to `true`.

There are three `emailVerified` states to consider:

1. `true` - the user confirmed his or her email address by clicking on the link Parse emailed them. `ParseUsers` can never have a `true` value when the user account is first created.
2. `false` - at the time the `ParseUser` object was last fetched, the user had not confirmed his or her email address. If `emailVerified` is `false`, consider calling `fetch()` on the `ParseUser`.
3. *missing* - the `ParseUser` was created when email verification was off or the `ParseUser` does not have an `email`.

Current User

It would be bothersome if the user had to log in every time they open your app. You can avoid this by using the cached `currentUser` object. Whenever you use any signup or login methods, the user is cached on disk. You can treat this cache as a session, and automatically assume the user is logged in:

```
ParseUser currentUser = ParseUser.getCurrentUser();
```

```
if (currentUser != null) {
```

```
// do stuff with the user
```

```
} else {
```

```
// show the signup or login screen
```

```
}
```

You can clear the current user by logging them out:

```
ParseUser.logout();
```

```
ParseUser currentUser = ParseUser.getCurrentUser(); // this
will now be null
```

Anonymous Users

Being able to associate data and objects with individual users is highly valuable, but sometimes you want to be able to do this without forcing a user to specify a username and password.

An anonymous user is a user that can be created without a username and password but still has all of the same capabilities as any other `ParseUser`. After logging out, an anonymous user is abandoned, and its data is no longer accessible.

You can create an anonymous user using `ParseAnonymousUtils`:

```
ParseAnonymousUtils.login(new LoginCallback() {
```

```
@Override
```

```
public void done(ParseUser user, ParseException e) {
```

```
if (e != null) {
```

```
Log.d("MyApp", "Anonymous login failed.");
```

```
} else {
```

```
Log.d("MyApp", "Anonymous user logged in.");
```

```
}
```

```
}
```

```
});
```

You can convert an anonymous user into a regular user by setting the username and password, then calling `signUp()`, or by logging in or linking with a service like [Facebook](#) or [Twitter](#). The converted user will retain all of its data. To determine whether the current user is an anonymous user, you can check `ParseAnonymousUtils.isLinked()`:

```
if (ParseAnonymousUtils.isLinked(ParseUser.getCurrentUser())) {
```

```
    enable SignUpButton();
```

```
} else {
```

```
    enable LogOutButton();
```

```
}
```

Setting the Current User

If you've created your own authentication routines, or otherwise logged in a user on the server side, you can now pass the session

token to the client and use the `become` method. This method will ensure the session token is valid before setting the current user.

```
ParseUser.becomeInBackground( "session-token-here" , new  
  
LoginCallback() {  
  
    public void done(ParseUser user, ParseException e) {  
  
        if (user != null ) {  
  
            // The current user is now set to user.  
  
        } else {  
  
            // The token could not be validated.  
  
        }  
  
    }  
  
});
```

Security For User Objects

The `ParseUser` class is secured by default. Data stored in a `ParseUser` can only be modified by that user. By default, the data can still be read by any client. Thus, some `ParseUser` objects are authenticated and can be modified, whereas others are read-only.

Specifically, you are not able to invoke any of the `save` or `delete` type methods unless the `ParseUser` was obtained using an authenticated method, like `login` or `signUp`. This ensures that only the user can alter their own data.

The following illustrates this security policy:

```

ParseUser user = ParseUser.logIn("my_username",
"my_password");

user.setUsername("my_new_username"); // attempt to change username

user.saveInBackground(); // This succeeds, since the user was authenticated on the
device

// Get the user from a non-authenticated manner

ParseQuery<ParseUser> query = ParseUser.getQuery();

query.getInBackground(user.getObjectId(), new
GetCallback<ParseUser>() {

    public void done(ParseUser object, ParseException e) {

        object.setUsername("another_username");

        // This will throw an exception, since the ParseUser is not authenticated

        object.saveInBackground();

    }

});

```


Security for Other Objects

The same security model that applies to the `ParseUser` can be applied to other objects. For any object, you can specify which users are allowed to read the object, and which users are allowed to modify an object. To support this type of security, each object has an [access control list](#), implemented by the `ParseACL` class.

The simplest way to use a `ParseACL` is to specify that an object may only be read or written by a single user. To create such an object, there must first be a logged in `ParseUser`. Then, `new ParseACL(user)` generates a `ParseACL` that limits access to that user. An object's ACL is updated when the object is saved, like any other property. Thus, to create a private note that can only be accessed by the current user:

```
ParseObject privateNote = new ParseObject( "Note" );
```

```
privateNote.put( "content" , "This note is private!" );
```

```
privateNote.setACL( new ParseACL( ParseUser.getCurrentUser() ) );
```

```
privateNote.saveInBackground();
```

This note will then only be accessible to the current user, although it will be accessible to any device where that user is signed in. This functionality is useful for applications where you want to enable access to user data across multiple devices, like a personal todo list.

Permissions can also be granted on a per-user basis. You can add permissions individually to a `ParseACL` using `setReadAccess` and `setWriteAccess`. For example, let's say you have a message that will be sent to a group of several users, where each of them have the rights to read and delete that message:

```
ParseObject groupMessage = new ParseObject( "Message" );
```

```
ParseACL groupACL = new ParseACL();
```

// userList is an Iterable<ParseUser> with the users we are sending this message to.

```
for (ParseUser user : userList) {
```

```
    groupACL.setReadAccess(user, true);
```

```
    groupACL.setWriteAccess(user, true);
```

```
}
```

```
groupMessage.setACL(groupACL);
```

```
groupMessage.saveInBackground();
```

Resetting Passwords

It's a fact that as soon as you introduce passwords into a system, users will forget them. In such cases, our library provides a way to let them securely reset their password.

To kick off the password reset flow, ask the user for their email address, and call:

```
ParseUser.requestPasswordResetInBackground("myemail@example.com"
```

```
, new RequestPasswordResetCallback() {
```

```

public void done(ParseException e) {

    if (e == null) {

        // An email was successfully sent with reset instructions.

    } else {

        // Something went wrong. Look at the ParseException to see what's up.

    }
}

```

This will attempt to match the given email with the user's email or username field, and will send them a password reset email. By doing this, you can opt to have users use their email as their username, or you can collect it separately and store it in the email field.

The flow for password reset is as follows:

1. User requests that their password be reset by typing in their email.
2. Parse sends an email to their address, with a special password reset link.
3. User clicks on the reset link, and is directed to a special Parse page that will allow them type in a new password.
4. User types in a new password. Their password has now been reset to a value they specify.

Note that the messaging in this flow will reference your app by the name that you specified when you created this app on Parse.

Facebook Users

Parse provides an easy way to integrate Facebook with your application. The Facebook SDK can be used with our SDK, and is

integrated with the `ParseUser` class to make linking your users to their Facebook identities easy.

Using our Facebook integration, you can associate an authenticated Facebook user with a `ParseUser`. With just a few lines of code, you'll be able to provide a "Log in with Facebook" option in your app, and be able to save their data to Parse.

Note: Parse is compatible with both Facebook SDK 3.x and 4.x for Android. These instructions are for Facebook SDK 4.x.

SETUP

To start using Facebook with Parse, you need to:

1. [Set up a Facebook app](#), if you haven't already.
2. Add your application's Facebook Application ID on your Parse application's settings page.
3. Follow Facebook's instructions for [getting started with the Facebook SDK](#) to create an app linked to the Facebook SDK. Once you get to Step 6, stop after linking the Facebook SDK project and configuring the Facebook app ID. You can use our guide to attach your Parse users to Facebook accounts when logging in.
4. Add `ParseFacebookUtilsV4.jar` to your Android project.
5. Add the following where you initialize the Parse SDK in your `Application.onCreate()`:

```
ParseFacebookUtils.initialize(context);
```

FACEBOOK SDK AND PARSE

The Facebook Android SDK provides a number of helper classes for interacting with Facebook's API. Generally, you will use the `GraphRequest` class to interact with Facebook on behalf of your logged-in user. [You can read more about the Facebook SDK here.](#)

To access the user's `AccessToken` you can simply call `AccessToken.getCurrentAccessToken()` to access the `AccessToken` instance, which can then be passed to `GraphRequest`s.

Files

The ParseFile

`ParseFile` lets you store application files in the cloud that would otherwise be too large or cumbersome to fit into a regular `ParseObject`. The most common use case is storing images but you can also use it for documents, videos, music, and any other binary data (up to 10 megabytes).

Getting started with `ParseFile` is easy. First, you'll need to have the data in `byte[]` form and then create a `ParseFile` with it. In this example, we'll just use a string:

```
byte [] data = "Working at Parse is great!".getBytes();
```

```
ParseFile file = new ParseFile("resume.txt", data);
```

Notice in this example that we give the file a name of `resume.txt`. There's two things to note here:

- You don't need to worry about filename collisions. Each upload gets a unique identifier so there's no problem with uploading multiple files named `resume.txt`.
- It's important that you give a name to the file that has a file extension. This lets Parse figure out the file type and handle it accordingly. So, if you're storing PNG images, make sure your filename ends with `.png`.

Next you'll want to save the file up to the cloud. As with `ParseObject`, there are many variants of the `save` method you can use depending on what sort of callback and error handling suits you.

```
file.saveInBackground();
```

Finally, after the save completes, you can associate a `ParseFile` onto a `ParseObject` just like any other piece of data:

```
ParseObject jobApplication = new  
ParseObject("JobApplication");
```

```
jobApplication.put( "applicantName" , "Joe Smith" );
```

```
jobApplication.put( "applicantResumeFile" , file);
```

```
jobApplication.saveInBackground();
```

Retrieving it back involves calling one of the `getData` variants on the `ParseObject`. Here we retrieve the resume file off another `JobApplication` object:

```
ParseFile applicantResume =
```

```
(ParseFile) anotherApplication.get( "applicantResumeFile" );
```

```
applicantResume.getDataInBackground( new GetDataCallback() {
```

```
    public void done(byte[] data, ParseException e) {
```

```
        if (e == null) {
```

```
            // data has the bytes for the resume
```

```
        } else {
```

```
            // something went wrong
```

```
        }
```

```
    }
```

```
});
```

Just like on `ParseObject`, you will most likely want to use the background version of `getData`.

Relations

There are three kinds of relationships. One-to-one relationships enable one object to be associated with another object. One-to-many relationships enable one object to have many related objects. Finally, many-to-many relationships enable complex relationships among many objects.

There are four ways to build relationships in Parse:

One-to-Many

When you're thinking about one-to-many relationships and whether to implement Pointers or Arrays, there are several factors to consider. First, how many objects are involved in this relationship? If the "many" side of the relationship could contain a very large number (greater than 100 or so) of objects, then you have to use Pointers. If the number of objects is small (fewer than 100 or so), then Arrays may be more convenient, especially if you typically need to get all of the related objects (the "many" in the "one-to-many relationship") at the same time as the parent object.

Many-to-Many

Now let's tackle many-to-many relationships. Suppose we had a book reading app and we wanted to model `Book` objects and `Author` objects. As we know, a given author can write many books, and a given book can have multiple authors. This is a many-to-many relationship scenario where you have to choose between Arrays, Parse Relations, or creating your own Join Table.

The decision point here is whether you want to attach any metadata to the relationship between two entities. If you don't, Parse Relation

or using Arrays are going to be the easiest alternatives. In general, using arrays will lead to higher performance and require fewer queries. If either side of the many-to-many relationship could lead to an array with more than 100 or so objects, then, for the same reason Pointers were better for one-to-many relationships, Parse Relation or Join Tables will be better alternatives.

On the other hand, if you want to attach metadata to the relationship, then create a separate table (the "Join Table") to house both ends of the relationship. Remember, this is information **about the relationship**, not about the objects on either side of the relationship. Some examples of metadata you may be interested in, which would necessitate a Join Table approach, include

Data

Valid Data Types

We've designed the Parse SDKs so that you typically don't need to worry about how data is saved while using the client SDKs. Simply add data to the `ParseObject`, and it'll be saved correctly.

Nevertheless, there are some cases where it's useful to be aware of how data is stored on the Parse platform.

Internally, Parse stores data as JSON, so any datatype that can be converted to JSON can be stored on Parse. The framework can also handle `Date` and `File` types. Overall, the following types are allowed for each field in your object:

String

Number

Boolean

Array

ObjecDate

`ParseFile`

`ParseObject`

`ParseRelation`

Null

The type `Object` simply denotes that each value can be composed of nested objects that are JSON-encodable. Keys including the characters `$` or `.`, along with the key `__type` key, are reserved for the framework to handle additional types, so don't use those yourself.

We do not recommend storing large pieces of binary data like images or documents in a `ParseObject`. `ParseObjects` should not exceed 128 kilobytes in size. We recommend you use `Files` to store images, documents, and other types of files. You can do so by instantiating a `ParseFile` object and setting it on a field.

Security

As your app development progresses, you will want to use Parse's security features in order to safeguard data. This document explains the ways in which you can secure your apps.

If your app is compromised, it's not only you as the developer who suffers, but potentially the users of your app as well. Continue reading for our suggestions for sensible defaults and precautions to take before releasing your app into the wild.

Client vs. Server

When an app first connects to Parse, it identifies itself with an Application ID and a Client key (or REST Key, or .NET Key, or JavaScript Key, depending on which platform you're using). These are not secret and by themselves they do not secure an app. These keys are shipped as a part of your app, and anyone can decompile your app or proxy network traffic from their device to find your client

key. This exploit is even easier with JavaScript — one can simply "view source" in the browser and immediately find your client key.

This is why Parse has many other security features to help you secure your data. The client key is given out to your users, so anything that can be done with just the client key is doable by the general public, even malicious hackers.


Class-Level Permissions

The second level of security is at the schema and data level. Enforcing security measures at this level will restrict how and when client applications can access and create data on Parse. When you first begin developing your Parse application, all of the defaults are set so that you can be a more productive developer. For example:


- A client application can create new classes on Parse
- A client application can add fields to classes
- A client application can modify or query for objects on Parse

RESTRICTING CLASS CREATION

As a start, you can configure your application so that clients cannot create new classes on Parse. This is done from the Settings tab on the Data Browser. Scroll down to the **App Permissions** section and turn off **Allow client class creation**. Once enabled, classes may only be created from the Data Browser. This will prevent attackers from filling your database with unlimited, arbitrary new classes.

 App Permissions

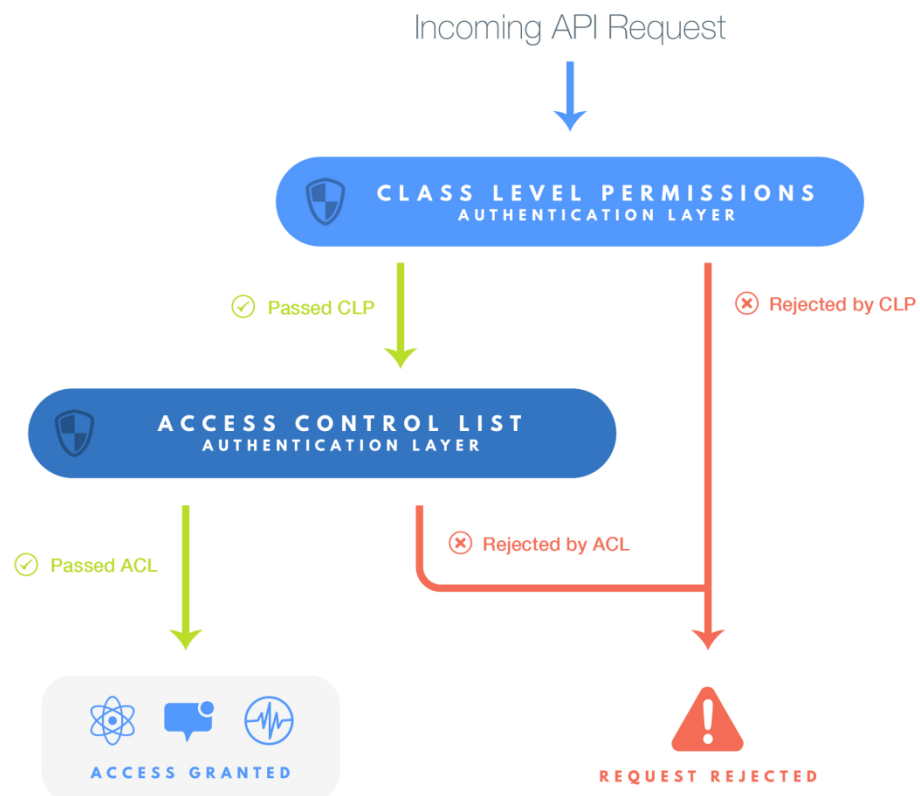
You can set application-wide permissions below.

Allow client class creation 

☐ OFF

CLP AND ACL INTERACTION

Class-Level Permissions (CLPs) and Access Control Lists (ACLs) are both powerful tools for securing your app, but they don't always interact exactly how you might expect. They actually represent two separate layers of security that each request has to pass through to return the correct information or make the intended change. These layers, one at the class level, and one at the object level, are shown below. A request must pass through BOTH layers of checks in order to be authorized. Note that despite acting similarly to ACLs, [Pointer Permissions](#) are a type of class level permission, so a request must pass the pointer permission check in order to pass the CLP check.



SECURITY EDGE CASES

There are some special classes in Parse that don't follow all of the same security rules as every other class. Not all classes follow [Class-Level Permissions \(CLPs\)](#) or [Access Control Lists \(ACLs\)](#) exactly how they are defined, and here those exceptions are documented. Here "normal behavior" refers to CLPs and ACLs working normally, while any other special behaviors are described in the footnotes.

	<u>User</u>	<u>Installation</u>
Get	normal behaviour [1, 2, 3]	ignores CLP, but not ACL

	<code>_User</code>	<code>_Installation</code>
Find	normal behavior [3]	master key only [6]
Create	normal behavior [4]	ignores CLP
Update	normal behavior [5]	ignores CLP, but not ACL [7]
Delete	normal behavior [5]	master key only [7]
Add Field	normal behavior	normal behavior

User Interface

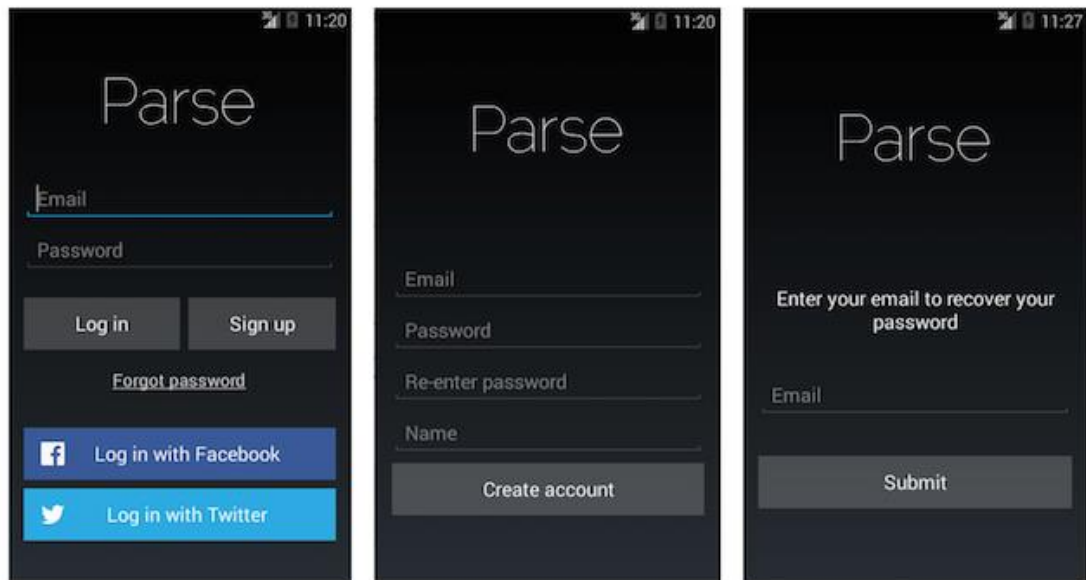
At the end of the day, users of your application will be interacting with Android UI components. We provide several UI widgets to make working with Parse data easier.

ParseLoginUI

If you are using Parse to manage users in your mobile app, you are already familiar with the `ParseUser` class. At some point in your app, you might want to present a screen to log in your `ParseUser`. Parse provides an open-source [ParseLoginUI](#) library project that does exactly this. Please note ParseLoginUI is not included with the Parse Android SDK; you need to import this library project from our Git repository into your Android Studio project. Here is an [example](#) for how to import a library project.

This library project contains an Android login activity that is ultra-customizable and easy to integrate with your app. You can configure

the look and feel of the login screens by either specifying XML configurations or constructing an Intent in code. In this guide, we first provide several ways to integrate with the login library. Then, we describe in detail how to customize the login screens.



Cloud Code

What is Cloud Code?

Parse's vision is to let developers build any mobile app without dealing with servers. For complex apps, sometimes you just need a bit of logic that isn't running on a mobile device. Cloud Code makes this possible.

Cloud Code is easy to use because it's built on the same JavaScript SDK that powers thousands of apps. The only difference is that this code runs in the Parse Cloud rather than running on a mobile device. When you update your Cloud Code, it becomes available to all mobile environments instantly. You don't have to wait for a new release of your application. This lets you change app behavior on the fly and add new features faster.

Even if you're only familiar with mobile development, we hope you'll find Cloud Code straightforward and easy to use.

Getting Started

On the computer you use for development, you will need to install Parse's command line tool. This will let you manage your code in the Parse Cloud. You can learn more about the features of the command line tool in the [Command Line Tool guide](#).

INSTALLING THE COMMAND LINE TOOL

MAC/LINUX

In Mac OS and Linux/Unix environments, you can get the parse tool by running this command:

```
curl -s https://www.parse.com/downloads/cloud_code/installer.sh  
| sudo /bin/bash
```

Performance

As your app scales, you will want to ensure that it performs well under increased load and usage. There are parts of optimizing performance that Parse takes care of but there are some things you can do. This document provides guidelines on how you can optimize your app's performance. While you can use Parse for quick prototyping and not worry about performance, you will want to keep our performance guidelines in mind when you're initially designing your app. We strongly advise that you make sure you've followed all suggestions before releasing your app.

Parse provides services out of the box to help your app scale automatically. On top of our MongoDB datastore, we have built an API layer that seamlessly integrates with our client-side SDKs. Our cloud infrastructure uses online learning algorithms to automatically rewrite inefficient queries and generate database indexes based on your app's realtime query stream.

In addition to what Parse provides, you can improve your app's performance by looking at the following:

Writing efficient queries.

Writing restrictive queries.

Using client-side caching.

Using Cloud Code.

Avoiding count queries.

Using efficient search techniques.

Keep in mind that not all suggestions may apply to your app. Let's look into each one of these in more detail.

Problem & solution

Road accidents are increasing day by day as a result

of poor implementation of the roads, as is happening on the ground in the way of Hurghada Red Sea, which washed away parts of it as a result of the floods, in addition to the lack of commitment by motorists as quickly as planned before last week's event in the bus accident which killed Suez road more than 24 people and wounding more than a dozen, Ring Road and Cairo Alascnrah agricultural as well as ways to level. Frequent scenario does not stop until travel has become risky allow the repercussions and accumulations of neglect of roads and other factors crises trip.

There are conflicting number of road accidents in Egypt numbers but with the growing importance of the system passenger and cargo transport on the roads and growing traffic by which led to the growing problem of road accidents, a report issued by the World Health Organization on the current situation of health and safety on the roads in 2012 confirmed that the rates of road accidents in Egypt higher than global averages, where up to about 12 per day clash entailed the death of about 12,300 people and injuring about 154 thousand other in 2009

The Central Agency for Public Mobilization and Statistics for 2011 data, she believes that Egypt is the highest state in the percentage of injuries and fatalities resulting from road accidents

Bringing the number of accidents caused by those deaths that year 7115 at a rate of 8.8 cases per 100 thousand people, equivalent to 19.5 deaths per day, while the number of injured reached 27 479 case equivalent to 7.3 infected daily

While the report of injuries center at the Ministry of Health in 2012 showed that pedestrians and cyclists account for 33.14 of the total injured in road accidents, pedestrian 10.65 and 19.90 riders motorcycles, riders and 2.59 ordinary bicycles

Contrary to popular belief most accidents occur in the less congested roads, especially on holidays and the combination, and the interpretation of it that the lack of road congestion encourages motorists to increase the speed and reckless driving, as well as population density few governorates such as South Sinai

And it reduced the Interior Ministry of the stated percentage from some quarters about road accidents where Colonel Dr. Ayman hyena General Administration of passage stressed that Egypt is not the first globally in road accidents, but we in the middle and this does not mean that we are satisfied with our situation, but we strive to be in the ranks of the few developed countries Alhawwat

He said that the reason for the increasing incidents is the evolution of cars and roads and increase with the increase in congestion on the roads is increasing the size and number of accidents, adding that the road has doubled and the number of vehicles with the growing population distribution makes traffic densities and confusion among pedestrians and passengers in addition to the problem of legislation. He explained that he die 3 members every second in the world with a total 1800 people a day and annual losses 518 billion dollars worldwide, while the middle-income countries are prone to the problem as the die .due to road accidents in the age range from 15 to 44 years

He stressed that the European countries and America of deaths on the roads is much less than the Middle East and North African countries and what happens from road accidents in Egypt shows that the number of deaths in a particular incident does not mean that this road is the highest in road accidents but in its edition. Confirmed d. Hala Henawy official health promotion programs the World Health Organization that the statistics in 2012 monitored by the World Health Organization for road accidents in Egypt that the number of deceased annually 719 thousand cases of whom 92 thousand deaths due to road accidents within 24 hours of the incident and there are 7.32 thousand cases die after 24 hours the incident and thus of death in hospitals affected by the incident increases the number of road accidents, but do not write in the report the cause of death, such as his colleague who died on the spot on the road

Maj. Gen. Medhat Koraytem Assistant Minister of Interior for the passage of Egypt lose three of annual income due to road accidents which cause a major impact on the national economy

In order to reduce road accidents, the ministry is installing cameras on highways to adjust the excess speed and a range of other measures to reduce accidents, as well as the installation of a large-screen display shows crowded places to change the route

Major-General Ambassador Noor board member of the Egyptian automobile club that requires Upgrading of roads, driver and car to reduce road accidents, stressing the importance of the human element, which escaped morals after the revolution, referring to some encroach on the traffic police and walk the opposite direction and other unhealthy behaviors that require emphasis on violator

He says d. Hamdi Leithy transportation expert and ATV and head Menatel company that the reason for road accidents in Egypt due to the design and quality of the road in addition to maintenance problems resulting in weakness efficiency

Leithy said that the behavior of motorists especially heavy transport a significant impact on highway accidents

To reduce road accidents Leithy said that the new technology can help raise the efficiency of the roads and reduce flight time and thus fuel, gasoline and time availability

He explained that during his meeting with Dr. Ibrahim El Demery Transport Minister to discuss the possibility of reducing road accidents, he offered him the application of technology - which considers the easiest and fastest - to reduce the aggravation of accidents problem on the roads, with maintenance to come up with a good rapid results in a short-term plan

He said that the application of technology using mobile cameras and gates to increase the capacity of traffic management, organized a scientific way and controlled from a central room to see busy roads to direct vehicles at least the busiest roads, with roads raise efficiency and reduce the bumps and the organization of signage

He stressed that the most important ways that will be initiated after the completion of the agreement with the Ministry of Transportation will be on the Ring Road and my way Egypt Alexandria Desert and agriculture, as well as ways to level the eastern and western

To reduce the incidents and bloodshed on the highways, said Dr. Ismail Abdul Ghaffar, President of the Arab Academy for Science and Technology said scientific research is the salvation of the Egyptian citizen of disasters and crises suffered by the rocking Bhadharh and its future development and threatens future generations

He said that with the recurrence of such incidents typically the Academy to conduct research necessary to prevent recurrence of such incidents and scientific studies to support the state's efforts to find solutions to those crises, pointing out that the cost of manufacturing and the operation of such systems does not exceed 1% of the losses incurred by the state because of these incidents

He said that the academy has signed an agreement with the Alexandria Governorate for the installation of cameras to control traffic violations and activating the traffic control room in the streets and public squares that have been providing them with technological devices developed and which is the first of its kind where The Chamber command and control through 221 private television camera on the Cairo Alexandria Desert covers the city of Alexandria whole to overcome the problem of traffic

Dr. Jalal Al-governor of Cairo said he was signing of a protocol with the National Service of the Ministry of Defense spends to cover 250 sites of important intersections to maintain the work of an integrated project to regulate the traffic, through the two chambers of key management Cairo traffic and the other building, the province ordered with control programs to control the traffic lights and cameras set irregularities and displays using the latest technologies with the ability to modify the times of the signals in accordance with current and sudden junctions with the requirements a total cost of 261 million pounds

Counselor Ahmed Amin, General Coordinator of the Egyptian Society for transportation and stressed that poor road conditions have a direct impact in the increase of road accidents attributed as the reports confirm that the rate of not less than 16 to 20 of road accidents caused by technical defects in road include defects in curves and flaws in bridges, roads fast which

He called Amin quickly work on the rehabilitation of all roads, bridges and maintenance work on fully

The d. Ibrahim El Demery and Minister of Transport, the Ministry of Transport is always keen to improve traffic safety level and the achievement of all the means that would provide maximum safety levels and safety for the users of the road network and raise the level of services provided through research and studies development in this area to get to know new in the field of road traffic and create a New matching international standards and methods of work as well as periodic maintenance of the existing roads and the use of modern systems and technologies that contribute to raising the level of these services

حوادث المرور			السنوات
وفيات	جرحى	عدد التدخلات	
33	211	106	2002
34	215	120	2003
50	272	136	2004
41	279	154	2005
73	373	208	2006
64	493	250	2007
77	775	376	2008
95	800	415	2009
63	724	374	2010
109	1230	617	2011
96	1263	1351	2012

PROJECT IMPLEMENTATION

Preface

While the sales volumes of PCs and feature cell phones stagnate, smartphone – and, since recently, tablet PC – sales boom.

Looking at operation system distributions in the mobile sector, iOS and Android are prevalent, squeezing other operation systems (like Symbian or Blackberry OS) out of the market (upcoming Windows Phone is still playing in its small niche).

Although Apple has pioneered the mobile market with their iPhone and iPad, Android device sales are growing much more rapidly in both relative and absolute terms. For smartphones, in year 2012, Android has a 59% market share (iOS: 23%), and 145% growth per year (iOS: 89%).

Still being a niche segment as of today, tablet PC sales are estimated to rapidly grow as well. In this segment it is forecasted that, while iOS will retain its leadership for several upcoming years, Android's relative growth is, and will be, higher. – See chapter Android Market Share for some resources on these issues.

Prerequisites

Unfortunately, there are more than two or three steps involved in order to get the sample application running, so this cannot be considered a Hello World sample.

Nevertheless, I've tested the deployment procedure, thus feel free to contact me in case of incidents – after "Google" hasn't brought you forward.

The application has been developed on

- Android Studio 1.3

The sample application makes use of Android's Location-Based Services (LBS) . Due to officially unresolved bugs related to using these with the Android Emulator in Android 4.2.2 the sample application uses the

- Android SDK 4.2.2 , API level 17, along with the corresponding
- Google APIs, API level 17.

Implantation and code

The project consists of classes and objects that performs its function:

1-Splash Screen

2- Login and Register

3- Main application screen (User Information Editing)

4-Contact Picker

5-Display screen and widget button

6-Datapinnig and storage

Let's Started with Splash Screen

- Splash screen: screen is a graphical control element consisting of window containing an image, a logo and the current version of the software. A splash screen usually appears while a game or program is launching.

Layout code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <ImageView android:id="@+id/title_image_splash"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:src="@drawable/splash"
        android:scaleType = "centerCrop"
        android:contentDescription="Image for splash screen"
        android:layout_gravity="center" />

</LinearLayout>
```

Splachscreen.java code:

```

package com.example.mohamed.iamindanger;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.view.Menu;

import com.example.mohamed.iamindanger.R;
import com.parse.Parse;
import com.parse.ParseUser;

public class Splash extends Activity {

    /** Duration of wait */
    private final int SPLASH_DISPLAY_LENGTH = 1000;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_splash);

        /* New Handler to start the Menu-Activity
        * and close this Splash-Screen after some seconds.*/
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                /* Create an Intent that will start the Menu-Activity. */
                Intent mainIntent = new Intent(Splash.this, login.class);
                Splash.this.startActivity(mainIntent);
                Splash.this.finish();
            }
        }, SPLASH_DISPLAY_LENGTH);
    }
}

```

Then We Using Parse.com APIs to have a cloud backend used for register and sign in:

Login layout code :

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent" android:gravity="center_horizontal"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.mohamed.iamindanger.login"
    android:background="@color/gray"
>

<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/sign_in_title">

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/username"
        android:layout_alignParentBottom="true"
        android:layout_alignParentStart="true"
        android:layout_marginBottom="228dp"
        android:layout_marginRight="12dp"
        android:layout_marginLeft="12dp"
        android:background="@color/white"
        android:hint="Username"
    />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPassword"
        android:ems="10"
        android:id="@+id/password"
        android:layout_alignTop="@+id/username"
        android:layout_marginRight="12dp"
        android:layout_marginLeft="12dp"
        android:layout_marginTop="35dp"
        android:background="@color/white"
        android:hint="password"
    />

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="SIGN IN"
        android:id="@+id/signin"
        android:layout_below="@+id/password"
        android:layout_marginRight="8dp"
        android:layout_marginLeft="8dp"
        android:layout_alignParentEnd="true"
        android:layout_marginTop="12dp"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="DON'T HAVE AN ACCOUNT!! SIGN UP"
        android:textColor="@color/white"
        android:id="@+id/signuptext"
        android:layout_below="@+id/signin"
```

Login.java code :

```

package com.example.mohamed.iamindanger;

import android.app.Activity;
import android.app.ProgressDialog;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

import com.parse.LogInCallback;
import com.parse.Parse;
import com.parse.ParseException;
import com.parse.ParseUser;

/**
 * Created by rufflez on 7/8/14.
 */
public class login extends Activity {

    private EditText usernameView;
    private EditText passwordView;

    @Override
    public void onCreate(Bundle savedInstanceState){
        Parse.initialize(this, "mWd74VseRTuVD3NJVotgM7QAehE0mwyvMe96OJsy",
"6AD5fj4tyGUamWys2alGUb8lRIDKoMIbVuzoM6dI");

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);

        // Set up the login form.
        usernameView = (EditText) findViewById(R.id.username);
        passwordView = (EditText) findViewById(R.id.password);

        // Set up the submit button click handler
        findViewById(R.id.signin).setOnClickListener(new
View.OnClickListener() {
            public void onClick(View view) {
                // Validate the log in data
                boolean validationError = false;
                StringBuilder validationErrorMessage =
                    new
StringBuilder(getResources().getString(R.string.error_intro));
                if (isEmpty(usernameView)) {
                    validationError = true;

validationErrorMessage.append(getResources().getString(R.string.error_blank_
username));
                }
                if (isEmpty(passwordView)) {
                    if (validationError) {

validationErrorMessage.append(getResources().getString(R.string.error_join))
;
                    }
                }
            }
        });
    }
}

```



```

        validationError = true;

validationErrorMessage.append(getResources().getString(R.string.error_blank_
password));
    }

validationErrorMessage.append(getResources().getString(R.string.error_end));

    // If there is a validation error, display the error
    if (validationError) {
        Toast.makeText(login.this,
validationErrorMessage.toString(), Toast.LENGTH_LONG)
            .show();
        return;
    }

    // Set up a progress dialog
    final ProgressDialog dlg = new ProgressDialog(login.this);
    dlg.setTitle("Please wait.");

    dlg.setMessage("Logging in. Please wait.");
    dlg.show();
    // Call the Parse login method

ParseUser.logInBackground(usernameView.getText().toString(),
passwordView.getText()
        .toString(), new LogInCallback() {

    @Override
    public void done(ParseUser user, ParseException e) {
        dlg.dismiss();
        if (e != null) {
            // Show the error message
            Toast.makeText(login.this, e.getMessage(),
Toast.LENGTH_LONG).show();
        } else {
            // Start an intent for the dispatch activity
            Intent intent = new Intent(login.this,
MainActivity.class);
            intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK
| Intent.FLAG_ACTIVITY_NEW_TASK);
            startActivity(intent);
        }
    }
});
    }

});
TextView textView= (TextView) findViewById(R.id.signuptext);
textView.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        startActivity(new Intent(login.this, signup.class));
    }
});
}

private boolean isEmpty(EditText etText) {
    if (etText.getText().toString().trim().length() > 0) {

```

```

        return false;
    } else {
        return true;
    }
}
}

```

Dispatch Activity: Activity that starts LoginActivity if the user is not logged in. Otherwise, it starts the subclass-defined target activity.

Dispatch.java :

```

package com.example.mohamed.iamindanger;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

import com.parse.Parse;
import com.parse.ParseUser;

public class dispatchActivity extends Activity{
    @Override
    public void onCreate(Bundle savedInstanceState){
        Parse.initialize(this, "mWd74VseRTuVD3NJVotgM7QAehE0mwyvMe96OJsy",
"6AD5fj4tyGUamWys2alGUb8lRIDKoMIbVuzoM6dI");
        super.onCreate(savedInstanceState);
        // Check if there is current user info
        if (ParseUser.getCurrentUser() != null) {
            // Start an intent for the logged in activity
            startActivity(new Intent(this, MainActivity.class));
        } else {
            // Start and intent for the logged out activity
            startActivity(new Intent(this, login.class));
        }
    }
}
}

```

```

=====
=====

```

Register Screen

Register.java code :

```

package com.example.mohamed.iamindanger;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Intent;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

import com.parse.ParseException;
import com.parse.ParseUser;
import com.parse.SignUpCallback;

public class signup extends Activity {
    private EditText usernameView;
    private EditText emailView;
    private EditText passwordView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_signup);
        usernameView = (EditText) findViewById(R.id.username);
        emailView = (EditText) findViewById(R.id.email);
        passwordView = (EditText) findViewById(R.id.password);
        // Set up the submit button click handler
        findViewById(R.id.signup).setOnClickListener(new
View.OnClickListener() {
            public void onClick(View view) {

                // Set up a progress dialog
                final ProgressDialog dlg = new ProgressDialog(signup.this);
                dlg.setTitle("Please wait.");
                dlg.setMessage("Signing up. Please wait.");
                dlg.show();

                // Set up a new Parse user
                ParseUser user = new ParseUser();
                user.setUsername(usernameView.getText().toString());
                user.setPassword(passwordView.getText().toString());
                user.setEmail(emailView.getText().toString());
                // Call the Parse signup method
                user.signUpInBackground(new SignUpCallback() {

                    @Override
                    public void done(ParseException e) {
                        dlg.dismiss();
                        if (e != null) {
                            // Show the error message
                            Toast.makeText(signup.this, e.getMessage(),
Toast.LENGTH_LONG).show();
                        } else {

```

```

        // Start an intent for the dispatch activity
        Intent intent = new Intent(signup.this,
Splash.class);
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK
| Intent.FLAG_ACTIVITY_NEW_TASK);
        startActivity(intent);
    }
}
});
}
});
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is
    present.
    getMenuInflater().inflate(R.menu.menu_signup, menu);
    return true;
}

```

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}
}

```

Register layout :

```

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context="com.example.mohamed.iamindanger.signup"
    android:background="@color/gray"
>

```

```

<RelativeLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_gravity="center"
    android:layout_alignParentTop="true"
    android:layout_alignParentEnd="true"
    android:layout_alignParentStart="true"
    android:background="@drawable/sign_in_title">

```

```

<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/username"
    android:layout_marginBottom="235dp"
    android:layout_marginRight="12dp"
    android:layout_marginLeft="12dp"
    android:layout_alignParentBottom="true"
    android:background="@color/white"
    android:hint="Username"
    android:layout_centerHorizontal="true"
/>

```

```

<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="textEmailAddress"
    android:ems="10"
    android:id="@+id/email"
    android:layout_marginRight="12dp"
    android:layout_marginLeft="12dp"
    android:hint="E-mail"
    android:layout_alignTop="@+id/username"
    android:background="@color/white"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="35dp"
/>

```

```

<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="textPassword"
    android:ems="10"
    android:id="@+id/password"
    android:background="@color/white"
    android:layout_below="@+id/email"
    android:layout_marginRight="12dp"

```

```

        android:layout_marginLeft="12dp"
        android:hint="Password"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="10dp"/>

<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="SIGN                                UP"
    android:id="@+id/signup"
    android:layout_marginRight="8dp"
    android:layout_marginLeft="8dp"
    android:layout_below="@+id/password"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="2dp"/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="sign                                in"
    android:id="@+id/textView2"
    android:layout_below="@+id/signup"
    android:textColor="@color/white"
    android:layout_alignStart="@+id/password"      />

</RelativeLayout>

</RelativeLayout>

```

Contact Picker

We'll then be adding functionality to allow a user to choose one of their existing contacts and send a canned message to them. We're going to dive right in, so have all of your code and tools ready. Finally, make sure your device or emulator has some contacts configured (with names and emails) within the Contacts application.

Contact Picker Layout code :

```
<RelativeLayout

    android:layout_height="wrap_content"

    android:layout_width="match_parent">

    <EditText

        android:layout_height="wrap_content"

        android:hint="@string/invite_email_hint"

        android:id="@+id/invite_email"

        android:inputType="textEmailAddress"

        android:layout_width="wrap_content"

        android:layout_toLeftOf="@+id/do_email_picker"

        android:layout_alignParentLeft="true"></EditText>

    <Button

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"
```

```

        android:id="@+id/do_email_picker"

        android:text="@string/pick_email_label"

        android:layout_alignParentRight="true"

        android:onClick="doLaunchContactPicker"></Button>

</RelativeLayout>

```

=====

ContactPicker.java code :

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (resultCode == RESULT_OK) {
        switch (requestCode) {
            case CONTACT_PICKER_RESULT:
                Cursor cursor = null;
                String email = "";
                try {
                    Uri result = data.getData();
                    Log.v(DEBUG_TAG, "Got a contact result: "
                        + result.toString());

                    // get the contact id from the Uri
                    String id = result.getLastPathSegment();

                    // query for everything email
                    cursor = getContentResolver().query(Email.CONTENT_URI,
                        null, Email.CONTACT_ID + "=?", new String[] { id },
                        null);

                    int emailIdx = cursor.getColumnIndex(Email.DATA);

                    // let's just get the first email
                    if (cursor.moveToFirst()) {
                        email = cursor.getString(emailIdx);
                        Log.v(DEBUG_TAG, "Got email: " + email);
                    } else {
                        Log.w(DEBUG_TAG, "No results");
                    }
                } catch (Exception e) {
                    Log.e(DEBUG_TAG, "Failed to get email data", e);
                } finally {
                    if (cursor != null) {
                        cursor.close();
                    }
                }
            }
        }
    }
}

```



```

        EditText emailEntry = (EditText) findViewById(R.id.invite_email);
        emailEntry.setText(email);
        if (email.length() == 0) {
            Toast.makeText(this, "No email found for contact.",
                Toast.LENGTH_LONG).show();
        }

    }

    break;
}

} else {
    Log.w(DEBUG_TAG, "Warning: activity result not ok");
}

```

We using Local Datastore for store data and retrieve :

The Parse Android SDK provides a local datastore which can be used to store and retrieve ParseObjects, even when the network is unavailable. To enable this functionality, simply call `Parse.enableLocalDatastore()` before your call to initialize.

```

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

public class App extends Application {
    @Override
    public void onCreate() {
        super.onCreate();

        Parse.enableLocalDatastore(this);
        Parse.initialize(this, PARSE_APPLICATION_ID, PARSE_CLIENT_KEY);
    }
}

```

There are a couple of side effects of enabling the local datastore that you should be aware of. When enabled, there will only be one instance of any given ParseObject. For example, imagine you have an instance of the "GameScore" class with an objectId of "xWMYz4YEGZ", and then you issue a ParseQuery for all instances of "GameScore" with that objectId. The result will be the same instance of the object you already have in memory.

Another side effect is that the current user and current installation will be stored in the local datastore, so you can persist unsaved changes to these objects between runs of your app using the methods below.

Calling the `saveEventually` method on a `ParseObject` will cause the object to be pinned in the local datastore until the save completes. So now, if you change the current `ParseUser` and call `ParseUser.getCurrentUser().saveEventually()`, your app will always see the changes that you have made.

Pinning

You can store a `ParseObject` in the local datastore by pinning it. Pinning a `ParseObject` is recursive, just like saving, so any objects that are pointed to by the one you are pinning will also be pinned. When an object is pinned, every time you update it by fetching or saving new data, the copy in the local datastore will be updated automatically. You don't need to worry about it at all.

```
ParseObject gameScore = new ParseObject("GameScore");
gameScore.put("score", 1337);
gameScore.put("playerName", "Sean Plott");
gameScore.put("cheatMode", false);
```

```
gameScore.pinInBackground();
```

If you have multiple objects, you can pin them all at once with the `pinAllInBackground` convenience method.

```
ParseObject.pinAllInBackground(listOfObjects);
```

Retrieving Objects

Storing objects is great, but it's only useful if you can then get the objects back out later. Retrieving an object from the local datastore works just like retrieving one over the network. The only difference is calling the `fromLocalDatastore` method to tell the `ParseQuery` where to look for its results.

```
ParseQuery<ParseObject> query = ParseQuery.getQuery("GameScore");
query.fromLocalDatastore();
query.getInBackground("xWMYz4YE", new GetCallback<ParseObject>() {
    public void done(ParseObject object, ParseException e) {
        if (e == null) {
            // object will be your game score
        } else {
            // something went wrong
        }
    }
});
```

Lock Screen Widget :

We can deploy it accessing those properties :

(FLAG_SHOW_WHEN_LOCKED and FLAG_DISMISS_KEYGUARD). FLAG_SHOW_WHEN_LOCKED works pretty consistently in that it will show on top of the lock screen even when security is enabled (the security isn't bypassed, you can't switch to another non-FLAG_SHOW_WHEN_LOCKED window).

If you're just doing something temporary, like while music is playing or similar, you'll probably mostly be okay. If you're trying to create a custom lock screen then there's a lot of unusual interactions on all the different android platforms. ("Help! I can't turn off my alarm without rebooting my HTC phone").

```
getWindow().addFlags(WindowManager.LayoutParams.FLAG_SHOW_WHEN_LOCKED);  
getWindow().addFlags(WindowManager.LayoutParams.FLAG_DISMISS_KEYGUARD);
```

Conclusion

Everyday there are a lot of deaths because of wrong Diagnosis especially if the patient is Unconscious. In accidents the most probably case is the injured is unconscious and if the rescuers want to know any information or contact any known body for him they found his phone is locked.

It will be an emergency system on Android devices doesn't need to unlock phone to contact with mobile owner emergency list by widget on lock screen.

The new system must include the following:

- Ability to locate and send the location through Emergency SMS.
- Ability to display on the mobile screen the medical history of the injured.
- Ability to incorporate automated routing and emergency notifications based on location.

And it's benefit is:

- Avoid and assist is diagnose.
- Automated connection with emergency agents and contacts (based on location).

Creating Simple Values

Strings

Externalizing your strings helps maintain consistency within your application and makes it much easier to create localized versions.

String resources are specified with the `<string>` tag, as shown in the following XML snippet.

```
<string name="stop_message">Stop.</string>
```

Android supports simple text styling, so you can use the HTML tags ``, `<i>`, and `<u>` to apply bold, italics, or underlining respectively to parts of your text strings, as shown in the following example:

```
<string name="stop_message"><b>Stop.</b></string>
```

You can use resource strings as input parameters for the `String.format` method. However, `String.format` does not support the text styling described above. To apply styling to a format string you have to escape the HTML tags when creating your resource, as shown in the following.

```
<string      name="stop_message">&lt;b>Stop&lt;/b>.  
%1$s</string>
```

Within your code, use the `Html.fromHtml` method to convert this back into a styled character sequence.

```
String          rString          =  
getString(R.string.stop_message);  
  
String    fString    =    String.format(rString,  
"Collaborate and listen.");  
  
CharSequence          styledString          =  
Html.fromHtml(fString);
```

Colors

Use the `<color>`

tag to define a new color resource. Specify the color value using a `#` symbol followed

by the (optional) alpha channel, then the red, green, and blue values using one or two hexadecimal numbers with any of the following notations:

➤ `#RGB`

➤ `#RRGGBB`

➤ `#ARGB`

➤ `#AARRGGBB`

The following example shows how to specify a fully opaque blue and a partially transparent green.

```
<color name="opaque_blue">#00F</color>
```

```
<color name="transparent_green">#7700FF00</color>
```

Dimensions

Dimensions are most commonly referenced within style and layout resources. They're useful for creating layout constants such as borders and font heights.

To specify a dimension resource use the `<dimen>` tag, specifying the dimension value, followed by an identifier describing the scale of your dimension:

- `px` (screen pixels)
- `in` (physical inches)
- `pt` (physical points)
- `mm` (physical millimeters)
- `dp` (density-independent pixels relative to a 160-dpi screen)
- `sp` (scale-independent pixels)

These alternatives let you define a dimension not only in absolute terms, but also using relative scales that account for different screen resolutions and densities to simplify scaling on different hardware.

The following XML snippet shows how to specify dimension values for a large font size and a standard border:

```
<dimen name="standard_border">5dp</dimen>
```



```
<dimen name="large_font_size">16sp</dimen>
```

Saving Objects

Let's say you want to save the `GameScore` described above to the Parse Cloud. The interface is similar to a `Map`, plus the `saveInBackground` method:

```
ParseObject gameScore = new ParseObject ( "GameScore" );
```

```
gameScore.put ( "score" , 1337 );
```

```
gameScore.put ( "playerName" , "Sean Plott" );
```

```
gameScore.put ( "cheatMode" , false );
```

```
gameScore.saveInBackground();
```

After this code runs, you will probably be wondering if anything really happened. To make sure the data was saved, you can look at the Data Browser in your app on Parse. You should see something like this:

```
objectId: "xWMyZ4YEGZ", score: 1337, playerName: "Sean Plott",
cheatMode: false,
  createdAt:"2011-06-10T18:33:42Z", updatedAt:"2011-06-10T18:33:42Z"
```

Retrieving Objects from the Local Datastore

Storing an object is only useful if you can get it back out. To get the data for a specific object, you can use a `ParseQuery` just like you would while on the network, but using the `fromLocalDatastore` method to tell it where to get the data.

```
ParseQuery<ParseObject> query =
```

```
ParseQuery.getQuery( "GameScore" );
```

```
query.fromLocalDatastore();
```

```
query.getInBackground( "xWMyZ4YEGZ" , new
```

```
GetCallback<ParseObject>() {
```

```
public void done(ParseObject object, ParseException e) {
```

```
if (e == null) {
```

```
// object will be your game score
```

```
    } else {
```

```
        // something went wrong
```

```
    }
```

```
}
```

```
});
```

If you already have an instance of the object, you can instead use the `fetchFromLocalDatastoreInBackground` method.

```
ParseObject object =
```

```
ParseObject.createWithoutData( "GameScore", "xWMyZ4YEGZ" );
```

```
object.fetchFromLocalDatastoreInBackground( new
```

```
GetCallback<ParseObject>() {
```

```
    public void done(ParseObject object, ParseException e) {
```

```
        if (e == null) {
```

```
            // object will be your game score
```

```
        } else {
```

```
            // something went wrong
```

```
        }
```

```
    }
```

```
});
```

Dispatch.java :

```
package com.example.mohamed.iamindanger;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

import com.parse.Parse;
import com.parse.ParseUser;

public class dispatchActivity extends Activity{
    @Override
    public void onCreate(Bundle savedInstanceState){
        Parse.initialize(this, "mWd74VseRTuVD3NJVotgM7QAehE0mwyvMe96OJsy",
"6AD5fj4tyGUamWyS2aIGUb8lRIDKoMIbVuzoM6dI");
        super.onCreate(savedInstanceState);
        // Check if there is current user info
        if (ParseUser.getCurrentUser() != null) {
            // Start an intent for the logged in activity
            startActivity(new Intent(this, MainActivity.class));
        } else {
            // Start and intent for the logged out activity
            startActivity(new Intent(this, login.class));
        }
    }
}
```

```
}
```

```
=====
```

Register Screen

Register.java code :

```
package com.example.mohamed.iamindanger;

import android.app.Activity;
import android.app.ProgressDialog;
import android.content.Intent;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

import com.parse.ParseException;
import com.parse.ParseUser;
import com.parse.SignUpCallback;

public class signup extends Activity {
    private EditText usernameView;
    private EditText emailView;
    private EditText passwordView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_signup);
        usernameView = (EditText) findViewById(R.id.username);
        emailView = (EditText) findViewById(R.id.email);
        passwordView = (EditText) findViewById(R.id.password);
        // Set up the submit button click handler
        findViewById(R.id.signup).setOnClickListener(new View.OnClickListener() {
            public void onClick(View view) {

                // Set up a progress dialog
                final ProgressDialog dlg = new ProgressDialog(signup.this);
                dlg.setTitle("Please wait.");
                dlg.setMessage("Signing up. Please wait.");
                dlg.show();
            }
        });
    }
}
```

```

// Set up a new Parse user
ParseUser user = new ParseUser();
user.setUsername(usernameView.getText().toString());
user.setPassword(passwordView.getText().toString());
user.setEmail(emailView.getText().toString());
// Call the Parse signup method
user.signUpInBackground(new SignUpCallback() {

    @Override
    public void done(ParseException e) {
        dlg.dismiss();
        if (e != null) {
            // Show the error message
            Toast.makeText(signup.this, e.getMessage(), Toast.LENGTH_LONG).show();
        } else {
            // Start an intent for the dispatch activity
            Intent intent = new Intent(signup.this, Splash.class);
            intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK | Intent.FLAG_ACTIVITY_NEW_TASK);
            startActivity(intent);
        }
    }
});

```