Mobile Application and Cloud Computing Notes

Giuliano Abruzzo

June 29, 2019

Contents

1 Cap 1: Introduction

A **mobile application** is composed by five *fixed blocks*:

- **Front-end**;
- **Back-end**;
- **Storage**;
- **Web-API**;
- **Cloud service**;

The **cloud computing** is a way to deliver remote *virtual resources* through *internet*. *Mobile application* are different from *desktop* ones cause they are very *easy to install* and few ones are used with respect to the number of installed ones. They are afflicted by *crash* and *bad UX* (user experience), and they are very slow. Right now there is a duopoly: **Android** and **iOS**, *iOS* devices are *closed-source* and work on one *market* only, while *Android* ones are *open source* and are used by several producers that can customize the basic *OS* of the device, and there are several markets. Apps have success cause they solve a problem simple and it's funny to use them. Apps doesn't have success cause they have *bad UX*, they have bad performance and for improper testing.

The *smartphones* are a **disruptive technology**, that is a technology that changes the *market* and replaces the *existing technology*, this process is called **dematerialization**, using less to produce more. They are also an *innovation* in the *media* in *banking* and *entertainment*. We call **digitalization** the process in which we decouple *information* from *physical support* (from a book to a ebook, ...). The mobile app can be classified in:

- **Web-site for mobile devices**: in which we deliver web content to mobile devices, using web site style navigation;
- **Web app**: web application that mimics native apps look and feel as well as navigation;
- **Native apps**: applications installed on the devices, purchased from app stores;
- **Hybrid apps**: applications that mixed web and native application;
- **[Native] Native applications**: application that direct access to C/C++ libraries;

2 Cap 2: Web Technologies for Mobile Apps

We want to understand how **web technologies** may be used in *mobile apps* and basic notation of **Responsive Web Design** or **RWD**. In *static web*, in which we don't have any *sever-side computation*, when we connect to a *web page* we execute it, and we download the *page* from the *server* which acts like a *dispatcher*, so the *web browser* acts like a VM.

- In **mobile web apps architecture** we just avoid fetching page from the site, we store the *web page* locally and the **web rendering engine** is run and we load the *web page* in the *engine*;
- In the **hybrid app architecture** after the run of the *web rendering engine*, a *native mapping* is made;
- In **native looking web apps** we don't have the web related user interaction (like scrolling or zooming) but we have native looking widgets and native features;

A *website* designed for *desktop* may provide a *bad UX* for *mobile*, so we should change the *layout* and use *images* suitable to screen resolution. We will use **vector images**, these are images described in terms of *points* connected by *lines* and *curves* in order to form *polygons*, the most popular is the **SVG** format, or **Scalable Vector Graphics** that is a *web graphics language* that allows to create *static* and *dynamic images*. *Vector images* are preferred cause they may be of any size instead the *scalar images* have their own pixel size.

If we want to display the same *image* on screens with different **aspect ratio** (from 16:9 to 4:3) we preserve the *original shape*, by scaling the *image* and *centering* it and then we add two *horizontal bars* called *letterbox*. A common smartphone screen *aspect ratio* is 9:16, so when we display it on a 16:9 TV we will see two *vertical lines*, similar to the *letterbox* called *pillarbox*.

Screen resolution is how many *pixels* are displayed per inch, and this is expressed in *PPI*. The eye has a *resolution* limit, so a $PPI > 300$ doesn't makes sense. When a *web browser* has to render a *web page* in order:

- Parse *HTML tags* to *DOM object* in a *DOM tree*;
- Parse *CSS* in order to define how to display the content of *HTML elements*;
- Attach *CSS properties* to *DOM objects*, creating a *render tree*;
- *Layout process* the *render tree*, where *physical position* are assigned to elements;
- Painting the *render tree*;

The **rendering process** is done with respect to the **viewport**, that is the *size* of the whole available *area* where the *page* is displayed. Using the *real device viewport* may look bad or break, so *rendering* is done using a *wider* **virtual viewport**, where a user can zoom and move to see different areas of the *page*. In order to deal with different *viewport* we can create a *CSS* for every different *screen* (so on *server side*), or define a *CSS per-screen* on the *client side*, also called **responsive web** and is made through *CSS media query*.

Sometimes we need to run some **logic part** of the *app* (the *backend*) remotely in some **services**, cause the *information* contained are a lot (like *authentication services*, *complex algorithms*, ...). A *mobile application* is connected to such *services* through different *protocols*, and several *APIs* are used to retrieve information. *AJAX* allows for exchanging *data* via *HTTP* and works in a *asynchronous multi-thread* flavor. The *caller* is notified when the *HTTP* returns. It is used when we have to retrieve *data* from the same origin of the loaded page, and to change the *web page* without blocking it. *XML* and *JSON formats* are used for the object exchanged by these services. In an *XML HTTP request object* interaction, we call the *server side* element and the response is sent to a *callback function*, which changes the *web page* properly. In order to retrieve data from different origins we have to use the $<$ *script* $>$ tag, where *JS script* can be loaded, using *JSONP* which allows to load a *JS file* with these functions that fetch *data*.

3 Cap 3: Android Native App

- **Android** is a *linux-based operating system* with important changes in *process* and *memory management*. An *application* runs inside a **process**, and *processes* running *applications* are all created from the same *process* called **Zygote** that is a main process which contains a pre-warmed execution *environment* required to all the *apps*, and this reduces the start-up time;

- In *android*, we have only *fork operation* not *exec*;

- In *Android* the **OOM Killer** (*Out of memory killer*), the mechanism that the *kernel* uses to recover *memory* by killing processes, is different from *tradition Linux* since the *processes* are **ranked** according to the *state* of the contained *application*, and an *application* can be resumed;

- Each *processes* that runs an *application* belong to a unique and different *user*, so the files created by an app cannot be read from other apps, and the **Intent** is the *app-to-app communication*, and can be *explicit* (target app) or *implicit* (any application that can perform an action required);

- There are *managers* for *package, telephony, location, activity, resource* and *notifications*;

- In the traditional way the *OS* manages *processes* providing the *execution environment*, and for each process we have a *fork* and a *exec*. Instead in the *Android* one, the server process contains all the *android managers*, and all the *processes* are forked from *Zygote*;

- Since in *Android* we have a problem of **fragmentation**, as new feature are added *support libraries* are developed, so that such features are also available to older *android versions*;

- The **Android kernel**, provides a level of *abstraction* between the *device hardware* and contains all the essential *hardware drivers*. The *binder* is a special *driver* designed to provide secure *communication* between *apps*;

- The **ART**, *Android run time*, runs directly on *hardware*, and each *app* runs in its own *process* and with its own instance of the *Android Run-time*;

- The *native libraries* (C++) are needed in the case an app handles *2D rendering, graphics, particular media format, ...* ;
- The entire feature set of the *Android OS* is available through *Java packages*, collected in the **Android Framework**: *apps*, *database, graphics, hardware API* are contained there;

4 Cap 4: Activity, Fragment, Navigation, Displaying large content

- The *Android screen* is composed by several elements, and the smaller one is called **View**. *Views* are organized in several ways thanks to **Layouts**, that are *Viewgroups* used to host other *views*, and **Containers** used to handle *dynamic* contents or content bigger than the screen size;
- *Dynamic* content management require special care in order to optimize performance, and a very common pattern is a *list* of item, for this there are special *views* like the **Recyler View**;
- A **style** is a collection of *attributes* that specify the appearance for a single *View*;
- A **theme** is a type of *style* that's applied on the *entire app*, *activity* or *view hierarchy*;
- An *application* is composed by at least one **Activity**, that is the *controller* of the *GUI*, and runs inside the *main thread* and it is reactive to *user input*. All the activity are managed by the **Activity manager**, via a *back stack* which contains all the previous *activities*, where the *current activity* (**foreground activity**) is on top and all the other *actives* that are not yet destroyed are below;
- The set of activities launched by a user is a *Task*, and each application runs inside its own *Process*, all the components of an *app* runs inside the only created thread (*main thread*), usually other *threads* are created to perform long running operations (services), and an *activity* in a *task* can belong to a different *application* and run in a different *processes*;
- Each *activity* in an *application* got its own **life-cycle** responding to a set of *Android* methods like:
  - *onCreate*, a method that runs when the *activity* is *created*;
  - *onDestroy* that happens when an *activity* is *shut down*;
  - *onPause* called when the system is about to start resuming another *activity*, generally used to commit *unsaved changes*;
- We can use a **Bundle** in order to save state but only for small data, instead we can use the **ViewModel** for large amount of data;
- The states of an **Activity** can be:
  1. **Running**: the *activity* is *foreground* and has the focus, so all the *events* are delivered to it;

2. **Paused**: the *activity* is *partially visible*, like with a *dialog box*, all state and information are maintained but the *system* can kill it;
3. **Stopped**: the *activity* is completely obscured by another one;

- An **Activity** is *explicitly* created if another *Activity* use an *Intent*;
- An **Activity** is *implicitly* created when an *Activity* declares to the system its ability to perform *actions* through **intent-filter**, and a calling *activity* ask to the *activity manager* who can perform an *action* required;
- **Fragments** are like small *activity*, hosted inside an *activity*, so they are attached to a *view* and they have their own *view*, and they have their own *life-cycle* (more complex than the *activity*) and these *fragments* can be added through XML or programmatically and they are handled by a **Fragment Manager**;
- The *user* interacts with the *screen* using fingers, and this generates **MotionEvents** which can be also **MultiTouch** based, where each *finger* is a pointer and the *events* are grouped into a *MotionEvent* object;
- **2D graphics** can be *animations*, *SurfaceView-based* or *simple View-based*;
- Since the *screen size* of the *device* is limited, many times we need to show a content that is bigger than the *screen*, so we special **layout** and classes to deal with this problem:
  - **ListActivity**: a subclass of *Activity*, in which there is no *layout* to inflate, that allows to display an array of *items* that are clickable and an *ArrayAdapter* is required to transform an item into a *view*, by default creates a view by calling an operation of *toString()*;
  - **ListView** and **GridView**: *layouts* similar to *ListActivity* but extend *AppCompatActivity*;
  - **RecyclerView**: faster and more flexible, automatically recycles views as they are no longer visible so is more efficient, use a *LayoutManager* to manage the *views*, and an *Adapter* to create and to update data of the *View*, also allows animations;
  - If *data source* is remotely located on a network we can use an additional *thread* to fetch data *asynchronously*, and the *adapter* is notified when data are available;

5 Cap 5: Storage Options

- As we seen, there is often the need of **saving data** and working at persistent across several *states* of a *process*, like in the case of GUI data. We saw that is possible to use the **ViewModel**, that is a class designed to stare and manage *data* related to UI in a *life-cycle* conscious way;
- The **LiveData** is a library class for *data observation*, that notify the observer when the data it holds changes;
- Observed *Data* are usually stored inside a *view model*, in this way, data persist over reconfiguration and changes are notified to the observer;

- For *dynamic content*, the *data* are fetched from a *data source* like a *local DB* or *network*, and a **repository** is used to decouple data source from data usage;
- There is also **Room DB**, that manages local data *SQLite* data source by using objects;
- Any app in *Android* is a **Linux user**, and an app can belong to one or more **Linux groups**, this property is used to implement the *android permissions*;
- *Android* uses **Virtual File System** also called *VFS*, so many different type of *file-systems* can be used. *VFS* is a single hierarchy and we have two main classes of file-system: *media-based* (stored in physical media) and *pseudo file-systems*;
- There are several *storage options* and they depend on the type of data we want to store, such options are:
  - **SharedPrefence**: small amount of data, like *Key-Value pair*, it can be private to an *Activity* or shared among *Activities*;
  - **Internal Storage**: small to medium amount of data, private to the *application*;
  - **External Storage**: not private data (like songs, video files);
  - **Database**: *SQLite*, structured data, private to the application;
  - **Content Provider**: an API which exposes *database* to another *process*, and widely used in clock, alarm, calendar, and widgets, they can be accessed by *Intent* even indirectly;
- **Files** represent by *File class* (*java.io package*), can be *Internal* (internal flash memory) or *External* (sd card)

6 Cap 6: Business Layer

The **business logic** of an *app* is the running of code, working on *input data* and producing an *output*. This is done through several solutions:

- **Worker Threads**: *Java standard Thread* and *Runnable classes* instances used in *Android apps*. They can communicate through a *Loop* which receive messages, while an *Handler* is used for sending and processing messages to a specific queue. In order to schedule actions on the *Main Thread* (called also *UIThread*) we have to append *Runnables* to *View* queues cause a *Thread* cannot modify the *Main Thread*;
- **AsyncTask**: a class which allows to perform background operation and publishes the result of the *UIThread*, without having to manipulate other threads;
- **Service**: an *application component* that runs in *background* and doesn't interact with the *user* for an indefinite period of time. The *services* run in the *main thread* of their hosting process, the difference with the *Thread* is that they are software components with a own *life-cycle*, and they can also run in *background*. If a *service* is destroyed by the OS they can be *re-created* according to the *restart* options, and they can be private or system-wide. There are 3 different types of services:
  - **Background services**: perform operations that isn't directly noticed by the user;

  - **Foreground services**: perform operations that it noticeable to the user;
  - **Bounded services**: it is the *server* in a *client-server interface*, it runs only as long as another application component is bound to it. Multiple components can bind to a single service, but when there are no more components bounded to it, the service is destroyed;

- **Job Scheduler**: used when a *Task* don't require an exact time, but it could be *scheduled* based on *system* and *user requirements*, so it allows to set conditions in order to own a specific *task*;

- **Broadcast receives**: is a *software components* which reacts to *system-wide events.* A *receiver* has to register specific *Intents* and this can be done *Statistically* (XML), so remain dormant and respond to the intent, or *Dynamically*, where receivers are alive as long as the activity who registered them is alive;

7 Cap 7: Native and Hybrid Apps

Some times we need to execute **C/C++ code** for performance reason or for specific libraries like *OpenCV*. This can be done simply loading the **native library** in an *Activity* and then by using methods to handle data returned by the *C/C++ code* and inserting it into *Java* again.

- The **system load library** produce a *shared library* that can be loaded at *run time*, the *shared library* contains the actual executable code. The code uses the *shared library* through an *address* and this is relocated by **dynamic linker**;

- A *Java call* to a *native method* triggers the execution of the native code;

- **Java → Native**: JNI, native methods translated into executable code, linked and available as a library;

- There are three possible way to do that:

  - Blocking call:
    * In which the main thread (Java), calls the native (C++), and the main thread blocks itself until the result;
  - *Native calls Java asynchronous*;
    * In which the main thread (Java), calls in an asynchronous way a native thread (C++), so the main thread doesn't stop;
  - Blocking *Java thread*, avoiding native to find and call java methods;
    * In which the main thread (Java), calls in an asynchronous way a Java thread that inside calls the native methods (C++) ,so the main thread doesn't stop, and he gets the results from the thread in Java and not in C++ like in the second method;

- **Native → Java**: exploits Java Reflection, no executable code produced at compile time;

- **JS → Android**: Android code is made available to the JS interpreter;

- **Android → Native**: evaluate JS code at run-time, and it is similar to reflection;

8 Cap 8: iOS

Every **iOS application** follows a **MVC patter** and the code is strictly related to this organization. The *application* is written in an *Object Oriented language*, and the app is a collection of managed objects (with a *life-cycle*) that responds to *events* (*GUI* and *OS*). Apps are *multi-thread* with one *main thread* looping on *UI* related *user event*, and other *threads* used for long *Tasks* running in background. The app behavior is defined through a *storyboard*, in which we define much of *navigation logic*. **Application Delegate** and **View controller** are the most important ones cause they are pre-created when we open a new project in *Xcode*. Other difference with *Android* are: *Intent* becomes *Segue*, and *Brodcast Receiver* becomes *Notification*.

9 Cap 9: Cloud Computing

Cloud computing is the *on-demand* availability of *computer system resources*, especially *data storage* and *computational power* without direct active management of the *user*. The *cloud computing* is enabled by factors like *distributed computing*, *internet technologies, hardware* and *system management techniques*. An example of this technology is **Dropbox**, it has 2 *access elements*, one for *user* (*web based* and *proxy*) and one for *developers* (*web-api calls and different development technologies*). The main characteristics of *cloud computing* are:

- **Pay-per-Use**;
- **Elastic capacity**;
- **Self-service interface**;

We have different *deliver models*:

- Classic:
  - **SaaS**: *Software* as a Service;
  - **PaaS**: *Platform* as a Service;
  - **IaaS**: *Infrastructure* as a Service;
- Others:
  - **BaaS**: *Backend* as a Service, that provides support to app in terms like DB, notification, auth, like firebase;
  - **MBaaS**: *Mobile Backend* as a Service, like Google Cloud Platform;
  - **FaaS**: *Function* as a Service;
  - **SaaS**: *Storage* as a Service, like Dropbox;

In order to use a **cloud service** we have to: register to the *console*, register the *application* and getting the **API-KEY**, and use the *service* from the *registered app* with **app authentication**. The *authentication* of the app usually is granted through an *API KEY* that the *provider* generate. Some *Web-API* allows to get access to sensible data only with *Access Token*.

Let's talk about **SaaS**, it's a complete environment to build, manage and deploy *apps*. We use a *virtualization* of the layers with the use of *containers* (*Docker*), that differently from *VMs*, the *containers* share the same *OS*, and the same *libraries* if needed. *Containers* include *application* and all of it *dependence*, they share the same *Kernel*, same *OS* with other *containers*. A key feature of *SaaS* is **scaling**, which can be:

- **Vertical**: when the response time of typical *application* depends on the rate of *requests* received, in order to avoid *congestion*, *physical machine* is upgraded as possible;
- **Horizontal**: where the *app* is divided into different *components* that can be replicated on different *physical machines* so the requests can be handled in *parallel*, or we can separate the *app* is *modules* with different roles connected each other;
- **Micro-services**: another way is to implement the *app* as a set of *micro-services*, where each *service* exposes public *API* computing a *asynchronous communication service to service*;
- **Automatic Scaling**: the *auto-scaling* is based on predicting when to scale with usage of *machine learning algorithms*;

Iaas instead is a *cloud infrastructure* that enables on-demand provisioning of *servers* running several choices of *OS* and a *customized software stack*. They are usually composed by *large-scale data centers*, and they offer *virtualized resources* on demand.

A **VM**, or *Virtual Machine*, is a *logic machine* M_L whose *ISA* is implemented exploiting *software* running on a *physical machine* M_F, and there are two main types:

- **Native**: in which $M_L = M_F$ we have same *ISA*, the instructions of the M_F are in large part executed on the real *CPU*;
- **Emulation**: in which $M_L \neq M_F$ so different *ISA*, emulation of *HW*, and installation of different *OS*;

The **virtualization** is realized a **Virtual Machine Monitor** or **Hypervisior** and can be *full virtualization*, or *para virtualization* in which OS can be modified. A classical *Virtual Machine Monitor* (VMM) executes *guest operating system* directly, but at reduced privilege level. *CPU* executes a *Kernel instruction* of *Guest OS* in case of privilege instruction, *CPU* generates a *trap* passing the control to *VMM* that emulates the *instruction*, these *instructions* are different but produces the same effect. *HW virtualization* allows running *multiple OS* and *software stacks* on a single *physical platform*. The *VMM* mediates access to the *physical hardware* presenting to each *guest* operating system a *VM*. There are two types of *VMM*:

- **Bare metal Hypervisior**: in which VMs run directly on top of HW;
- **Hosted Hypervisior**: in which VMs run on top of a Host OS.

In the **Hardware Assisted Virtualization** processors are designed to help *virtualization* so *VMM* can use these instructions to improve the performance. The two main characteristic of VMs are **Isolation** and **Application mobility**:

- **Isolation**: is a property of virtualization where all programs instruction are fully confined inside a VM, so we have better security, reliability, and performance;

- **Application mobility**: allows a better *HW maintenance* just by encapsulating a guest OS state in a VM allowing it to be suspended or migrated to a different platform;

The **HW consolidation**, is *VMM* that consolidate multiple *workloads* into a *single physical platform*. The **IaaS** challenge is to build a *cloud infrastructure* that manage *multiple physical and virtual resources* in an *integrated way*. The software that is responsible of this orchestration is called **Virtual Infrastructure Manager** or **VIM**, that aggregates *resources* from *multiple computers* generating a *uniform view* to *user* and *applications*.

OpenStack is a *cloud operating system* that control large amount of compute, networking and storage in a big data center managed from a dashboard by administrators, while users can use resources via a web interface.

The **mobile cloud computing** exploits *cloud approach* in order to boost the performance of an *application*, and reduce the energy consumption. **CloneCloud** is a *flexible application partitioner* that transforms a *single-machine execution* into a *distributed execution* automatically.