
Internship Report

Master's Degree in Computer Engineering – Mobile Computing

# *Development of a framework to create plugins in Android Apps*

**Rafael Alexandre Marques da Costa**

Leiria, *September 2016*


Internship Report

Master's Degree in Computer Engineering – Mobile Computing

# *Development of a framework to create plugins in Android Apps*

**Rafael Alexandre Marques da Costa**

Master's degree internship report elaborated under Dr. Luís Filipe Fernandes Silva Marcelino supervision, Teacher at Escola Superior de Tecnologia e Gestão of Instituto Politécnico de Leiria.

Leiria, *September 2016*

# *Acknowledgments*

I would like to take this opportunity to express my sincere gratitude to the people that somehow made it happen or helped me through this internship and work in “Plugins Framework”

I sincerely thank Professor Luis Marcelino, for his guidance regarding the school tasks. During the internship, undoubtedly the main person that really taught me so much, Mr. Miguel Nunes, Software Developer on WIT Software and my tutor.
Besides them, a big thanks to everyone in WIT Software and Professor Carlos Grilo for making this internship possible.

On a more personal note, I can never leave unmentioned for their emotional help to my girlfriend Marta Rodrigues and my parents Fátima Marques and Ernesto Costa.

# *Abstract*

The present work aims to allow developers to implement small features on a certain Android application in a fast and easy manner, as well as provide their users to install them on-demand, i.e., they can install the ones they are interested in. These small packages of features are called plugins, and the chosen development language to develop these in was JavaScript.

In order to achieve that, an Android framework was developed that enables the host application to install, manage and run these plugins at runtime.

This framework was designed to have a very clean and almost readable API, which allowed for better code organization and maintainability.

The implementation used the Google's engine "V8" to interpret the JavaScript code and through a set of JNI calls made that code call certain Android methods previously registered in the runtime.

In order to test the framework, it was integrated with the client's communication application RCS+ using two plugins developed alongside the framework. Although these plugins had only the more common requirements, they were proven to work successfully as intended.

Concluding, the framework although successful made it clear that this kind of development through a non-native API has its set of difficulties especially regarding the implementation of complex features.

Key-Words: Plugins, JavaScript, Android, Framework, API

# *Acronyms*

API - Application Programming Interface
HTTP - HyperText Transfer Protocol
IDE - Integrated Development Environment
JSON - JavaScript Object Notation
RCS - Rich Communication Services
SDK - Software Development Kit
UI - User Interface
URL - Uniform Resource Locator
XML - eXtensible Markup Language

# *Index*

# *Introduction*

Nowadays, the smart phone industry has become more than a social need, it has become a part of the people's everyday life. With the growing market of companies that offer solutions in this area, the competition for fast and quality work is fiercer than ever. Having that in mind, finding a way to easily meet the needs of the users as well as doing that on a fast clock is more important than ever, and so, a quest to find a better and easier way to develop software has been taking place more frequently. The internship that will be reported here, aims to aid in that direction, enabling developers to easily create small pieces of software that can be downloaded and used by the users of a certain application in an on-demand manner.

The present work is a report for the internship that took place at WIT Software from September of 2015 to June of 2016.

WIT Software is a company in the mobile development business and it has been around since 2001. The company focuses mainly on creating white-labeled products for the mobile telecommunications industry and its main product is RCS+ which is a mobile application that allows users to communicate through chat and calls.

This internship aimed to find a solution for creating and executing JavaScript plugins in an Android application. The solution should be a separate module that, integrated with an application, would provide ways to install, manage and execute at runtime these plugins in a native way. The module should also ease the communication with a web app store, where the user could download and upgrade plugins at any time.

During the internship, all the tasks were done following a SCRUM methodology. Meaning that an initial product backlog was created with all the user stories that needed to be implemented for the end result. This was done in conjunction with the client, which in this context was WIT Software. Then, following an initial sprint meeting, the tasks/user stories for the next sprint were chosen. The sprint is a fixed period of time where the team focuses on finishing the chosen user stories, and, in this case, it had a duration of two weeks. After each

sprint, another sprint meeting was attended to discuss what was done during the spring, what could have gone better or worse and how to improve. During these sprint meetings, some more user stories were chosen from the product backlog to the next sprint. The team was, in this case, composed only by the author of the present work.

During the implementation phase, the process was closely followed by the inner tutor from WIT Software who supported the trainee.

# *Technology assessment*

The present work aims to find what is the best way to execute plugins written in JavaScript in Android apps. The chosen solution was to implement a framework to manage the execution of those plugins. Two distinct ways of doing that were discovered: using WebView's or using a JavaScript engine. The first method was quickly discarded since the performance is far away from native functionalities and, since there is already a base app in which the plugin will be run, this solution would not work as pretended.

After an investigation about engines compatible with the platform, for names were discovered: "SpiderMonkey", "JavaScriptCore", "Rhino", and "V8". Unfortunately, "Nashorn" is incompatible with Dalvik's virtual machine used by the Android operating system. [1]

While investigating that, other available products that also made their choices for a JavaScript engine were determined. Even though some of these products were not usable for this work, they helped to understand some tendencies that exist in their engine choices. In the below table, some of these products can be seen: [2][3]

**Table 1 - Framework and chosen engines**

| Framework | Android chosen Engine |
|---|---|
| ngCore [4] | V8 |
| Titanium [5] | V8 or Rhino |
| Tabris.js [6] | V8 (J2V8) |
| CocoonJS [7] | JavasScriptCore |
| Cocos2D-x JavaScript [8] | SpiderMonkey |

Some of these frameworks chose the same engine to support other platforms, and therefore, their choices might have been influenced by that fact. However, a certain tendency for the Google's "V8" is very noticeable.

All above engines except "Rhino" are written in C++, which means that an implementation of "JNI" (Java Native Interface) would need to be implemented. This would allow the communication with that engine through Java code, used in the Android app. The J2V8 framework presents this kind of implementation "out of the box", providing certain parts of the V8 API to Java using a JNI layer. These three engines, the most promissing was V8 which is already used in Android for Google Chrome and has better performance as seen in the next diagram.

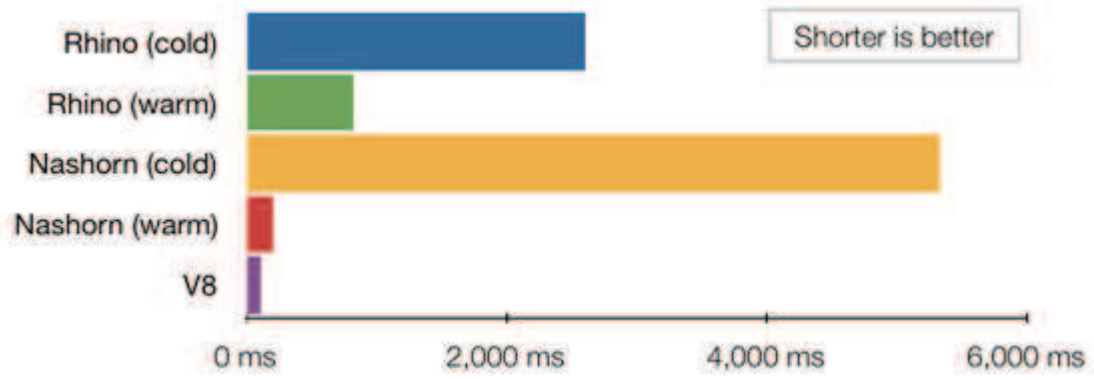


**Figure 1 - Performance in different JavaScript engines [9]**

The "Tabris.js" development team even refers that they first began using "Rhino" and that in certain animations they were getting only 10-15 FPS (Frames per second) and after changing to J2V8 they managed to get 60 FPS. [3] They also tested a script that produces the first one hundred fibonacci numbers. Using "Rhino" it took around seven and a half seconds and with "J2V8" it took around one second. [1]

Having this in mind, it was decided to test in more detail these two engines with the goal to find out the main differences between them and, lastly, choose one of them. In order to do this, some applications were developed to test the performance of simple JavaScript scripts that would create native visual components. Times were measured to see how both engines would do executing a specific task. The results of these tests can be accessed in annex A. From these results, one can conclude that the heaviest tasks in the integration between Java and JavaScript give a small advantage to "Rhino", since this does not require the JNI layer. However, the heaviest tasks in JavaScript execution itself give a clear vitory to "J2V8" engine. In addition, tests to memory consumption were done, setting up 10 running plugins. In

these tests “Rhino” used much more memory (~14mb) than the memory used by the application that used “J2V8” (~4mb).

The engine “Rhino” is written in Java which makes it very easy to implement in Android. However, its API is not very elegant, and its development is almost inactive which makes their use potentially unstable. In addition, the documentation is incomplete and out of data on certain points. When using the API “Rhino” some several difficulties were also found such as the lack of documentation on how it functions.

The API “J2V8” is a set of Java bindings for the engine “V8”, and so it also becomes easy to implement. These API is on active development and takes advantage of development and improvements made by Google on the engine itself. Its documentation is more complete and updated, which made the implementation of certain difficult features to be easier solved. As mentioned earlier, the performance of the two engines also gives a clear win for “V8”. Also worth mentioning is the fact that the final size of the application is a win for “Rhino” which needed 5mb less than using “J2V8”.

In the end of all the tests, two applications using the same executing plugins were successfully developed, one for each API. Despite the application size, it was concluded that the API “J2V8” would be the most reliable for the development of plugins management framework both for its documentation and its frequents updates.

## 1.1 Used technologies

*PROGRAMMING LANGUAGES*

- Java (with Android SDK)

Imposed by the client since the objective was to build a framework for Android apps to consume. It was used to implement the framework itself and to integrate it with the client’s app.

- JavaScript

Used to implement the JavaScript plugins that will be installed and run within the client’s app. These plugins will consume some methods that the framework registers to each JavaScript runtime instance.

*ANDROID LIBRARIES*

- J2V8

Java library that allows java applications to run JavaScript code in the Google's V8 engine as well as register callbacks for the JavaScript runtime and calling certain functions in a JavaScript script. When compared to similar solutions it gave the best results in the executed tests.

- nv-websocket-client

Java library that implements a WebSocket protocol for Android clients. It allows the creation and management of web sockets to connect to a web server. Chosen due to its simplicity, active development and low size.

*OTHER TOOLS*

- Android Studio

Official and widely used IDE for developing to Android.

- Redmine

Free online tool to manage tasks in a Scrum methodology

- SVN

Used to keep track of changes and to commit them to a web repository.

# Architecture

In this chapter the architecture of the solution will be discussed in two perspectives: general and framework specific.

## 2.1 General

Given the objective of promoting the ability to execute JavaScript plugins in Android apps, the chosen solution was to implement a framework that, integrated with certain app, would ease the execution and management of those plugins as well as the communication with a server to download the plugin source files and communicate with other plugin users as can be seen in below figure.

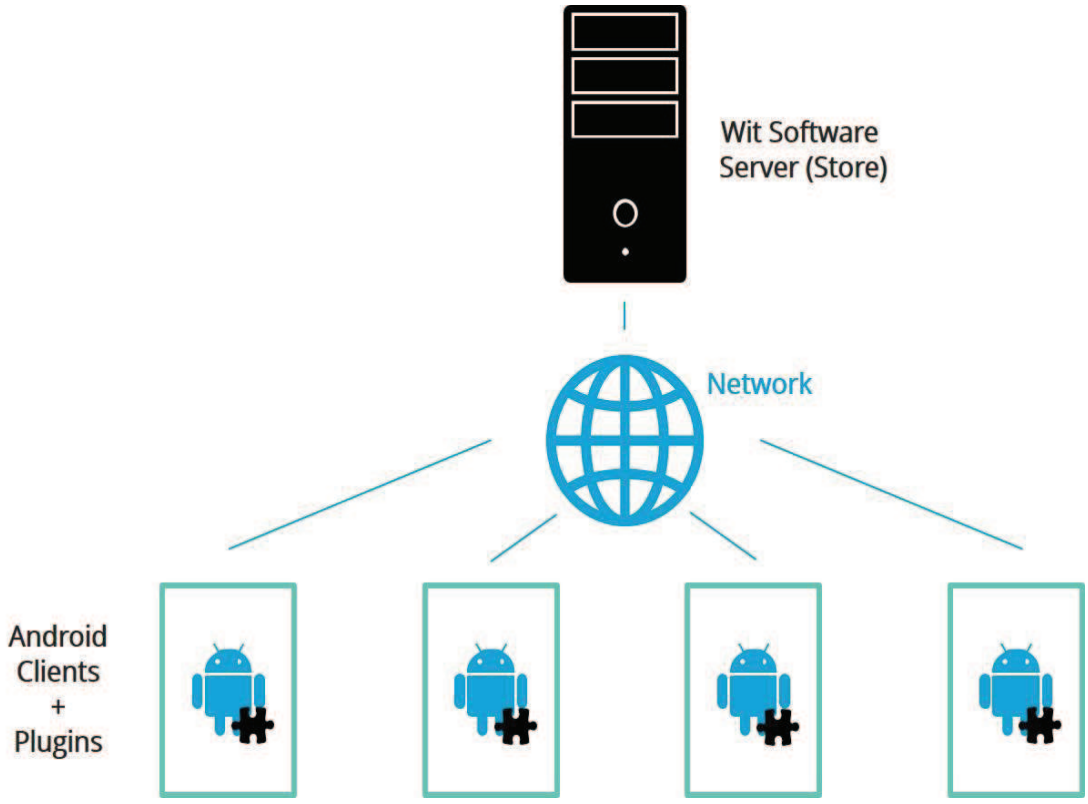


**Figure 2 - Plugins framework general architecture**

This communication between plugins can be made in two different ways: RCS File Transfer or WebSockets.

RCS File Transfer is a file that is sent between users with a certain MimeType attached. The app of the receiver will detect that the file received has the MimeType correspondent to a plugin message and transmits that message to the plugin responsible for treating it. This type of communication is used by the host app (RCS +) and is registered in the framework under the form of an API that the plugins may consume. The communication uses the mobile numbers of the users to determine who is to receive the message.

WebSockets are a way of communication directly between plugins without the host app receiving any notification of a message received. To do that, a server in which one can create a session sending a HTTP request was used. After the session is created, it gets shared with the other plugin user, using the type of communication explained earlier. After that initial handshake, the plugins communicate through WebSockets sending messages to one another.

This way, the developer of the plugin may choose the type of communication that better suits its needs.

## 2.2 Plugins

In this context, plugins are small pieces of software coded in JavaScript. They contain the following set of file structure:

```
Plugin/
  |
  |----- js/
  |
  |----- resources/
  |
  |----- boot.json
```

As one can see, plugin is composed with two folders and one configuration file. The “js” folder is where the plugin has its JavaScript code with the main function. The “resources” folder has everything needed by the plugin that is not the code itself, such as images or other configuration files. Lastly, the “boot.json” file is where the plugin specifies the files it

contains plus some other configuration parameters. The possible parameters are:

**"id"** - plugin's id;
**"author"** - plugin's author;
**"title"** - plugin's title;
**"version"** - plugin's version;
**"description"** - plugin's description;
**"icon"** - name of the image file inside the "resources" that is the icon of the plugin;
**"platforms"** - array of platform names supported by the plugin, example: android, ios, web;
**"extensions"** - string array of possible extensions that the app can send to the plugin's main method that will determine how it will behave;
**"media_types"** - if the plugin expects to receive messages of a certain MimeType, these should be placed here as well as the function that handles a message of that type;
**"js_files"** - array of files present in the "js" folder;
**"languages"** - default language and supported languages by the plugin;
**"main_function"** - main function name;
**"requires"** - array with the names of APIs necessary for the execution of the plugin.

This file is used by the "Engine" (explained in detail in chapter 3.2) which will read it and initialize the "BackgroundPlugin" with its parameters.

The framework uses two types of plugin classes: "BackgroundPlugin" and "ForegroundPlugin". "BackgroundPlugin" is initialized once and uses a background thread to operate. The app will make its requests and get answers from this instance. This plugin remains executing in the background until the app asks the framework to stop the plugin explicitly.
Since its initialization, the app may call the main function specified in the "boot.json" and pass to it an extension and an object. This object is usually associated with a message or specific user of the app, but it can contain all kinds of data important to the plugin.
The "BackgroundPlugin" instance can decide it's time to request for its own User Interface. When that happens, the request is sent to the app which will then prepare a view where the plugin will build the UI components in. The framework will pass that view to a new plugin instance: "ForegroundPlugin" calling the function that the "BackgroundPlugin" registered to

create the UI. This plugin has its own runtime and operates in its own thread, which this time is the main thread the app is running on, since it will create and change UI components. Unlike the other instance, this plugin will only remain active while the UI is visible.
These two instances of the same plugin work as separate entities but they can invoke functions from each other. This way, since the J2V8 limits calls to a certain runtime to the thread who created it, the UI gets created in the main thread and all the rest of the plugin's work runs in a background thread.

Besides these two classes, which are subclasses of "EnginePlugin" and represent running plugins, there are other classes that describe a plugin in the framework: the "StorePlugin" and the "LocalPlugin" which represent a store plugin and an installed plugin, respectively. All these classes are also subclasses of an abstract class named "Plugin" which has all the common behavior.

## 2.3 API

The framework was implemented aiming to be easily integrated and used in an Android's app. The API has a Fluent Interface [10] principle and the main point of communication is a component named "PluginManager". From this component a developer can make calls to different parts of the framework to install plugins from certain store, manage previously installed plugins and manage plugins that are currently running or about to be. Each of these "parts" is mapped with a method which returns a new instance of a new "Manager" that has a smaller portion of framework methods responsible for managing that specific part. A typical call to the API would, for instance, be:

```
PluginManager.local(context).uninstall(pluginId);
```

This call would uninstall the plugin with id "pluginId" if this plugin was installed in the device. The first part of the expression is used to initialize the "LocalManager" which contains methods for managing installed plugins or the installation itself. This way the developer has just one component of entry to the framework but, from that, he can invoke different managers of the framework in a clean and clear manner. The developer can also use two other "managers" like this:

PluginManager.store(storeUrl)....
PluginManager.engine(context)....

These two “managers” are in charge of communicating with the store (“StoreManager”) and managing the running plugins (“EnginePlugin”) respectively.
Note that all the “managers” need a specific parameter, the Android “Context” in the case of the “Local” and “Engine” managers, needed to make disk operations or to create views, etc, and the store URL in the case of “StoreManager” to which the connection will be made.

## 2.4 Components

The main component in the framework is called “PluginManager”. This is the base for all others and is the component that distributes “work” between the rest. On top of this there are three other components that act as sub parts of the first: “LocalManager”, “StoreManager” and “EngineManager”, as seen in below figure.

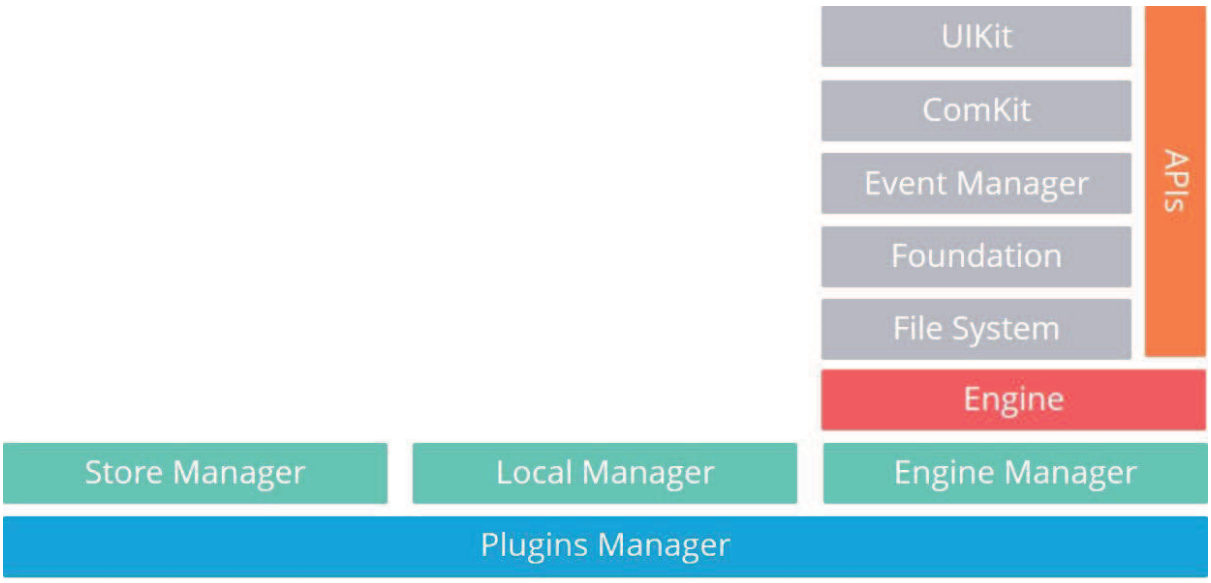


**Figure 3 - Plugins framework architectural components**

The “LocalManager” does disk operations related with plugins and uses a local SQLite database with the currently installed plugins. This component also deals with the “LocalPlugin” instances which are instances with information about certain installed plugin.
The “StoreManager” will make all connections to certain plugin store and return “StorePlugin” instances. This “manager” also uses components to parse the server response

and turn it into instances usable in the code.

Lastly, the “EngineManager” which starts operations related with the functioning of installed plugins. This “manager” directly communicates with other very important component in the framework, the “Engine”, which has a Singleton architecture. The “Engine” has a very important role in the framework, since it starts installed plugins, registers APIs in them and controls their lifecycle.

When the app asks for a plugin instance the first time, the plugin gets initialized by the “Engine”. This will then save that instance in a cache, where are all running plugins, and, if the app asks again for the same instance, that saved instance is returned. This “BackgroundPlugin” instance has a certain JavaScript runtime associated that operates in a single background thread. This runtime corresponds to a “V8” (J2V8) instance where all the callbacks to Java are registered. A set of those callbacks with the same “theme” of operations they can do is called an API. When the plugin gets initialized, the “Engine” will register all available APIs that are requested by the plugin’s “boot.json” file. There are some APIs that get registered by default in all plugins, even if not explicitly asked by the plugin itself.

This way, the JavaScript plugin can make calls to these APIs to create and manage UI elements, HTTP connections, etc.

The framework itself has the main APIs available for plugins to use in their code:

**“UIKit”** - API to create and manage user interface native Android elements;

**“ComKit”** - API to make HTTP or WebSocket connections to a server;

**“Foundation”** - registers methods usually present in a JavaScript develop environment like “setInterval”, “alert”, etc;

**“EventManager”** - default API that allows the plugin to register to a certain system event, such as low battery, headphones plugged, etc and then be able to respond to those events.

**“FileSystem”** - default API through which plugins can create and manage their own files in a sandboxed way, i.e, they will only able to do it inside a certain root directory.

Some of these APIs are internally divided into “sub APIs” in order to better organize and the framework code. When an API gets registered, all its sub APIs will also be registered in the runtime.

Besides this APIs present in the framework, the app can also easily create and add an API to the framework with specific functions. To do that, the developer should create a subclass of the abstract class “Api” and register functions to the plugin’s runtime.

Each method registered in the runtime is stored on a hashmap of the J2V8 [11] framework, and so is associated with a cost. Some APIs create Java objects, associate that object to a JavaScript one and register in this last one a set of methods required to manage it. This way, the plugins can create and manage Java instances. However, this can easily get out of hand with a number of methods registered in the order of thousands which may be consuming way too much memory for an Android device and even be the reason for some unpredicted memory leaks. To solve this, each API that can create and manage instances like this would only register one set of static methods once, meaning that when a plugin creates a new instance the Java method will return the id of that instance instead of a new JavaScript object with all the functions inside it. Then when the plugin wants to manage a specific instance, it will call the static method desired and pass the returned id and all the parameters necessary to make a change. This way, the number of methods registered is always the same, no matter how many instances the plugin has created. However, with this methodology the plugin's code will lose the notion of instances. To solve this, new JavaScript APIs were created that will call the Java methods. This way, these JavaScript APIs create JavaScript objects that hold the id returned from Java, and will automatically include that id when the plugin is managing that instance, keeping the notion of instances and ease with which plugins can create and manage their elements.

This concept was then taken a step further on the "UIKit" API. Since this API had a lot of similar methods which would take an id of the element to change, the name of the attribute to change and the parameter to change it to, it was created a general static method that would receive all three of these arguments and make the change. This way the number of methods registered in the runtime was reduced from hundreds to only one without increasing the complexity for plugins, since the JavaScript APIs encapsulate it, giving the plugin's developer a final set of functions to manage the instance.

# *Implementation*

In this chapter the framework will be discussed in the matter of the implementation and decisions that took place.

## 3.1 Managers

The managers are one of the key components of the framework since they provide the way to communicate with it in a easy and clean manner. The way the framework is designed, the app will only communicate with one manager: the “PluginManager”. This manager is the base for the other ones and does not actually do any work. It instead delegates work to a different manager regarding the type of operation the app is asking the framework to do. There are three other managers in the framework: the “StoreManager”, “LocalManager” and “EngineManager”. All these managers are plain old Java classes that will be instantiated when the app requests for an operation that concerns them.

This part of the framework was designed with the Fluent Interface model [10], which means the app has an easy and almost literate way of calling for an operation to the framework. If the developer uses this line of code:

PluginManager.local(context)...*

What is actually happening is an instantiation of the “LocalManager” by the “PluginManager” that will pass the given context to the “LocalManager” constructor and return that new instance. From there the developer can complete the statement with one of the public methods of the “LocalManager” which turns out this way:

PluginManager.local(context).install(storePlugin);*

*note that the “context” is the current Android Context [12] and the “storePlugin” is an instance of the class “StorePlugin”.

If instead the app wants to perform a connection to the plugins' store it should use the following:

PluginManager.store(storeUrl)...*

*note that the "storeUrl" is the url to make the connection to.

Which will create an instance of the "StoreManager" and the developer can keep going through the method calls until he reaches the end of the statement. Once again the only methods available to the developer at this point are the methods regarding the operations to the store.

And lastly, if the developer wants to manage the running state of installed plugins he should write:

PluginManager.engine(context)...*

*note that the "context" is the current Android Context [12]

This will create a new instance of the "EngineManager" class and, once again, the developer will have available to him a number of methods to manage the running state of plugins.
This encapsulation made it easy to keep the code well organized, clean and keep the API fluent.

### 3.1.1 Local Manager

This manager is responsible for operations that take place in the disk of the device. This includes installing and uninstalling plugins, enabling and disabling plugins, return instances of installed plugins and update plugins. It accomplishes those things by using a local SQLite database [13] in which it adds new entries when new plugins are installed and removes them when they are uninstalled. When an operation to install a plugin starts, the "LocalManager" will first verify it the plugin is already installed and if it isn't then it begins to download all

the plugin files to the internal storage of the app in a specific folder where all the plugins are. After that, if the operation was successfully completed, it adds an entry in the database saving some data about the plugin. If the developer only wants to disable certain plugin or enable it then the manager will simply change that field in the database in the correspondent entry. All the operations to the database are made through a data source component named "PluginsDataSource" which encapsulates all the queries made to it in a simple method call.

### 3.1.2 StoreManager

The "StoreManager" is responsible for making connections to the plugins' store, parsing the answers and return Java instances of "StorePlugin" class. It is capable of parsing both XML and JSON formats and creating those instances using a factory class that receives a content type of the answer and creates a difference instance of parsers for each type. The connection itself uses the HttpUrlConnection [14] provided by the Java language. It can make two types of connections: store content or package. The first retrieves list of the available plugins in the store with the URL needed to make the second connection, which results in a single "StorePlugin" instance.

### 3.1.3 EngineManager

This component is responsible for starting, stopping and enquiring the "Engine" for the currently running plugins. It also returns an instance of the interface "RunningPlugin" (implemented by the "BackgroundPlugin" class) which then lets the developer communicate directly with the plugin itself for doing some limited operations like calling its main method and asking it to refresh certain view with new data. Besides this, it is also the component where the app may want to register callbacks for when the plugin asks for its own UI. These callbacks are methods of an inner interface of "PluginManager" named "OnUIChangeRequestedListener". All other operations are delegated to the "Engine" component that will execute them.

## 3.2 Engine

The “Engine” is a core and very powerful component in the framework that is responsible for executing operations regarding the execution of plugins. This includes starting and stopping a plugin, getting all the currently running plugins, adding new APIs to the list of available APIs, and invoke methods between foreground and background plugins. This component is meant only for internal use, and the developer should stay away from using it directly, instead he should use methods of the “EngineManager”. It contains a reference to another component: “PluginCache” which saves all currently running plugins. Each plugin is only started once, and if the app needs an instance of a plugin already started, then that is returned. Besides this, it also contains an instance of the “AvailableApis” class which has all the different APIs that are available to the plugins either from the framework itself or APIs that the app has defined and added to the framework.

When the “Engine” is ordered to start certain plugin, it will verify it the plugin is already in the “PluginCache” and if it isn’t then a new instance is created and saved there. The initialization process of a plugin consists of reading the “boot.json” file and the source code of the plugin, creating a new instance of the “BackgroundPlugin” class with the read data, starting the runtime of the plugin and registering the required APIs to that runtime. At this point the plugin is started and placed in the “PluginCache” but no code is actually run except if the app requests it or if there is some new message or event that makes the instance call a specific JavaScript function.

The stopping of a plugin is much simpler, the “Engine” only removes the instance of the “PluginCache” instance and calls the “stop()” method of the plugin itself verifying that the plugin is in a state in which it can be stopped.

If the app has its own APIs meant for plugins, then they can register them in the framework. When that happens the “Engine” adds that Class object and its name to the “AvailableApis” instance. Then when plugins are initialized, if they require an API with that name, it gets registered to that runtime.

At any point in its execution, the plugin can invoke functions in its foreground/background

sibling. In order to invoke methods from different runtimes, the “Engine” must first parse the JavaScript array of arguments to a JSON string and then call the other function parsing it again in the new runtime and calling the function with that array of parameters. This parsing is done using the “JSON.stringify()” and “JSON.parse()” methods of the JavaScript V8 runtime.

These methods are actually declared in the “Foundation” API of the framework, but they are executed in the “Engine” which is the only component with knowledge of all the active running plugins.

## 3.3 Plugins

Plugins are entities in the framework that describe a certain piece of software that can be run within the host app. There is an abstract superclass named “Plugin” from which all the plugin classes inherit and which has the fields common to all those subclasses: “LocalPlugin”, “StorePlugin” and “EnginePlugin”. The “Local” and “Store” classes add some fields that help describe a plugin installed or in the store, respectively. The “EnginePlugin” however, adds a lot of behaviour to the superclass and it is also an abstract superclass from which the “BackgroundPlugin” and “ForegroundPlugin” inherit. This plugin adds some other fields and some methods to help the “Engine” manage its lifecycle or the APIs make changes in the plugin’s runtime. When the “Engine” starts a plugin, a new instance of “BackgroundPlugin” is created and the method “start()” is called on that instance. This method will create a new JavaScript runtime where the “Engine” will register the APIs and the plugin will run. At this point each subclass will create its runtime in a different way depending on if its a “BackgroundPlugin” or a “ForegroundPlugin” since both override the “start()” method. The major difference is the “BackgroundPlugin” creates the runtime on its own thread, and so it has to make all the calls in that same thread. Besides, since the app will only communicate with the background plugin, the “ForegroundPlugin” doesn’t need the methods for calling functions in the runtime.

With all these classes the framework can better describe and work with plugins in different contexts.

## 3.4 JavaScript Runtime

The runtime is where the JavaScript code will run. All the calls made to a single runtime have to be done from the same thread that initially created it. In the framework there are classes that encapsulate the runtime: "JSRuntimeImpl", "JSRuntimeForeground", "JSRuntimeBackground". The first is an abstract superclass from which the others inherit fields and behaviour. All of them contain the JavaScript runtime itself (V8 class from J2V8 [11]) and some methods that help making calls to it, ensuring that they are done from the same thread. Each running "BackgroundPlugin" will have an instance of "JSRuntimeBackground" each with its own thread. When the "Engine" starts a background plugin, the "JSRuntimeBackground" is created, which creates an instance of a "HandlerThread"[15] and when the "Looper"[16] in that thread is ready, the V8 runtime is created in that thread. After that, a "Handler"[17] is saved in the instance, to help make calls in the same thread. This "HandlerThread" is a specific type of thread that prepares its own "Looper" and after executing certain task it remains alive waiting for other tasks or messages to process or for someone to call "quit()" method on it. This behaviour makes it perfect for this case, since there was the need to keep a reference to the thread that created the runtime in order to make calls from it. When this plugin requests for its own UI, an instance of "ForegroundPlugin" is created which then initializes a "JSRuntimeForeground" with a handler for the main thread. Since this plugin operates from the main thread, that handler is used to make sure all calls are made from that thread.

This way, all the APIs can make changes in the runtime without worrying about what thread they are making it from or which plugin it is (foreground or background) they just make calls to the "JSRuntime" interface they have and the correspondent implementing class will deal with the call to the V8 runtime itself. This calls include everything since creating JavaScript objects, calling specific functions or registered callbacks to retrieving properties from given JavaScript objects.

## 3.5 JavaScript APIs

These APIs are what expose some Android or app specific classes and methods to the plugins so that they can make user interfaces, HTTP connections and whatever the app registers on

the framework. The framework has some APIs itself that allow plugins to do the more important operations. In the time of a plugin initialization, the "Engine" registers the APIs required by the plugin's "boot.json" that he has available in the "AvailableApis" class. This class has a hashmap where the keys are the API name and the values are the API "Class" itself. This Class object is then used to initialize the API and register its methods in the runtime using reflection [18]. Doing this allows the app to add its own APIs to the plugins that will run, achieving a completely extendable framework.

In the framework, there is an abstract superclass "Api". It has a constructor which receives the "EnginePlugin" class where the API will be registered and an abstract method that will make that registration. When the "Engine" initializes a plugin, it calls this constructor and this method. After that, each API is responsible for registering the methods in the given V8 runtime. The full process can be seen in detail in the following sequence diagram.

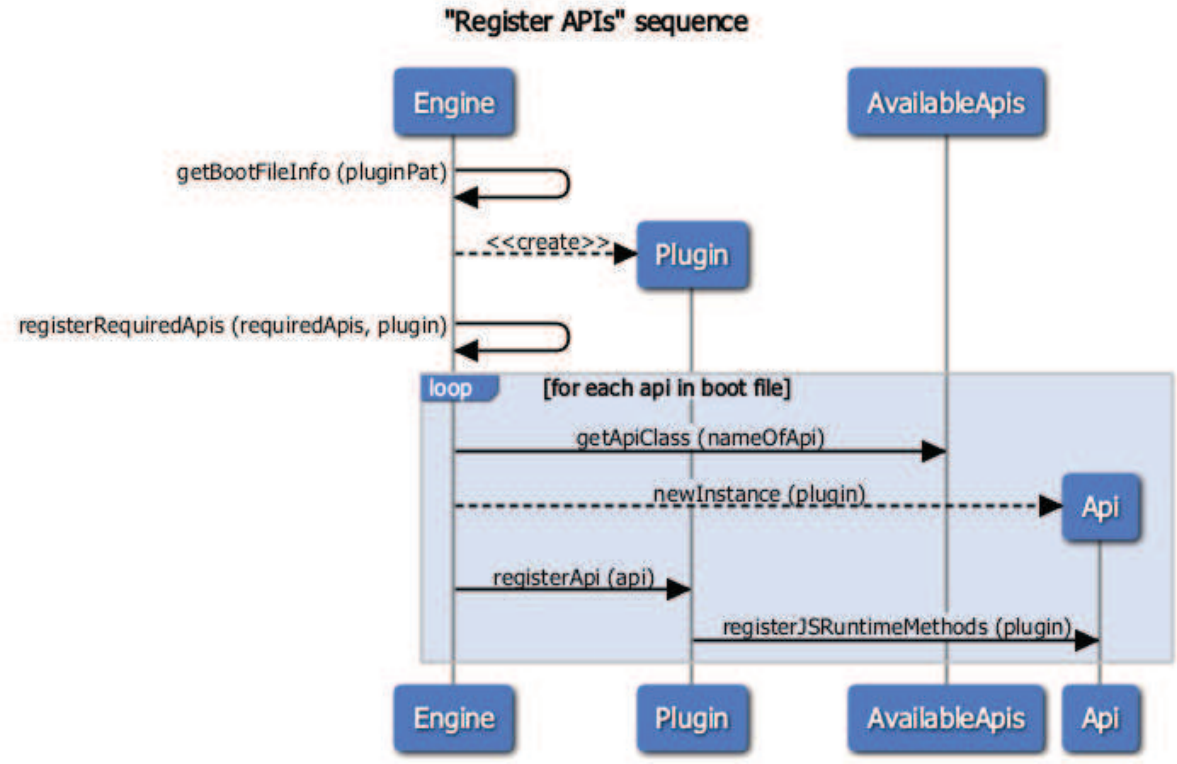


**Figure 4 - API registration sequence**

These API instances will not be saved anywhere in the framework's code, they will only be present in a "MethodDescriptor" used by the J2V8. This means that their methods can only be called from the plugin's code itself (since we don't hold any reference). There are however some exceptions to this, since there are some APIs that might be interested in allowing other APIs to interact with them. To achieve this, there is a Java annotation meant to be used in

the top of the class definition called “SingletonApi”. This annotation requires a function that returns the name of the method to get the instance. This method should receive the plugin as the single parameter, since these APIs are only singletons within the plugin, i.e., each plugin has its own instance. To make this happen, when the APIs initialization happen, the existence of this annotation is verified, and the creation of the API is different in case it exists. Since this APIs save the instances on a static field, they should remove them in case the plugin they are associated with stops (to avoid memory leaks).

## 3.6 JavaScript Helpers

In order to make the calls to a certain JavaScript runtime easier and encapsulated, there are four Java classes in the framework: “JSFunctionCallBuilder”, “JSObjectClassRegister”, “JSObjectCreator” and “JSObjectValueHelper”.

The “JSFunctionCallBuilder” is the most used and the most useful since it allows APIs to call functions in the runtime in a easy and clean manner. The code for calling functions using J2V8 can be hard to read and maintain, and so, by using this helper, the APIs can generate function calls with parameters much cleaner and no matter the thread they are on. This helper takes the runtime has a parameter, and so it can use its handler to call the functions directly on the thread correspondent to that runtime.

The “JSObjectClassRegister” has a single method that helps certain APIs to register their methods. It does this by taking the class that is going to be registered, reads each method from it reflectively and registers in the runtime. Normally when registering methods in the J2V8 runtime, the developer would need to explicitly say the name of the method and their parameters, which can be very hard to maintain. This way the registration process will follow the code changes, the developer does not need to change the name or parameters of a method registration each time he makes alterations to code.

The “JSObjectCreator” and “JSObjectValueHelper” are both for facilitating the work to create V8Objects as well as getting values from them.

# *Tests and Results*

In this chapter, the tests that were made and the resulting plugins and applications will be discussed.

## 4.1 Integration with RCS+

The framework was tested by integrating it in the client's communication application. The application is called RCS+ and it allows the user to communicate with their phone contacts with the RCS protocol.

Besides importing the framework as an Android library, some other changes were made to the application to allow it to install and manage the available plugins. When the application starts, it initializes the framework registering a set of callbacks for certain plugin's requests such as the request for its own user interface. It will also register a set of APIs that allows interested plugins to use certain features of the application, like its file transfer communication.

The application has a chat feature where the plugins may be started. The installed plugins that have the "chat.share" extension will appear in the share menu of the application. From there, the user can start it and share it with their contact. Below, there is a figure that shows a chat screen of the application (left), and an example of a share menu (right).

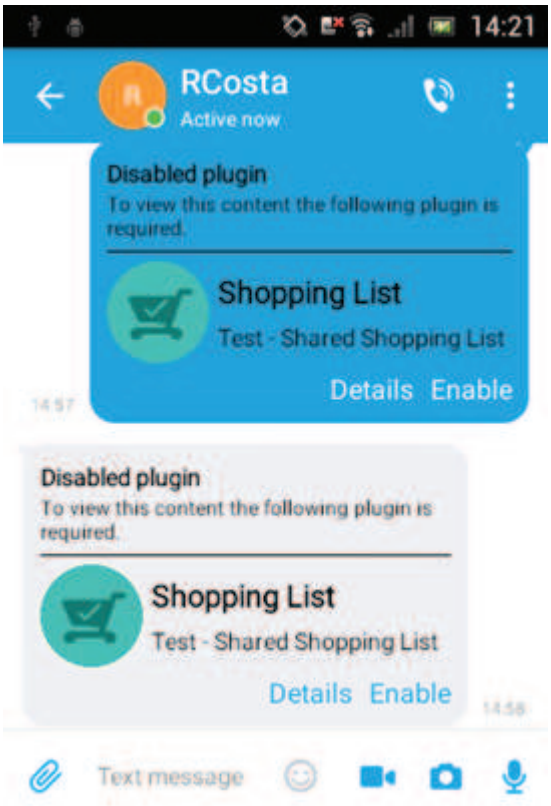


Figure 5 - Chat screen

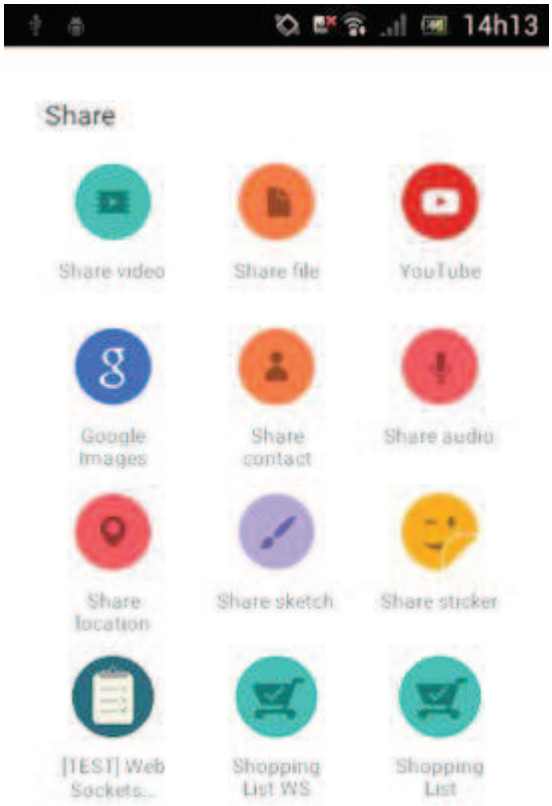


Figure 6 - Share menu

In addition, the application also has a “Settings” menu, where a “Plugins” entry was added. From there, the user is able to manage his plugins as well as install new ones. These menus can be seen in the following figures: on the left side there is the “Settings” menu, and on the right the “Plugins” menu showing all plugins available.

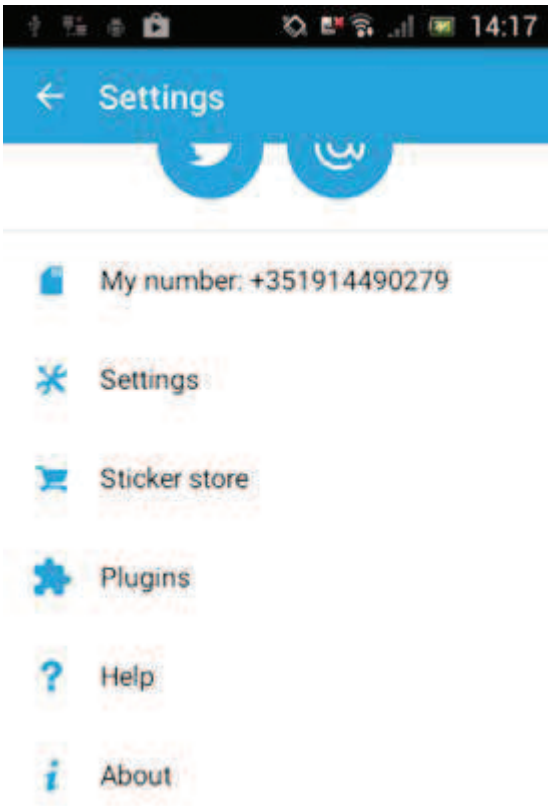


Figure 7 - Settings menu

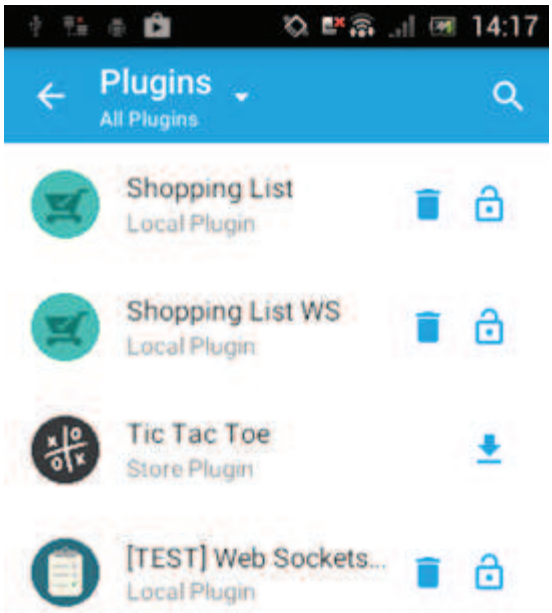


Figure 8 - Plugins management menu

Once the user is in the menu illustrated by Figure 8, he can press on any plugin and he will enter the screen shown in below figure.

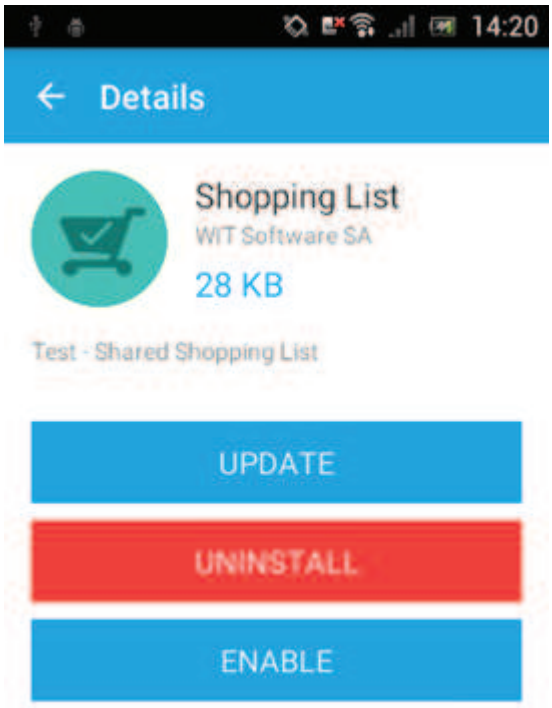


**Figure 9 - Plugin management screen**

## 4.2 JavaScript Plugins

In order to further test the developed framework, it was decided to create some JavaScript plugins that could take advantage of the provided features. These plugins helped creating a real scenario, and thus, were very important to improve the framework. Throughout its development they helped detecting bugs and other issues with the framework, for example, the problem with the huge memory hole present in the way the J2V8 library was being used.
They also served as examples to show results to the client, as well as show improvements as they were being made.
A plugin consisted of a set of JavaScript files and assets. The application could start a new instance of a certain plugin any time, and then send events to it through the framework. While responding to these events, the plugin may choose to start its own UI interface, which would then ask the application to open a new Android Activity or Dialog depending on what the plugin wants. The user interactions with this UI trigger events on the plugin which acts accordingly.

Two plugins were developed: the “Shopping List” and the “Tic Tac Toe”.

### 4.2.1 Shopping List

This plugin allows the user to create and send a shareable shopping list to a friend. This list automatically updates whenever one of the users changes anything like adding or removing items, marking them as complete and changing the name of certain item. The plugin saves the current state of the list in a local JSON file.

When the user starts the plugin, the screen with the list appears as a dialog, as can be seen in the following image.

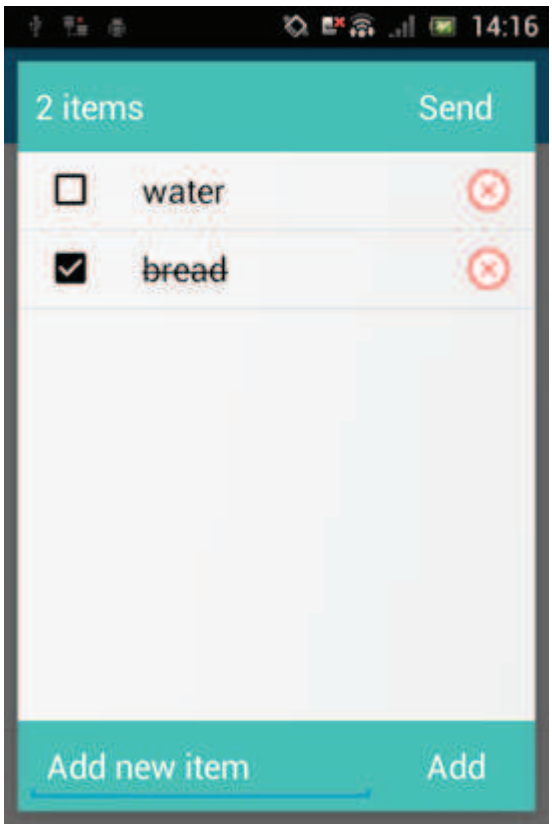


**Figure 10 - Shopping List main screen**

From here the user makes all changes he wants to the list. If his friend is also online, these changes will be sent by WebSocket to his plugin that will receive events and update the list. If his friend is offline, then they will synchronize both local lists merging them once they are both online.

### 4.2.2 Tic Tac Toe

This plugin is a game where one user plays against a friend to try and beat him in the classic game Tic Tac Toe. It will save the results between both users and show the number of ties, wins and losses.

The plugin's main screen is where each player will take their turn, and it looks as follows:

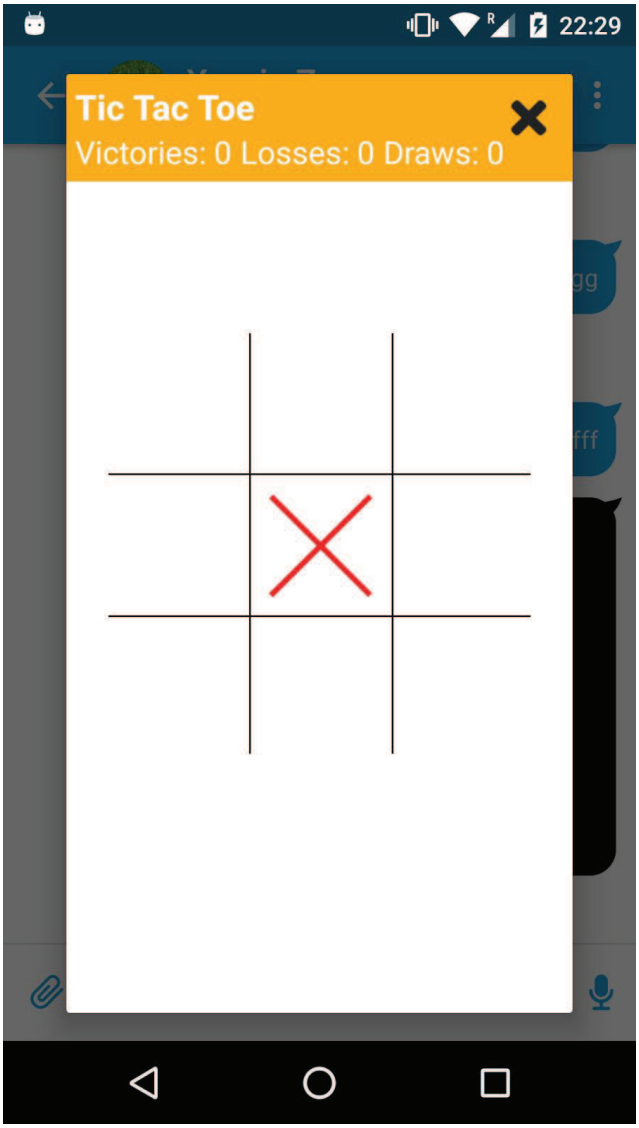


**Figure 11 - Tic Tac Toe main screen**

Once a player begins a game taking the first turn, his friend will receive an invitation through the File Transfer of the client's application. The transferred file has information such as the

URL for the WebSocket communication and the game's id. When the user accepts the invitation, the players will take turns sending his play to the other user through the WebSocket session.

# *Conclusion*

With the ever growing smartphone industry, the need to be better and faster in the development of software has become a big priority for companies in this field. The present work aimed to aid in this direction, giving developers an alternative way to bring small features to a host application. This would also allow applications to have their main features available at install time, and provide a set of additional ones that a user would be able to choose and download from, making each user's application its own.

In the end of the implementation phase, an Android framework that allowed the host application to download and manage JavaScript plugins in runtime was developed, however this framework allowed plugins to use the very most basic Android native features. This means that in order to take this work further a reality, some serious work would have to be invested in enlarging the feature set of the framework as well as maintaining it.

There is still a lot that could be added to the framework as far as features go. If the plugins' requirements become more and more complex, this approach would take a lot of commitment to take forward. Due to the time frame of this internship only the more common features that a plugin would need were implemented and, all the other features will be considered future work. All the important and discussed features with the client were successfully implemented.

While trying to implement the framework it was clear that it was a constant struggle to "defeat" the Android SDK, i.e., trying to do things that it was not originally intended to do, and, despite the success on implementing them, one can conclude that the work to maintain such a framework on a production level would probably not be worth the effort.

However, if someone puts the work and manages to find a common ground for Android and iOS and implements an API on top of the JavaScript API designed on this internship, the results might actually be worth it, since from then on developers would only need to implement some features once for both platforms. It is however important to be aware that a

framework like this would never be able to be so flexible and capable as developing natively. The present work and the internship were very helpful to the trainee to dig deep into the Android SDK and JavaScript language. On a less technical look, it was very helpful to learn how things are done in a workplace environment.

# *Bibliography*


[1] – *Highly Efficient Java & JavaScript Integration* – October 8th, 2015 at:
http://eclipsesource.com/blogs/2014/11/17/highly-efficient-java-javascript-integration/
[2] – *Part I: How to Choose a JavaScript Engine for iOS and Android Development* - October 8th, 2015 at:
http://openaphid.github.io/blog/2013/01/17/part-i-how-to-choose-a-javascript-engine-for-ios-and-android-apps/
[3] – *Performance, Top Tabris.js Feature #6* - October 8th, 2015 at:
http://eclipsesource.com/blogs/2015/05/13/performance-top-tabris-js-feature-6/
[4] – *The Mobage ngCore: Getting Started* - October 8th, 2015 at:
https://docs.mobage.com/display/WWNGCORE/Getting+Started
[5] – *Appcelerator – Titanium* - October 8th, 2015 at:
http://www.appcelerator.com/product/
[6] – *Tabris.js* - October 8th, 2015 at:
https://tabrisjs.com/
[7] – *ludei: CocoonJS* - October 8th, 2015 at:
https://www.ludei.com/cocoonjs/
[8] – *COCOS2D-JS* - October 8th, 2015 at:
http://www.cocos2d-x.org/wiki/Cocos2d-JS
[9] – *Nashorn: The New Rhino on the Block* - October 8th, 2015 at:
http://ariya.ofilabs.com/2014/03/nashorn-the-new-Rhino-on-the-block.html
[10] – *FluentInterface: by Martin Fowler* - September 27th, 2015 at:
http://martinfowler.com/bliki/FluentInterface.html
[11] - *Getting Started With J2V8 by Ian Bull* - September 27th, 2015 at:
http://eclipsesource.com/blogs/getting-started-with-j2v8/
[12] – *Android Developers, Context* - September 27th, 2015 at:
https://developer.android.com/reference/android/content/Context.html
[13] – *SQLite, About* - September 27th, 2015 at:
https://www.sqlite.org/about.html

[14] – *Android Developers, HttpURLConnection* - September 27th, 2015 at:
https://developer.android.com/reference/java/net/HttpURLConnection.html
[15] – *Android Developers, HandlerThread* - September 27th, 2015 at:
https://developer.android.com/reference/android/os/HandlerThread.html
[16] – *Android Developers, Looper* - September 27th, 2015 at:
https://developer.android.com/reference/android/os/Looper.html
[17] – *Android Developers, Handler* - September 27th, 2015 at:
https://developer.android.com/reference/android/os/Handler.html
[18] – *ORACLE, JAVA Tutorials, Trail: The Reflection API* - September 27th, 2015 at:
https://docs.oracle.com/javase/tutorial/reflect/index.html

# *Attachments*

## Attachment A

J2V8

| | Start Time | Finish Time | Elapsed Time | |
|---|---|---|---|---|
| | ms since boot | ms since boot | ms | seconds |
| TESTE 1 | 10981337 | 11074723 | 93386 | 93,386 |
| TESTE 2 | 16014476 | 16046819 | 32343 | 32,343 |
| TESTE 3 | 16046819 | 16070863 | 24044 | 24,044 |
| TESTE 4 | 16567383 | 16567676 | 293 | 0,293 |
| TESTE 5 | 326086796 | 326102671 | 15875 | 15,875 |
| TESTE 6 | 327183349 | 327189681 | 6332 | 6,332 |
| TESTE 7 | 433596103 | 433600110 | 4007 | 4,007 |

| Teste1 | | | | Teste2 | | | |
|---|---|---|---|---|---|---|---|
| Fazer 10 000 pedidos a um servidor dentro de um ciclo. | | | | Fazer 10 000 pedidos a um loop dentro de outro loop com 100 teracoes cada | | | |
| Winner | Rhino | by | 2,098 | Winner | Rhino | by | 3,606 |
| | | | | | | | |
| Teste3 | | | | Teste4 | | | |
| Receber todas as respostas do teste2 | | | | 100 primeiros numeros fibonacci 10 000 vezes | | | |
| Winner | Rhino | by | 1,488 | Winner | J2V8 | by | 38,584 |
| | | | | | | | |
| Teste5 | | | | Teste6 | | | |
| 500 pedidos e por as respostas numa ListView | | | | Executar um plugin com 10 componentes de ui 50 vezes | | | |
| Winner | Rhino | by | 0,82 | Winner | J2V8 | by | 1,694 |
| | | | | | | | |
| Teste7 | | | | | | | |
| Executar 10 plugins em simultaneo | | | | | | | |
| Winner | J2V8 | by | 1,067 | | | | |

Rhino

| | Start Time | Finish Time | Elapsed Time | |
|---|---|---|---|---|
| | ms since boot | ms since boot | ms | seconds |
| TESTE 1 | 11577513 | 11668801 | 91288 | 91,288 |
| TESTE 2 | 15873218 | 15901955 | 28737 | 28,737 |
| TESTE 3 | 15901955 | 15924511 | 22556 | 22,556 |
| TESTE 4 | 16604277 | 16703154 | 38877 | 38,877 |
| TESTE 5 | 325740795 | 325755850 | 15055 | 15,055 |
| TESTE 6 | 327015097 | 327023123 | 8026 | 8,026 |
| TESTE 7 | 431435751 | 431440825 | 5074 | 5,074 |

**Attachment B – Tasks**

__Summary (Hours per Project)_____

#HoursProject
1525:17 Geral

__Summary (Tasks per Project)_____

Geral (100%)
+ First Day
+ Instalação e apresentações aos escritórios
+ Estudar o estágio proposto / Instalar software
+ Pesquisar sobre o trabalho / preparar apresentação
+ Preparar apresentação de 5-10 / Pesquisar sobre o tema
+ Preparar apresentação inicial / Pesquisar
+ Levantamento de Funcionalidades/Requisitos com Dúvidas
+ Reunião com orientador
+ Experimentar com o J2V8
+ Experimentar com o Rhino
+ Apresentação esquema da arquitectura
+ Experimentar com Rhino / J2V8
+ Buscar dados a um servidor J2V8 / Rhino
+ Multiple Threads, Javascript runtimes exprimentações J2V8
+ Preparar apresentação inicial de 5 de Outubro
+ Multiple Runtimes / Plugins J2V8 Tests
+ Multiple Plugins Rhino Tests
+ Multiple plugins Rhino tests
+ Remaking UI when plugin transictions to foreground J2V8
+ Rhino Multiple plugins & background
+ Rhino - Remaking UI when plugin transictions to foreground
+ Implement basic Javascript functions - J2V8 / Rhino
+ Prepare document - State of the Art

+ Prepare document - State of the Art
+ Write document - State of the Art
+ Finish document - State of the Art
+ Memory and performace tests - J2V8 vs Rhino
+ Memory tests Rhino vs J2V8
+ Preparing the presentation
+ Attending to presentations
+ Memory testes Rhino vs J2V8
+ Reading iOS plugin Framework documentation
+ Writting functionalities for the Android plugins framework
+ Spikes for testing some complicated functionalities
+ Reunião com o orientador
+ Investigating future user stories for possible problems
+ Trip to Coimbra
+ Meeting Sprint #0
+ Preparing presentation about SoA for Sprint #1
+ Trip to Leiria
+ Scrum workshop
+ Add user stories to redmine's product backlog
+ Estimating backlog's user stories
+ Estimating sprints' velocity
+ Reviewing backlog
+ Checking previously done Plugin Store Android App
+ Creating Android App Demo PluginStore
+ Meeting Spring #1
+ Adding sub tasks to redmine's user stories
+ User Storie: see list of available plugins in the store.
+ Retrieve the list of available plugins from the store.
+ Parse list of plugins from the store
+ Talking to advisor
+ Implement interface Parcelable in Plugin class
+ Junit tests
+ Investigating Fluent interface API's

+ Implementing Logger for the framework
+ Creating Javadoc for current US
+ Measuring Time of getting all the plugins of the store
+ Testing everything before commiting
+ Starting US Redmine #53976 - As a user, I want to see a full description of certain store plugin.
+ Getting full info of a plugin US.
+ Planning and starting new US - As a user, I want to be able to download and install a plugin.
+ Downloading plugin zip
+ Unzipping plugin file
+ Implementing a SQLite to save info of installedPlugins
+ JUnit tests for this US
+ Javadoc for current US
+ Sprint DoD tasks
+ Backlog's re-estimation
+ Sprint Meeting #2
+ US - List of installed plugins
+ US - As a user, I want to see a full description of certain installed plugin.
+ US - Enable/disable certain plugin.
+ US - Uninstall certain plugin.
+ Rethink about backlog's user stories.
+ US -As a user, I want to start a plugin.
+ US - As a user, I want to start a plugin.
+ US - List of running plugins.
+ US - Stop a plugin.
+ US - Run multiple plugins at a given time.
+ Workshop - Software Arquitecture
+ Sprint Meeting #3
+ SubTasks for sprint#3 US's
+ Feature #57484 - As a developer, I want the Engine to be able to respond to system events and notify interested running plugins.
+ Studying and refactoring the code.

+ Feature #58190 - As a developer, I want to be able to use console.log funcion in my plugins.

+ Feature #58188 - As a developer, I want to be able to use alert function in my plugins.

+ Feature #58191 - As a developer, I want to be able to use setInterval function in my plugins.

+ Feature #58194 - As a developer, I want to be able to use setTimeout function in my plugins.

+ Feature #58195 - As a developer, I want to be able to use clearInterval function in my plugins.

+ Feature #58197 - As a developer, I want to be able to use clearTimeout function in my plugins.

+ Feature #58198 - As a developer, I want to be able to use confirm function in my plugins.

+ Feature #59006 - As a developer, I want to unit test the foundation methods done so far.

+ Feature #59141 - As a developer, I want to be able to use prompt function in my plugins.

+ Projecting an api for file management.

+ Holiday

+ Sprint Meeting #4

+ Sub tasks for sprint #4

+ Feature #53956 - As a developer, I want plugins to be able to manage files in their own storage directory.

+ Feature #59640 - As a developer, I want plugins to be able to have their own User Interface.

+ Feature #53959 - As a developer, I want plugins to be able to create and manage buttons.

+ Feature #56361 - As a developer, I want plugins to be able to create and manage ListViews.

+ Spike - Test running multiple plugins in the UI thread.

+ Preparing Sprint #6

+ Fixing Unit Tests for UI Threaded plugins version.

+ Feature #53782 - As a user, I want plugins to be able to share data with other peers using Web Sockets.

+ Creating a solution for better building and testing multiple plugins.

+ Week of Report / Presentation preparation

+ Integrating plugins lib with rcs app

+ Feature #62970 - As a developer, I want to make shopping list extend the share dialog with its own button.

+ Feature #63316 - As a developer, I want to start the plugin action triggered by the chat.share extension click.

+ Mid term presentations

+ Meeting with advisor

+ Feature #63686 - As a developer, I want to refactor the framework to better suit the needs of the RCS app.

+ Preparing the mid term presentations.

+ Changing the way plugins can set attributes to their views.

+ Feature #64749 - As a developer, I want to have the possibility to add extensions (APIs) to the plugins framework.

+ Feature #64699 - As a user, I want to be able to send a shopping list item to a contact.

+ Feature - Show the new received shopping list as a plugin message in the chat.

+ Feature - Add click listener for when user clicks on plugin message view.

+ Improve balloon view refresh performance

+ Refactor, Javadocs.

+ Feature - Make plugins able to subscribe to chat events.

+ Feature #66572 - As a developer, I want to send the plugins data objects as RCS file transfer.

+ Feature #66558 - As a developer, I want the plugin to be able to receive message events in background

+ Feature - As a developer, I want to centralize all notifications to the plugins in the framework.

+ Feature - Creating a Sync FileSystem API.

+ Sprint Meeting #9

+ Feature - As a developer, I want to run all the "own UI" tasks of a plugin in the main thread, and the rest in a background one.

+ Wit Snow Trip

+ Feature - As a developer, I want to be able to display a plugin's UI in fullscreen and dialog modes.

+ Feature - As a developer, I want to improve the shopping list plugin's UI.

+ Preparing Sprint#10 - Review Presentation

+ Sprint Meeting #10

+ Bug - Fix memory leak in V8 runtime;

+ Feature #69323 - Set the plugins store up and running.

+ Sprint Meeting #11

+ Feature - As a developer, I want to enable plugins to make Http requests to a server and get the response.

+ Feature - Set the content store up and running.

+ Feature - As a user, I want to be able to install plugins from the store.

+ Feature - Plugin missing message.

+ Feature - Use CloudPlayRoom WebSockets to communicate between plugin users.

+ Sprint Meeting #13

+ Internship final Report

+ Android Intermmediate Course

+ Final testing - websockets Shopping List

+ Feature #73882 - As a developer, I want to merge my changes with latest stable RCS verstion.

+ Sprint Meeting #14

+ WIT Blue Friday

+ Feature - Keep a Shopping List balloon in the end of the chat to make the user aware of it.

+ Feature - Make internationalization possible for plugins.

+ Feature #53961 - As a user, I want to be able to update a plugin.