POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
MSc Computer Science & Engineering

POLITECNICO
MILANO 1863

Mobile Studio

A Framework for the Development of NativeScript Mobile Applications

Supervisor: Prof. Luciano Baresi

Thesis of the student:
Luigi Russo
Personal code (matricola): 836025

Academic Year: 2015/16

Acknowledgements

It was a long journey. This work is more than a simple document, coming at the end of my two-years Master of Science course and, in general, concludes my five-years long student history in Politecnico di Milano. Looking back, over my shoulder, I cannot see anything that could be described with the word "*easy*". Since the very first time I left my hometown, L'Aquila, and I came to Cremona and after when I moved in Milan, everything was difficult, complex, hard, also in non-university tasks. I wouldn't have managed to complete them successfully without the help of many people that believed in me and gave me strength to go on. My thoughts go immediately to people from Cremona campus, **L. Baresi**, **F. Renga**, **G. Ferretti**, **E. Vannutelli**, that gave me the opportunity to win a scholarship in mobile application development and still involve me in related activities in that campus, that I'm always happy to see again. I absolutely would not be able to cope with any difficulty without the help of my friends, new and old ones, particularly to the furthest ones in L'Aquila that never stopped to support me, although I was able to see them only few times in these years.

Nothing of this could be achieved without the help of my parents, and in particular my mother **Maria**, that never stopped to support me in any time and had to do huge sacrifices to let me study in Politecnico. Times were absolutely difficult in my hometown, before I enrolled in this University: my mother had the strength to push me outside that situation and to let me see how the world is like outside that small and wounded city in the centre of Italy. My greatest hope is that my small achievements can be a return to these investments she did.

I would spend some final words to thank the professor **L. Baresi**. Not only he followed me in the preparation of this document, but thanks to him I was able to follow a path, that has roots in my Bachelor in Cremona, totally oriented to one of my passions, Mobile App Development, that finds a natural outlet in this document. I have fantastic memories of courses to high school student organized by him in Cremona campus, where I could help students to program simple apps. Regarding this document, the professor has given me assistance in each step, being a reference point that gave me a quick support each time I needed it, demonstrating a huge patience considering times where I wanted to be faster. I'm proud to had the occasion to work with him and I hope occasions like this will present again in the future.

Luigi Russo

Table of Contents

Index of Figures

Index of Tables

Abstract

English

In the following work, we analysed *cross-platform* programming approaches in 2017, both in desktop and in mobile world, stressing the latters. We introduce Mobile Studio, a software written with JavaScript and that uses frameworks Electron and NativeScript in order to develop native Android and iOS apps, analysing in deep how this could be achieved.

Italiano

Nel seguente lavoro abbiamo analizzato gli approcci alla programmazione *cross-platform* nel 2017, sia nel mondo desktop che in quello mobile, concentrandoci sui secondi. Presenteremo Mobile Studio, un software scritto in JavaScript e che usa i framework Electron e NativeScript per sviluppare applicazioni Android e iOS native, analizzando in tutti i dettagli come questo possa essere stato possibile.

1 Introduction

Communication problems are not a novelty in human history and affect how we relate with technology, since ages. The curious reader may know about the legendary construction of the *Tower of Babel*, made impossible because the workers of that marvellous building, according the biblical myth, spoke different languages. But we don't have to follow thousands of years old books to prove these difficulties that still survive and provide huge money loss in the current society. A not far in time example of this situation is the *Mars Climate Orbiter*, a spacecraft intended to study the atmosphere of the red planet. Launched in 1998, it positioned itself too low in the Martian sky because its software used measures in Imperial system, while the navigation team on Earth expected data in the International one. The result was that a 328 million dollars' spacecraft was destroyed by the friction with Martian atmosphere.

Software Engineering is not immune of these problems. Since the dawn of Computer Science multiple vendors have proposed their machines, each of them "speaking a different language": numerous operating systems, totally different architectures, hundreds of programming dialects spiralled over the years, causing the proliferation of an enormous amount of protocols that tried to flatten these differences, not always succeeding in this purpose.

A *cross-platform programming,* the idea to write software in one language and see it running in multiple platforms, became the dream of every developer and has been significantly more hunted when mobile architectures were started to be used. In this scenario, in fact, the time-to-market is critical and having multiple teams working on the same app, but in different platforms, is an approach that potentially can lead to various problems, like functionality misalignment or basically the waste of money that two team, working on the same software, can represent.

In the years, numerous approaches to the dreamed cross-platform programming have been proposed, both in the desktop world and in the mobile one. It is impossible to not think of Java Virtual Machine, one of the first solution that, with specific versions in both worlds, detained the throne of best cross-platform approach for years and still is the idea – extremely and continuously improved from the original one – that let all the plethora of Android devices to be programmed in the same way.

This, however, is not sufficient. Java software has been criticized in desktop world for its slowness and, in the mobile one, it totally cuts out one of the most important mobile operating systems: Apple's iOS. Other solutions have been proposed in the years and in this document we will analyse the most important ones in details, trying to suggest to developers which one use in various situations.

While such an approach is not a novelty in the academic world (see Related Studies paragraph), the various resources that we have found seems to be outdated, thus our objective is to propose again a comparison of technologies considering the context of 2017. The real novelty that we are proposing with this document is an analysis of cross-platform approaches also in the desktop world, considering totally new technologies like Electron, that uses JavaScript at their core. We can safely say that this is something new in academic world. The reason is quite simple: JavaScript based desktop approaches are brand new and still are not widespread.

We have performed more than tests, though. We chose two of these cross-platform approaches, the ones that, for each world, we considered the most innovative, and we built a complex software, an Integrated Development Environment able to compile Android and iOS applications. The choice to build such a software was inspired from the absence, in our analysis, of one single IDE able to provide an easy and visual approach to the usage of these new mobile cross-platform methods. It is curious how this program is written in one language and is able to output real apps using the very same language. This is the reason that can lead us to state that the resulting IDE, that we called Mobile Studio, is a perfect synthesis of how it is possible to address literally all platforms, both desktop and mobile, with one single language. This “grail” jargon is JavaScript and it is famous for its flexibility and easiness, that met an incredible evolution in the years and is one of the most popular programming language to date. Born simply to add some interaction to web pages, the most modern frameworks allow to perform complex tasks, like creation of new processes or access to the file system, with simple JavaScript functions. We analysed how cutting edge technologies allow a program with the logic written using it – and the rest in the standard web format – to be launched as a first-class software in desktop systems, and we built Mobile Studio in this way. The IDE is then able to build a level of abstraction from the app that the developer is building inside it, creating a model that is independent from the framework that will be used to translate it to a real native app. This was a crucial point in this work because it makes the IDE easily expandable to the usage of different frameworks that will accomplish this task. In the implementation proposed with this document, we specifically chose one, called NativeScript: Mobile Studio is able to translate the model into the specific NativeScript syntax – that, as we will discover, has the logic written always in JavaScript with only few customizations needed – and then using this framework to compile it to a real mobile app, that can run in iOS and Android. We will analyse in deep the potentiality of this software and of this approach in general, not forgetting the inevitable limitations that it bears. In synthesis, we can summarize our work with the following points:

- We analysed the various existing methods to develop a mobile software, discussing advantages and disadvantages of each one.
- We investigated on new, cutting-edge methodologies to develop both mobile software and desktop one, both having strong roots in web programming.
- We tested these approaches, programming an IDE that use both to deliver simple, but fast and reactive native apps, using an easy UI. This software, extensively explained in this document, will be able to support the clear majority of native UI components and permits to specify simple interactions visually, that can be furtherly enriched with an embedded code editor.

- We tested the IDE, and thus all the approaches that it uses, trying to convince the reader about the performance and the easiness to program complex software using them.

In order to achieve these result, we structured this document in four main chapters:

1. The *first*, this, contains a brief introduction to the problem and the main purpose of this work, comparing it also to other scholar studies.
2. The *second*, contains the very same introduction, but in Italian.
3. The *third* will provide a bird-eye look to the general software context in 2017, both in desktop world and in mobile one, stressing the latter, considering its popularity nowadays. We will provide information on how it is possible to build a program in both worlds and how are the new cutting edge ways of doing that.
4. The *fourth* will introduce Mobile Studio, the IDE developed in Politecnico di Milano to build cross-platform mobile applications. This section will act as a documentation of this software, exploiting all its details, how it has been built, which choices have been made in the implementation process and how it works.
5. The *fifth* will discuss our results and what are the possible evolutions of the project.

Keywords Mobile app programming, cross-platform developing, Node.js, Electron, JavaScript, React Native

2 Introduzione

I problemi di comunicazione non sono una novità nella storia umana e condizionano il nostro rapporto con la tecnologia da lungo tempo. Il lettore curioso conoscerà la leggendaria meraviglia della *Torre di Babele,* resa impossibile perché i costruttori di quell'opera, secondo il mito biblico, parlassero lingue diverse. Tuttavia, non è necessario seguire libri vecchi millenni per provare queste difficoltà che ancora sopravvivono e causano immense perdite economiche nella società attuale. Un esempio non lontano nel tempo di questa situazione è il *Mars Climate Orbiter,* una sonda spaziale avente l'obiettivo di studiare l'atmosfera del pianeta rosso. Lanciata nel 1998, si è posizionata troppo in basso nel cielo marziano perché il suo software usava misure nel sistema Imperiale, mentre il team di navigazione sulla Terra si aspettava dati con quello Internazionale. Il risultato è stato che una sonda da 328 milioni di dollari è stata distrutta dall'attrito con l'atmosfera marziana.

L'Ingegneria del Software non è immune da questi problemi. Fin dall'alba dell'Informatica, più produttori hanno proposto le loro macchine, ognuna di esse in grado di "parlare una lingua diversa": numerosi sistemi operativi, architetture totalmente diverse, centinaia di linguaggi programmazione si sono evoluti negli anni, causando la proliferazione di un numero enorme di protocolli che ha tentato di appiattire queste differenze, a volte non riuscendoci.

Una *programmazione cross-platform,* l'idea di scrivere software in un linguaggio e vederlo eseguito su più piattaforme, divenne rapidamente il sogno di ogni sviluppatore ed è stata significativamente più ricercata all'avvento delle architetture mobile. In questo scenario, infatti, il time-to-market è critico e avere più team che lavorano alla stessa app, ma in piattaforme diverse, è un approccio che può portare potenzialmente a vari problemi, come disallineamenti di funzionalità o, banalmente, alla perdita monetaria che più team, lavorando allo stesso identico software, può rappresentare.

Negli anni, numerosi approcci alla sognata programmazione *cross-platform* sono stati proposti, sia nel mondo desktop che in quello mobile. È impossibile non pensare alla Java Virtual Machine, una delle prime soluzioni che, con versioni specifiche per ciascuna piattaforma, ha detenuto il trono di miglior approccio cross-platform per anni ed è ancora l'idea – estremamente e continuamente migliorata da quella originale – che permetta a tutta la pletora di dispositivi Android di essere programmati nella stessa maniera.

Questo, ad ogni modo, non è sufficiente. Il software Java è stato criticato nel mondo desktop per la sua lentezza e, in quello mobile, non considera totalmente uno dei più importanti sistemi operativi mobile: iOS di Apple. Altre soluzioni sono

state proposte negli anni e in questo documento analizzeremo quelle più importanti nel dettaglio, cercando di suggerire agli sviluppatori quale usare in varie situazioni.

Questo approccio non è affatto nuovo nel mondo accademico (v. Related Studies nel capitolo successivo), tuttavia le risorse che abbiamo analizzato ci sono sembrate superate, pertanto uno dei nostri obiettivi è quello di proporre ancora una simile comparazione delle tecnologie, considerando però il contesto del 2017.

Una novità che proponiamo con questo documento è una simile analisi di questi approcci cross-platform anche nel mondo desktop, considerando tecnologie totalmente innovative come Electron, basate su JavaScript. Possiamo dire con confindenza che ciò è qualcosa di nuovo nel mondo accademico. La ragione di ciò è semplice: applicazioni desktop basate su JavaScript sono nuove e ancora con una bassa diffusione.

Abbiamo fatto qualcosa molto più che semplici test, inoltre. Abbiamo scelto due di questi approcci cross-platform, quelli che, per ognuno dei mondi considerati, abbiamo considerato i più innovativi, e abbiamo costruito un software complesso, un Ambiente di Sviluppo Integrato (IDE) in grado di compilare applicazioni Android e iOS. La scelta di sviluppare un simile software è stata ispirata dall'assenza, nella nostra analisi, di un singolo IDE in grado di utilizzare un facile metodo grafico per sfruttare i più innovativi metodi di programmazione mobile cross-platform. È curioso come questo programma, scritto in un certo linguaggio, è in grado di sfruttare lo stesso linguaggio per sviluppare applicazioni mobile. Questo è il motivo che ci spinge ad affermare di come l'IDE finale, che abbiamo chiamato Mobile Studio, possa rappresentare una perfetta sintesi di come sia possibile sviluppare per qualsiasi piattaforma, sia desktop che mobile, con un singolo linguaggio. Si tratta di JavaScript: famoso per la sua flessibilità e semplicità, ha conosciuto un'incredibile evoluzione negli ultimi anni ed è ora uno dei linguaggi più popolari. Nato semplicemente per aggiungere della dinamicità alle pagine web, i framework più moderni gli permettono di eseguire compiti complessi, come la creazione di nuovi processi o l'accesso al File System, attraverso semplici funzioni. Abbiamo analizzato come le tecnologie più moderne permettano di avere un programma scritto con la logica in questo linguaggio – e il resto secondo il classico paradigma web – che possa essere lanciato come un software desktop in alcun modo diverso dagli altri sviluppati con tecnologie più classiche, costruendo Mobile Studio in questa maniera. L'IDE è poi in grado di costruire un livello di astrazione dall'app che lo sviluppatore sta costruendo al suo interno, creando un modello indipendente dal framework che sarà utilizzato a livello sottostante per tradurlo in un'app nativa vera e propria. Questo è un punto cruciale di questo lavoro, che rende l'IDE facilmente estendibile al supporto di diversi framework che possono effettuare tale traduzione. Nell'implementazione proposta con questo documento, ne abbiamo scelto uno in particolare, chiamato NativeScript: Mobile Studio è in grado di tradurre il modello nella sentassi specifica di NativeScript – che, come scopriremo, ha sempre la logica scritta in JavaScript, con solo pochi accorgimenti necessari – e usa questo framework per compilarlo in una vera applicazione mobile nativa, che può essere eseguita in Android o iOS. Analizzeremo nello specifico le potenzialità di questo software e di questo approccio in generale, non dimenticandoci degli inevitabili limiti che soluzioni del genere portano. Sintetizzando, possiamo riassumere il nostro lavoro con i seguenti punti.

- Abbiamo analizzato i vari metodi esistenti per sviluppare un software desktop, discutendo di vantaggi e svantaggi di ciascuno.

- Abbiamo approfondito le nuove metodologie per sviluppare sia software mobile, che desktop, che hanno radici nella programmazione web.
- Abbiamo testato questi approcci, programmando un IDE in grado di sviluppare applicazioni mobile semplici, ma veloci e reattive, usando una facile interfaccia utente. Questo software, spiegato in ogni suo dettaglio in questo documento, è in grado di supportare la maggioranza delle componenti grafiche di una app e di specificare interazioni in maniera visuale, che possono essere arricchite ulteriormente grazie a un editor codice integrato.
- Abbiamo testato l'IDE, e quindi tutti le metodologie che esso utilizza, cercando di convincere il lettore sulle performance e sulla facilità di programmare software complesso attraverso essi.

Per raggiungere questi risultati, abbiamo strutturato questo documento in quattro capitoli principali:

1. Il *primo,* contiene una breve introduzione al problema e lo scopo di questo lavoro;
2. Il *secondo,* questo, traduzione in lingua italiana del primo.
3. Il *terzo* darà una vista d'insieme al contesto del software nel 2017, sia nel mondo desktop che in quello mobile, concentrandosi sul secondo, considerando la sua popolarità oggigiorno. Daremo informazioni dettagliate su come è possibile costruire un programma in entrambi i mondi e su quali siano le più avanzate tecnologie che lo permettono
4. Il *quarto* presenterà Mobile Studio, l'IDE sviluppato all'interno del Politecnico di Milano per costruire applicazioni cross-platform mobile. Questa sezione sarà la documentazione di questo software, spiegando ogni suo dettaglio, come è stato costruito, quali scelte sono state fatte nel processo di implementazione e come funziona.
5. Il *quinto* discuterà dei vari risultati e degli sviluppi futuri del progetto.

Parole chiave Programmazione app mobile, sviluppo cross-platform, NativeScript, React Native, Node.js, Electron, JavaScript

3 General context

3.1 Related Studies

As briefly said in the introduction, comparison of cross-platform mobile technologies is something far from new in the academic field, where we can find interesting studies that try to compare them both analytically – e.g. analysing performances of the final output – and from a higher point of view – e.g. questionnaire to developers. The main problem that we found with these works is that they are not able to follow the rapid development of technologies and becomes outdated rapidly. As an example, the interesting survey by M. Latif et al. [1], does a comparison of the technologies similar to ours' but gives little emphasis on cross-interpretation methods that will be crucial in our work.
We thus decided to run a more comprehensive analysis of what this world is like in 2017, giving to the academic world a new "snapshot" of this world.

Our analysis is anyway deeper and touches cross-platform methods also in the desktop field, always using JavaScript as a main programming language. To the best of our knowledge, this is the very first academic work that analyses this scenario. Yet, we based our analysis, and the implementation of the project, on technologies already seen in a scholar way – the main example could be Node.js, where academic studies heavily documented its performances [2] and its architecture [3].

The more general idea of building a software able to output an app in a model-driven way is not a novelty either and proposed by other works of the Politecnico di Milano itself, with the thesis of Perego et Pezzetti [4] or with the study of Fino [5]. Both used a model to capture all the details of a native application – the first both for iOS and Android, the second only Android – and techniques that let translate this model in a real native app. The principal difference with the current work is that we are inserting another abstraction layer between the model and the final app: a cross-interpreted language – in the implementation we propose, NativeScript. Such a system allows us to use a far simpler model than the one used in these works, and permits us to model a broader number of components, but, as we will see in the conclusions, will bound our flexibility to modify the app in environment different from the ones that uses NativeScript.

Finally, it is worth mentioning another important contribute that used web technologies – in particular Node.js – in order to build quickly a prototype of the app: we find it in the work of Natali [6]. Many of the concept used in is thesis are present also in the current work inside the Visual Editor of Mobile Studio.

3.2 The mobile world in 2017

On 9 January 2007 Apple CEO Steve Jobs introduced the device that many consider the first “smartphone”: the iPhone. This device was far from being as powerful as its contemporary PCs and provided little functionalities, but gave the user a new way to interact with technology, being disruptive with the ecosystem that lived before it. Usage of touch and sensors – e.g. accelerometer – gave to people, used to interact with technology with mouse and keyboards, a totally new experience. This is the reason that lead at the incredible diffusion of mobile devices, now ubiquitous, irreplaceable part of our lives.

This would not have been possible without the involvement of third-part developers, not directly employee of Apple or Google, but persons that could sell their products via the stores present in the operating system in these new smartphones. This was the rise of the *apps*: a totally new paradigm of software with the principles to be fast, lightweight, single-purpose. Apps and developer were, and are now, crucial in the success of a certain mobile operating systems and thus of the company that owns it: it is noticeable the case of Microsoft that, arrived late in the market with its Windows Phones, had to do huge investments in the field to be competitive – and still is a small player in this world.

The spiralling of these new technologies lead to an incredible evolution in their architectures [1] and thus a tremendous improvement in performances. Totally new technology arose: VR or smartwatches, for instance. Devices that can only little resemble their common ancestor, the smartphone, but still bear in mind the original idea to give the user new way to interact with the technology.

The reader could immediately think that such a plethora of devices would be difficult to manage and in fact, although new standards had born, many companies have developed their own technologies that have little compatibility between each other. A developer is usually asked to choose *a priori* the potential users of his app, and therefore to choose an operating system and a language to use. This, at least, is the common approach, that many have used and are still using. But, as we will see in the following paragraphs, in the last few years new techniques that helps developer to reach a broader audience are being introduced and this thesis describes one tool, developed in Politecnico di Milano, that using such technologies will try to reach this specific purpose: an Integrated Development Environment that we have called Mobile Studio.

Before going in deep into explaining such a tool it is fundamental to have a clear idea on how the ecosystem of mobile app is composed in 2017.

3.2.1 Background

As written at the beginning of the last paragraph, it is common practice to say that the smartphone era started with the introduction of the iPhone. This device, built by Apple, ran a custom operating system derived from Mac OS X, the usual desktop OS of this company. Years later from its presentation, to better emphasize the difference between the desktop and the mobile one, the latter OS has been called “**iOS**” and it is still the operating system that every mobile device built by Apple runs. Currently, iOS is in 10^{th} version.

The main competitor of iOS is without any doubt **Android** OS, that has been presented to the public in the same year of the first iPhone and it is developed and supported by a consortium of technological companies, the Open Handset

Alliance, led by Google. Currently, Android is the most widespread operating system ever built – also considering desktop ones – being installed in millions of totally different devices, that range from microcontrollers to PCs, passing obviously by smartphone and tablets.

Figure 1 – Most diffused mobile OSs in specific countries, Sept to Nov 2016, in green Android, in grey iOS

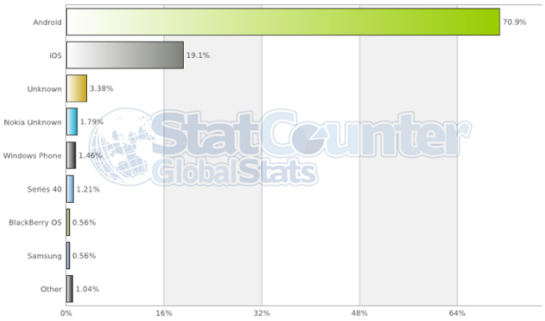


Figure 2 - Percentages of diffusion of mobile OSs, September to November 2016

While other platforms have been presented – an example could be Windows Phones – and older ones tried to evolve, Android and iOS dominate the market of smartphones and tablets. This is the reason why most of the developers try to build their app in one of these two platforms, if not both.

As we have said the success of these platforms was not given principally by the platform *per se*, but from third-part developers. Both Android and iOS provide in fact places – Play Store for the first, App Store for the latter – where is possible to download and install on the device apps programmed by external developers and only analysed by Google or Apple – to prevent virus and any other type of malware.

As the “store” word says, these apps, or the content inside them, could be sold: and it is from this basic idea that all the apps-world exploded. Near anyone could be a developer and get paid a small amount of money that scaled quickly with the diffusion of these devices. To dimension this phenomenon, we can consider that only on Apple platforms, according to company data [2], App Store had 140 billion download in its 5 years history and gave to developers 50 billion of U.S. Dollars.

The reader could immediately think on how this context was attractive to developers, who rapidly started programming apps for the two platforms. In the next paragraph, we will investigate on how is it possible to build an app.

3.2.2 State of art of mobile app programming

We can say that the programming of an app presents various issues *a priori*: while it is certainly possible for a developer to choose one method to develop an app, this is not unique, maybe not the easier and, at the end, maybe not the smartest. This choice, in fact, will determinate precisely which user she will be able to reach and, often, this cannot be modified at the end without reprogramming entirely the app, with huge time, and thus money, loss.

We will explore all these methodologies, shading light on all the advantages and disadvantages of each, in the following sections.

3.2.2.1 Native programming

It is possible to say with no regret that the *native* programming approach is the most "classical" one, because directly supported from the companies that build the operating systems of smartphones. These companies have all the interest to have quality apps that can promote their OS, so they usually provide first class frameworks that are deeply engineered to give to the developer all the tools that he needs to build a complex app. Apps that can, by construction, be lightweight, fast and coherent with the visual language that the OS defines. This is due to the usage of specific tools for specific platforms, which can fully unleash the power behind these, often accessing directly to the hardware of the device without additional software layers. This is the case of iOS and Windows Phone, where the high-level language is compiled directly in machine code and then executed. In Android, the situation in a little more complex, because a Java Virtual Machine – "Dalvik" for Android versions inferior than 5 – is positioned between the high-level language code and the hardware. Yet, this is the only layer of this kind and it is continuously subject to improvement[1].

The closeness of the native languages to the machine code and the presence of few level of abstraction translates into fast and reactive apps. There is also another big advantage that, in the writer opinion, is not fully understood by many companies: the owners of mobile operating systems enforce a "continuity" between all the apps that should run on one platform, in terms of aesthetic, graphic and user experience. This is often done seamlessly, without the awareness of the developer, but it is able to let the user feel that the app "is just right" for her phone, integrating perfectly with the ecosystem of other apps.

All of this does not come free, though: it is impossible to have apps able to run in multiple platforms without programming them multiple times, for each platform. If this can sound difficult, it is important to notice that this is the usual *modus operandi* of companies, nowadays, thanks also to the small number of platform to address – principally only Android and iOS. Yet, having different teams working on different version of the same app can lead to many problems: it could easily happen that the functionalities of the app are not aligned on all the platforms. It is clear, also, that having multiple teams working on the same app could represent an expensive overhead that companies will always try to avoid: the need of a new multi-platform approach arose quickly and developed in the years. We will investigate on these new methods in the following paragraphs.

3.2.2.2 Web Apps

The first of the multi-platform approaches has born considering that a language able to run in quite every operating system existed already**:** web pages, built with standard web technologies – HTML, JavaScript and CSS – could be rendered in every device that had a standard browser. Web apps, that were nothing more than a web page styled to be more accessible to the smaller screens of smartphones, appeared rapidly and still exist nowadays, like we can see in Figure 3 and 4.

[1] Since Android version 6 Dalvik VM, that used a "Just-in-time" compiler, has been replaced by the ART runtime, that used an improved "Ahead of Time" compiler. Moreover, now the OS, thanks to the new NDK, supports code written in C that can be directly translated to machine code.

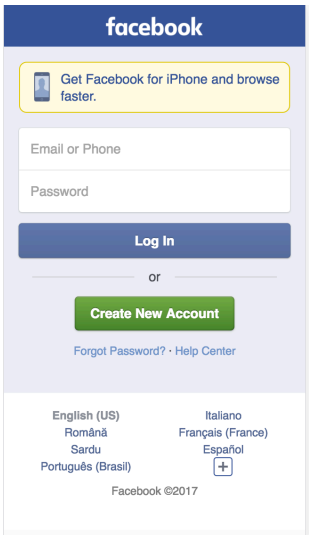


Figure 3 – Login on the Facebook Mobile WebApp

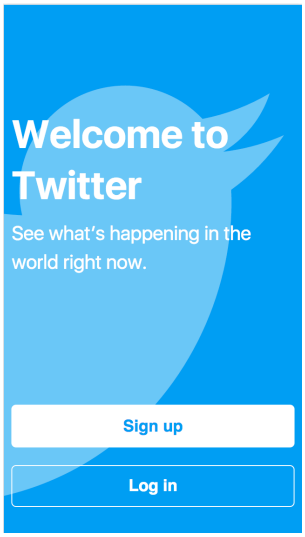


Figure 4 - Login on the Twitter Mobile WebApp

Designing websites for mobile was challenging, considering new screen sizes and new interactions – e.g. swipe and other multi-touch gestures – and required a totally different way to program the web, especially the UI.

Framework to help web developers to address this issue arose and HTML itself was updated in order to introduce tags and elements that helped the building of *responsive* websites: websites that are able to understand the context – browser, device, etc. – where they are displayed and adapts their components in order to provide the best visualization of the content. We will talk more about HTML evolutions in the paragraph 2.2.1, but an easy visualization of how components can be resized and repositioned in a responsive website is shown in Figure 5.

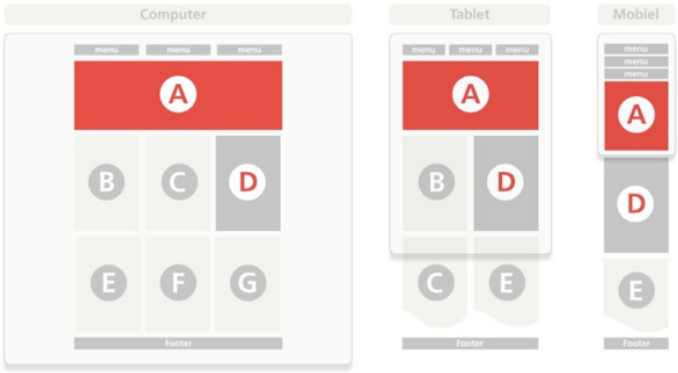


Figure 5 - Responsive web: how elements adapt in different platforms

To let the reader be able to fully understand this approach we propose a brief example that uses the framework **jQuery Mobile** [2]. Build on top of the famous jQuery framework, jQuery Mobile, not differently from all the framework of its kind – like its main competitor, **Bootstrap** – is nothing more than a set of HTML stylesheets and scripts giving a "mobile" look to a web page. To use it, in fact, the programmer has only to insert these files her HTML document, as shown in the following snippet of code.

```
<link rel="stylesheet" href="jquery.mobile-1.4.5.min.css" />
<script src="jquery-1.11.1.min.js"></script>
<script src="jquery.mobile-1.4.5.min.js"></script>
```

Where the files have been assumed to be positioned in the same directory of the HTML document. Just with the inclusion of these libraries, we can see how all the HTML starts to have a more “mobile” visualization. Let consider, for instance, the case of a simple button, written with the following HTML code.

```
<input type=”button” value=”Button”>
```

Figure 6 and 7 show the differences between how this line is visualized with and without the inclusion of jQuery Mobile libraries.

Figure 6 - HTML button without jQuery Mobile styling

Figure 7 - HTML button with jQuery Mobile styling

jQuery Mobile does not only style pre-existing HTML components, but also introduce new ones and adds elements that allow an easy navigation and complex transitions that are typical of native apps. Yet, we can see how this is prevalently an UI library. To enrich the logic of a web app in order to let it use basic mobile events many more libraries can be found in the internet and we can easily present an example to the reader, with **Hammer.js**. This library is specifically built to handle multi-touch, giving to the programmer the ability to recognize gestures like pan, swipe, pinch, etc. As typical of a web-like approach, the usage and the inclusion of such a library is simple, putting a single line in the HTML of the page:

```
<script src="hammer.js"></script>
```

Then, using JavaScript this time – thus inside a `<script>` tag or in a specific `.js` file – it is possible to create a listener, specific to a DOM element, that will check if some multi-touch action will be performend to that element and then acts accordingly. This is shown in the following snippet of code:

```
var hammertime = new Hammer(myElement, myOptions);
hammertime.on('pan', function(ev) {
        console.log(ev);
});
```

In the example, in fact, a new `hammertime` listener is created and bound to the element `myElement`, the listener then will react to the pan gesture performed on that element showing specific data of the event – for example x, y coordinates.

Hammer.js is only one of the many more libraries that can be found in the internet and that allows to expand the capability of a web app in an incredibly simple way, as we have seen it. Easiness to be programmed can translate into huge economic savings for a company that wants to build an app using this approach, especially considering that web-skills are often already inside the knowledge base of quite every technological company. Unfortunately, this is the only advantage of pure web-based mobile programming: applications built with this method are several times slower than their native counterparts [4] [5] and living

inside the sand-boxed environment of the browser doesn't allow them to have any complex communication with the hardware of the device – e.g. access to accelerometer data, microphone. Therefore, typical web services like Facebook and Twitter, although they still have a web application, have developed more complex apps in other technologies and strongly invite their clients to use the latter instead of the first.

3.2.2.3 Hybrid apps

The main problem that we have encountered when analysing the app development process using web technologies was the browser, a big – and thus slow – software layer of every OS, not specifically built to access directly the hardware of the device. To allow this communication and improve performances, while still using web technologies to build apps, a basic idea has been to let the web app run inside a native component able to visualize web pages, lighter than the browser itself. In both Android and iOS this native component is called *WebView*. The usage of such components was the base to the realization of real "wrappers" for the web app that, written in native languages, allowed the developer to use advanced features. Usually, these native functions called by the wrapper are masked inside JavaScript ones that can be written by the developer. In this way, the programmer can use only one script language for the whole app.

The reader can fully understand how this approach works looking at the most famous implementation of it, that was done by **PhoneGap**, a software later acquired by Adobe that uses an engine provided by Apache, called **Cordova**.

In PhoneGap, the wrapper of the web application is a *chrome-less* browser, a browser that has no windows decorations in order to display the content in full-screen. This, as we have said, is nothing different from a native WebView component, that, in fact, is an *UIWebView* in iOS and an *android.webkit.WebView* in Android. It is crucial to notice that these views are not the same and can display contents in different ways: the programmer must be aware of that.

PhoneGap can then use Cordova Plugins in order to give to allow the programmer to use native functions and sensor, all using JavaScript as main language. The skilled developer can also build its own plugins in native languages and then build JavaScript functions able to use them.

In Figure 5 is shown the architecture of a PhoneGap/Cordova application [3]

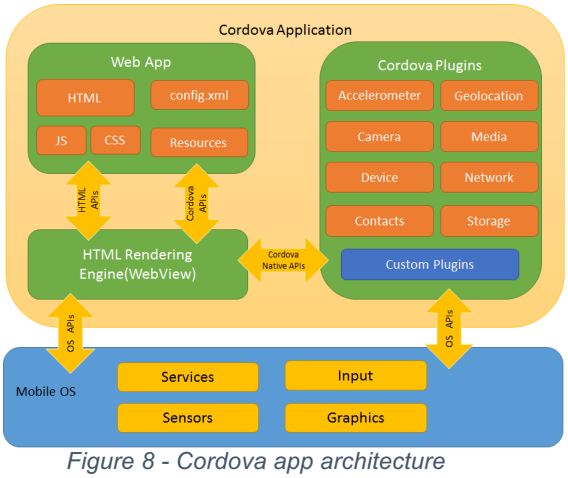


Figure 8 - Cordova app architecture

As we can see, Cordova is composed by three modules:

- Web App
- Web View
- Plugins

Where the Web App are the actual HTML, JS and CSS files that compose the pages that should be presented to the user. The Web View is the real wrapper that we were explaining before, a small software written in native language able to browse the Web App and, thus, that acts like a browser. The most important module is the last one: the plugins. Communicating directly with the wrapper, they allow a direct access to the underlying OS API. The most noticeable fact that Cordova allows is that these plugins are still usable by the programmer with JavaScript code, although they are written in platform specific language. We can look at the usage of one of these plugins: in the following example, a function is written to display the battery level of the device when it changes by at least 1 percent.

```
window.addEventListener("batterystatus", onBatteryStatus, false);

function onBatteryStatus(status) {
    console.log("Level: " + status.level + " isPlugged: " + status.isPl
ugged);
}
```

As we can see, a `status` JavaScript object is provided and, in its attribute `level` we find the current battery level of the device.

Plugins, not differently from the external libraries that we have encountered in the pure web programming, tries to be a bridge that allows the web technologies to access specific device feature. Differently from those, plugins are coded in native languages, as thus are faster than pure web libraries. Yet, performance is a strong issue also using this approach. While a performance test is not straightforward to do with Cordova [4], a Microsoft study of 2015 [5] proved that an app programmed with such technology is usually slower and has more memory consumption of a native one in a factor that depends on the device OS, but can be up to six times (use case of a cold startup of the same app programmed in Cordova and in Native in Android 4.4).

In the writer experience this approach is far from being able to offer a user experience comparable to native apps and it could be used only for simple software, like the one for shopping or e-commerce. The main advantage of this methodology is that let the company use skills that often are already present inside it – web developing – and, for the programmer alone, the curve of learning of these web languages is far easier than native ones.

3.2.2.4 Cross-compilation

As we have seen, the usage of the techniques that we have introduced is not able to provide apps with the performances typical of the one programmed with a native approach. That is due to the software layers – that can be browsers or wrappers – where the app must live, that will necessary slow down the overall execution.

To solve this problem a “back to native” approach has been taken by some companies: the idea here is to write the app in a language that tries to map the

functions and the elements of the native ones, that will be then properly compiled in native code. This approach has been called **cross-compilation**.

A compiler able to accomplish the translation above would be incredibly complex and difficult to be maintained, because should map new native function as soon as they come out from official native SDK. Such a complexity could be managed only by big companies, in fact the main example of this approach is provided by Microsoft with its **Xamarin**. With this development kit is possible to write in a language (C#) that gets compiled in the machine code of multiple platform, having at the end a real native application. Unfortunately, a perfect one-to-one mapping between functions in different platforms is impossible and the code must always be tailored for each of these. The idea, anyway, is to let this part of customization be as little as possible, often reserved only to the UI, maintaining the logic of the app the same: the main target of this approach is to maximize the code reuse, leaving only a little layer to be tailored to the specific platform.

It is curious to analyse how the complex compiler of Xamarin works [6] and why we call it a “cross-compiler”. Compilers of such a kind basically are able to create executable code for a platform different than the one on which they are running [7], in the particular case of the one of Xamarin, it has to behave differently for the three platforms it supports – iOS, Android, Windows Phone. The differences in the compilation on these platforms are noticeable:

- **iOS** – C# is ahead-of-time (AOT) compiled to ARM assembly language. The .NET framework is included, with unused classes being stripped out during linking to reduce the application size. Apple does not allow runtime code generation on iOS, so some language features are not available.
- **Android** – C# is compiled to instruction level and packaged with a VM (MonoVM) and a just-in-time (JIT) compiler. Unused classes in the framework are stripped out during linking. The application runs side-by-side with Java/ART (Android runtime).
- **Windows** – C# is compiled to instruction level and executed by the built-in runtime, and does not require Xamarin tools.

As we can see, the easiest work for the compiler is the one that it should do for Windows. This is not a surprise also because the main IDE that uses Xamarin is **Visual Studio**, the famous Microsoft product to build every kind of software on its platforms. Note that, on OSs which do not support this IDE – e.g. macOS – a lighter **Xamarin Studio** is provided. We will look in detail to these software when talking about IDE.

Programming with Xamarin allows the developer to use an advanced programming language [8], way more similar to Native ones rather than web programming, that allows the production of apps with performances comparable to the pure Native approach and that need only a small amount of platform-specific customization to be up and running.

However, this cross-compiling approach has some downfalls: the programmer has always to rely to a third-part stakeholder – that in the case of Xamarin is Microsoft – that implements the native functions he wants to use. This is critical when new functionalities are introduced by device companies and the programmer must wait that the third-part stakeholder makes the translation to let him use it. Nowadays time-to-market is a crucial point to let an app have success, so this can be painful. Another big disadvantage of these approach is that, as we have seen,

a cross-compiler is an incredibly complex piece of software to be programmed and maintained, and someone has to pay that complexity. These approaches are expensive: for instance, the business subscription to Xamarin costs $999 per developer, per device platform. While this amount is not critical for a big company, it actually is for a small one, or for a single programmer: two kind of stakeholder that, as we have seen, compose the vast majority of the mobile developers base.

3.2.2.5 Cross-interpretation

A fusion between the idea of cross-compilation – able to produce native applications – and the web technologies – that have an easy language to deal with – lead to the development of the last technology that we will consider in the programming of an app: the one that we can call **cross-interpretation**. In this approach the app is written in a web-like language, pseudo-JavaScript or directly JavaScript, and an enriched layout file is specified, usually in XML, that can be styled in a subset of CSS rules. This is the case of **NativeScript** and **React Native**: the first one is the technology at the base of the project associated to this thesis, so we will analyse in deep these two methodologies, comparing them and explaining why the first one has been chosen in the next paragraph.

It is important to say that these technologies exist thanks to the evolution of JavaScript engines, which are able to behave like compilers and translate JavaScript code directly in machine code. We will speak about these in detail in paragraph 2.2.1.1. In the mobile world, these kinds of engines are usually built in the host OS itself – this is the case for iOS and Windows – or should be included in the final app – Android case.

3.2.2.5.1 React Native

Before talking about React Native is important to say something about its main ancestor library: **React**. Built by Facebook, React is a JavaScript library with the main purpose to build user interfaces in an efficient, declarative and flexible way based on components. In order to understand this concept, we can see an example on how a component is built declaring the following `ShoppingList`:

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}
```

As we can see, a component is basically a JavaScript class that, using the `renderer()` method, is able to return a small piece of UI, written in HTML. Note

how the class can be instantiated with different values, as usual in Object Oriented programming. For instance, the previous example can be used writing the line:

```
<ShoppingList name="Mark" />
```

It is left to the reader to see how it is possible to enrich React syntax with functions and data persistence. [9]

React Native is nothing more that the usage of React inside a container that is not the standard DOM of a webpage, but a mobile app screen, where real native components are masked inside React ones. Let us take a look at the code below.

```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

class WhyReactNativeIsSoGreat extends Component {
  render() {
    return (
      <View>
        <Text>
          If you like React on the web, you'll like React Native.
        </Text>
        <Text>
          You just use native components like 'View' and 'Text',
          instead of web components like 'div' and 'span'.
        </Text>
      </View>
    );
  }
}
```

Where we can see the usage of components, like `Text` one, that will be converted in real native elements, specifically the `UILabel` on iOS and the `TextView` in Android. This, as Figure 6 shows, is done creating a virtual DOM where the programmer can put its React components, as before in standard React code. The virtual DOM, then, communicates asynchronously with the underlying OS API and translates its components in real native ones. Working asynchronously, the native UI thread is not blocked and that translates to a fast UI, but the communication is complex and a bottleneck to this approach is that an application that needs, for any reason, to do a lot of native calls, can be slow.

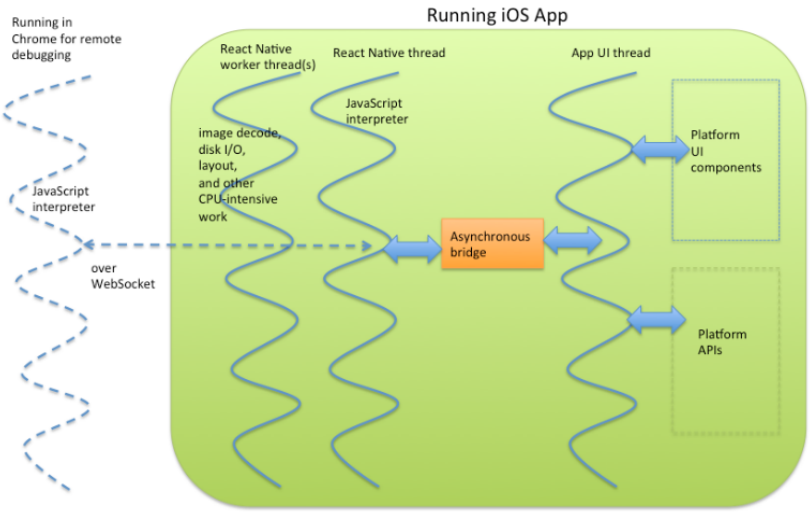


Figure 9 - React Native architecture schema

One advantage of React Native is that the bridge can load also custom native modules. In this way, the programmer can reuse parts of the app written in native languages in an easy way. [10]

3.2.2.5.2 NativeScript

Built by Telerik and supported by Google, NativeScript shows a similar behaviour as React Native, although the architecture of the two are quite different.

From the programmer point of view, the main difference is that it uses a more "web-like" approach to develop an app. In fact, it exists the usual separation between JS, CSS and XML (that take the place of standard HTML) of a web project. This is convenient because enforces a strong separation between the structure (XML), the presentation (CSS) and the logic (JS), in a classical MVC style, which is not present in React Native where, for instance, is mandatory to use inline styling to change the appearance of the component.

A NativeScript application has always a *package.json* file that, in a way that closely recalls the Manifest in Android, specify all the details of the app. We can see an example of such a manifest in the following code.

```
{
	"description": "NativeScript Application",
	"license": "SEE LICENSE IN <your-license-filename>",
	"readme": "NativeScript Application",
	"repository": "<fill-your-repository-here>",
	"nativescript": {
		"id": "org.nativescript.Coffee",
		"tns-ios": {
			"version": "2.3.0"
		}
	},
	"dependencies": {
		"tns-core-modules": "2.4.2"
	}
}
```

As we will see, this is the standard JSON file of a Node.js web app, that specifies the name, the license and which node modules are needed in order to let the application run (*dependencies*). After the *package.json* file has been written, NativeScript will try to lunch by default the *app.js,* that is the real entry point of the app. The simplest example of the app.js file is the following.

```
var application = require("application");
application.start({ moduleName: "main-page" });
```

That, as we can see, is something very lightweight used principally to start the first screen – in the example “main-page” – of the app. When a screen is written in this way, NativeScript will look for the corresponding XML file and eventually load the associated JS and CSS, always using the basic strategy that each file must have the same name.

In NativeScript, everything orbits around the entity *Page*, a container that represent a separate screen of the app. Each Page must have an XML file that defines its structure and can have one CSS file or one JS file that can specify its appearance and its behaviour. It is important to notice that these files are not totally equivalent to their web counterpart. Let us look to an XML that defines a simple page, with the following code.

```
<!-- main-page.xml-->
<Page loaded="onPageLoaded">
  <Label text="Hello, world!"/>
</Page>
```

We can immediately spot how the tags used are not the ones of classical ones of HTML, these one, not differently from the React Native components, are used to mask real native modules.

As seen that Mobile Studio uses frequently these components, we write in the following table which one are supported by NativeScript and how they are converted to Native ones.

NativeScript component	Android Native	iOS Native
Button	android.widget.button	UIButton
Label	android.widget.TextView	UILabel
TextField	android.widget.EditText	UITextField
TextView	android.widget.EditText	UITextView
SearchBar	android.widget.SearchView	UISearchBar
Switch	android.widget.Switch	UISwitch
Slider	android.widget.SeekBar	UISlider
Progress	android.widget.ProgressBar	UIProgressView
ActivityIndicator	android.widget.ProgressBar	UIActivityIndicatorView
Image	android.widget.ImageView	UIImageView
ListView	android.widget.ListView	UITableView
HTMLView	android.widget.TextView	UILabel
WebView	android.webkit.WebView	UIWebView
TabView	android.support.v4.view.ViewPager	UITabBarController

SegmentedBar	android.widget.TabHost	UISegmentedControl
DatePicker	android.widget.DatePicker	UIDatePicker
TimePicker	android.widget.TimePicker	UIDatePicker
ListPicker	android.widget.NumberPicker	UIPickerView

Table 1 - Correspondence between Native and NativeScript components

Moreover, a separate module is reserved to create **Dialogs**. It is important to notice that, with the paid upgrade of NativeScript called NativeScript UI, more components are supported (e.g. the lateral panel in Android). [11]

We would like to focus again the reader attention to the previous snippet of code, specifically on the line `loaded="onpageLoaded"`. It is a syntax that can resemble the one used in HTML for buttons, e.g. `onclick="myFunction();"`, where the `myFunction` is bound to *onclick* event of the button. We can see how custom events are provided, like the `loaded`, but also the one corresponding to the click on a button, the one called `tap`, will be used often. Going forward over our running example, we want to show now how a JS file is written.

```
// main-page.js
function onPageLoaded(args) {
    console.log("Page Loaded");
}
exports.onPageLoaded = onPageLoaded;
```

The code here is standard JavaScript that the reader can use easily also in web pages. There is a small detail that make the difference and needs to be pointed out: the last line. We can see that the functions, in order to be seen in the XML and, generally speaking, in other files, must be exported with the syntax written above.
The JavaScript in NativeScript can be way more complex, supporting features like data binding and observables, but these are concept that will fall outside the scope of this work. We suggest to the curious reader to look at NativeScript official documentation in order to discover how to use these advanced features [12] and expand them even further with the usage of Angular [13].

After having seen how XML and JS are made, we rapidly have a glance on how styling is done looking at the following CSS line.

```
.title { font-size: 32 }
```

Nothing special for the web-expert reader: we select one or more element in the DOM, using a standard CSS selector, and then apply a styling rule. The devil is in the details, though: as it is possible to see in the line written above, we have not specified any standard measure unit (px, em, etc.): it is not an error, NativeScript does not accept any of them.

Finally, we can say that not the whole amount of web CSS rules are supported, but only a small subset. The supported styles, and also advanced techniques to use them, can be found in the official documentation. [14]

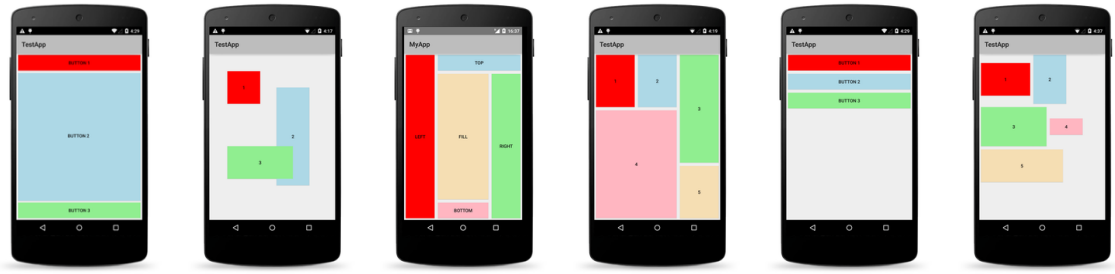
It is important to note that the CSS animation are supported and perform quite well: in the writer opinion, they are an easy and fast way to add interactivity to an app.

Although components in NativeScript can be put as simply as we have seen in the XML example written before, this is not a best-practice: it is far more convenient to put, between the page and the components, another structure: the **layout**.

Layout are crucially important when building a user interface, because they offer a convenient way to position components. This idea is not a novelty, because used also in native programming, both in iOS and in Android. NativeScript supports several layouts [16]:

- StackLayout
- AbsoluteLayout
- WrapLayout
- DockLayout
- GridLayout
- FlexboxLayout

These layouts are shown in the following table



Flexbox Absolute Dock Grid Stack Wrap

Until now we have seen how to use NativeScript as a black box, without knowing what happens “under the hood” and how the JS code is converted to native one. To discover more on this topic, we should look at the NativeScript architecture, shown in figure 10.

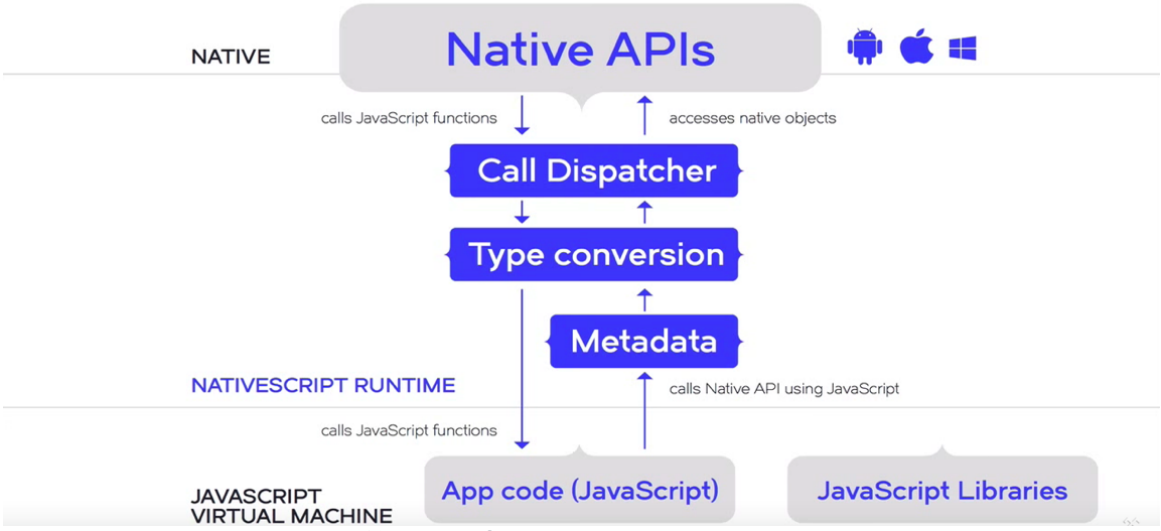


Figure 10 – NativeScript architecture schema

NativeScript is based on two JavaScript engines, Google's V8 and JavaScript Core: the first one is used for Android and the latter for iOS. These virtual machines do not have specific tools to work with these two mobile OSs, so the NativeScript runtime module injects code – C++ for V8 Engine – that customizes the ambient of the VM and insert all the native calls needed. In the case of Android platforms, NativeScript injects a global variable, with several custom JavaScript functions that are able to invoke call-backs of the JS engine, that will produce C++ code. While this is sufficient on iOS – the C++ can directly access to native Objective C code – in Android an addition layer is needed to use native components. This layer is the Android's JNI – Java Native Interface – that in Android is able to translate the JavaScript into native Java. TJ VanToll, a developer from Telerik, has provided an interesting example-based exploration of the NativeScript architecture [16].

3.2.2.5.3 Comparison of cross-interpretation technologies

Rect Native and NativeScript are both cutting-edge technology that promise to deliver the fastness of native application with the usage of easy web-like programming languages. We have talked in detail about these technologies, but choosing between them could be difficult.

As we have shown, in fact, NativeScript and React Native present strong differences. While the first is closer to the real web developing, the second seems to be more like "standard" mobile programming. React Native, in fact, allows the programmer to use real native code in Java or Swift, element that can give a huge improvement in the final app if the developer knows how to deal with these languages. On the other hand, it could be easier to have a uniform programming language that manages the logic of the app – and this is the main reason why Mobile Studio uses NativeScript at its core – and having a more classical web development approach, e.g. styling the app with CSS.

The problem becomes even tougher if we consider the main aspect that is interesting for the final user: the performance of the app. While the strong differences in their architecture, tests [18] shows that the two approaches are able to deliver apps with similar speed, both performing better on iOS rather than Android.

As the reader can see, there is no clear winner in this scenario and, in a real use case, differences in performance are negligible. We can safely say that the difference in the two approaches proposed is up to the developer, if she has a strong background in native languages she would prefer using ReactNative, where it is possible to include modules written in such languages, else if she prefer a more web-like approach the NativeScript way would be preferable.

To conclude this paragraph, it is important to say that Mobile Studio works with NativeScript – although future expansions to let it work with React Native are possible. The reason that lead to the choice of this framework resides in a more consistent usage of web programming languages, without mixed native code, that would have increased the complexity of the IDE.

3.3 The new desktop environment

3.3.1 Background

While the platforms involved are far less than the ones present in the mobile, the problem of building software that can run in every machine has always be present also in the desktop world. However, in this context, as caused by the absence of a “strong” central app store in each platform, software requisites are quite different. For example, here it is desirable to have support for automatic updates and crashes, elements that, in the mobile world, are often left to the Operating System.

While the cross-platform idea of software also in desktop is not a novelty – we can think that the Java Virtual Machine could be one of the first ways to achieve it – the implementations that inherits concepts from the mobile world are quite new. We will spend a lot time talking about the usage of web technologies in desktop software developing: the idea here, as Phonegap does in the mobile world, is to use such technologies like JavaScript, HTML and CSS to write the software, that is then “wrapped” in an environment able to give to this web-app integration to all the functionalities that the host OS provides. This is allowed by the usage of JavaScript both as a front-end and a back-end language, thanks to well know libraries like Angular and Node.js.

In the writer opinion is it worth to spend a bit of time talking about the evolution of such undelaying technologies, that made a simple scripting language made to have interactive DOM elements a powerful tool able to access file system, write files, etc.

3.3.1.1 A kick start introduction to HTML5, JS and CSS3

Since the web was born, **HTML** – Hyper Text Markup Language – has been used to define the content of a web page: it was the first type of language that, using a specific syntax, allowed to define the structure of all the elements inside it. We can see an example of such language and such structure in Figure 11.

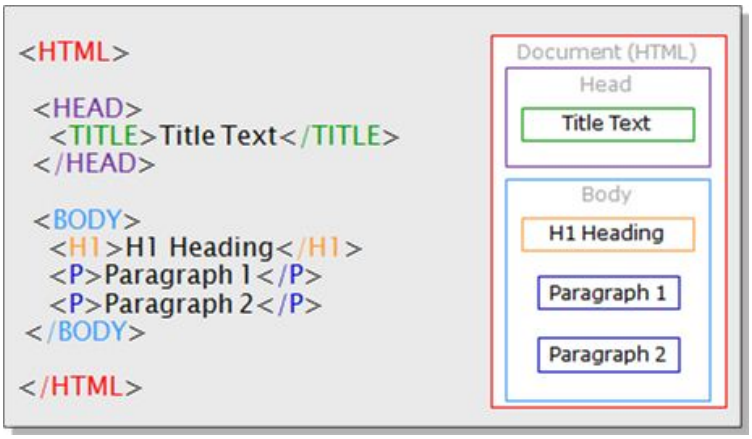


Figure 11 - HTML and the structure of a web page

As we can see, HTML uses *tags* in order to define the structure of the web page. In the language, a tag is an element enclosed by curly brackes, like for example <HTML>. An HTML element is thus specified by the combination of an opening tag – like the <HTML> that we have seen before – and a closing one – e.g. </HTML>

– that define its scope: inside it, other tags can be present and thus other elements contained by the parent.

The latest version of HTML, the 5th, has full support for all the most modern devices, like smartphones. It includes new tags – like *section, aside,* etc. – in order to build more complex layouts and tries to be less dependent from external programs: before HTML5 it was common practice to embed in a page modules written in other languages – e.g. Flash videos, Java applets – that could potentially slow down the rendering of that page.

In particular, we can see the new elements introduced in HTML5:

- Semantic elements like <header>, <footer>, <article>, and <section>
- Attributes of form elements like number, date, time, calendar, and range
- Graphic elements: <svg> and <canvas>
- Multimedia elements: <audio> and <video>

But more importantly new API's were added:

- HTML Geolocation
- HTML Drag and Drop
- HTML Local Storage
- HTML Application Cache
- HTML Web Workers
- HTML SSE

These API allowed to give access to the sensors present in new devices, handling touch better and, overall, speed up the visualization of a page.

While HTML is used to define the content of a webpage, its dynamic behaviour is left to **JavaScript**. This language, introduced in 1995 by Netscape, was born with the idea to add interactivity to static DOM elements. JavaScript is an *interpreted* language, this means that the instructions are not executed by the target machine, but from other programs – we will discuss about JavaScript Engines in the next paragraph – that allow a "on the fly" execution, because there is no need to run a compilation stage. Moreover, the fact to be such a kind of language allows JavaScript to be incredibly flexible, supporting multiple methodologies of programming, like imperative, functional and object-oriented. It's in fact a recent addition to the language the possibility, for the programmer, to use classes, a well know concept in Software Engineering that was absent from the original specifications of the language. Object could still be done by the usage of closures. As in our software classes were used in order to build the model, we show how to declare one in the following code.

```
class Screen {
    constructor(id){
        this.id=id;
        this.components = {};
        this.numberOfComponents = 0;
        …
    }
}
```

The reader used to OOP will immediately notice some differences with the standard usage of classes: as usual in JavaScript, there is no need to declare variables, neither their type, that is automatically inferred from the value they will contain. There is no *information hiding* concept, so neither *getters*, nor *setters.* All attributes, can be visible in every scope where the class is instantiated.

The instantiation of a class is not different from other languages:

```
var newScreen = new Screen("win"+newId);
```

Where a new object is created, calling the constructor of the class and passing to it the parameters.

As we can see, JavaScript met a great evolution during the years and a lot of libraries extended the capabilities of this language, letting its usage expand also to the back-end side of web applications. Moreover, thanks to modern operating systems that detached the concept of the JavaScript virtual machine from the browser, putting it in lower level of abstraction, made the programs built with it incredibly fast and reactive, and so JavaScript became a perfect language to make software also for desktop.

The last component of the trio needed to build a proper web app is **CSS** – Cascade Style Sheet. CSS is a simple language, consisting in a set of rules that change the appearance of DOM elements. We can see a simple rule in the following snippet of code.

```
.android.label {
    margin-top: 2px;
    margin-bottom: 2px;
    color: black;
}
```

The first element that we can see from the CSS example is the *selector,* a string used to select one or more elements from the DOM. Common selectors are a tag name – e.g. `body` – a class, preceded by a dot – e.g. `.android` – or IDs, preceded by an hash – e.g. `#Label1`. It is important to say how these selectors are nestable, in the example we are looking at the element that has class `label` of the ones that have class `android`.

After having selected one or more items, we can find, in the scope of the element, all the styling rules that can will be applied to it.

In the years, the set of these rules expanded rapidly, sometimes also stealing graphic effects – e.g. *box-shadow* – that could be done with JavaScript. This was to enforce the separation between content, logic and visualization that it's a perfect real life example of the pattern MVC.

3.3.1.2 Advanced Concepts

As we have said, in the year web technologies were incredibly successful and simple languages became more and more complex, allowing to perform tasks usually reserved to ones far from the web world. We will look to some of the ideas and framework that allows the web methodology to be more powerful, ending in a discussion of how is possible to really build desktop software with these.

3.3.1.2.1 JavaScript Engines

When talking about JavaScript we stated that this is a *interpreted* language, a language so that must be executed by another software: a JavaScript Engine, that is any Virtual Machine able to interpret JavaScript code and execute it. Usually, these engines are built-in in the browser, but more and more we can see their integration in the OS itself [19]. This, joint to the incredible evolution in performances of these engines, made them mature for desktop-class application development.

To understand what truly a JS Engine is, we can look at how the most two famous ones work: WebKit's JavaScriptCore and Google's V8.

JavaScriptCore performs a series of steps to interpret and optimize a script:

1. It performs a lexical analysis, breaking down the source into a series of tokens, or strings with an identified meaning.
2. The tokens are then analyzed by the parser for syntax and built into a syntax tree.
3. Four JIT (just in time) processes analyze and execute the bytecode produced by the parser.

Google's **V8** engine, written in C++, also compiles and executes JavaScript source code, handles memory allocation, and garbage collects leftovers. Its design consists of two compilers that compile source code directly into machine code:

- **Full-codegen**: a fast compiler that produces unoptimized code
- **Crankshaft**: a slower compiler that produces fast, optimized code.

If Crankshaft determines that the unoptimized code generated by Full-codegen is in need of optimization, it replaces it, a process known as "crankshafting".

3.3.1.2.2 Node.js

Node.js is one of the most popular JavaScript framework to date. Based on Google's V8 JS Engine, Node.js has the ability to manage I/O operations asynchronously with its event-driven architecture. It is used frequently as a back-end language, where it has replaced the old PHP for the simple reason that most functions in this latter language are blocking – commands execute only after previous commands have completed – while Node.js, with the usage of parallelism and call-back functions, is totally designed to be non-blocking.

Node.js is based on modules, able to handle most of its core functionalities. Using a module is simple and it is done with the require statement.

```
const osModule = require('os');
```

The above line of code load the `os` module in the program and let its usage by accessing to the `osModule` JavaScript object. An example of this usage could be the following.

```
if (osModule.platform()=="darwin"){
```

```
        //Some specific code for macOS
  });
```

Modules can be way more complex and allow the programmer to interact with the OS. The following example shows how a new process is created and how it is able to execute a terminal command.

```
var childProcess = require('child_process');
var cmd = ”cd /Desktop”;
childProcess.exec(cmd, {cwd: path }, function(err, stdout, stderr) {
        //Callback function to execute when command is executed
});
```

Where the `exec` method of the object `childProcess` allows the execution of the command `cmd` in the path specified in the variable `path`. It is important to notice that this will result in an asynchronous execution of the process that, when it will end, will invoke the call-back anonymous function passed as third parameter of the `exec`.

More complex action, like file managing, are possible with Node.js and we invite the curios reader to follow the official documentation [20] and look to the interesting paper of San Souci and Lemarie [21] to have a glance on the complex Node.js architecture.

3.3.1.2.3 Chromium

It is worth to spend some time talking about Chromium, an open-source web browser at the hearth of the more popular Google Chrome. Born in 2008, the main feature of this browser, that in the time was unique, was the presence of tabs to browse internet pages. Each tab runs in a separate process called *renderer*. Thanks to sandboxing such processes have separate memories that, by default, cannot be accessed by each other. Communication between them is possible only thanks to a message-passing style protocol called IPC – Inter Process Communication. The process that should use most this functionality is the *browser* one, that, differently from the *renderers*, is unique and has the main scope to manage all the *browsers*. This architecture is shown in figure 12. Detailed information on how this is done and works can be found in the official website of the chromium project. [22] [23]

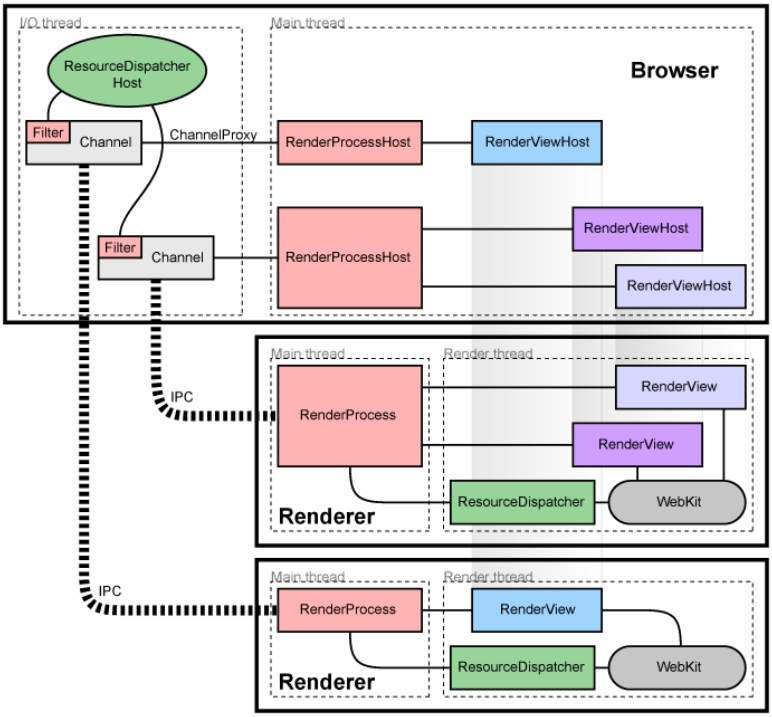


Figure 12 – Chromium architecture

3.3.2 State of art of desktop programming with web technologies

As we have seen in the mobile scenario, a common way to build a modern software is to write a web app and then wrap it around a container that allows it to communicate with the host Operating System. This new way of programming is starting to be diffuse also in desktop world, considering the faster JS engines and the presence of more computational power present in this world. This new kind of software bears an *inertly* multiplatform experience and a strong easiness to be programmed. Some examples of successful apps programmed with this methodology are the following:

- Spotify – music player
- Brackets – web code editor
- Visual Studio Code – code editor
- Slack – team messaging

And many others. While many of these implementations defined a custom web wrapper, one solution let us investigate on how is possible to adopt this style of programming: the one called Electron. We will talk in-depth of this framework in the following paragraph.

3.3.2.1 Electron (former Atom Shell)

The first name of Electron was Atom Shell, a name that strongly underline the ancestor of this project: Atom code editor. This editor supported by GitHub, was written with web technologies, wrapped in a native container that used node.js to communicate with the filesystem. GitHub decided to let this wrapper be open

source and strongly supported the community that, using it, is developing more and more software with this method.
Electron has an architecture closely related to the open source web browser Chromium, being composed by three main parts:

- One main process, the *browser*
- Multiple *renderer* processes
- A set of utility modules

Every Electron app, is composed by three file types:

- JavaScript scripts
- HTML and CSS markups
- Package.json

Where the latter is a file that specifies the entry point of the application. An example of such of file could be the following:

```
{
  "name"    : "sample-app",
  "version" : "0.1.0",
  "main"    : "main.js"
}
```

Where we can see that the entry point is a JavaScript file of the *browser* process, able then to initialize the *renderers* and thus present an UI to the user.
An example of such a `main.js` script file can be one in the following snippet of code.

```
//Modules initialization
const electron = require('electron');
const {app} = electron;
const {BrowserWindow} = electron;

//Variables definition
let window;

//Browser creation
function createWindow() {
  window = new BrowserWindow({
      width: 800,
      height: 600
  });
}

window.loadURL(`file://${__dirname}/index.html`);

window.on('closed', () => {
    window = null;
});
```

Where we can see how a new *renderer* window is created instantiating the object `BrowserWindow` and specifying some parameters, like the width and the height, in the constructor. After that the browser has been created it is possible to let it display some content for the user, specifying an HTML document with the `loadURL` method. The window is not different from any other JavaScript object and, as we can see in the example, some listeners could be attached to it: it's the case of `on('closed',…)` present in the code.

Both in the *browser* and in the *renderers* Electron allows the usage of modules. Modules are crucial to let the web app communicate with the host OS and are something that resemble the *plugins* that we have seen in Phonegap, when talking about mobile hybrid development. Electron modules are a little more complex and powerful, though: while the programmer could use Phonegap plugins writing pure JavaScript, Electron modules are Node.js modules, so they should be written and used according to Node syntax.

Let us see an example of how a module of such a kind can work.

```
var fs = require("fs");

fs.readFile(__dirname + "/test.txt", function (err, data) {
  if (err) throw err;
  console.log(data.toString());
});
```

We specifically used the File System Node.js module in order to read the file `test.txt`, using the `readFile` method of it. The content of the file is then displayed in the console of the *renderer*.

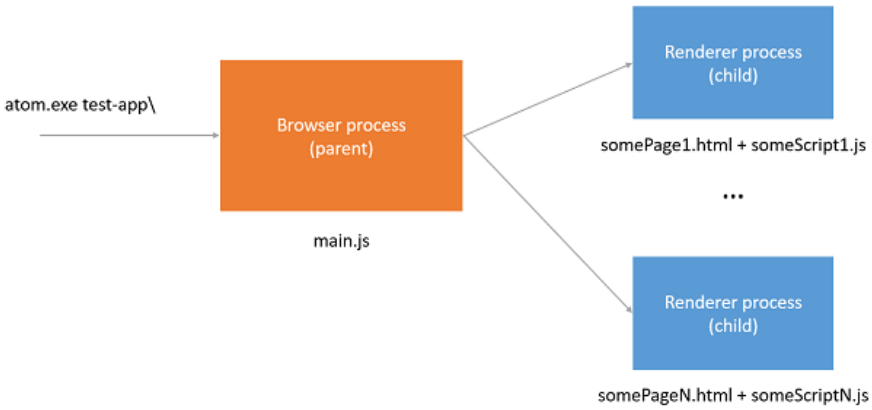


Figure 13 - Browser/Renderer execution in Electron

The communication between *Renderers* and Browsers is not as straightforward as one can imagine: the careful reader can recall that the processes in Chromium are sand-boxed and cannot access the memory of each other directly. When we have talked about Chromium we spoke also of the IPC message passing protocol that filled the gap, but Electron has also another method that allows *renderers* to invoke functions of the *browser:* the one called *remote*. Let us consider first the case of the classical IPC, with a simple example.

```
// In the browser

var ipc = require('ipc');
ipc.on('asynchronous-message', function(event, arg) {
  console.log(arg);
  event.sender.send('asynchronous-reply', 'pong');
});

ipc.on('synchronous-message', function(event, arg) {
  console.log(arg);
  event.returnValue = 'pong';
});
```

```
// In the renderer

var ipc = require('ipc');
console.log(ipc.sendSync('synchronous-message', 'ping'));

ipc.on('asynchronous-reply', function(arg) {
  console.log(arg);
});

ipc.send('asynchronous-message', 'ping');
```

Where we can see that each process has to instantiate the IPC Node.js module, then the renderer sends, both in a synchronous – and thus blocking – way and in an asynchronous one the message “ping” to the *browser,* that displays it and then reply sending back a “pong” message in both cases. We can see that we have to listeners, in the browser, one for the synchronous message and the other for the asynchronous one.

The usage of *remote* is by far simpler, but the interaction between renderers and browser is by far less complex, allowing only the firsts to access objects and methods of the latter. We can see an usage example of this technique in the following example.

```
var remote = require('remote');
var BrowserWindow = remote.require('browser-window');
var win = new BrowserWindow({ width: 800, height: 600 });
win.loadUrl('https://github.com');
```

We can see how the *remote* module is instantiated, then it is used from the *renderer* to build another *renderer* window, behavior permitted by the invocation of the `BrowserWindow` object that is owned by the *browser* processes and accessed by the *remote* module.

As we have discovered, IPC and remote are modules. Modules, in Electron world, can be used by *browser*, by *renderers* of by both and allow a complex communication between them and with OS.

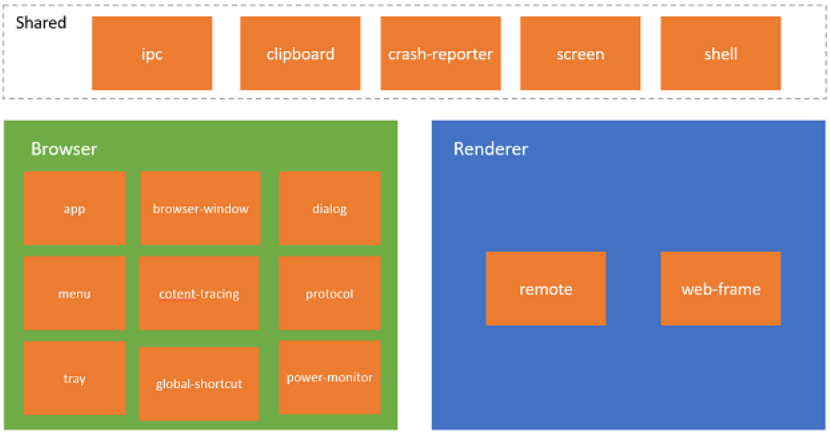


Figure 14 - Modules in Electron

3.3.2.2 A brief comparison with classical technologies

Unfortunately, a consistent way to compare performances among software produced by Electron and software produced by standard programming – Java, .Net, etc. – is not present and difficult to realize, falling outside the scope of this work. It's important to state that, as we have discovered, the main difference is, more than in performance, in the approach that the programmer can take when developing a new software. As we said a lot of times, JavaScript can be seen as a more flexible way to program an app: with standard software developing – e.g. with Java – the programmer is asked to choose a programming style *a priori*. This does not happen in JavaScript, where the code can be used in an Object-Oriented, Imperative or Functional way, also mixing all of them, depending on the skills of the developer. It is important to state that this is not always an advantage, because it can lead to confusion in a team.

3.3.2.3 Brackets: the state of art of JavaScript programming in desktop

We would like to conclude this section giving to the reader an example of application of the technology that we introduced, the one that let to program a desktop software with JavaScript. While in fact we can define this technology quite novel, as no scholar studies have been done regarding it, some popular products arose. As we have said, one example could be Adobe Brackets. [26]

Brackets praise itself to be a web editor programmed with web technologies, in this way easily expandable and customizable. It does not use Electron as a framework to run the HTML that compose it, but uses a custom solution, that is still based on Chromium [27] and has been called Brackets-shell [28]. This module manages all the back-end of the software, letting it be able to communicate effectively with the file system, not differently on how Electron allows its developer to do. It is crucial to notice that the main difference between these two approaches is that Electron allows developer to use its platform to build apps, while Brackets-shell is a software owned by Adobe and used only for Brackets. Yet, being open-source, the possibility to use it to run a custom web app, in a similar way as we have seen is possible to do with Electron, exist [29], but no official support is provided.

The front-end of brackets uses standard web libraries, interestingly many of them are also used by the solution that we provided: an example could be CodeMirror a library used by Brackets to display a Code Editor that we also used in Mobile Studio.

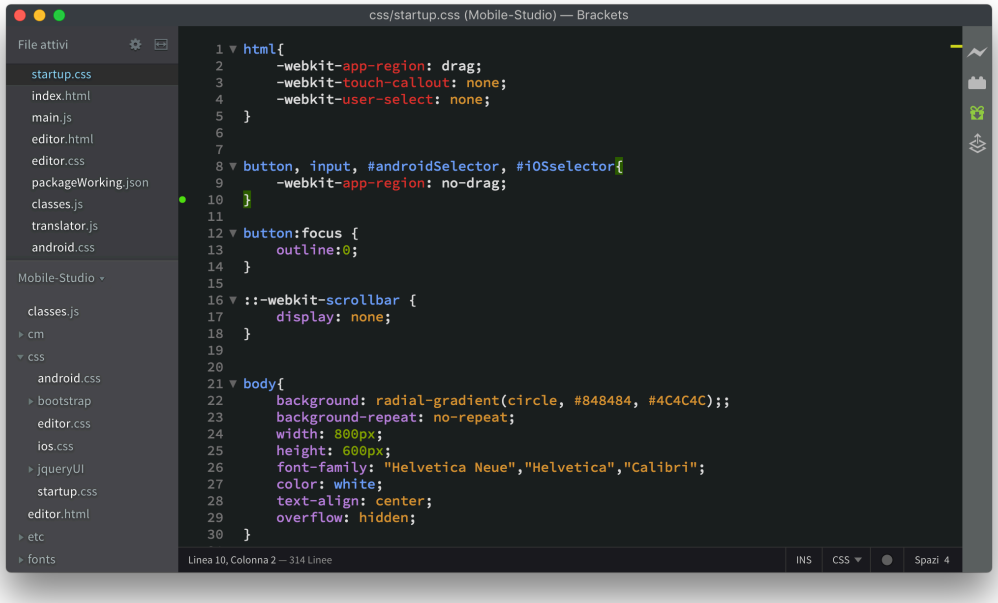


Figure 15 - Adobe Brackets main screen

3.4 The link between the two worlds: IDEs & SDKs

In previous paragraphs we have discussed about the latest technologies in the development of desktop and mobile software, but we have not considered the applications that let the user build an app for mobile from desktop platforms, linking these two worlds. Two macro-suites of software are often used to this specific purpose: SDKs and IDEs. The difference behind these two terms is not often understood well and we will try immediately to shed some light.

An SDK – Software Development Kit – includes the necessary building blocks that let the programmer develop a certain application: compilers, debuggers, libraries, framework, even hardware – in case of embedded system software development. An IDE – Integrated Development Environment – is sometimes contained in an SDK and provides a better way for the programmer to use all the tools inside the kit. In its basic form, an IDE consist of a source code editor and a debugger, but many IDE encapsulate also tools of SDK, like compilers.

The main part of an IDE, where the programmer will spend most of its time, it's without any doubt the code editor that, providing a visual UI and advanced features like code-completion, let the programmer write code in the easiest way possible.

Considering that, with the current work we are proposing a new IDE, it is in our opinion important to let the reader have a clear picture of what these suites of software are and what are the most used tools of these kind.

3.4.1 Native Examples

It is clear that the most advanced IDE and SDK are the one devoted to native app programming, because supported directly from the software houses that build the OS of the devices – Apple and Google. We will look in detail at each of these official solution.

3.4.1.1 Android SDK

In Android the distinction between SDK and IDE is clearly denoted because the latter is not in the default installation of the first and many IDEs can be used with Android SDK. In the kit we can find tools in order to provide a virtualization of a device – e.g. Android Emulator – specific development programs, – e.g. SDK Manager – software that allow an easy debugging, tools able to build the app itself – e.g. *apksinger,* that allows to check and create digital signatures for final apps – and, finally, platform-specific tools. A comprehensive guide to all these tools can be found in Android SDK official documentation [26].

Theoretically, with just these software is possible to interact with the SDK command line and thus build software for Android. This is rarely done, though: often an IDE is used to build an app in an easier way.

Android developers often used third-part IDE like Eclipse or Netbeans to this purpose, that can communicate with the SDK using plugins – e.g. NBAndroid for Netbeans [27] – but since 2014 the most used IDE for this platform is the official one, Android Studio.

Based on IntelliJ IDEA and built by Google, Android Studio offers advanced functionalities, like Instant Run or the intelligent code editor, that help the developer in each stage of the construction of the app [28]. In particular, we would like to focus the reader attention both to the code editor itself and to the UI builder. The latter, shown in Figure 15, allows an easy construction of the User Interface of the app simple by drag and drop of components. Differently from others IDE, like Xcode, Android Studio permits also to edit the UI simply modifying the XML associated.

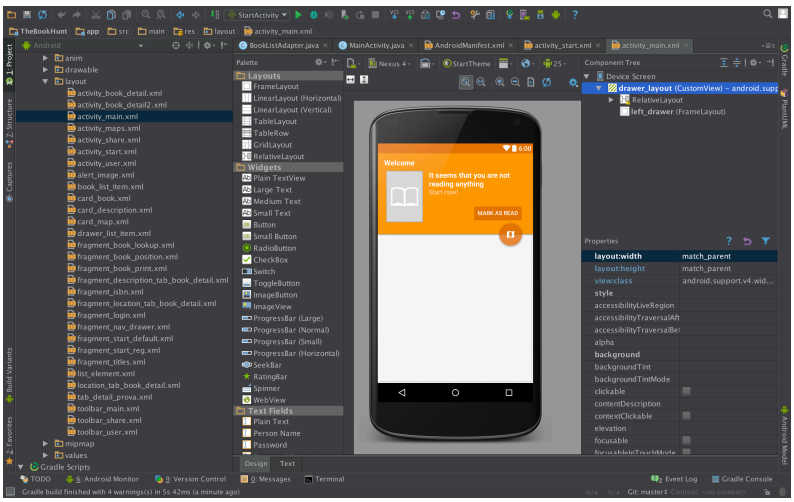


Figure 16 - Android Studio with UI builder opened

3.4.1.2 iOS SDK

While we have seen that in Android SDK and IDE were two separate elements, and a programmer can use different IDEs with the kit, in iOS this does not happen. The IDE found as part of iOS SDK, named Xcode, is the only one that can work with it and build iOS applications. The fact that this software runs only on macOS has always been a difficulty for a programmer in order to work with this popular platform.

iOS SDK is composed by a set of different tools, based on pre-existing macOS X technologies [29]:

- **Cocoa Touch**
- **Media**
- **Core Services**
- **Core OS (Mac OS X Kernel)**

The firsts are able to handle smartphone specific features – like multi-touch, or accelerometer – while the latest are closer to macOS system and provide access to low level features – OS X kernel, Sockets, Power Management, etc.

It is important to mention a cause of heavy criticises to this framework and to iOS itself: it does not allow the execution of other executable code different from the one written in Xcode. This is the reason why no Java Virtual Machine, and thus Java software, or .Net runtime can run on iOS. This rule, stated in the iOS SDK Agreement, lead also to the decline of the popular Flash content in the web, that could not be seen on iOS.

Interestingly, iOS apps can be programmed in two languages: Objective-C and Swift. While the first in an evolution of the popular C language and shares many features with C#, the latter is a technology recently introduced by Apple itself – it first appeared in 2014 – with the idea to be more easy, flexible and able to deliver more performance in respect to the classical Objective-C, that Apple has been using since its birth – the language itself has roots in 1980s.

The details of these languages fall outside the scope of this work, but detailed guides on Swift can be found from Apple [30] and interesting comparison between them are argument of debate since the introduction of Swift [31], but we can say without fear that Apple supports this new language and it is clear that this should be the way to program iOS app – and, in general, any software in Apple ecosystem – from now on.

As we said, iOS SDK can only be used with the standard IDE of Apple: Xcode. Similarly to Android Studio, this software has advanced feature, like code completion, that allows the developer to easily build its software. Also similar to Android Studio is the UI creator, that in Xcode is called Interface Builder – and, before Xcode 3, was a separate program – that let the developer create an UI visually with drag and drop, as shown in Figure 16. It is important to notice that, differently from Android Studio, Xcode does not permit to access to the XML code of the UI.

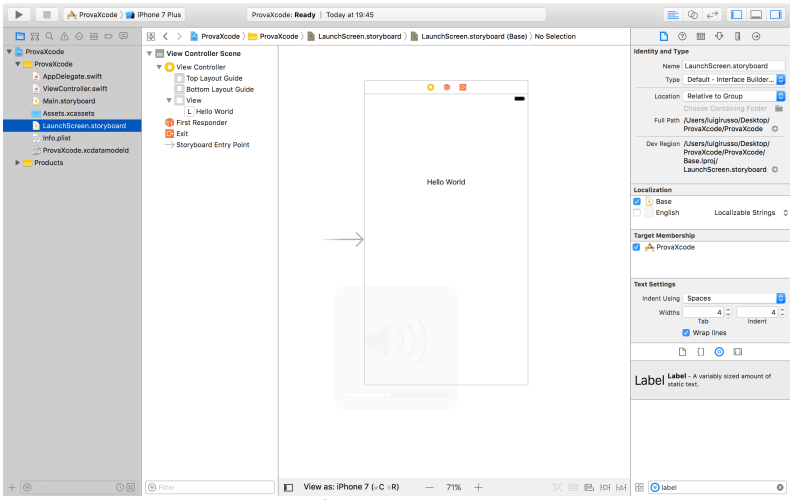


Figure 17 - Interface Builder inside Xcode

In Xcode, the visual elements are more predominant than in Android Studio. Screens are grouped in *Storyboards,* elements that should follow the interaction of the final user. Actions can be inserted by drawing lines between elements – that can be buttons, screens, but also functions in code.

When the developer is ready to test her app, he can use another element of the SDK that communicates perfectly with Xcode: the iOS Simulator.

3.4.2 Cross-platform Examples

While SDK are present for every cross-platform method that we have discussed, only a few can compete with the complexity and the richness of native ones. When considering hybrid development, for example, an SDK can be the one of Cordova, that simply let the programmer be able to call some specific JavaScript functions and wraps up all the web files to output an application. Yet, no advanced debugging technologies are provided and no advanced tools – like a device emulator – neither. The only approach that is able to offer a complex SDK, joint with an advanced IDE, is the one of cross-compilation, delivered by Xamarin. We will look in detail on its IDE in the following paragraph.

3.4.2.1 Xamarin Framework

As we have seen Xcode and Android Studio are IDE intended to develop an app in a native way and are tailored to the platform that they support. Xamarin, instead, is a cross-compilation framework that is able to address iOS, Android and also Windows Phone. To do this, two IDE are supported: a specific Xamarin Studio for macOS and Microsoft's Visual Studio for Windows.

The Xamarin Framework is composed principally by one of these IDE and the complex compiler that is able to produce native platform specific code. There are also many more modules, that let the programmer perform complex tests or build rich UIs, but they are additional – and expensive – add-ons. The reason to this apparent simplicity can be found in the fact that the compiler has to communicate with other SDK – the ones of the platform – in order to build the final running app.

In particular, Xamarin let the programmer access to native functionalities using C# talking in different way with different SDK, specifically:

- **iOS** – Xamarin.iOS exposes Apple's CocoaTouch SDK frameworks as namespaces that you can reference from C#. For example the UIKit framework that contains all the user interface controls can be included with a simple using MonoTouch.UIKit; statement.
- **Android** – Xamarin.Android exposes Google's Android SDK as namespaces, so you can reference any part of the supported SDK with a using statement, such as using Android.Views; to access the user interface controls.
- **Windows** – Windows apps are built using Visual Studio on Windows. Project types include Windows Forms, WPF, WinRT, and the Universal Windows Platform (UWP).

The programmer can write C# code in Xamarin Studio or Visual Studio, depending on which OS she has. Both these IDE have advanced features, in particular Visual Studio has been long prised for its auto-completion module, *IntelliSense.* Xamarin allows both these editors to build a mobile UI visually, with an interface that strongly recalls the one of the specific editors for the platforms. For instance, we can find the same *Storyboard* metaphor seen in Xcode when programming for iOS.
An example of this interface is shown in figure 17.

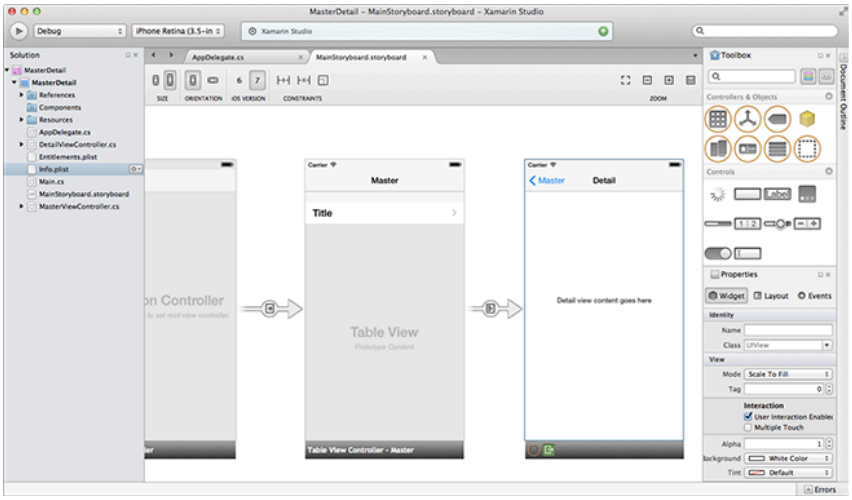


Figure 18 - Storyboard in Xamarin Studio

3.5 Time for stocktaking

In this chapter, we analysed the most used options to develop software both in mobile and in desktop world. Considering the first, we have seen extensively which technologies are now present in the field, hoping to have given to the read a clear idea of advantages and disadvantages of each one, and letting her choose between them with critical spirit if asked to build an app. To further help in this side, we present a table that sums that takes the most important KPIs that emerged from our analysis.

Technique Name	Device Supported	UX	Access to Sensors	Development Easiness	Development Language
Native	---	+++	+++	-	Java/Swift
Pure Web	+++	---	---	+++	HTML5
Hybrid	++	--	-	++	HTML5
Cross-compilation	++	++	++	--	C#
Cross-interpretation	++	+	+	++	Simil-JS

Table 2 – KPI comparison of various mobile app programming methodologies

We considered also desktop software, coping with technologies that, in our opinion, are innovative and able to let programmer build complex programs with ease. In particular, we analysed how web technologies are able to deliver software also in this world, with the same easiness to be programmed that characterize them. At the end, we have seen briefly some examples of complex desktop programs, the IDE, that are able to connect mobile and desktop programming, being complex software that let the programmer build apps for mobile.

The challenge that we will face in the next chapter will be to try to build such complex programs with the technologies that we have seen: using Electron for desktop part and NativeScript for the mobile. This will be a real stress test on how these “trendy” approaches, that are getting more and more popularity in these time, can really be used in serious software developing.

4 Mobile Studio

4.1 Introduction

As we have seen from the last chapter, the clear majority of IDE and SDK are devoted to the “classical” native programming and only a few exist for the new technologies that permits innovative ways of building mobile apps. This has the effect that many developers are not aware of the benefits that these approaches can give, or find too difficult to use them. To fill this gap and to test cross-platform approaches in desktop world, we built an IDE able to use cross-interpretation methods. This IDE offers an easy visual UI that allows the programmer to build an app with easy and familiar methods, like drag and drop of components. The programmer can have immediately a glance of how the UI of her app will be in multiple platforms, simply looking at the preview offered by the editor, both for Android and iOS. Finally, the editor, constantly updating an underlying data layer that models the app, is able to output code in cross-interpretation languages, convert it into a native application and run it in a real device, or in an emulator.

It is important to state that, although the model was done to be as more development method agnostic as possible, in the current version it is translated to the cross-interpretation language NativeScript, that was the one that in our opinion was the most flexible and the most easily usable with an IDE written in pure JavaScript. The IDE translates the model and does further editing using the specific syntax of the language. Using NativeScript compiler then, the code is converted in real Android and iOS native projects, that can be opened and further modified using specific tools – e.g. Android Studio and Xcode. This, in our opinion, is another strength of our program: the ability to let the developer decide at which point stopping being “cross-platform” and start being “real native”, opening the project in a specific IDE for the specific platform and customizing it for that platform. This is totally up to developer: she can use Mobile Studio for the entire process and write everything in NativeScript language, or she can use only a little our IDE, for instance to build a common UI for Android and iOS, and then use specific tools to write the logic of the app in a native language.

As we will discover in the following paragraphs, the IDE has been developed with Electron, thus building a heavy software using this framework can represent an interesting “stress-test” for it, that let us investigate on how efficiently this method is able to build reactive cross-platform desktop applications.

All the logic of the software has been written in JavaScript, with large usage of Node.js modules, while the structure of the page with HTML and all the graphic with CSS. We will see if this paradigm is able to provide in a desktop environment

the same rich, and definitely successful, user experience that it is able to offer right now in web environment. Before going in deep into the technical explanation of how each component of the software works, we think that a high-level view of its structure would let the reader understand more easily this complex software.

We can see it as the stack visualized in Figure 19.

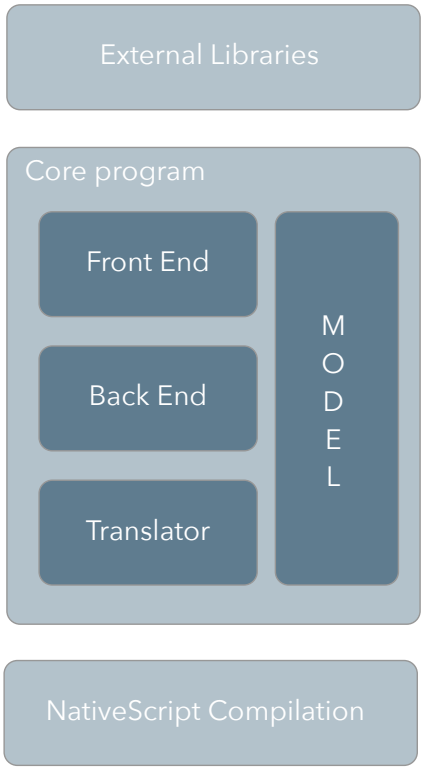


Figure 19 - High level schema of Mobile Studio

The reader will immediately be able to spot two extra layers external to the main program. The first is composed by external libraries: as we will see, Mobile Studio inherits all the characteristics of a web application, for this reason it uses many functions that have been developed from third party – an approach that is common in web developing, but it is not rare to be found in all Computer Science projects. A fast example of these libraries, for the web-expert reader, could be the inclusion of jQuery. This very example let us immediately point out that these libraries were used horizontally in any of the other modules of the software – jQuery in particular is the most widespread one.

Outside the core of the program we still find another module: the one of NativeScript, the framework that we have chosen to use in order to build real native apps from the code produced by Mobile Studio. It should be clear that all the logic behind this framework cannot be controlled in any way by Mobile Studio, so it uses this module like a black box. It is also important to point out how external this module is, because, as the translator module is currently able to output only NativeScript-compatible code, it is in our intention to let the core program be agnostic about the framework used to convert JavaScript in real app code. This, in order to support more framework as possible – e.g. React Native.

We can now introduce the main part of the software: its core. For a high-level view, we can say that, as any web-based application, Mobile Studio is composed by a Front-End and a Back-End part. The first is the one that is responsible to deliver to the user an interface to let her use the software and its challenges are to

let this interface be as easy and responsive as possible. We tried to achieve this result using the most modern HTML5 techniques, that involved for example CSS3 animations. This part should communicate with the Back-End one, that is responsible of everything that happens “under the hood”: access to disk, creation of new processes, etc. In the Back-End Node.js have been used extensively, a framework that let us be able to write all the software in one language, JavaScript – as we have seen there still exist web projects where this part is written in other languages, like PHP.

We will see how this logical view will be more complex in reality, because more screens are present and each of them will have a front end and back end part.

The last two modules that we are going to introduce are less common in classic web app developing. We can start from the Model, a data layer that will contain each detail of the app that the developer is building and that should thus be updated every time she does a change on her project. As we said, the presence of such data layer is not common in classical web developing, still the idea is not a novelty, because used in certain enterprise-level software – e.g. Google Analytics. Like in these implementations, our data layer must be active in the clear majority of the execution time of the software, being frequently updated and properly stored in the hard drive of the user, when she closes the software. The model, in fact, containing all the details of the app, is the data structure necessary to restore the app that is being developed in any time.

The real uncommon module is the last one, the Translator, that is something totally customized for Mobile Studio that contains all the functions and objects that let the software convert the model to a code that can be taken in input from the NativeScript logic and, thus, being converted to a real running app. As we will discover, this process is totally transparent to the user, that only clicks on the “compile” button, but hides a strong complexity that we will fully explain in the next paragraphs.

4.2 Core Ideas

After a bird-eye view on the structure of Mobile Studio, we think that it is important to analyse all the ideas that lies behind this software and, at the end, let it work. This is crucial because, as we will see, many were totally custom for this new environment where Mobile Studio lives: the desktop.

4.2.1 A web software living in the desktop

One novelty that characterize Mobile Studio is that it is a software written using web technologies. This means that the structure of the views is written in HTML, its UI defined with CSS and all its logic with JavaScript, as we have seen in chapter 2. Such an approach is made possible by the Electron framework, that uses Chromium in order to build *browsers* processes that, as we have seen, are not different from the classical browser that one can use to surf the Internet, but they have also other functionalities specific to interact with the OS, like a software programmed in any other “classical” programming language. We have seen how this approach is the dual, in the mobile world, of the one used by PhoneGap (Cordova).

Mobile Studio inherits also the concepts of a Single Page Web application [42], that we can basically define as a singular web page that uses JavaScript in order to display different content, rather than a more classical approach where the content is defined in HTML and, to change it, the user should move herself to a different webpage. The latter, old, approach, is less and less used nowadays, because the need of a new page loading creates an overhead noticeable for the user and thus worsen the overall experience.

The usage of the new SPA approach adds, though, a level of abstraction, because it enforces a separation between *physical* and *logical* screens: the first is the one defined in a single HTML document, but as said multiple screens can appear in such a document, we will call them logical. The hierarchy between physical and logical screens present in Mobile Studio is shown in Figure 19.

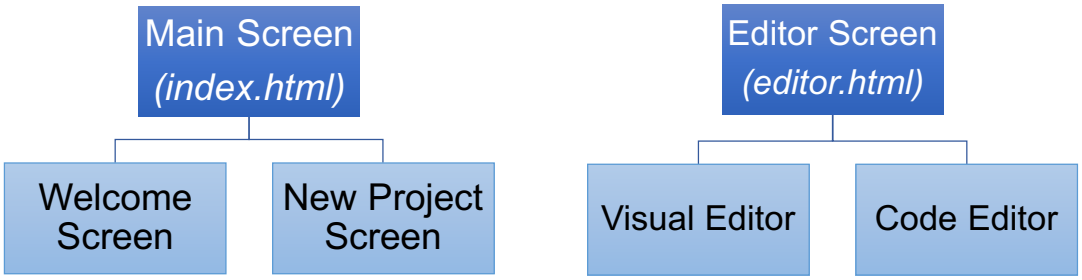


Figure 20 - Hierarchy between physical pages (dark blue boxes) and logic ones (light blue ones)

Each physical screen is characterized by an HTML structure, one – or more - JavaScript file that defines its logic and a set of CSS sheets that define its styles. We will look in detail how to present multiple interfaces in a single HTML file when we will talk specifically of the UI of the program.

Each physical screen is executed in a specific *browser* Chromium process, that present a window and opens the HTML. The software, thus, will have two processes of this kind. Yet, it important to notice that these are not the only processes present, because a Main one is needed to manage the browsers – as we have seen when talking about Chromium – and several other thread are executed by node.Js in order to perform very specific tasks (e.g.: access to file system).

4.2.2 The model based approach

In the introduction of this chapter we have discussed how we Mobile Studio uses a model-centric programming style. This means that the information of the app that is being developed inside the editors is not directly translated to NativeScript code, but it stored in an object that is agnostic of the framework used to translate it and it is external to all the module of the software itself, in order to be easily shareable and edited by any of them.

This object is updated constantly, as soon as the user interact with the app that she's developing: each UI edit – e.g. the change in the color of a button – should immediately be pushed to the model. It is thus mandatory to have a perfect one to

one correspondence of actions that could be done visually and properties of the model. Interestingly, in the various stages of the app could exist a *submodel,* an object that, having the same syntax, has only a subset of the information of the original model. The submodel is used principally when the Visual Editor should display only one specific screen, instead the complete set of them present in the project. In this context, the submodel can be considered a *projection* of the real model, containing the all the information of it, but only relative of the specific screen that should be displayed. The Visual Editor can accept this other object because indistinguishable from its syntax is the same as the one of original model. It is crucially important to merge the information in the submodel, eventually modified by the user with the Visual Editor, with the original model, in order to not lose other screens. At the end, in fact, the Translator module expects the real model to translate it in specific NativeScript syntax and execute it.

We will cover in depth the syntax and how the model has been realized in paragraph 3.6.1.

4.2.3 Visual and Code Editor – 2-way programming style

As we briefly seen in paragraph 3.2.1, the editor physical screen can be divided in two logical screens: the Visual Editor and the Code Editor. We can see the UI of both in figure 21 and 22, respectively.

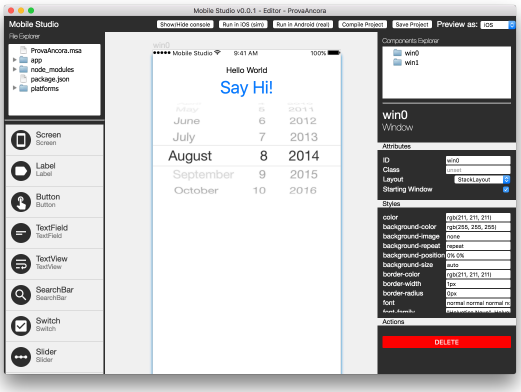


Figure 21 – Editor Window – Visual Editor

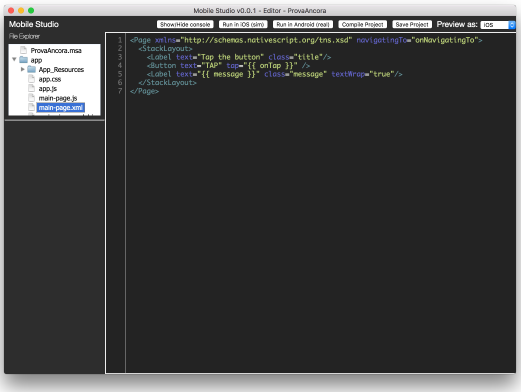


Figure 22 – Editor Window – Code Editor

The first, that we can see in figure 21, is the one able to provide to the programmer a quick preview of how her app will look on different platforms and permits her to build it using drag and drop of components. On the other hand, the second screen, shown in figure 22, allows a more advanced editing of the app, permitting the programmer to customize totally its behaviour, letting her modify the code of the structure, the logic and the appearance. The crucial difficulty we had to cope with was to let these two different editors communicate, being able to work with the very same file. This was done with some tricks, as it was a feature added in the latest part of the development process.

It is important to notice that the Visual Editor contains by default an overall view of all the screens inside the project, in a way similar to what happens in Xcode storyboards, while the Code Editor, by construction, can show the details of only one screen – or, better, of a file – at a time.

4.2.3.1 From Visual Editor to Code Editor

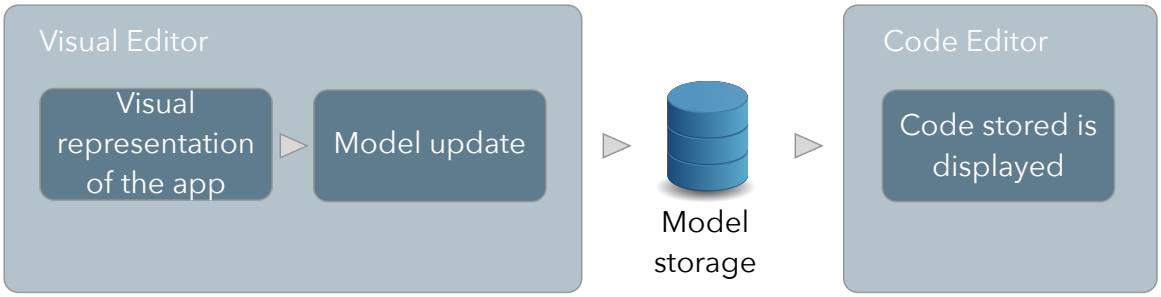


Figure 23 - From Visual Editor to Code Editor process

The developer can open the code editor simply clicking on a relative button inside the toolbar – we will look in depth when talking about UI in the relative paragraph. As we have already seen, in any time of the execution of the editor the model of the app is present and updated in order to reflect perfectly the app developed. When the user decides to switch from the Visual Editor to the Code one, all the modification done by the user are "on the fly" stored into the disk, then the code editor opens the file selected and display their content. It is important to notice that the Visual Editor stores at once all the files relative to screens opened inside it. It should be considered that these are usually three files (XML structure, CSS style, JS logic) per screen and this editor can display multiple screens.

There is also another use case: instead of being in a Storyboard-like view, the Visual Editor can display details of a single screen, selected by the user among the others. This is the case where a submodel is present, instead of the complete model of the app: in the model update phase the two are merged, in order to restore consistency. Clearly, such a merge is performed also if the user wants to run the application: Mobile Studio must always give to NativeScript compiler a consistent model.

4.2.3.2 From Code Editor to Visual Editor

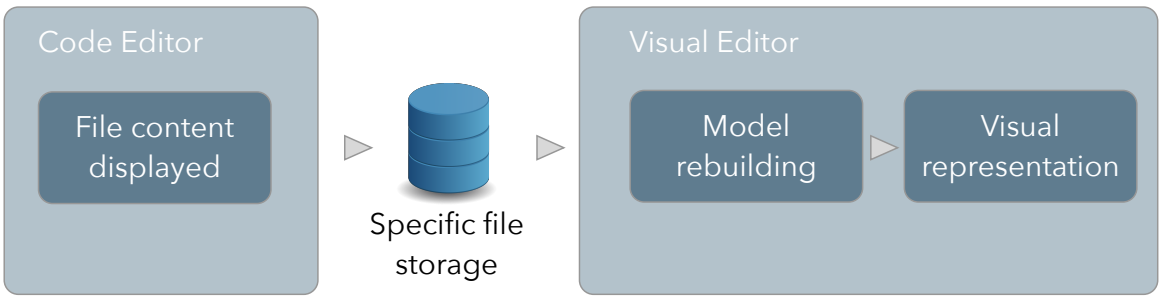


Figure 24 - From Code Editor to Visual Editor process

It has to be considered that only few files support a visual representation: in particular only XML ones, that defines the layout of a single screen, and MSA one, that are the specific files built by Mobile Studio that contain all the information of

the project, thus their visualization is a collection of all the screens and the interaction between them, in a similar way of what happens in Xcode editor.

When the user switch from the code editor to the visual one, the file inside the code editor is written “on the fly” to the disk. Then, depending on the type of the file, a submodel or a model is generated translating the stored files: the first solution when the user is opening a single screen – XML file – the second when it the case of the whole project – MSA file. Either models type, sharing the same syntax, are accepted by the Visual Editor and correctly rendered.

4.2.4 Mapping HTML elements to native mobile ones

As we have discussed the Visual Editor is able to provide a consistent preview of how the final app will look like in Android and in iOS. It is clear, though, that the components that appear inside the Editor are not the same as the one in the real device, being them platform-specific. In order to enrich as further as possible the preview offered by the editor, allowing it to be also interactive, we tried to map the real native components to HTML ones. It is important to point out that these components will be then converted in NativeScript ones, and it will be only this framework to provide to the final user real native components. The mapping between HTML and NativeScript components that we proposed can be found in table 3.

NativeScript (mobile) component	HTML element that visualize it
Label	
Button	<button>
TextField	<input type=”text”>
TextView	<input type=”textarea”>
SearchBar	<input type=”search”>
Switch	
Slider	
Progress	<progress>
ActivityIndicator	
Image	<img>
ListView	
HtmlView	<iframe>
WebView	<iframe>
TabView	
SegmentedBar	
DatePicker	
TimePicker	
ListPicker	

Table 3 - Mapping between NativeScript component and HTML element in Mobile Studio

While some of this correspondence may sound obvious to the reader – e.g. the pair Image and <img> – some of them are quite complex. The reader can see, in fact, that when an easy correspondence was not found we decided to fall-back to the classical that, as the web-expert reader already knows, it’s nothing more than a standard HTML container. Then, to make this container look like the real mobile component, we used specific CSS rules and images. This approach can give to the developer a sense of how her UI will be, but eliminates any interactivity of the component in the editor. Yet, this is not a big loss, considering also that many

other solution explored – Android Studio, Xcode – does not allow the component in the editor to be interactive, but only to be a simple preview of the UI. The usage of CSS rules allowed us to provide different component visualizations for the different operating system supported by Mobile Studio. We will investigate more technically on how this was done in paragraph 3.6.4.2.

4.2.5 NativeScript Compilation

When the developer thinks that her app is ready, she can run or debug her app in a real or simulated device. When this happens, Mobile Studio should convert the model in the file hierarchy accepted by NativeScript: here the Translator module will take the model and properly convert it to the needed files. After all, Mobile Studio executes the shell commands needed in order to send these files to NativeScript and let this framework build the app and run into a simulator or a real device, depending on the user choices.

4.3 Programming choices

After the high-level view of the software and the explanation of its main concepts we can start being more technical of how the solution has been programmed. Before going in deep into this side, we would like to immediately point out some choices that have been done during the very first phases of the development. These choices, mainly program-side ones, will be listed and detailed in this section.

4.3.1 Usage of objects instead of arrays

As we will see extensively when talking about the model, we will discover that, in run-time, it is nothing more than a JavaScript object that contains other JavaScript objects. The reader will immediately imagine that a collection of objects contained in another one – e.g. the components inside the screen – could be represented with an *array,* that is the simplest way to declare a collection of objects of the same kind in the clear majority of programming languages, JavaScript included. In the project, tough, we preferred to use objects collections rather than arrays, a more abstract data structure that has the advantage to provide a simple direct way to access to the element of the collection, using a key. For example, it is possible to write the following line:

```
application.screens[“win0”]
```

Or the equivalent with dot notation:

```
application.screens.win0
```

While with the arrays the programmer is forced to used indexes to access to elements. This advantage bares several disadvantages tough: for example, it is impossible to enforce an order between the elements of the collection and it is not straightforward how to count the elements: something that comes for free if using arrays. These features were necessary and, in order to be implemented, we added

a counter in each object that used a collection of this kind, that was able to track the number of elements inside it. When it was needed to add or to remove an element in a specific position inside the collection, we decided to simply destroy the entire object, then recreate it and add one element of the collection at a time, inserting/ignoring the one that should be added/deleted at the right position.

4.4 External Libraries

As something typical of a web application, but also of any modern software, Mobile Studio uses several external libraries to perform different tasks. We will list then and discuss how were useful and how to use them in the following paragraphs.

4.4.1 jQuery

The library jQuery represent nowadays a standard for every web project, allowing a faster and easier ways to use common JavaScript functions. For example, it is possible to select a DOM element with the simple statement:

```
$("#myElement");
```

While in vanilla JavaScript we should have written:

```
document.getElementById("myHTMLElement");
```

As the careful reader has surely noticed, the hash inside the jQuery function *$* is nothing more than a CSS selector specifying that the following string is an ID: we could use any CSS selector, here, in order to select elements of the DOM easily.

JQuery does not provide only a simpler way to select elements, but also a convenient access to standard JavaScript functions. A classic example could be the following:

```
$(document).load(function(){
        //Do something here
});
```

The *load* method binds the following standard JavaScript one:

```
document.onload =  function(){
        //Do something here
}
```

That execute the core of the anonymous function when the DOM has fully loaded. Several utility functions like this are provided by jQuery, that can be easily found in the library documentation [32]

It should be noted that, although jQuery accelerates the writing of the code, it is a heavy library that can slow down the overall execution [33] [34]. Therefore,

jQuery compatible libraries, that supports only a subset of the methods provided by the original library, are born – e.g. ZeptoJS [35]. Yet, for the purpose of this work we decided to maintain jQuery, but we don't exclude the migration to these other lighter libraries (see Future Developments paragraph).

Finally, jQuery, while being a widespread library in the web, has still some bugs and we have to cope with the zoom one (see Issues).

4.4.1.1 JQuery UI

JQuery UI is a set of user interface interactions, effects, widgets and themes built on top of jQuery library, and focused to enrich the user experience of a website. In the project, mainly the first components of JQuery UI was used: the interactions. An example of this could be represented by the *draggable()* jQuery UI function that, with a single line of code, is able to permit a HTML element to be dragged around the screen. The usage of this is straightforward:

```
$( "#HTMLelement" ).draggable();
```

Is interesting how these functions accept other parameter that permits to customize the interaction. For instance, we can specify that the element should be dragged only inside its parent and only in horizontal position with the following line of code:

```
$( "#HTMLelement" ).draggable({ containment: "parent", axis: "x" });
```

Other supported interactions are:

- Droppable
- Resizable
- Selectable
- Sortable

Widget, instead, are complex pre-built HTML elements that can be easily put inside a webpage and customized. An example is the *accordion,* a component often used in web pages to display collapsible content with a simple visualization that can be seen in Figure 25.

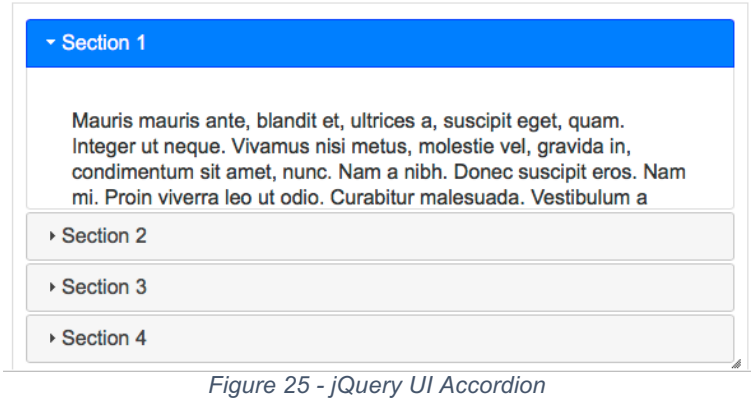


Figure 25 - jQuery UI Accordion

As typical in jQuery UI, creating such an element is straightforward and can be done with a single line of code:

```
$( "#accordionHTML" ).accordion();
```

Where the *accordionHTML* element has the following structure:

```
<div id="accordion">
  <h3>Section 1</h3>
  <div>
    <p>
    Mauris mauris ante, blandit et, ultrices a, suscipit eget, quam. In
teger ut neque. Vivamus nisi metus, molestie vel, gravida in, condiment
um sit amet, nunc. Nam a nibh. Donec suscipit eros. Nam mi. Proin viver
ra leo ut odio. Curabitur malesuada. Vestibulum a velit eu ante sceleri
sque vulputate.
    </p>
  </div>
  <h3>Section 2</h3>
  <div>
    <p>
       <!-- Second content -->
    </p>
  </div>
  <!-- Other sections -->
</div>
```

The last component of jQuery UI is dedicated to effects, a set of functions able to add animation to HTML elements in order to improve the user experience of the website. A quick example of these functions can be the *show* one, able to display elements using custom effects.

```
var options = { percent: 50 };
$( "#element" ).show( "scale" , options, 500, callback );
```

Where the effect `scale` is performed to the element with id `element` with the options contained in the relative variable declared in the first line, that simply will specify that the element should be scaled of 50 percent of its original width. The `callback` function will be executed when the effect will finish.

While effects and widgets are a consistent part of jQuery UI, we have not used them in the project, where instead the interaction part was fundamental in order to build the drag and drop logic inside the visual editor. We invite the reader to check the online documentation of jQuery UI [36] for more details on these parts not used in the project, while we will discuss extensively in the following sections how we used the first ones.

4.4.1 Bootstrap

Bootstrap is another famous framework intended to help the developer create rich user interfaces and experiences in her websites. It can be seen as the main competitor of the previously written jQuery UI and the reader may ask herself why we used both in our project. The fact is that, in the writer opinion, Bootstrap offers more updated and mobile-oriented components, that can naturally have a more modern look in respect to the old-fashioned jQuery UI ones. Yet, we were not convinced by the events that are also present in Bootstrap (*draggable, droppable,* etc.), preferring the easier to implement jQuery UI's equivalents.

So, at the end, we decided to use both these UI frameworks, jQuery UI for the more logical part, the events, and Bootstrap for the real UI.

We won't dive in Bootstrap usage, because perfectly symmetrical to jQuery UI, with the same structure of interactions, components and effects, inviting the reader to look for more information on this framework on the rich official site [37]. The reader will immediately spot the similarities looking on how we created tooltips in the editor using Bootstrap, that it is simply done with the line of code:

```
$(document).tooltip();
```

That enables tooltips in the whole document. An element that should present a tooltip must have a pair of additional attributes: one that states that a tooltip is needed and the other that contains the text of the tooltip. For example:

```
<button data-toggle="tooltip" title="Run the app"></button>
```

Bootstrap is, in the writer opinion, a far more complex framework than jQuery UI that can be used to create more advanced website with components and events tailored to the mobile world. While in this work we tried a "best of both worlds" approach, it is probable that only one of these frameworks should be maintained, in order to lighten the program and improve performance.

4.4.2 CodeMirror

As we will see, the IDE provides a simple code editor that allows the developer to further customize her app, being not only bound to the visual editor.

A code editor of such a kind, that support basic features like copy and paste, but also more advanced ones, like syntax highlighting, would be painful to develop, taking a lot of time. Fortunately, a library that let us build a code editor exist and it is called CodeMirror. This library has been used successfully in many products, the most famous of them is Adobe Brackets.

CodeMirror allows to append to the body an HTML document a full code editor with many advanced features:

- Support for over 100 languages out of the box
- A powerful, composable language mode system
- Autocompletion (XML)
- Code folding

- Configurable keybindings
- Vim, Emacs, and Sublime Text bindings
- Search and replace interface
- Bracket and tag matching
- Support for split views
- Linter integration
- Mixing font sizes and styles

Its usage is very simple and, after having included in the project, we can build a CodeMirror editor, simply appending it to the `body` of the document, using the simple line of code:

```
var myCodeMirror = CodeMirror(document.body);
```

It is possible to instantiate the editor in a more complex way, like the following.

```
var myCodeMirror = CodeMirror(document.body, {
  value: "function myScript(){return 100;}\n",
  mode:  "javascript"
});
```

Where the editor is initialized with a value, so it won't start empty, and we specify that the content will be a JavaScript code, in order to let the editor highlight it correctly. Adding a CodeMirror component in these ways is not common: usually, as it is also done in Mobile Studio, a `textarea` HTML element is positioned and then "coverted" to a CodeMirror element. This is understandable, because as the `textarea` is added statically it could be easier to position or to style rather than an element added in runtime. CodeMirror allows a fast and easy way to convert a `textarea` in a CodeMirror object:

```
var myCodeMirror = CodeMirror.fromTextArea(myTextArea);
```

Also the CodeMirror created in this way can be customized with many options, like the presence of line numbers, how highlight the code and many more. We invite the reader to look at the official documentation to have a complete look on all the options that CodeMirror can accept [38].
While we will look in depth on how CodeMirror has been used in Mobile Studio, we can have a look on its basic form in Figure 26.

```
1  function getCompletions(token, context) {
2    var found = [], start = token.string;
3    function maybeAdd(str) {
4      if (str.indexOf(start) == 0) found.push(str);
5    }
6    function gatherCompletions(obj) {
7      if (typeof obj == "string") forEach(stringProps, maybeAdd);
8      else if (obj instanceof Array) forEach(arrayProps, maybeAdd);
9      else if (obj instanceof Function) forEach(funcProps, maybeAdd);
10     for (var name in obj) maybeAdd(name);
11   }
12
```

Figure 26 - CodeMirror example

4.4.3 FancyTree

A common way to represent file hierarchy in the File System is to use a tree. Most software, IDE and non, that have to interact with the File System offer such a visualization, that is the simplest in order to understand which files and directories are in the project.

To offer a file tree visualization also in Mobile Studio we have used the JS library called FancyTree, both to visualize the project files and to show the hierarchy of components in the screens.

As all the libraries that we have seen, its usage it's not difficult and we can have a FancyTree component writing just one line of JavaScript, after having included the library in our project:

```
$("#tree").fancytree({ ... });
```

Where the FancyTree object will be created inside the HTML element `tree`. However, it is mandatory to pass an object inside the FancyTree function that will configure the tree and, in particular, specify its elements. Elements can be loaded in three ways, inside the source attribute of the configuration object:

1. Passing a nested data structure, e.g. an array of nested objects
2. Load data in JSON format via AJAX call
3. Converting a pre-existing HTML hierarchy in the root element (composition of `<ul>/<li>` tags) into a FancyTree tree.

In the project, the first approach has been used extensively. After having initialized the tree, the nodes will show up with a structure similar to the one that we can see in Figure 27.

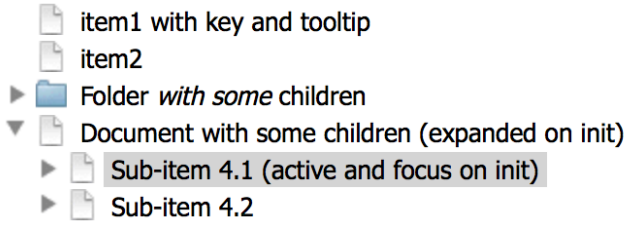


Figure 27 - FancyTree example

Another feature of FancyTree used in Mobile Studio is the possibility to bind an action to the click of an element of the tree. This can be done specifying a function that should be called in this situation inside the activate attribute of FancyTree's configuration object. We can see an example in the code below.

```
$("#tree").fancytree({
    activate: function(event, data){
      // A node was activated: display its title:
      var node = data.node;
      console.log(node.title)
    }
  });
```

Where we can see how an anonymous function has been specified in the `active` attribute and its event and data parameters will be automatically populated with the event that caused the activation of the node and the data linked to that node. We can see, in fact, how it is possible to access to the node title navigating inside the data parameter.

It is important to state that, in the writer opinion, FancyTree can be difficulty defined “flexible”: the object that should populate the tree must have a predefined strict syntax. As we will see, the model inside Mobile Studio should be converted in a proper way in order to be displayed as a tree, and we defined a custom function – `adaptModelForTree()` – in order to do this translation. To understand this specific syntax, we can look at the output of this function for a simple project: an object that we can see in Figure 28.

Figure 28 - Example of FancyTree object

As we can see, it is mandatory for the object to define a `children` array – with this specific name – that can contain other objects with, eventually, other array of such a kind. Each element of the array is a node and should contain a `title`, the string that will be displayed near the icon in the tree, and the boolean value `folder` that indicates if the node is expandable or not. Other custom attributes can be bound to the node – like the `parentWindow` in the example – and be easily accessible when selecting one of them – as we have seen, they will appear in `data.node` object.

4.5 Project

4.5.1 File Structure

Mobile Studio present the typical file structure of a software built with Electron, with some files specific to this framework, like the *package.json*, but the vast majority are typical of a web application and will not surprise the web-expert reader. When possible, we decided to organize software files in a classic web app structure – e.g. all the Cascade Style Sheet files inserted in the directory *css*.

We will list the files and directory present in the root of the project, explaining them, in table 4.

File Name	File Type	Description
package.json	JSON property file	Required by any Electron app, it describes the dependencies of the project and other general information, like the name and the icon.
main.js	JS source file	Contains the source code of the Main process, the only one that does not require an HTML to be visualized.
classes.js	JS source file	Contains the classes that build up the model.
index.html	HTML document	Contains the structure and the logic, written as in-page script, of the Welcome Screen.
editor.html	HTML document	Contains only the structure of the Editor Screen
mainIcon.ico	Icon	Icon of the project
js	Directory	Contains all the logic of the software, written in JS source files.
js/editor-backend.js	JS source file	Contains the logic that allows the software to communicate with file-system and execute shell command.
js/editor-frontend.js	JS source file	Contains all the logic that manages all the changes in the UI of the Editor
js/[other files]	JS source files	Inclusion of all the external plugins needed to accomplish certain tasks
css	Directory	Contains all the documents that specify the style of each element of the UI of the software, written in Cascade Style Sheet format
css/ios.css **css/android.css**	Cascade Style Sheet files	Specify how the default look of the components to be similar to the native ones of iOS/Android
img	Directory	Contains all the icons used in the UI
node_modules	Directory	Contains all the node.js modules that allow the application to work.
cm	Directory	Contains all the file of the CodeMirror plugin

Table 4 - Project file structure

4.6 Architecture

Mobile Studio has been structured in several Chromium processes, principally divided per different user interactions, being able to load an HTML document and to present an UI to the user. As we have seen when talking about Chromium and Electron, a main *browser* process is always present and, working in the background without offering any UI to the user, initializes the program and let the *renderers* process communicate via IPC. A simple message protocol has been created in order to let this communication be efficient and expandable in the future.

Yet, many other processes are needed by the Editor window: they are not Electron renderers, but small Node.js threads intended to perform very specific tasks – file creation, command line command execution, etc. The need of such processes lead to the division of the Editor windows in two parts: one of front-end, able to visualize the UI of the Editor itself, the other of backend, that enclose these processes and it's thus able to communicate with the underlying OS.

The most important piece of the puzzle is not a process, though: the **model** is an abstract representation of the app that the developer is building and it is updated in real time, as soon as she does any modification to the app. The model can be seen thus like a real data layer that lies behind the UI of the whole program, like often happens in complex web services, like Google Analytics. Treated as a global JavaScript object variable, it can be translated in any time in specific NativeScript code and, using NativeScript compiler, it can be converted in native project files and also executed in real devices or simulator.

Mobile Studio, while being a first class desktop application, has still the nature of a web software and thus it uses several external libraries in order to accomplish numerous task, especially in the UI: from the drag and drop in the editor to the file tree representation, for instance. We will investigate in all these components in relative paragraphs.

A bird's eye view of all of them can be found in Figure 29.

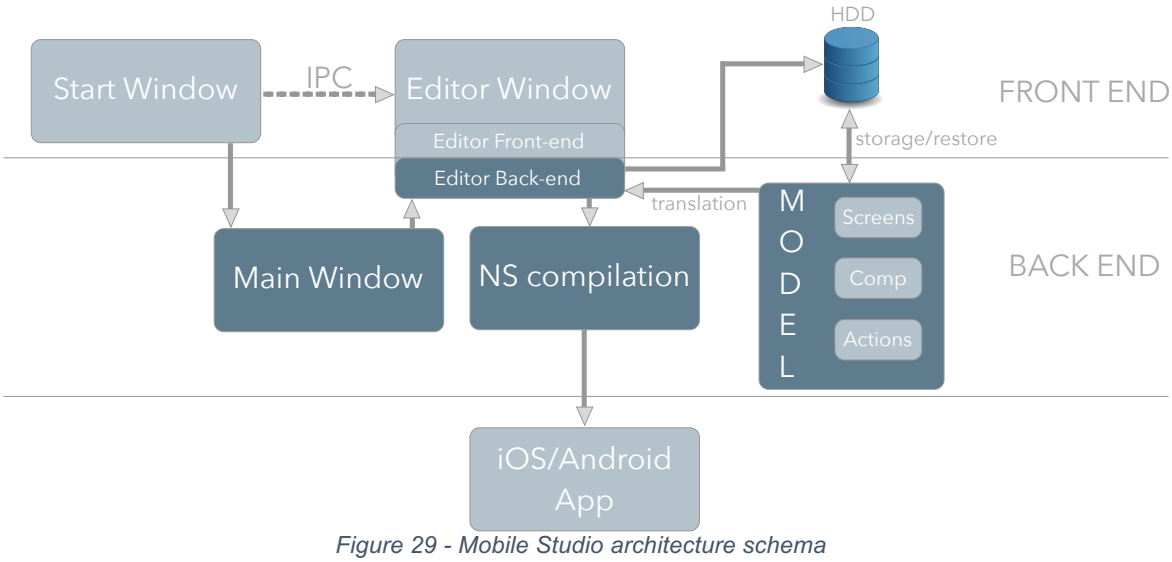


Figure 29 - Mobile Studio architecture schema

4.6.1 The Model

When we first discussed the creation of an innovative IDE for developing mobile applications, it immediately appeared clear that a complex data layer was needed in order to abstract from the application inside the editor. This layer encloses all the details of the project, needed to translate it in NativeScript syntax and thus in the final running application.

Inspired to famous data layers used in enterprise area – e.g. Google Analytics – this substrate has the natural form of a global JavaScript variable, bound to the Editor window. However, is important to say that the model can be found in other forms in the execution of Mobile Studio, in particular it can be translated into a

JSON string in order to be passed among the different parts of the software, or it can be stored into the disk of the user in a specific format, the MSA, in order to be easily used to restore the project. An example of how the model can encapsulate all the information of an app can be found in Figure 30 and 31.


```
▼Object {screens: Object, numberOfScreens: 2, startingScreen: "win0"}
  numberOfScreens: 2
  ▼screens: Object
    ▼win0: Object
      ▼components: Object
        ▼Button4: Object
          ▶actions: Object
            id: "Button4"
            isDynamic: false
          ▼specificAttributes: Object
              text: "Calcola"
            ▶__proto__: Object
            style: "left: 86px; top: 131px;"
          ▼supportedActions: Array[3]
              0: "tap"
              1: "doubleTap"
              2: "longPress"
              length: 3
            ▶__proto__: Array[0]
            type: "Button"
          ▶__proto__: Object
        ▶Label2: Object
        ▶Label5: Object
        ▶Label6: Object
        ▶TextField1: Object
        ▶TextField3: Object
        ▶__proto__: Object
        id: "win0"
        isStartingWindow: true
        layout: "absoluteLayout"
        numberOfComponents: 6
        style: "top: 76.7969px; left: 49px;"
        type: "Window"
      ▶__proto__: Object
    ▶win1: Object
    ▶__proto__: Object
  startingScreen: "win0"
```


Figure 30 - A sample model of an app

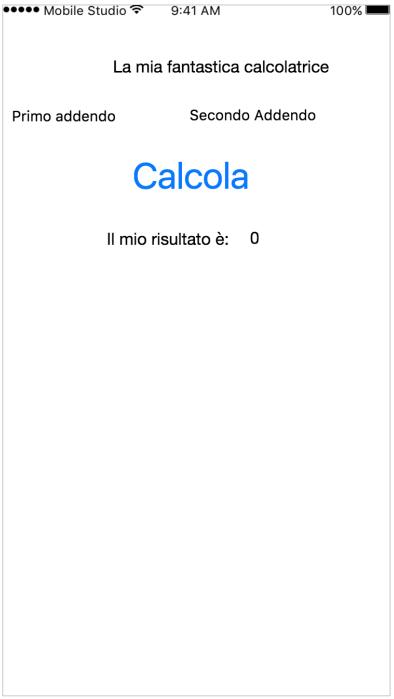


Figure 31 - The app that produced the model in Figure 23

As we can see from the figures, the model, in runtime is a nested object. Following a Software Engineering style approach, we decided to build it with *classes*, a well-known concept still not used frequently in JavaScript.

The structure of the model is shown in the Figure 23.

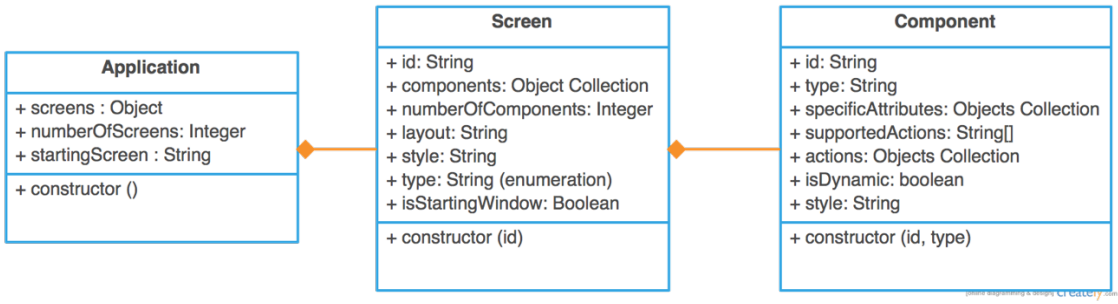


Figure 32 - UML Diagram of the Model

As we can see the three main parts classes that builds up the model are the following:

- Application
- Screen
- Component

We'll dive in each of these in the following paragraphs.

4.6.1.1 Application class

The Application class models the whole app that should be developed in a simple way: we consider an app a set of screens where the user will navigate. Only one other attribute is needed in this class: the one that specify which, of all the screens, should be the first to be launched, the entry point of the future app. The reader can easily see all the attributes of this class in the following table.

Attribute	Description	Possible Values
Screen	Object set of the components that builds up the screen	{Screen1, Screen2,…} where ScreenX is a Screen object
numberOfScreens	Counter of the size of the previous set	Natural number greater than zero
StartingScreen	String with the id of the first screen to show to the user	"win0", "win1", …

Table 5 - Attributes of Application Class

We have not listed the only method of this class, that is the simple constructor that, without accepting any argument, instantiate the class. As we will see when talking about the editor, the Application class is instantiated as soon as a project in opened in the Editor.

4.6.1.2 Screen class

The Screen class models a screen of the final application. As its parent class, Application, a Screen can also be seen as a set of other elements, the components inside the view presented to the user. This time, tough, more custom attributes are needed in order to specify some properties of the screen itself, like its aspect, defined with CSS syntax inside the style attribute. The list of all the attributes of this class can be read with the following table.

Attribute	Description	Possible Values
id	Id of the screen: must be the same of the HTML element that present the screen	"win"+<number> by default, possibly any string
components	Object set of the components that builds up the screen	{Com1, Comp2, …} where CompX is a Component object
numberOfComponents	Counter of the size of the previous set	Natural number greater than zero
layout	Each screen has a NativeScript layout	"stackLayout", "wrapLayout", "absoluteLayout", …
style	A string that contains all the custom CSS that the user	CSS properties stringified

	enters to change the visualization of the screen	
type	To be coherent with the Component class, a type attribute has been introduced	Constant string “Window”
isStartingWindow	Boolean flag that indicated if the windows is the first one to be presented to the user	True/False

Table 6 - Attributes of Screen Class

Also in Screen class we have only one method, that is the constructor, but this time is slightly more complex than the one seen in the previous class, because it accepts a parameter: the id of the Screen.

4.6.1.3 Component Class

The last class that builds up the model is the Component one, that represent a singular element inside the screen – e.g. a label – of the app. It is crucial to mention that these components can be very different among each other’s: this class needs to consider these differences. The attributes of this class are in the following table.

Attribute	Description	Possible values
Id	Id of the component: must be the same of the HTML element that present the screen	String composed by a concatenation of the component type and a natural number
type	Type of the component	“Label”,”Button”, etc.
specificAttributes	Objects containing custom attributes for the component. In a Label, for instance, we can find the object “text” inside this.	{customEl1, customEl2} where customEl1 is for istance {“text”: “Hello Word”}
supportedActions	Array of strings that represent the supported actions that an user can have with the component.	E.g. for buttons: ["tap","doubleTap","longPress"]
actions	Objects that specify the user-defined actions that are actually bind to the component	E.g. for buttons: ["tap","doubleTap","longPress"]
isDynamic	A flag that indicates if a certain component should be changed dynamically	True/False
style	A string that contains all the custom CSS of the component	“#win0{background-color:red;}”

Table 7 - Attributes of Component Class

Components are differentiated between each other thanks to the constructor, that initialize properly the attributes. A switch statement, depending on the attribute type – passed as parameter to the constructor – will initialize properly `specificAttributes` and `supportedActions` variables.

4.6.1.3.1 specificAttributes

This variable is intended to better specify the attributes of the component, that can be different between them, depending on the type of the component itself. For instance, a popular attribute of such a kind is `text`, that contains the text inside the component like a label, a TextView, etc. The attribute `value` is instead present when the component is bound to a number: this can happen in sliders, for instance. It is important to notice that `specifiAttributes` is an array, because if in the current implementation all components have only one attribute, in future implementations it will be possible to add other attributes. In the following table, we list the `specifiAttributes` already implemented.

Component	specificAttribute	Description
Label	text	String Text that contains the text that should be seen inside the component
Button	text	
TextField	text	
TextView	text	
SearchBar	text	
Switch	set	Boolean value that indicates if the switch is set or not
Slider	value	A floating point value that is bound to the component
Progress	value	
ActivityIndicator	busy	Boolean flag that indicate if the activity indicator should be acrive or not
Image	src	A string containing the url of the image
ListView	listElements[]	An array that contains the elements of the list
HtmlView	html	A string that contains the HTML to be visualized
WebView	src	A string that contains the url of the website that should be visualized
TabView	tabs[]	An array containing the tabs in the tab view
SegmentedBar	items[]	An array containing the items inside the segmented bar
DatePicker	date	A string containing the date of the DataPicker
TimePicker	time	A string containing the time of the TimePicker
ListPicker	items	An array containing the elements of the ListPicker

Table 8 - specificAttribute for each component

4.6.1.3.2 SupportedActions

This attribute indicates what are the actions that can be bound to a particular component and it is simply an array of strings.

Up to now, only the `button` component has this attribute, initialized with tree supported actions, as follows:

```
supportedActions = ["tap","doubleTap","longPress"];
```

4.6.1.3.3 Actions

It is important to denote the difference between the `supportedActions` and `Actions` attribute. While, as we have seen, the first is intended to list all the actions that a component can support, the latter indicates the actual action that the programmer has linked to the particular component. This is the reason why it is initialized with an empty array. We will see how the Visual Editor will populate this field and how the Translation logic will convert it to executable code.

4.6.1.4 The model seen as MSA

The model needs to be stored to the disk, in order to let the user be able to restore the project after closing the program. To do this, the model is converted in JSON format and then written in a MSA custom file, where it is concatenated with other parameters as a string. As we will talk when introducing the translator library of the project, the MSA file is built with the following structure.

```
toStore = projectName + "|" + workingPath + "|" + JSON.stringify(applic
ation) + "|" + componentsCounter;
```

Where, as it is easily understandable, the other parameters in the MSA file are the project name, its path and the current count of the components.

4.6.2 Main process

As we have seen chromium processes can be of two different kinds: one main *browser* and many *renderers*. In Mobile Studio the Main process is responsible to instantiate the two *renderers* windows, specifying how they should appear in term of size and other attributes, e.g. if the window should be resizable or not.

We can dive into an example of how this instantiation is done, looking on how the Welcome Screen is built. In the `main.js` we find the following code:

```
function createStartWindow() {
  if (os.platform()=="darwin"){
    startWindow = new BrowserWindow({
        width: 800,
        height: 600,
        resizable: false,
        icon: `file://${__dirname}/img/startupIcon.png`,
```

```
        titleBarStyle: 'hidden'
    });
  }else{
     startWindow = new BrowserWindow({
        width: 800,
        height: 600,
        resizable: false,
        icon: `file://${__dirname}/img/startupIcon.png`,
        frame: false
     });
  }
  startWindow.loadURL(`file://${__dirname}/index.html`);
```

It is important to notice how we can specify different behaviours of the windows in different platforms. This is done using the node.js variable `os`, that stores all the details of the operating system where the app is running. In the specific code written above, we can notice how the `startWindow` has a `titleBarStyle: 'hidden'` attribute in Darwin platforms (macOS), while in all others (Windows) it has a `frame: false` one. This is only for aesthetic purpose, because in Windows the first attribute, that let the close, minimizing and expand button of the window collapse inside the content, is not supported.

The Main process is also responsible to build the menu that appears on below the bezel of the windows – on Windows platform – or on top of the screen – in macOS. The menu is built simply using the following lines:

```
const menu = Menu.buildFromTemplate(template)
Menu.setApplicationMenu(menu)
```

But the `template` object is a complex array that contains all the info that should be displayed. An example of such an array is the following:

```
let template = [
    {
      label: 'File',
      submenu: [{
        label: 'New Project',
        accelerator: 'CmdOrCtrl+N',
        click: function(){
            createStartWindow();
        }
      },{
        label: 'Open...',
        accelerator: 'CmdOrCtrl+O',
        click: function(){
            toImplement();
        }
      }, {
        label: 'Save All',
        accelerator: 'CmdOrCtrl+S',
      }]
    },{
        label: 'Edit',
        ……
```

Where we cut off all the other elements in the menu because with similar syntax. We can see how the array has to specify the label of the menu, its elements, if it contains submenu, accelerator – combination of key used to enable a certain function – and the JavaScript function to call when an element is clicked.

Yet, Main process has still one other important role. In fact, being the background process always present when the application is running, provides the only way for the browsers processes to communicate via **IPC**. In the specific case of Mobile Studio, such a communication is needed only when the user has specified all the details of its app in the Start screen and wants to open the Editor screen, that should read that details in order to build the app. To realize this communication, done in asynchronous way, the Main process listen for specific messages sent from the Welcome one and, if the message *"startEditor"* is received, it builds up the Editor Window with the parameters acquired. The listener function that accomplish this is the following:

```
ipcMain.on('asynchronous-message', (event, arg) => {
  console.log(arg);
  if(arg[0]=="startEditor"){        //arg[0] is the name of the message
    createEditorWindow(arg[1], arg[2], arg[3]);   //arg[…] other param

  }
});
```

And instead the possibility to send the message to the editor window is specified in the `createEditorWindow()` function, always in main.js:

```
function createEditorWindow(path, prjName, existingPrj){
 […]

  editorWindow.webContents.send('asynchronous-message', ["initializatio
n", path, prjName, existingPrj]);

 […]

}
```

To provide such a communication a simple protocol has been built, structuring all the messages that are sent in an array containing the following variables:

Variable	Type	Cardinality	Mandatory
Title of the message	String	1	YES
Message Attributes	Any kind (strings, Boolean, etc.)	1 to N	NO

Table 9 - Variables in the communication protocol

An example of this array could be the following array:

```
[“initialization”, “/Users/luigirusso/Desktop”, “HelloWorldApp”]
```

Up to now only two messages are used, the *initialization* and the *startEditor* one, sent according to the schema in Figure 26.

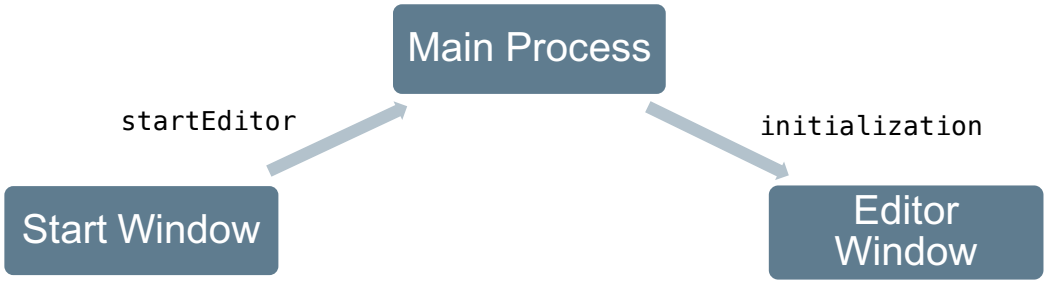


Figure 33 - Main Window Communication Schema

4.6.3 Welcome Screen

The Welcome Screen is the first browser process presented to the user. As a process of this kind, in fact, it is able to provide a UI that is specified in an html document – *index.html* – and styled by a CSS file – *startup.css*. Yet, all the logic of the screen is managed by JavaScript that, considering only a few lines of such a code were needed, it has been included in the html – enclosed in the usual <*script*> tag – without building a separate file.

This process displays the starting dialog of the IDE, allowing the developer to specify the folder where he/she wants to save the project and asking him which platform he/she wants to address. It is able to retrieve the list of old projects, reading a history file positioned in the OS user data directory (<spcify directory in macOS/Win). A simple example of how Electron let the web app be able to communicate with the OS is the usage of system dialogs when the developer has to choose the directory of the project.

This is done with the incredibly simple one line code that follows.

```
workingPath = dialog.showOpenDialog({properties: ['openDirectory']});
```

Although the logic of this screen is quite simple we still can find some interesting features, like the usage of Node.js functions in order to acquire the history of the last projects where the user has worked on, stored in the filesystem. We will report this function in the following snippet of code:

```
function retriveProjects(){
  const {app} = require('electron').remote;
  var homePath = app.getPath("userData");
  fs = require('fs');
  console.log(homePath+"/history");
  fs.readFile(homePath+"/history", 'utf8', function (err,data) {
  if (err) {
    return console.log(err);
  }
  strData = "["+data+"]";
  strData = strData.replace(/}{/g,'},{');
  var dataJSON = JSON.parse(strData);
  for(var i = 0; i<dataJSON.length; i++){
    $("#listContainer").append(`
                    <div class="content-prj">
                        <h2>`+dataJSON[i].name+`</h2>
                        <h3>`+dataJSON[i].path+`</h3>
                    </div>
```

```
    `);
  }
  $(".content-prj").click(function(){
    const {ipcRenderer} = require('electron');
    ipcRenderer.send('asynchronous-message',
                 ["startEditor",$(this).find("h2").text(),
                    $(this).find("h3").text(),true]);
  });
}
```

Where we can see how the Node.js filesystem module is called, reads the *history* file stored inside the disk and, for each entry of that file, display an element of the visual list – that is a standard HTML *div.* Note also that a listener function is attached to each of these list elements, able to send the asynchronous message to Main process in order to start the editor with the parameter in the list.

4.6.4 Editor Screen

The browser process that corresponds to the Editor screen is the real hearth of Mobile Studio. It inherits the concept of a single page web application and in fact, all the main components of it are in a single HTML file, *editor.html*.

As usual in web applications, all the styling of these HTML files is done by CSS files, that, for improved readability, have been put in a specific folder. We will not spend a lot of time talking about these files, while instead we will analyse in deep the JavaScript ones, that contains all the logic of the application.

Few JS files are needed to achieve this goal. Always keeping readability in mind, we tried to split the code useful for the front-end (mainly UI) from the one of the back-end (e.g. file system communication): this is a common pattern in web developing, usually because often the language is different – it is common to use JS for front-end and PHP for backend. This is not the case of Mobile Studio that, while this separation is still present, both parts are written in JavaScript.

We will start explaining the front-end part, talking about the processes that, as we have seen, characterize an Electron application. It is important to notice that, as we have seen in previous paragraphs, the single page web application is split in two different parts: the one relative to the Visual Editor, that allows the developer to visually build its app, and a more classical Code Editor, that let the developer do the same operations writing code. These two parts are not independent: as we have seen these module have to share the same files and thus they must maintain them consistent. This allow the developer to use them both, switching between them frequently: this 2-way programming, described in Core Ideas paragraph, is something not new in IDEs and in the following section we will discover how it has been realized in Mobile Studio. Yet, a strong differentiation between the files of the two parts of the editor is not present, where we can find only files relative to the general Front-End and the general Back-End.

4.6.4.1 Global Variables

Although it is a famous bad practice to use too many global variables in JavaScript, this system provides an incredible easy way to share data between all the parts of

one application and, in the specific case of the Editor, share data in all the modules that build it up. The global variable used are listed in the table below.

Variable Name	Possible values	Description
componentsCounter	Integer greater or equal than 0	The number of the components present in all the screens. It allows to use an unique component id for each component
dirty	true/false	A flag that indicates if the user has done some changes in the project and still has not saved it
supportedCSS	[“color”,”background-color”,…]	An array that indicates the CSS tags supported by NativeScript
application	{screen:”…}	The object that represent the model of the project
platformToPreview	“ios”/”android”	A flag that indicates which platform should be previewed in the visual editor
visualEditor	true/false	A flag that indicates if the visual editor is enabled or not
fs	{…}	File System module of Node.js, used to access to File System
path	{…}	Path module of Node.js, used to manage file system paths.
os	{…}	OS module of Node.js, provide information about the current OS.
workingPath	“/User/Desktop”	A string containing the absolute path where the project is located
projectName	“HelloWorldApp”	A string containing the name of the project
codeEditor	null | {…}	The object containing all the information about the code editor or *null* if it is not visualized

Table 10 - Global Variables of the Editor

4.6.4.2 Editor – Front End

The functions of the Front End are not supposed to communicate directly with the file system, nor the OS: they are primary designed to provide to the developer an interface that he can use to interact with the software. The Front-End part of the editor is characterized by several of these functions:

- `(global variables)`
- `window.on(...)`
- `deleteElement`
- `buildProjectForAndroid`
- `setNewCss`

- `setNewAttr`
- `buildPropertiesPanel`
- `removeAllLayoutClasses`
- `normalizePosition`
- `selectElement`
- `initializeWindowEvents`
- `initialization`
- `deleteAction`
- `drawLine`
- `createCodeEditor`
- `codeToggle`
- `toggleComponentPanelVisualization`

All these functions, enclosed in the file *editor-frontend.js*, will be now explained in detail.

- `$(window).ready(…)`

This jQuery listener fires when the DOM is ready, so when the all the html, script and styles have been loaded. We can consider it as an entry point of the JavaScript execution in the Editor. Its main purpose is to define the initial properties of the layout of the page and listeners for events. In particular, the most important is the one that will respond to the asynchronous message sent by Main process, that will contain the parameters that the user has entered from the Starting screen.

We can analyse more deeply this listener looking at its code:

```
ipcRenderer.on('asynchronous-message', (event, arg) => {
       switch(arg[0]){
           case "initialization": {initialization(arg); break;}
       }
});
```

This listener will take in consideration every message sent by the main process, that will be stored in the variable `arg`. As we have seen when talking about the communication protocol between the processes, this variable will contain the name of the signal, the name of the project and its path. In the current implementation, only the *initialization* message is needed and it is the only one present in the switch statement. In future, it will be easily possible to expand this section to manage different messages.

The initialization message will call the initialization function, the most important function of the front-end.

- `initialization(config)`

The initialization function is the first one called when the Starting Window “calls” the Editor Window. The parameter of the function, immediately divided and put in global variables, is composed by an array containing the name of the project, its path and a flag that indicates if it is a new project or one already stored in the hard disk of the user. The first thing that the function does is to check this flag, stored in

`existingPrj` variable: depending on its value, the proper first back-end call is accomplished.

A second back-end call, this time to the function `createFileExplorer`, is accomplished in order to create the lateral panel where all the files will be displayed.

Then, the lateral list of components is built, reading their names from the array `components` and building, for each one, the relative html element. Once created, these elements are made draggable, using jQuery UI, and the main area of the visual editor is made sensitive to the drop of them inside it. This is done always using jQuery UI, specifying that the main area should be droppable and indicating what should happen when a drop occurs. jQuery UI provides a function that manages this case, that can be customized and, in our implementation, controls which component has been dropped, checking the title text of the droppable element, that is unique.
It is important to notice that it is possible to drop, in the main area, only the component *Screen*, while all the other components should be droppend inside one object of such a kind. When a Screen is dropped, it is positioned correctly and the model is updated to represent the new component added. This means that a new object is instanced from Screen class and inserted into the global `application` variable. The last things done relative to the dropped screen are the initialization of all its events and the update of the components explorer on the right, that will visualize all the components inside the screen. These are done by the *initializeWindowEvents* and *updateModelExplorer* function that we'll see right now.

- `initializeWindowEvents()`

This function acts like a continuation of the one before, exploiting specific behaviour of a Screen dropped in the mainContent. The first action done is, always using jQuery UI, to let the Screen be draggable around the mainContent, then the Screen is made selectable when the user clicks on it, finally, as it has been done for the mainContent, also the Screen is made droppable. This time the drop function is more complex: not only it needs to take into account all the different components that can be dropped inside a Screen – everyone except another Screen – but also two radically different behaviour: the first, when a component is dragged from its panel to the screen, in order to add to it, the second is when a component is dragged inside the screen in order to be repositioned.

To decide in which case we are, we analyse the classes of the object that is being drag: if it is has the *component* one, we are in the first behaviour. In this case the id of the new component is built with a concatenation of its type - e.g. "Label", "Switch", etc. – and a number, that will be incremented. Note that this number is stored in the `componentsCounter` global variable and also stored into the disk, inside the MSA file, in order to ensure that a unique id will be generated. After having decided the id for the new component, a proper HTML element is built to be visualized inside the Screen in the Editor. Note that a perfect one to one correspondence between mobile components and html elements does not exist, so the strategy used to map the firsts to the latter was to build a "similar" html element when possible – often styling it with CSS in order to let it be as more similar as possible with the mobile one – and, when this wasn't possible, use a div with a simple image that shows the mobile component. We have seen this correspondence in paragraph 3.2.4.

Once a component has been visualized, it is crucial to add it to the model. This is done right after the insertion of the HTML of the component. Finally, it is made draggable and the components explorer on the upper part of the right sidebar is updated.

When we have to consider the **second** behaviour, so the user is dragging a component that is already inside the window, it is crucial to understand which layout is set in the Screen. Up to now, three layouts have been exploited: the StackLayout, the RelativeLayout and the AbsoluteLayout. While we don't need to pay a lot of attention when considering the latter one – the position of the component doesn't need any further code adjustment rather than simple CSS styling – for the first one is crucial to understand which component is behind the one that is being dragged, in order to be correctly positioned before it.

This is done thanks to the standard JavaScript method `getElementFromPoint`, that returns the DOM element that is in a certain position specified in x,y coordinates. While the insertion of the new visual component is easy, thanks to the `insertAfter` method of jQuery, the update to the model is not straightforward. Being structured as an object, rather than an array, is not possible to order directly the components. To do so, the trick that has been done – and that the writer discourages to use in context not based on Google Chrome – is to destroy the model and recreate it, adding one component at a time and inserting the new one at the correct position, now possible because the components are inserted sequentially.

- `buildPropertiesPanel(winId, compId)`

This function is the one intended to build the panel that we can see on right part of the editor (see UI), where we can found several details of the object currently selected and where we can edit these details. As we will discover in the UI section of this document, the panel is nothing more a complex html element that is visualized and populated at run-time using this function, that, as we can see from the parameters, takes as input the specific element currently selected in the editor. The very first lines of the function are intended to this scope: they should understand if the object in input is a Screen or a Component.

Then, it starts building the html of the element, that, as we will see, is composed by three main part: attributes, styles, actions of the object. We thus scan all the CSS supported by NativeScript iterating into the `supportedCss` global variable, and, for each of these, we build an input box.

A simpler approach is dedicated to the attributes of the element that has been selected: depending on its kind – screen or component – specific attributes are displayed. These specific attributes and their scope are displayed in the following table:

Attribute	Description	Presence	
		Window	Component
Id	The id of the object	YES	ALL
Class	The class of the object	YES	ALL
Layout	The specific NativeScript layout of the window	YES	NONE

isStartingWindow	Flag that indicates if the app should start from the current Window	YES	NONE
Set	Flag that indicates if the checkbox is checked or no	NO	Checkbox
Src	String that indicates the path where the source of the image to visualize is present	NO	Img

Table 11 - Specific properties in the right panel

We would like to focus the reader attention to the fact that only few components need specific attributes entries different from the classical ones of a string, that is the default case when a specific attribute is not found.

Finally, the function build specific html for the actions. In this case, the function must iterate on all the `supportedActions` of the specified element and check if a real action is already bound to it. An easy visualization of these situation is provided to the developer, that is able to remove action or add others – we will discuss this more in the UI section of the document.

- `toggleComponentPanelVisualization()`

This function permits to easily close or open the properties panel that we have discussed before, using the simple jQuery functions `show` and `hide`. It also adjust accordingly the button that permits to open and to close the panel.

- `selectElement(element, window)`

This function is activated whenever a user click on a component or a screen in the visual editor. Mainly, it is a wrapper for the `selectInModelExplorer` backend function, but it also called the `buildPropertiesPanel` front-end one, properly initializing it considering if the selected element is a window or a standard component.

- `deleteElement(idToDelete, winId)`

This function is totally devoted to the removal of an element, that can be a window or any other component of the app, from the visual editor. It important to notice that this is not only a visual change because also the model should be updated accordingly. To do this, also the counter of screens or components, depending on the element that is going to be updated, should be decreased accordingly.

- `createCodeEditor()`

This function builds the skeleton of the main other visualization inside the Editor: the code one. For this, everything in the main visualization of the Editor window – the `mainContent` container – is deleted and replaced with the html element `textarea`. Then, the CodeMirror default function `fromTextArea` is used to build up a CodeMirror editor from that textarea. The snippet of code that do this and configure the editor is the following:

```
codeEditor = CodeMirror.fromTextArea($("#code")[0], {
  lineNumbers: true,
  mode: "htmlmixed",
  viewportMargin: Infinity,
  lineWrapping: true,
  theme: "lesser-dark",
});
```

Where we can see that the editor is set in order to display line number and highlight htmlmixed – html, javascript and css – language. Other visual properties, like the theme, are included.

- `codeToggle()`

This function represents the click listener that handles the click on the Visual/Code selector inside the Editor. Where supported this behaviour should change the visualization from a Visual one, where the user can utilize drag & drop to position components, to a more customizable Code View, where the software offers to the user a full code editor in order to write her JavaScript functions. The behaviour of this function is thus split in two parts, depending on which visualization is currently presented to the user.

If the Visual Editor is currently active, the function createCodeEditor will be called, eliminating the current visualization and building the skelethon of the code editor. It is important to notice that, after this skelethon is built, it will be populated calling the back-end function `openFileInCodeEditor`, that will read the content of the file and insert it into the editor just created.

If the Code Editor is currently active, we should analyse which element visualize, depending on the file selected on the lateral component. In order to build a two-way editing from the two parts of the editors, the content of the code one is converted into a submodel, that then will be rendered as a window and displayed in a new istance of the Visual Editor.

- `drawLine()`

The drawLine function permits the user to create visually a simple action: the developer can click on the action button in the lateral panel, then a line will be displayed and she can click on the target element, inside the editor, that will be the “ending point” of the action. For instance, if the developer wants to allow a navigation from one screen to the other at the press of the button, she can simply click on the *navTo* action in the lateral panel, when the button is selected, and then click to the second screen.

This approach is very similar to the one used in Apple’s Xcode and, in the writer option, allows the developer to perform tasks in an easier a faster way, instead of writing code. Up to now, only the navigation use case is supported, but we plan to extend to more and more actions, in order to automatize as much as possible simple operations for the developer.

It can be curious how the actual line is made: we appended to the HTML of the editor a SVG line, that follows the cursor using the `document.onmousemove` listener. As seen as this is a total different language from the other that we have seen until

now, and used only in this very specific application, we report it in the following snippet:

```
<svg height="100%" width="100%" style="position: absolute;">
  <line id="lineAction" x1="`+event.clientX+`" y1="`+event.clientY+`" x
2="200" y2="200" style="stroke:rgb(0,0,255);stroke-width:2" />
</svg>
```

Where we can see that the ending point of the line are dynamic, depending on the `event` variable.

The function not only draws the line as we have seen, but also create the real action in the model. Finally, when the action is bound to the element, a visual element appear in the Visual Editor – for the *navTo* event, it is an arrow that links the first screen to the second screen, always created in SVG.

- setNewCSS() | setNewAttr()

This pair of function does the simple job to attach the style of an attribute, specified in the lateral panel, to the real element. While for the first one is not needed to update the model in this step – the entire CSS will be read from a specific function in the backend – for the second this should be done and it is not as straightforward as one can imagine. Certain attributes need more attention, in fact: is the case of the id, that, by definition, should be unique. To understand if the new id has been already used we convert the whole model to a string and we search for the id inside it: a simple, but effective, method. Other attributes need specific care in order to align their visualization on the editor to what the attributes are saying: for example, the value text on the textfield.

- Minor functions

We have discussed of the main functions present in the front-end. Yet, a couple of small ones are present, that comes in handy to perform very specific task. They are the following:

- `removeAllLayoutClasses()`: removes all the HTML classes relative to a Nativescript layout from the input element
- `deleteAction()`: deletes the action that should be assigned to an element (e.g. a button), simply removing the relative entry in the model.

4.6.4.3 Editor – Back End

The main purpose of the back-end module of the editor is to allow I/O from the file system and the execution of shell commands. <allunga>.
The functions present inside this module are the following:

- `buildNewProject`
- `compileForNS`
- `run`
- `buildManifest`
- `saveProjectToMainScreen`

- `storeModel`
- `restoreProjcetFromFile`
- `restoreProjectFromString`
- `restoreFromSubmodel`
- `openFileInCodeEditor`
- `dirTree`
- `populateFileExplorer`
- `adaptModelForTree`
- `createFileExplorer`
- `updateFileExplorer`
- `updateModelExplorer`
- `selectInModelExplorer`
- `createComponentsExplorer`
- `saveProjects`
- `pushVisualChangesInCodeEditor`
- `pushEditorChangesInVisualEditor`
- `importImage`

We will cover them in depth in the following paragraphs.

- Global Variables

As we have seen in the front-end module, also in the back-end one global variables appears and are shared in the whole program. Yet, as this is not a good practice, few variables of such a kind have been inserted and will be listed in table 12.

Variable Name	Possible values	Description
fs, path, os	NodeJS Object	NodeJs modules that allow to communicate with filesystem, get the path of a file and information about the OS
workingPath	String	Contains the path of the file currently opened in the editor
projectName	String	Contains the name of the project currently opened in the editor
globalApplication	Application Instance	Contains the “backup” of the model, while the original will be split in parts in order to edit specifi screen (2-way)
codeEditor	CodeMirror object	Containes the object representing the CodeMirror editor that has been inserted in the page

Table 12 - Global Variables in Editor Back-End

- `buildNewProject(config)`

This function creates the file hierarchy of the project in the file system. The parameter that has in input is the variable sent by the Start Screen to the Editor,

thus it contains all the data that the user has specified in that first view. In particular, the function uses the second and the third element of this array, that represent the absolute path where the project is located and its name. These values, that have paramount importance in the whole project, are stored in specific global variables. Then, the shell command needed to create a NativeScript application is prepared and executed asynchronously in a child process. Finally, the function builds the manifest file of the project and prepares the content of the components explorer using the fuctions `buildManifest()` and `createComponentsExplorer()`.

- `compileForNS()`

This function performs all the operations needed in order to start the translation of the model in the real files needed by NativeScript compiler, that will generate a running application for Android and iOS. To do this, the function uses Nodejs *writeFile* function in order to save to the *app.js* file, considering the starting screen selected by the programmer. Then, iterating on all the other screens, it calls the proper functions from the translator module in order to write XML, CSS and JavaScript for all the screen present. At the end, it updates the visualization of the file explorer on the left calling the proper front-end function.

- `run(mode)`

This function, called after NativeScript files are written, execute the proper shell command in order to execute the application in a specified environment. Thanks to a switch statement that analyses the platform selector input box in the UI, specific commands are executed, different in the platform (iOS/Android) and the environment (real device/emulator). The command is executed in an asyncrouns way, in order to be not blocking for the ui: this allow also to display the output of the shell inside the console of the editor, in order to provide to the developer a useful feedback of what's going on. The developer is also notified when the process is finishing or has an error thank to a notification.

- `buildManifiest()`

This function has the simple objective to create the manifest file needed by NativeScript in order to compile the app. It simply builds a JavaScript object that has an attribute name, the name of the project, and a main one, that specifies where the entry point of the app is located, that is fixed in app.js.

Thanks to a proper nodeJs call, the JSON derived for this object is written to the disk, saved as the package.json of the app. Finally, a proper line is written to the console in order to notify the user of the success of the operation.

- `saveProjectToMainScreen()`

This function writes basic details of the current project – its path and its name – in an history file, located a shared location in the file system. This allows the Welcome Screen to read these details and show the project in the "last projects" section, in order to let the programmer open projects quickly. It is interesting to notice that the shared folder, that is different depending on the OS where the software is ran, can be identified using an Electron-specific nodeJs function `getPath:`

```
const {app} = require('electron').remote;
var homePath = app.getPath("userData");
```

That, as said, will return different paths depending on the OS:

- `%APPDATA%` on Windows
- `$XDG_CONFIG_HOME` or `~/.config` on Linux
- `~/Library/Application Support` on macOS

- `storeModel()`

This function allows to store the model as a file in the disk. The main idea is to convert the model to a string and then store it, but some operations should be performed before. The first one is that the model should be enriched with all the CSS styles of all components and screens that are present in the front-end. Then, the model is converted to a string, but the string is concatenated to other important information needed to restore the project. These are its name, its path and the component counter. Such a string is finally stored as an MSA file (Mobile Studio Application) and, if not already done, written in the history in order to be displayed in the Welcome Screen.

It is important to notice that, as after this storage is performed the content of the disk is perfectly consistent with the app that the developer is building, the `dirty` flag is set to false.

- `restoreProjectFromFile()`

This function reads an MSA file from the disk and calls immediately the restoreProjectFromString, that will be able to builds up a project with the information contained in the file. It also performs some simple front-end changes, in order to let the software be ready to house the new project.

- `restoreProjectFromString(msaString)`

This function is intended to do the opposite job of the storeModel one, being able to take a string – that is read from the disk thanks to the restoreProjectFromFile() function – and translate it to a proper model.

The function expects, as an input, a string composed by three substrings, containing the name of the project, its path and the string representation of the model. Immediately, the function splits these components and assign them to the proper variables. It decode the string of the model, and starts to rebuild the UI of the program. This is the most difficult part of the function, that should explore all the model in order to rebuild the proper screens and components that build it up.

The HTML elements corresponding to the model entries are then appended inside the main UI of the Editor.

Finally the loading screen is hidden.

- `restoreFromSubModel(sm)`

This function is a wrapper from the restoreProjectFromString one, allowing the latter to work also with a model object instead of a string. Its name specifies that it is intended to use with a submodel, a portion of the original model that refers only to a single screen.

- `openFileInCodeEditor(filePath)`

This function take as input the path of a file and visualize its content inside the Code Editor. To do so, it analyses the full path of the file passed as parameter, considering the extension of the file. It prepares the editor according to that, setting a CodeMirror option.

It is important to notice that, if the extension is MSA, so the full project has been selected, the function does not start the code editor, rather it start the full visual one with every screen of the project opened.

- `dirTree(filename)`

This function is needed in order to create a FancyTree-compatible object that represent the files and the directory that are present in the path `filename`. It is interesting how this function skips big and useless platform-specific folders, using a simple if statement.

- `populateFileExplorer(info)`

This function prepare the file explorer HTML element to receive the data from the file hierarchy. Specifically, it also enables to create a new code editor when an element inside the file explorer is clicked

- `adaptModelForTree()`

This function, similarly to the previous one, prepares the component explorer to receive data from the model and display its parts. It is important to notice that this is needed because, although the model is already present as a JavaScript object, it does not meet the requirements of the library FancyTree, used to display such object in a visual tree format. The model is thus converted with a specific syntax that has been explained when talking about this library.

- `createFileExplorer()`

This function is simply a wrapper for the `populateFileExplorer()` function, that simply prepares the UI of the file explorer, builds the actual string that represent the full path of the project and then call that function.

- `updateFileExplorer()`

This function is called whenever the File Explorer has new data to display and its visualization needs to be updated. It does the very same job of the previous `createFileExplorer()` function, but at the end uses a specific Fancytree function – the reload one – to push new data on the explorer

- `updateModelExplorer()`

A simple function that does the same job of the updateFileExplorer, but this works with the Model Explorer. It need thus to call the adaptModelForTree in order to put the model in the correct format to be displayed in that component.

- `selectModelInExplorer(id)`

This function is needed in order to visually select a component inside the Model Explorer.

- `createComponentsExplorer()`

This function populates the Components Explorer converting the model to a proper format, using the adaptModelForTree function. It also binds the action to select one element – open it in visual editor or code editor – when clicking on the element in the selector.

- `saveProject(silentMode)`

This function stores the model into the disk and prepares all the files needed in order execute the app, still not compiling it, if the visual editor is currently active. If instead the Code editor is active, the function uses a node.js call to write the current active file in the code editor into the disk.

- `pushVisualChangesInCodeEditor()`

This function is one of the pair that allows a two-way code editing style in the IDE. This one specifically assumes that the editor is in the Visual mode: it cuts the model into the specific file that should be opened and stores it to the disk. Then, it prepares a the code editor opening that file.

- `pushCodeEditorChangesInVisualEditor()`

This function is the dual of the last one. It is able to store the files in the code editor, then istantiate the visual one considering the new model created and, thus, with the modification inserted by the user.

- `importImage (completeImgPath)`

A special care is needed for the image component, because it has to be included in the root of the project, in order to be visible in the final app. To do this, the function uses node.js to analyse the image file, in order to get its name and its path. It the uses again node.js to execute a shell command that copies the image file in the root of the project, in order to let it be used, returning the resource name. It is important to notice that in future an approach like this can be extended to every external resource, not only images.

4.6.5 Translator Module

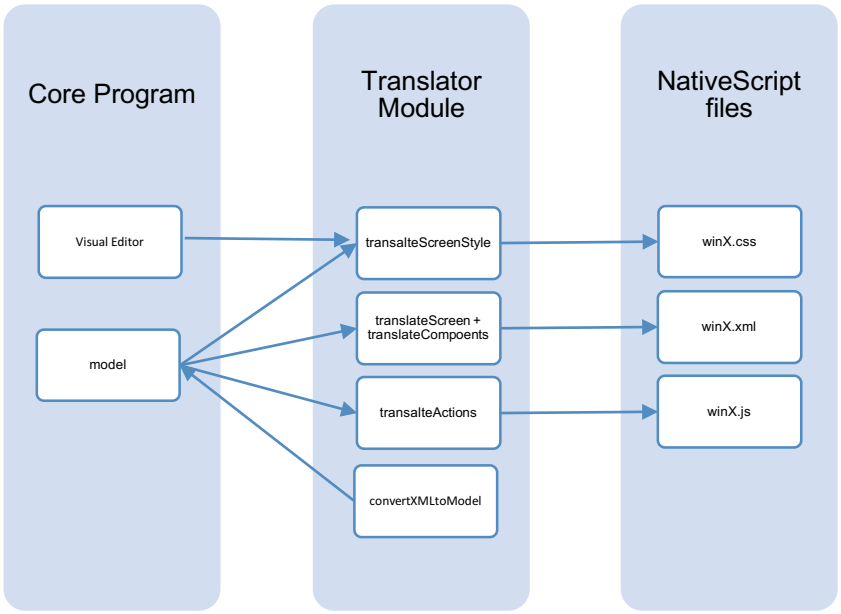


Figure 34 – Translator Module Schema

Until now, we have seen how Mobile Studio is able to work with HTML elements, but, as we know, to be compiled by NativeScript and thus run in real devices, these components should be translated from HTML to the specific NativeScript syntax.

A translator module, contained in the file *translator.js*, is present to this specific purpose. The functions contained in it do not access to the HTML elements, but analyse the model and can extrapolate from it the XML, JS and CSS of the NativeScript app that will be compiled. The translator module is composed by the following functions:

- `translate`
- `translateScreen`
- `translateComponents`
- `translateScreenStyle`
- `translateActions`
- `convertXMLtoModel`

We will look at each function of this module in detail.

- `translate(application)`

This function takes in input the `application` object that represent the model and then builds an array of strings, where each of them is the translation of one screen of the project. To do so, it iterates on the `application.screen[]` array and calls the `translateScreen` function, present in the same module, for each screen found.

- `translateScreen(screen)`

This function takes in input one object that represent a screen of the app, taken from the model, and returns its translation in XML, in format of a string. To do so, it builds the skeleton of the document according to the proper layout of the screen

and then calls the `translateComponents` function, that will populate the XML of the screen with the one of the components. We can have a look on how the translation is made for the stackLayout:

```
case "stackLayout":
            return `<Page id="`+screen.id+`" loaded="loadView`+screen.i
d+`">
                        <StackLayout>
                            `+translateComponents(screen)+`
                        </StackLayout>
                    </Page>`;
```

As we can see, the page is built with some attributes, the id – taken from the model – and a `loaded` event – the equivalent of the `onload` in standard browsers.

- `translateComponents(screen)`

This function takes a screen as input from the module and iterates on the components in order9rings that represent them in XML. It is crucial, to distinguish between different type of components and access to specific attributes of them that should be propagated from to model to the XML.

The reader will easily understand this concept looking on how such a translation can be done. Let's take the example of the *Switch* component: the specific code that translate it is written below.

```
case "Switch": componentsTranslated += `
                  <Switch id="`+screen.components[i].id+`" checked="`+sc
reen.components[i].specificAttributes["set"]+`" />`;
    break;
```

As we can see, the *specificAttribute* `set` is accessed. If we remember from the Switch explanation in the model, this attribute is a flag that indicated if the switch should be activated or not. This flag is inserted in the xml attribute `checked` that has the same purpose, but in NativeScript syntax.

When a component is translated, its XML, in form of a string, is appended to the variable `componentsTranslated`. Iterating on all the components, at the end we will have the proper translation of all of them in one screen. The function end appending at the XML created the closing tags and the returning it.

- `translateScreenStyle(screen)`

This function is not much different from the others we have seen until now, but it is totally dedicated to define the styles of the app, having in output string formatted as a CSS file. To do so, the styles are read directly from the Visual Editor and then formatted in the standard CSS syntax. The core of the translation is performed with the following lines of code.

```
var styleWindow=`#`+screen.id+`{
                `+usefulStyles+`
```

```
            }`;
var styleComponents = "";
for (var i in screen.components){
    if(!$("#"+i+"").attr("style")) continue;
    styleComponents +=`
        #`+i+`{
            `+$("#"+i+"").attr("style")+`
        }`;
}
```

Where we can see how the CSS rule are built.

- `translateActions(screen)`

Similarly to the other functions of the translator module that we have seen until now, the `translateAction` takes in input a screen from the model and outputs a string, that this time contains the JavaScript code of all the functions that the components in the screen are able to do. If the reader has a clear picture of how the a NativeScript JavaScript file should be composed, this translation may seem straightforward.

The function reads from the model if a certain component has to be *dynamic*, this means that will be used from a JavaScript function: if it is so, that component should be declared. Then, the code iterates on the actions declared for that component and build up the real function that will perform the action. For now, only button taps are supported. We can see the code of this function below.

```
var js = [header js code]
for (var i in screen.components){
    if (screen.components[i].isDynamic)
         js += `var `+i+`=view.getViewById(page, "`+i+`");`;

js += `}
exports.loadView`+screen.id+` = onLoad;`
if (screen.components[i].actions)
    for (action in screen.components[i].actions)
        switch(action){
            case "tap":  js += `
                exports.`+action+`Action = function (args){
                        frame.topmost().navigate({moduleName: "`+screen
.components[i].actions[action]+`"});
                }`
         }
return js;
```

- `convertXMLtoModel(xml)`

This function is a little different from the others in the Translator module because it is the only that do a translation in the opposite way as the others. Instead of writing the content of a file from the module, it does the opposite: from an XML file it is able to reconstruct the model. This is needed when the user makes some changes in the XML, from the code in the editor, and then opens the Visual Editor. It is clear that the user wants to see the changes done in the code editor.

To do so, the string in converted in XML – thanks to the parseXML() jQuery method – then is explored using standard both HTML methods – like `.childNodes` or `.tagName` – and JQuery ones - `$(currentPage).children()`.

4.7 User Interface

Figure 35 - UI of the welcome screen

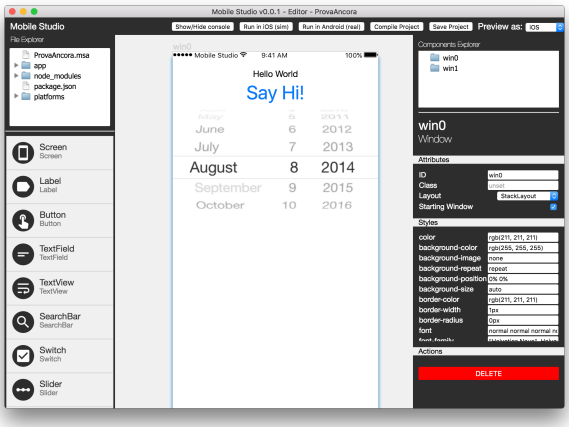


Figure 36 - UI of the main editor

The UI of Mobile Studio has been thought to be the most user friendly as possible. This vision was challenging in many ways: an IDE is usually a difficult software, often intended to be used by skilled people. In this sense, we tried to simplify the process, allowing also unexperienced people to start developing. This was done greatly privileging the Drag&Drop UI instead of focusing on the code editor, still present and necessary in order to develop advanced features. This is an open challenge that will follow the further development of the IDE: migrate more and more functionalities from the Code editor to the Visual one.

4.7.1 Design

In the writer opinion, it is worth spending a little time speaking about the design choices that lead to the development of the UI of the software. The usage of HTML and, in particular, CSS, lead to an incredibly flexible user interface, that tries to be the easiest to use as possible using animations and minimalism. In all the windows of Mobile Studio we can find visual styles recently popular: in the welcome screen the usage of the shadows to differentiate elements and real world styled ones (e.g. the “Beta” ribbon) reminds Apple’s skeuomorphism used before iOS 7. Ideas to renew this screen, in order to let it be lighter and cleaner are present and will be discussed in the future developments paragraph. The UI of the Editor screen is instead clearly inspired to the Metro UI of the latest versions of Microsoft Windows, with a monochromatic high-contrast palette that let the screen be as minimal as possible.

The icons, everywhere, are the one of Google Material Design [inserisci link a material icons]. Last, but not least, the main icon of the program is inspired to the one of Politecnico di Milano, with same colour palette and features to clearly remark the origin of the software.

4.7.2 Welcome Screen

Figure 37 – First view presented to the user

Figure 38 – New project view

It is common for IDE to start up with a dialog that shows the latest project and allows the user to create a new one. Mobile Studio is not different and the first window presented to the user, shown in Figure 36 has these very purposes.

As we have explained in paragraph 3.1, the physical Start Screen is composed by two logical screen, the Welcome one and the New Project one, that can be seen in figure 36 and 37. Inside the first, the user is able to quickly open a recent project, that appear in a list populated by the back-end logic, that analyses the content of an history file; while in the second the user can specify the initial details of her app – e.g. the platform where it will run – and then start the editor.

Both logical screens exist in the same HTML file, index.html, and are composed by two `<section>` elements, put side by side thanks to CSS positioning: this allows to use a CSS smooth transition in order to pass from one to the other. This transition is made up with two CSS animation, one to show the second screen and the other that allows the first to be hidden. It is interesting to discover how such a animation is specified with the following code.

```
@keyframes hideFirstScreen {
    from {left: 0px;}
    to {left: -800px;}
}

#firstScreen{
    position: relative;
    top: 0px;
    left: 0px;
}

#firstScreen.animate{
    animation-name: hideFirstScreen;
    animation-duration: 1.5s;
    animation-fill-mode: forwards;
}
```

4.7.3 Editor Screen

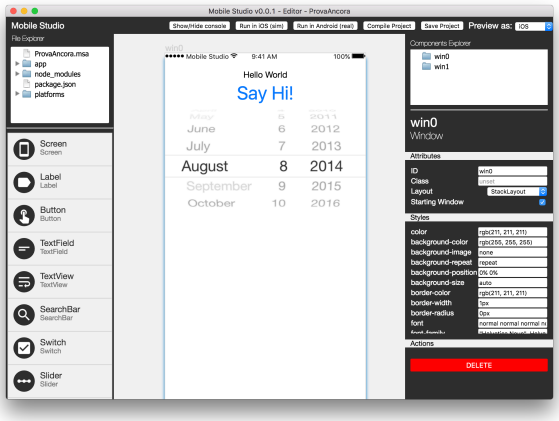


Figure 39 – Editor Window – Visual Editor

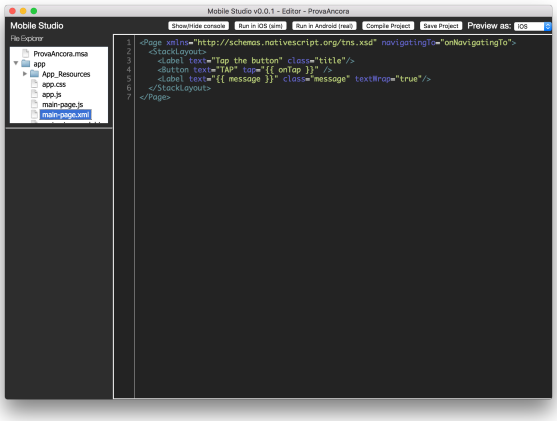


Figure 40 – Editor Window – Code Editor

Similarly to the Welcome Screen, also the Editor one is composed by two logical views. They are the Visual and the Code editor, that allow to work on same files but in different way. The first allows the building of the app by dragging and dropping of components in a visual way. It is able also to specify basic interactions – e.g. navigation – between the views.

The code editor, instead, is able to provide a file-specific view that makes the programmer able to add further interactions and dynamicity to her program. As we have seen it uses mainly CodeMirror library to display a complex code editor. We will now see in detail how these screens have been composed.

4.7.3.1 Shared Modules

Both in the Visual and in the Code Editor appear few identical components: we will list them and analyse each one.

4.7.3.1.1 Menu

Menus are an element used frequently in desktop software and are one of the most noticeable differences of the UI in various platform. Basically, the allow a faster way to access to functions in a program and, in Windows, are displayed below the frame of the windows of the program, while in macOS, as we can see in figure 41, are embedded in the OS and displayed in a bar always at the top of every screen.

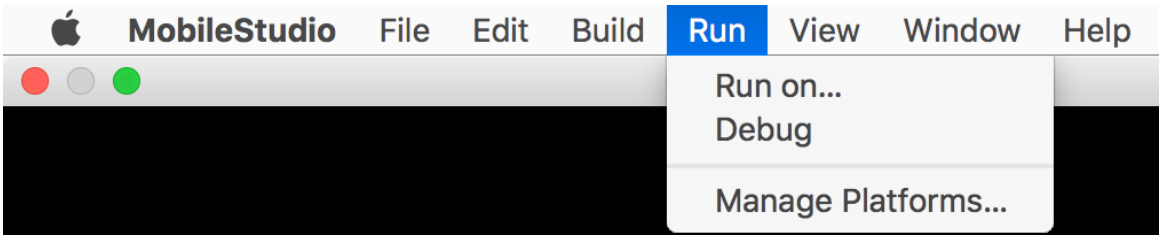


Figure 41 – Menus in Mobile Studio

In Mobile Studio, no menu is giving a function and in the current implementation we specified some of them only to test this interaction that an Electron software must have with the operating system. We discussed on how to implement such menus in the relative development paragraph.

4.7.3.1.2 Console

The console is basically a container, that by default is hidden, able to show the textual output of Mobile Studio, that mainly is related to NativeScript compilation. Most of JavaScript output in console is in fact replicated to this other console, thinking that the developer should not have access to the Electron Developer Console. This component, by default hidden, can be show at any time clicking on the relative button on the toolbar. The component can be seen in figure 42.

Figure 42 – Console

4.7.3.1.3 Toolbar

The toolbar is a component present in each screen of the editor and it is shown in figure 43.

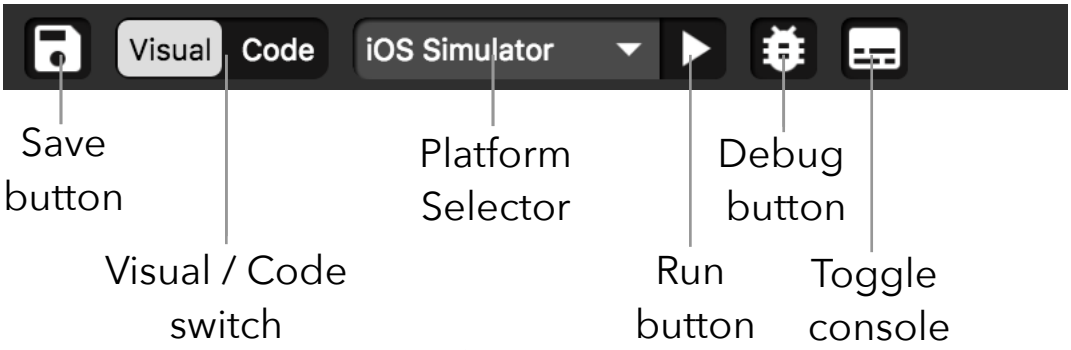


Figure 43 - Toolbar in the editor

As we can see, it contains the buttons that allow the developer to access to the main functions of both editors. We quickly explain each of these buttons:

- Save Button: when in Code Editor, it saves to the disk the current opened file. When in Visual Editor, saves the changes made at the whole project.
- Visual/Code Editor switch: permits to quickly change editor between the two present, when this is a legal operation – JS and CSS file can be opened only in the Code Editor and do not support a visual representation.
- Platform Selector: this crucial component is needed in the Visual editor to both set the preview that can be seen inside it (iOS / Androi) and to select to which environment output the code (real device / emulator)
- Run button: accordingly to the environment selected, when the user clicks on the run button all the operation needed to compile and run the app in the selected environment are performed.

- Debug button: similarly to the run button, the debug one compiles the project for the environment selected in the Platform Selector, but runs the project in a debug mode, where the developer can analyze its app
- Toggle console: shows or hide the console.

4.7.3.1.4 File Explorer

The File Explorer module is present in the left lateral panel of both Editor. In the Code one, it is expanded to get all the panel length. In figure 44 we can see an example of this explorer, seen instead reduced in the Visual Editor.

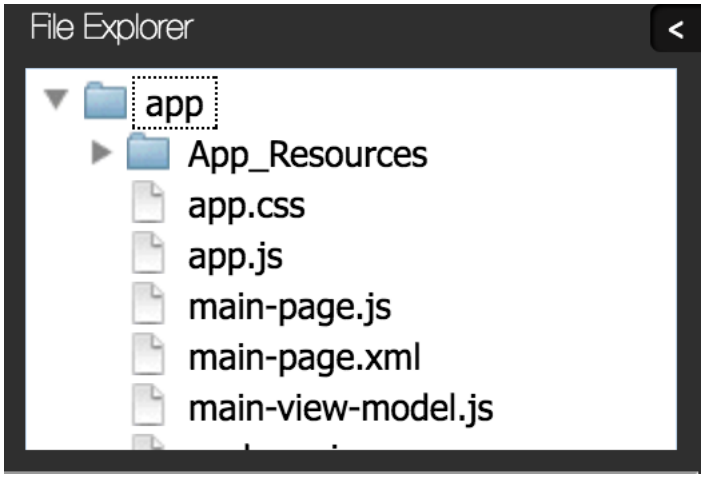


Figure 44 - File Explorer

It contains a hierarchical visualization of the file inside the project. As we have seen in the development section, this is provided by the plugin FancyTree. The user is able to select each of the items, that will be properly visualized in the Editor.

4.7.3.2 Visual Editor

The Visual Editor permits, as the name says, to build the app visually, letting the developer use drag & drop to build the UI of the app and allowing her to specify all the details. It is composed by many elements that we will analyse in the following paragraphs. We can see the Visual Editor, where all its main part have been properly highlighted, in figure 45.

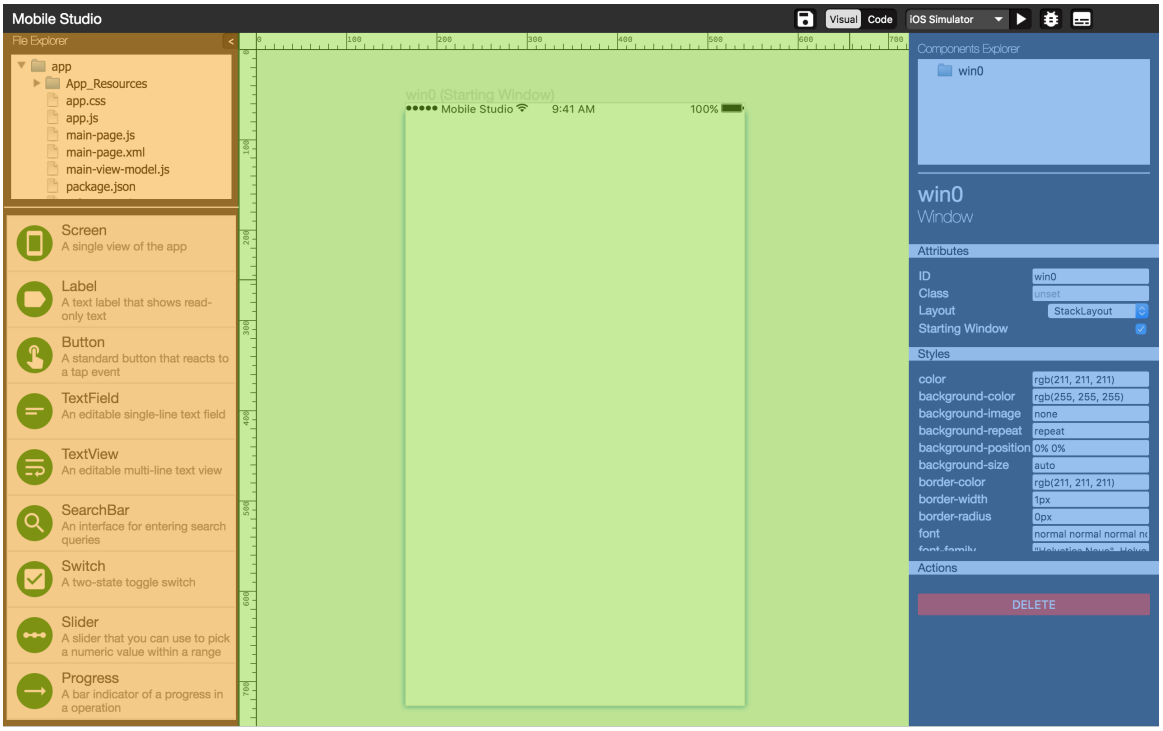


Figure 45 - Visual Editor and its components

We will explain each of the component highlighted in the following paragraphs.

4.7.3.2.1 Main Content

The Main Content is the space where the user can visualize the preview of its app and build it visually, dragging components from the Left Panel. The Main Content inherits the concepts of a Zoomable User Interface, a concept not new – present in software like Xcode, but also in design ones like Sketch – that allows the user to freely pan and zoom the container view.

In this container, the user is allowed to drop screens, that are the only components that allows this behaviour. All other components, in fact, should be dropped inside a screen. The screen can be freely moved around and so can be the components inside it, although they must obey to the layout specified in their parent screen.

4.7.3.2.2 Left Panel

The left panel contains the File Explorer, a component that is shared between the Visual and the Code Editor that we have seen in paragraph 3.7.3.1.4, but also a list of visual UI elelements that the programmer can add to her app. To do this, she has to drag on element of the list to the main content, in a drag&drop approach typical of Visual-based IDE.

In the list it appears the name of the component, a brief description and also an icon. In this beta stage, the icon is coloured in green, yellow or red, depending on how the component is supported – totally, partially or not – in the current build of Mobile Studio.

4.7.3.2.3 Right Panel

The right panel, shown in figure 46 allows the programmer to set the appearance and some simple behaviour of a specific component.

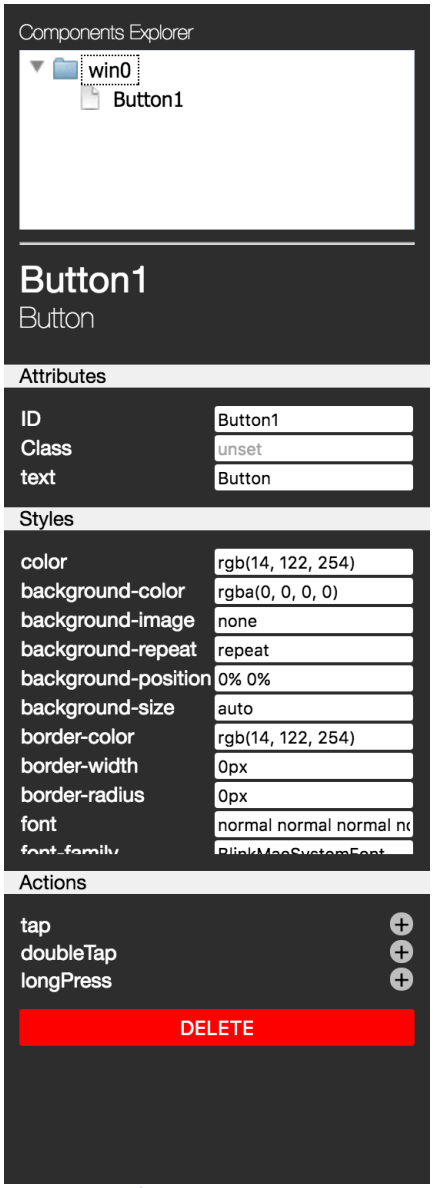


Figure 46 - Example of the right panel with a button selected

From the upper part down the first element presented to the user is a Component Explorer. This component presents strong similarity with the File Explorer seen in the left panel and in fact is built always using FancyTree external library, as we have seen in the development description. This explorer, though, is not intended to browse file of the project, but instead the components that the developer has inserted in her project. All the components are visualized in a hierarchical tree, that has always, as roots, the view where they have been inserted. The user can easily select one component simply clicking on it, both in the Component Explor and in the real visualization of the component in the Main Content.

The details of the component selected appear immediately below the Explorer: the id of the element and its type – window, button, label, etc.

Then, in the panel three section are present: the attributes, styles and actions ones. These reflect the separation of structure, visualization and logic that will be reflected in the files generated by the project – XML, CSS, JS.

The first section, the Attributes one, allows the developer to edit the properties of the structure of the element. It is important to notice that some of them are common for all elements – e.g. the ID – while some other depends from the element type – e.g. `isStartingWindow` for windows. Being built at runtime, the panel should take into account the type of the element selected and display the correct attribute.

The second section show all the CSS properties that the selected element presents. It is important to notice that only NativeScript compatible CSS properties are displayed, in order to prevent problems in the translation. Yet, thanks to jQuery, all the CSS present of an element inside the main content is extrapolated and put inside this view. The user can edit these fields and immediately see the effect on the visual preview in the Main Content. All NativeScript compatible CSS rules are displayed for every component.

In the third section the developer can find the actions that a certain element can present. In an opposite way of the section before, this is totally dependent from the element selected, because different elements can have totally different actions. Moreover, some elements can have no action at all: windows are a clear example of this situation.

When instead a component supports one or more action – this is inferred by the supportedActions attribute of the model that we have discussed – they are listed in this section. The developer can give an action to the element by clicking to the plus button near the action name: in an Xcode fashion, a line is displayed and allows the developer to link the action, and thus the element, to another element in the screen. In the current release, only other screens are supported as receiving elements of this interaction: when the developer do this, the action of navigation from one screen to the other is assigned to the action selected.

Such an action compare instead of the plus button and its highledt by an arrow also in the Main Content. The user can easily click on the action name in order to delete it.

Finally, the right panel feature a button that allows the user to delete the element selected. Pressing it, the element will be deleted both in the Main Content and in the model, disappearing total from the project.

4.7.3.3 Code Editor

The other main visualization of project files, far easier than the Visual one, is offered by the Code Editor, that can be seen in figure 47. When enabled, the screen is cleaned and all the elements present in the Visual Editor are deleted, excepted for the left panel, where the component list is deleted and the file explorer expanded. All the rest of the screen is occupied by a text editor that displays the code of the specific file selected, highlighted depending on the file. As we have seen in the development part, this is made possible thanks to Code Mirror external library.

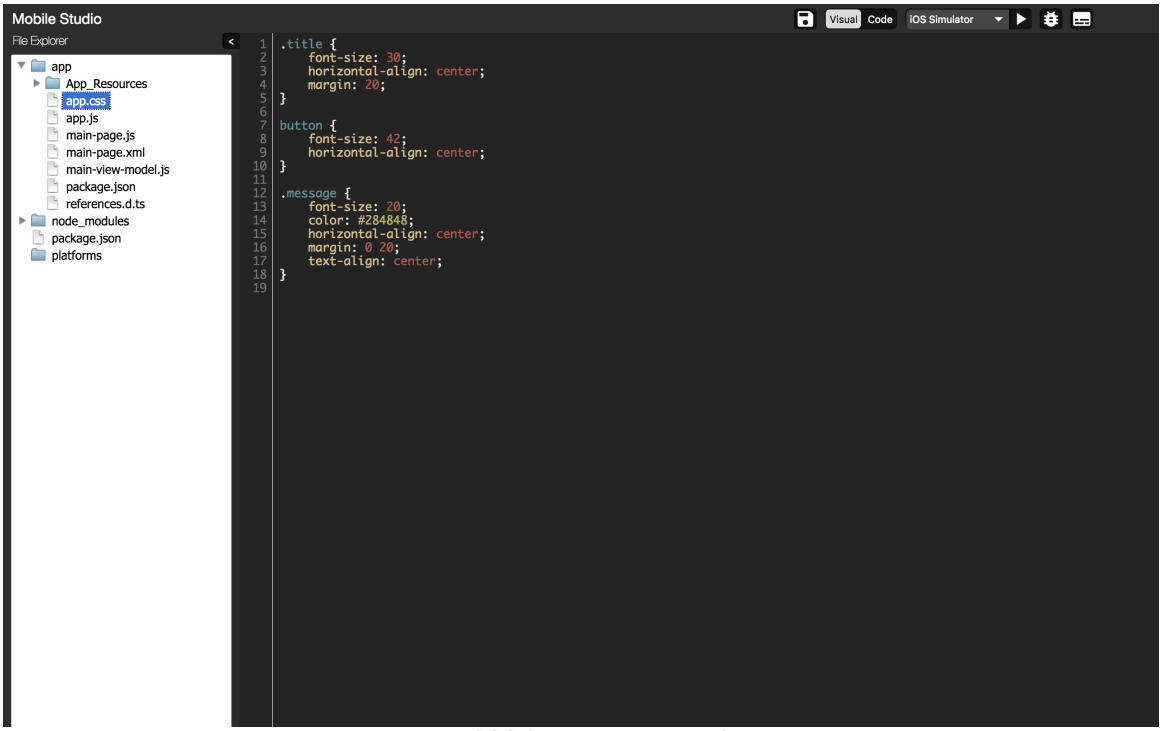


Figure 47 - A CSS file opened in the Code Editor

4.8 The app in output

Until now we have seen a full tour of Mobile Studio, from its core concepts to how it was actually developed. We still have to figure out what are the products that this software can build. This will allow us to take some conclusion on all the approaches that we have seen, the cross-platform JavaScript based both in the mobile and in the desktop world, understanding the power and the limits of these techniques. Firstly, we will analyse the outputs of our program. As we told, NativeScript is able to build directly a native application, but it can also stop and build only the Xcode or Android Studio project that the developer can further modify. We will analyse both situations.

4.8.1 Directly to the app

For our test, we built a very simple app with two screens and the ability to navigate between them. With Mobile Studio, no coding was required and everything could be done in the Visual Editor. We can see in figure 48 the project and in figure 49 the output in the iOS simulator.

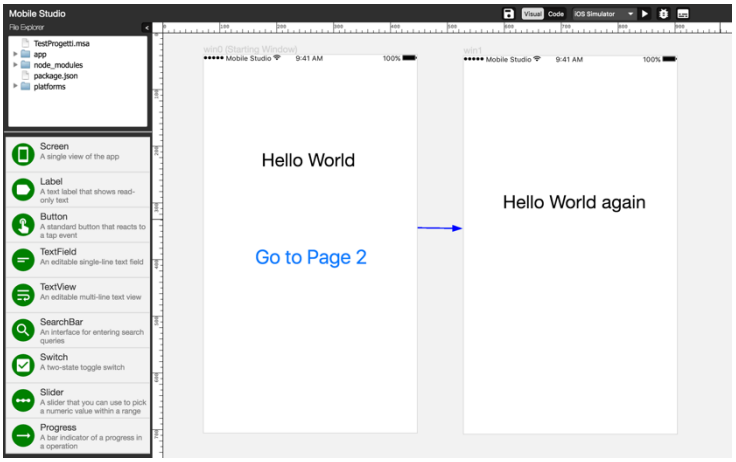


Figure 48 - Hello World app in Mobile Studio

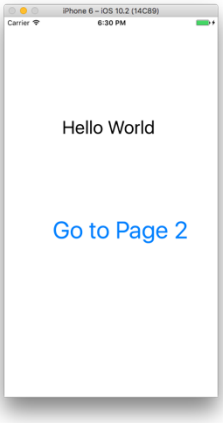


Figure 49 - Hello world app in iOS Simulator

We can see how a perfect visual correspondence is achieved thanks to the CSS rules written in Mobile Studio.

4.8.2 Xcode and Android Studio projects

Analysing the Xcode and Android Studio project derived from the compilation of NativeScript we discovered that only few modification to the NativeScript app could be done. In fact, as we can see looking in figure 50, we cannot access directly to the components and the logic that we wrote with NativeScript syntax: we can see only a screen where the NativeScript runtime is initiated, with the path to the web project.

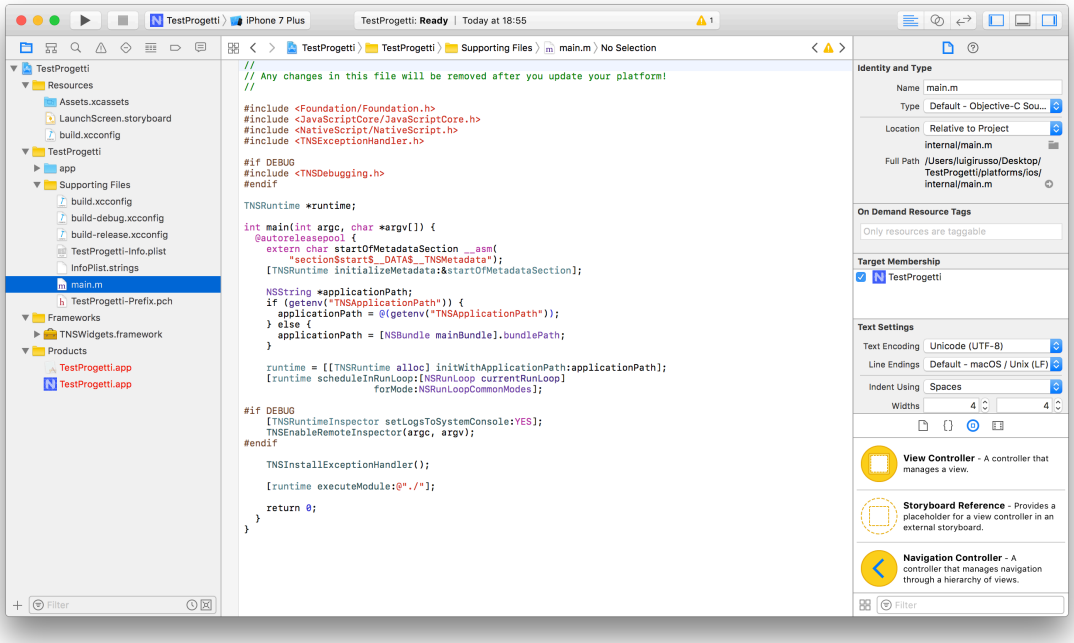


Figure 50 - NativeScript Hello World project opened in Xcode

This is something quite surprising for us, because it consisted in an approach not different from the hybrid ones, like PhoneGap. A real translation is not performed on the fly and we were quite uncertain if this approach could lead interesting performances.

4.9 Performances

4.9.1 Mobile Studio performances

It is quite difficult to measure the performances of an IDE such as Mobile Studio. As a general writer opinion, the Electron way of programming delivered surprisingly fast applications and Mobile Studio make no difference. To better have an idea of the performances of this product, we can compare events in multiple similar IDEs.

4.9.1.1 Comparison with other IDE

In table X we can see a table on various IDE performances, among them is present Mobile Studio. All the test have been run in a MacBook Pro Early 2015, with 2,7 GHz Intel Core i5 processor, 8 GB of RAM and Intel Iris Graphics 6100 1536 MB graphic card.

	Android Studio	*Xcode*	*Mobile Studio*
New project screen opening	8,65 s	2,35 s	1,32 s
Already existing project loading	50 s	3,28 s	0,8 s
App execution	10 s *	17 s	52 s

Table 13 - IDE timing comparison. (*) App executed on a real device (Nexus 5)

Looking at our experiments we can state safely that Mobile Studio, and we could also extend this notion to every Electron-based software, runs smoothly and are quite indistinguishable from the software programmed with classical methods. Many actions are quite instantaneous, while we have seen standard software – e.g. Android Studio – that can be painful slow. Yet, when Mobile Studio has to use other services, like NativeScript, we can see that it has difficulties and force the user to wait longer. This slowdown is due to the usage of NativeScript, that, maybe still being a very young project, is not sufficiently optimized to build the project as other well known products are.

4.9.2 App in output performances

Although we have seen various comparison between apps programmed in different methods, we have not seen anything similar dedicated to NativeScript and thus we set up our custom tests in order to understand if this approach is able to really deliver fast apps, like it states. We conducted this analysis in two ways: the first, more rigorous, will be based on specific tests done in two identical apps, one programmed with Native methodologies, the other with NativeScript, while in the second will be more empirical and based to our experience with this new platform.

4.9.2.1 Calculated

As we have seen, an analytic comparison between NativeScript apps and Native ones is not present, and actually not straightforward to do. We have tried to

accomplish that, taking as inspiration the analysis done by Microsoft when comparing Native apps to PhoneGap ones [13], extending it to another dimension that we thought be important in this context. In particular, we built the very same app and run on a real device and get some performance data. Unfortunately, the only device that we had was an Android one, so our analysis will focus on this platform. We thus created a very simple application in Android Studio and then in Mobile Studio and we run some of the following test. It is important to notice that all our tests were run in a Nexus 5 smartphone with Android 6.0.1. Each event was analysed in a “cold” way, as soon after the boot of the device, without the app in memory, and in a “warm” one, after the app had already been started once. All the times are in seconds.

4.9.2.1.1 App lifecycle and Memory

It is crucially important, for the user, how fast an app will start and how it will resume from background. To test this, we dynamically coloured the background of the app at each *create/resume* event, analysing how much time was needed in order to have the background coloured as so the app ready. We also analysed the memory required, using, for both, the Android monitor. The results are in table 14.

	Android Native	*NativeScript*
Create	1.979	3.089
Restart	0.868	0.617
Memory required	16.39 MB	25,68 MB
Space on disk	3,96 MB	36,86 MB

Table 14 - Performance comparison - App Lifecycle and Memory

We can immediately see how the NativeScript app is the heavier one, but we can notice how small the gap is between the create time. Surprisingly, the NativeScript app was faster to resume execution compared to the Native one.

4.9.2.1.2 Communication performance

Another metric that we thought important in the performance of an app is how quickly it can handle communication with remote servers. We thus let our app send a get request to Google Books API, retrieving then the details of a book, and measured the speed of this in both Native and NativeScript methods. It is important to notice with both methodologies we used an asynchronous request, that in Android Native was performed by an *AsyncTask,* while in NativeScript we used the standard *fetch* JavaScript function. We were able also to monitor the timing in a better way that we did with the lifecycle because we were could use specific functions to retrieve the timestamp of an event in both platforms – in Native Android thanks to the class *Calendar*, while in NativeScript with the object *Date*. The results are in table 15.

	Android Native	*NativeScript*
Cold Request	0,525	0,567
Warm Request	0,483	1,091

Table 15 - Time to load a remote JSON - Native Android vs NativeScript

We can see how the request time are very similar. Yet, NativeScript seems to perform worse when doing a request after the resume of the app.

4.9.2.1.3 Computational performance

A very technical aspect that cannot be ignored when talking about software performance is how quickly a certain algorithm can run in different platform. To test also this dimension, we created a very simple function, both in Java and in JavaScript, able to understand if a certain number is prime or not. We then gave to this function a very high prime number, in order to understand how much time was needed, both for the native app and the NativeScript one, to understand that it was prime. The results are summed up in table 16.

	Android Native	*NativeScript*
Cold Calculation (ms)	6.496	5.848
Warm Calculation (ms)	6.165	5.707

Table 16 - Time to understand if a number is prime or not - Native Android vs NativeScript

The reader can see how the computational times are quite identical in both platforms and surprisingly are a little faster in the NativeScript app.

4.9.2.1.4 Development complexity

Our last dimension to compare the two methods is from the developer side: we wanted to understand how easily the app can be built with both methodologies. We summed up some metrics in the table 17.

	Android Native	*NativeScript*
Actual Lines of Code	253	77
Development Time	2 h	2 h

Table 17 - Development complexity comparison - Native Android vs NativeScript

Where we can see how JavaScript can be more considered more concise in respect to Java, achieving the same tasks using three times less code. Yet, it is important to notice that in our experience this hasn't accelerated the development: while Android is well documented and all its resources can be found easily on the Internet, the same does not holds for NativeScript: we often used a JavaScript function without knowing if that was supported by NativeScript or not. NativeScript obviously has a documentation, but it is not as comprehensive and complete as the one of Android and the community of developer of the first is a fraction of the one of the second.

4.9.2.2 Empirical evaluation

As the reader could have understood looking at our calculated analysis, when using a NativeScript app and its native counterpart it is difficult – in the writer opinion quite impossible – to understand which one is the native or the cross-interpreted one. The usage of real native components and performances that are basically the same cannot give clues on who is who. Yet, it important to notice that many components are not present with the basic version of NativeScript – other

neither in premium one, like canvas for games – so we can always say that a native app could be more complex and manage this complexity better rather than a NativeScript app. Yet, considering how easy is the development of such an app, also thanks with our solution proposed, Mobile Studio, a developer should really ask herself if she wants to manage that complexity or can use something easier.

4.10 Building an app, step by step

In this section, we will go through the creation of a classical “Hello World” app on iOS, using Mobile Studio and NativeScript, in a step-by-step guide:

4.10.1 Project creation

Figure 51 – Step 1: welcome screen

Figure 52 - Step 2: we insert details of the app

When opening Mobile Studio, the Welcome Screen will be displayed, as we can see in Figure 51, showing the last projects where we have worked on. We select the button “Create a New app” and a transition replace the screen with a more detailed view, where we can insert the name of the app, the path where it will be stored and on which platform it will run. We select iOS and then we hit the button “Start Editor”. The Editor screen appears and a loading bar is displayed, as we can see in figure 53: the files of the project are being written into the hard drive.
When the loading is complete, the Editor Window will appear, with the Visual Editor enabled, as in figure 54.

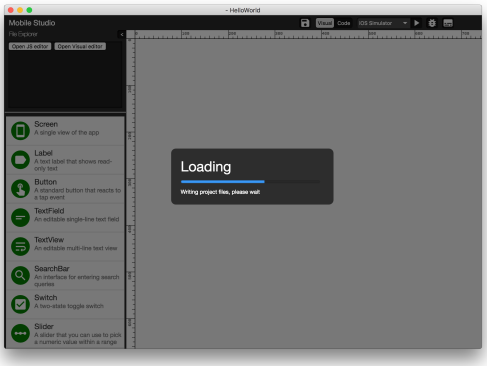


Figure 53 – Step 3: editor loading

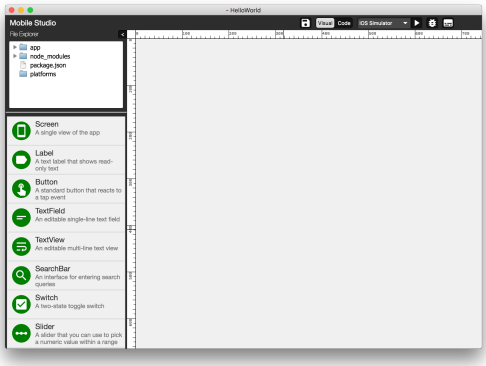


Figure 54 - Step 4: blank Visual Editor

We then drag a list element of type *Screen* from the component list on the left panel inside the Main Content. We do the same for a Label, dropping it inside the screen. We will have the step of figure 55.

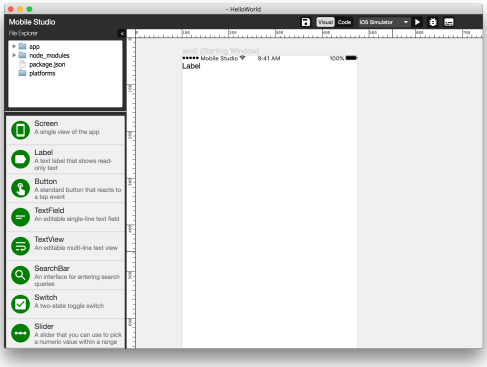


Figure 55 – Step 5: label dropped

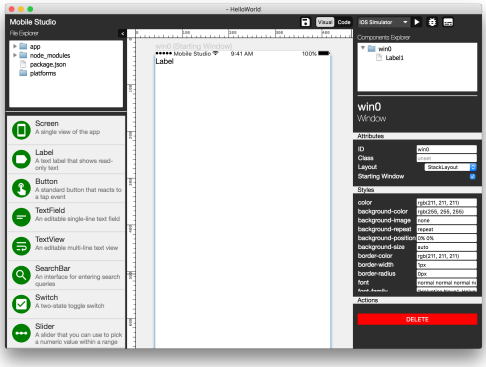


Figure 56 - Step 6: screen right panel opened

We will then click on the screen, inside the Main Content, and the right panel will appear, as in figure 56. We select, from the layout dropdown, the *AbsoluteLayout* instead of the predefined *StackLayout*. Nothing changes, but we are now free to position the label wherever we want, without any restriction. Before doing so, we can change the background of the screen, writing "*lightgray*" to the `background-color` style attribute that appear in the right panel and hit the key Enter in our keyboard. The result is shown in figure 57.

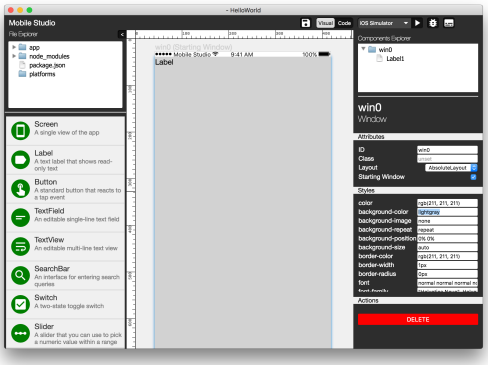


Figure 57 – Step 7: screen style modified

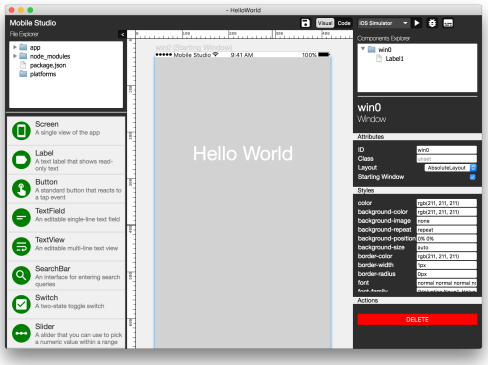


Figure 58 - Step 8: label style modified

We then move our label to the centre and clicking on it, we change its text to “Hello World”, its colour to “white” and its font size to “42px”. The result is in figure 58.

To let our application be a little more complex, we create another screen, in the same way we have done with the first, putting aside it. We will leave it white but this time we will put a button inside it, always using the drag and drop method seen for the first label. We put a button also in the first screen, putting a text of “Go to Screen 2” in the one inside the first screen and “Display alert” in the one of the second. At the end, we will have the screen show in figure 59.

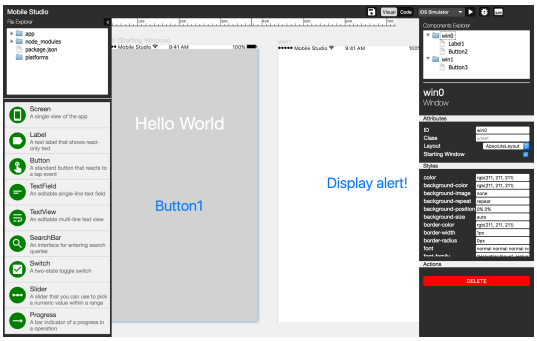


Figure 59 – Step 9: screen and buttons added

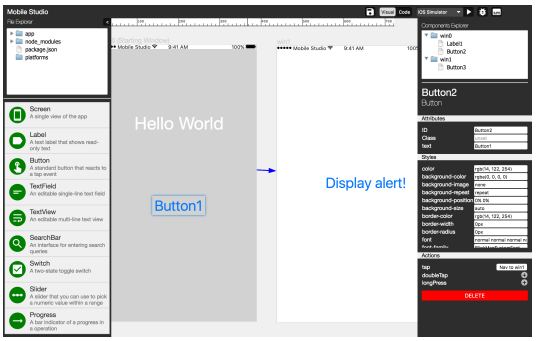


Figure 60 - Step 10: action between screens

Now, we select the button in the first screen. We click on the plus button near the “tap” label in the Actions part of the right panel and, with the blue line that appear, we click on the second screen. The action is enabled and displayed as an arrow between the screens, like in figure 60.

We now save everything with the save button in the toolbar. We check out the ID of the “Display alert!” button, that, in our case, is “Button3”. We then select `win1.js` from the File Explorer panel, inside *app/appResources* folder. The code editor will appear and display the content of the JavaScript file relative to the `win1` screen, like in figure 61.

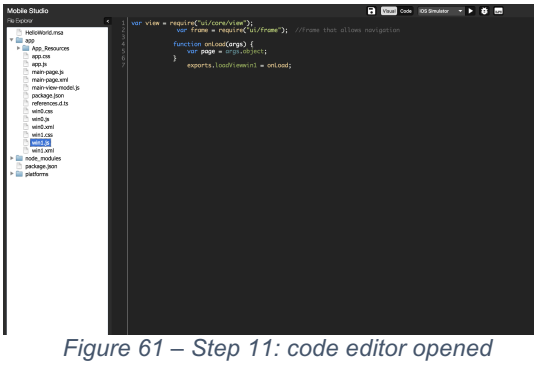


Figure 61 – Step 11: code editor opened

Figure 62 - Step 12: console displayed

We put this line of code at the very beginning of our code, that let us use the button module of NativeScript accessing to the `button` variable.

```
var button = require("ui/button");
```

And then these other lines at the end of the `onLoad` function, that will bind to our second button the event to display an alert box:

```
var btn = view.getViewById(page, "Button3");

btn.on(button.Button.tapEvent, function (args) {
   alert("Hello world!");
});
```

We hit again the save button and we are now ready to run our application, pressing the run button in the toolbar. To monitor how the compiling is going we can press the console button, the console will show up like in figure 59.

When the “Build Succeed” notification pops out, the iOS emulator will start with our application, like in figure 63. We can tap on the button on the first screen and see how we can navigate into the second screen, shown in figure 64. It is important to notice of a native iOS transition is performed and how it is possible to go back to the first screen by clicking the Back button on the top left of the screen, added automatically. Finally, if we click to the button in the second screen we can see how a native alert box appears, with the message that we have written in JavaScript, shown in figure 65.

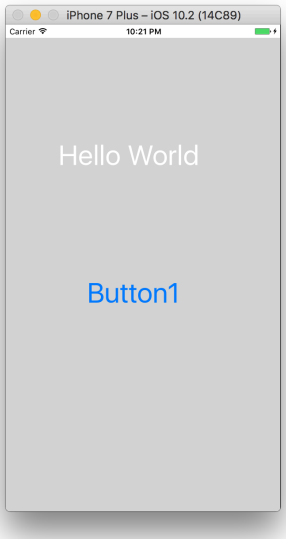


Figure 63 – Final app first screen

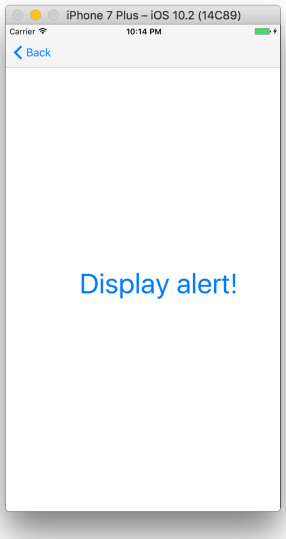


Figure 64 – Final app second screen

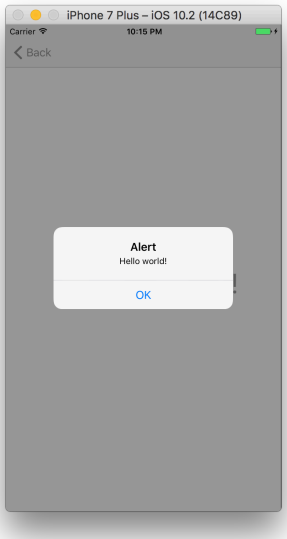


Figure 65 – Final app popup

With this incredibly simple example we hope to have given to the reader a glance on how Mobile Studio can build easily an app in a visual way and can enrich it using some code. A final remark about the code: while concepts are very similar to JavaScript we have seen, also in this little example, that NativeScript code is not perfectly identical to standard JavaScript – e.g. to access to an element knowing its id we used the function `getViewById`, and not the standard JS `getElementById`, that it's not supported. At the same time many function are the very same, e.g. the *alert()* to display a message box. It is thus important to read the documentation of NativeScript and to not be fooled by these tricky errors.

5 Conclusions

In this chapter, we will understand the limitations of our solution, what we have learnt from this experience and how we evaluate the JavaScript programming, both in desktop and in mobile environment, after having had a clear picture of the main aspects.

5.1 Issues

No software is perfect and unfortunately the one that we presented with this work is far from this perfection. Yet, it should be noticed that our intention was not to build a commercial product, but an application that helped us evaluate the JavaScript technologies that are becoming more and more popular nowadays. We will look in detail on what are the main problems that we encountered and that we weren't able to solve.

5.1.1 jQuery related issues

As we have seen extensively in the development chapter, Mobile Studio is written in JavaScript and uses the popular library jQuery quite everywhere in the code of its logic, while in the UI is frequently used jQuery UI. In the writer opinion, this is a perfect solution for the scope of this work, but it is definitely not the one to use if planning to extend this software in a commercial field. As we have already seen in jQuery specific paragraph, this library is slow and heavy and alternatives do exist – e.g. Zepto.js, Vanilla JS, Umbrella JS or directly using vanilla JavaScript.

The limitations of jQuery are more evident in its UI extension, jQuery UI. Working with that framework gave us the sensation of working on something "old", not able to cope with modern UI design pattern present in most web applications. Moreover, jQuery UI bugs heavily compromised the user experience of Mobile Studio. The most noticeable example is surely represented from the ZUI – Zoomable User Interface – that we tried to realize in the main container of the visual editor of Mobile Studio. While we were able to use pinch and pan gestures to move – in computer with multitouch trackpad, in the other using the scrolling wheel of the mouse – the logic is unable to work when components are resized, because all the measure are not updated. Unfortunately, we have discovered that this is a known issue in jQuery UI, still not solved [50].

Modern jQueryUI alternatives are not rare [51] and we think that, for future expansions of this project, they should be absolutely analysed and considered.

5.1.2 Installation needed

Mobile Studio uses many frameworks – NativeScript and Node.js are the most noticeable two, for instance. In the current implementation is left to the developer install all the dependencies that let the software run. This, if acceptable for our testing objectives, it is not in a final product that should be given to a real developer. It is important to notice that Mobile Studio does some tests in order to be able to understand if it can be executed in a particular machine – e.g. it tests if NativeScript

is present, or it alerts Windows user that they will not be able to build project for iOS – but cannot actively install the required software on user's computer. We analysed some approaches that could be used to achieve this concept, like the promising *electron-builder* package [52], but advanced concepts like *code-signing* for macOS apps, or a particular file structure for Windows, came to play and we considered all of this outside the scope of this work.

5.1.3 Android SDK location issue

Slightly related to the precedent described issue, we run into some troubles when developing an app to a real Android device. This, because the NativeScript configuration that lies behind Mobile Studio is sometimes not able to identify the correct folder of the Android SDK. An interesting solution that we thought would be to add to the menus or into the toolbar an Android SDK manager that would be also able to launch official tools of the SDK, like the Virtual Device Manager. Up to now such a view does not exist and, when a developer wants to run an app in Android platform, if the SDK is not in its default location, she should open the terminal and write the following statements:

```
export ANDROID_HOME= pathToAndroidSDK
export PATH=${PATH}:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools
```

Substituting to *pathToAndroidSDK* the real absolute path of the Android SDK. After, she has to move to the project folder and finally execute the following command:

```
tns run android
```

The app will start correctly and use that console for output.

5.1.4 Limited components support

While we tried to support the broader set of components among the ones provided by NativeScript, not all of them were easy to implement and so not all of them are supported in the version of Mobile Studio related to this work. An example of this is represented by the *ListView* component, that needs a custom structure in order to work that we considered too time-consuming to implement for this first Beta version. We instead focused on giving to the user a full set of basic components that allows her to build visually the UI of the software, and then use these advanced components with the code. It should be noticed that, in fact, thanks to the Code Editor every NativeScript component is supported: yet, a visual preview of the most advanced ones is not present, with this implementation. The original idea of this software was to give a simple way to access to new JavaScript cross-interpretation mobile programming methods, and we think that that simplicity should be achieved moving more and more user interaction from the Code Editor to the Visual one. Is something not impossible to do, in which we believe strongly.

5.2 Future Development

As we have discussed until now, many improvements should be done in order to consider Mobile Studio a full featured IDE that can be used also outside academicals studies. Yet, in order to land in this market, issues should not be the only elements to solve: many other improvement can be done. We will list the one that came into our mind to let Mobile Studio grow.

5.2.1 A new User Interface

In the writer opinion User Interfaces have paramount importance in the usability of any software. Mobile Studio is not different and we though its UI basing on concepts that can be considered “old”, like skeuomorphism. This was the first approach, also forced from the technology that didn’t allow the development of more modern ideas. As a prove of how rapidly the technologies we used are evolving, a new release of Electron has come out at the end of the development of Mobile Studio, that supported a crucial ingredient for modern user interfaces: blur, transparencies, accent colours. We have immediately though a new UI with these concepts in mind, that we show in Figure 66.

Figure 66 - Mobile Studio Welcome Screen Concept UI

As the reader can see, this approach let the software blend more naturally with the desktop environment where it runs. In the newer version of Electron this effect can be achieved simply, specifying only the attribute *vibrancy* to a window, like shown in the following code.

```
startWindow = new BrowserWindow({
    width: 800,
    height: 600,
    […]
    vibrancy: 'dark'
});
```

It is important to notice that these effects are only supported on macOS.

5.2.2 More advanced editor features

To be usable daily an IDE should have some feature that speeds up the work of the developers. Auto-completion of code, advanced copy and paste of components, code search and substitution are only few features that a developer will give for granted while installing a new IDE, but are not actually present in Mobile Studio. Always bearing in mind its analytical purpose, we let these features out of the main developing as *nice to have,* but not implemented in the version that comes with this work. Obviously, these will be mandatory features when thinking about expanding our product that should be considered.

5.2.3 Framework support

As we have seen extensively, the translator module of Mobile Studio is currently able to output only NativeScript code. We have discussed why we have chosen this framework, but we also said that a great potential of Mobile Studio is that its model is framework-agnostic, and, thus, can be translated in any other code different from the one of NativeScript: it needs only a proper translator module. In future, an interesting expansion for this work could be to support also React Native, a cross-interpretation technology that is being more and more popular nowadays.

Still thinking about NativeScript, it is important to say that Mobile Studio is not yet taking the full advantages of this approach. In the writer opinion, an interesting feature of this framework – and all cross-interpretation in general – is the ability to run an app without the need of compiling, but only doing some slightly changes to the code. This is called *livesync* in NativeScript and it is a feature that, in the writer opinion, is able to drastically cut-off development times.

5.2.4 Plugins

The idea of having a software written in a simple language like JavaScript is interesting also for its expandability. Like it has happened in Brackets, we think that Mobile Studio could be prone to be expanded by third part developers using plugins. We think that this would lead to a fast increase of quality in the product and could represent an interesting future for our small software.

5.3 Overall

With this work, we have proposed a new IDE able to use cutting-edge technologies in order to build native mobile applications. A cross-platform IDE able to output cross-platform mobile applications, always using one language: JavaScript. We analysed deeply all the approaches to build software, both in mobile and in desktop world, in 2017, we compared them and we understood that no one is perfect and each has its advantages and disadvantages. The writer hope is that the curious reader now will be able to perfectly decide which one to use in different occasions.

The writer also thinks that we have fully proved how it is possible to write complex software using simple languages, in a fraction of time that was needed with more classical approaches. In only few months, we built a simple IDE: a software usually developed by complex teams, instead Mobile Studio was written by only one student and supervised by only one professor.

This work, to the best of our knowledge, is the only one that analysed this approach in such a deep way, and the biggest hope of the writer is that it could be inspiring for other developers, that more and more are thinking on JavaScript-based approaches to build software.

We also ran numerous analysis. It can be defined funny how some of them were similar to other done on approaches that the writer used many years ago, when the cross-platform idea had ineffective solutions. The very same tests now prove that these approaches can be used really in development of apps, inheriting all the simplicity that JavaScript bares.

The final IDE is the first step in filling the gap between these innovative way of programming mobile native apps and the developers, that may find these technologies difficult to use. Mobile Studio, with its easy Visual Editor, permits this, not forgetting an advanced approach with its Code Editor.

As people of Politecnico di Milano we are proud of these results and we will never stop to let the technology be as easier to use as possible, not cutting important functionalities. At the end, “*everything should be made as simple as possible, but no simpler”* (Albert Einstein).

5.4 Resources

The interested reader can find all the resources of the project – executables for Windows and macOS, an installation guide and all the code of the project, also for the testing apps – to its dedicated home page in GitHub:

```
https://github.com/lurus92/Mobile-Studio
```

We suggest the reader to keep in touch with this page, as all the future development will be pushed there.

6 Bibliography

[1] Y. L. E. H. N. N. E.-S. Mounaim LATIF, “Cross platform approach for mobile aplication development: a survey,” in *International Conference on Information Technology for Organizations Development (IT4OD)*, 2016.

[2] Y. M. Z. T. Kai Lei, “Performance Comparison and Evaluation of Web Development Technologies in PHP, Python and Node.js,” in *IEEE 17th International Conference on Computational Science and Engineering*, 2014.

[3] S. V. Stefan Tilkov, “Node.js: Using JavaScript to Build High-Performance Network Programs,” *IEEE INTERNET COMPUTING,* no. 1089-7801/10, pp. 80-83, November/December 2010.

[4] S. P. Gregorio PEREGO, “Un Approccio Model-Driven per lo Sviluppo di Applicazioni Mobili Native,” Milano, 2012/2013.

[5] L. P. Fino, “Progettazione e implementazione di un generatore Model Driven di applicazioni per la piattaforma Android,” Milano, 2013/2014.

[6] M. Natali, “Prototipizzazione rapida per applicazioni mobili multipiattaforma,” Milano, 2013/2014.

[7] M. P. Singh, “Evolution of Processor Architecture in Mobile Phones,” *International Journal of Computer Applications,* vol. 90, no. 4, pp. 34-39, 2014 March.

[8] Fortune, “Apple Has Paid Almost $50 Billion to App Developers,” 13 June 2016. [Online]. Available: http://fortune.com/2016/06/13/apple-has-paid-almost-50-billion-to-app-developers/.

[9] The jQuery Foundation, “jQuery Demos,” [Online]. Available: http://demos.jquerymobile.com/1.4.5/button/.

[10] D. Crawford, “Mobile web apps are slow,” 6 May 2013. [Online]. Available: http://sealedabstract.com/rants/mobile-web-apps-are-slow/. [Accessed 13 February 2017].

[11] D. Crawford, “Why mobile web apps are slow,” 9 July 2013. [Online]. Available: http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/. [Accessed 13 February 2017].

[12] The Apache Software Foundation, “Cordova Architecture Overview,” [Online]. Available: https://cordova.apache.org/docs/en/latest/guide/overview/.

[13] Microsoft, “Measure the performance of a Cordova app,” 10 August 2015. [Online]. Available: https://taco.visualstudio.com/en-us/docs/measure-performance/.

[14] Microsoft, “Evaluate the performance costs of a Cordova app,” 10 8 2015. [Online]. Available: https://taco.visualstudio.com/en-us/docs/cost-cordova/.

[15] Xamarin Inc. (Microsoft), “Understanding the Xamarin Mobile Platform,” [Online]. Available: https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_1_-_understanding_the_xamarin_mobile_platform/.

[16] Wikipedia, “Cross compiler,” [Online]. Available: https://en.wikipedia.org/wiki/Cross_compiler.

[17] Xamarin Inc. (Microsoft), “Architecture,” [Online]. Available: https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_2_-_architecture/.

[18] Facebook Inc., “Tutorial: Intro To React,” [Online]. Available: https://facebook.github.io/react/tutorial/tutorial.html#storing-a-history.

[19] Facebook Inc., “Native Modules,” [Online]. Available: https://facebook.github.io/react-native/docs/native-modules-ios.html.

[20] NativeScript, “UI for NativeScript,” [Online]. Available: http://www.telerik.com/nativescript-ui.

[21] NativeScript, “NativeScript Documentation,” [Online]. Available: https://docs.nativescript.org/.

[22] NativeScript, “NativeScript with Angular,” [Online]. Available: https://docs.nativescript.org/angular/start/introduction.html.

[23] NativeScript, “Styling,” [Online]. Available: https://docs.nativescript.org/ui/styling.

[24] NativeScript, “User Interface Layouts,” [Online]. Available: https://docs.nativescript.org/ui/layouts.

[25] T. VanToll, “How NativeScript Works,” 16 February 2015. [Online]. Available: http://developer.telerik.com/featured/nativescript-works/.

[26] V. Stoychev, “What are the key difference between ReactNative and NativeScript?,” 11 May 2016. [Online]. Available: https://www.quora.com/What-are-the-key-difference-between-ReactNative-and-NativeScript. [Accessed 30 January 2017].

[27] Microsoft, “JScript,” [Online]. Available: https://msdn.microsoft.com/en-us/library/72bd815a(v=vs.100).aspx. [Accessed 30 January 2017].

[28] Node.js, “Node.js v7.4.0 Documentation,” [Online]. Available: https://nodejs.org/api/. [Accessed 30 January 2017].

[29] B. San Souci and M. Lemaire, “An Inside Look at the Architecture of NodeJS,” McGill University.

[30] The Chromium Projects, “How Chromium Displays Web Pages,” [Online]. Available: http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome. [Accessed 30 January 2017].

[31] The Chromium Projects, “Multi-process Architecture,” [Online]. Available: https://www.chromium.org/developers/design-documents/multi-process-architecture. [Accessed 30 January 2017].

[32] Adobe, “Brackets,” Adobe, [Online]. Available: http://brackets.io/. [Accessed 23 March 2017].

[33] Adobe, “Architectural Overview,” [Online]. Available: https://github.com/adobe/brackets-shell/wiki/Architectural-Overview. [Accessed 23 March 2017].

[34] Adobe, “Brackets Shell,” [Online]. Available: https://github.com/adobe/brackets-shell. [Accessed 23 March 2017].

[35] Adobe, “Building Brackets Shell,” [Online]. Available: https://github.com/adobe/brackets-shell/wiki/Building-brackets-shell. [Accessed 23 March 2017].

[36] Android, “Command Line Tools,” [Online]. Available: https://developer.android.com/studio/command-line/index.html. [Accessed 19 February 2017].

[37] NBAndroid, “NBAndroid Home Page,” [Online]. Available: http://nbandroid.org. [Accessed 19 February 2017].

[38] Android, “Everything you need to build on Android,” [Online]. Available: https://developer.android.com/studio/features.html. [Accessed 2017 February 2017].

[39] MacRumors, “Apple Releases iPhone SDK, Demos Spore, Instant Messaging,” 2008 March 06. [Online]. Available: https://www.macrumors.com/2008/03/06/apple-releases-iphone-sdk-demos-spore-instant-messaging/. [Accessed 2017 February 19].

[40] Apple Inc., The Swift Programming Language, Apple Inc., 2014.

[41] Quora, “Should I use Swift or Objective-C to learn iOS development?,” 21 September 2016. [Online]. Available: https://www.quora.com/Should-I-use-Swift-or-Objective-C-to-learn-iOS-development. [Accessed 19 February 2017].

[42] J. M. A. Santamaria, “The Single Page Interface Manifesto,” 21 September 2015. [Online]. Available: http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php. [Accessed 27 March 2017].

[43] jQuery, “jQuery API Documentation,” [Online]. Available: http://api.jquery.com/jquery/. [Accessed 26 February 2017].

[44] jsPerf - JavaScript performance playground, “jQuery vs JavaScript Performance Comparison,” [Online]. Available: https://jsperf.com/jquery-vs-javascript-performance-comparison/22. [Accessed 26 February 2017].

[45] j. -. J. p. playground, “jQuery vs Native Element Performance,” [Online]. Available: https://jsperf.com/jquery-vs-native-element-performance. [Accessed 26 February 2017].

[46] ZeptoJS, “ZeptoJS Introduction,” [Online]. Available: http://zeptojs.com. [Accessed 26 February 2017].

[47] The jQuery Fundation, “jQuery UI Home,” [Online]. [Accessed 26 February 2017].

[48] Bootstrap, “Getting Started,” [Online]. Available: http://getbootstrap.com/getting-started/. [Accessed 26 February 2017].

[49] CodeMirror, “CodeMirror Manual,” [Online]. Available: https://codemirror.net/doc/manual.html. [Accessed 26 February 2017].

[50] jQueryUI, “jQuery doesn't take into account CSS scale transform when getting height/width of element,” 20 December 2011. [Online]. Available: https://bugs.jquery.com/ticket/11114. [Accessed 2 April 2017].

[51] S. Codrington, “Top 5 jQuery UI Alternatives,” 15 March 2017. [Online]. Available: https://www.sitepoint.com/top-5-jquery-ui-alternatives/. [Accessed 2 April 2017].

[52] GitHub, “electron-builder,” [Online]. Available: https://github.com/electron-userland/electron-builder. [Accessed 2 April 2017].

[53] Electron, “Adds vibrancy effect for macos,” 11 November 2016. [Online]. Available: https://github.com/electron/electron/pull/7898. [Accessed 2 April 2017].