MASTER THESIS

TITLE: VacApp: Mobile tool for the child vaccination control

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Rubén Gómez Pérez

DIRECTOR: Juan López Rubio

DATE: November 26th 2014
Overview

Following the prediction of eMarketer, an independent market research company from New York whose clients include two-thirds of the 500 biggest companies in the United States, in 2014 the total number smartphone users would rise to 1.75 billion; and, following the figures of past years, this number will continue to increase in the future.

We have arrived at a point in which having a smartphone is a common thing, and with them, the usage of mobile applications has become a regular thing for consumers. This popularization of smartphones and mobile applications opens a way for different areas to expand through it, and health is one of them.

Combined with the possibilities offered by smartphones, there are many applications regarding health that reach lots of people. However, there are places where smartphones cannot be afforded and where humanitarian help is needed.

These workers could take advantage on something as usual as a smartphone, even their own ones, to keep record of their actuations easily using a mobile application; and, in the same way, that application could be used by people in richer areas to keep an eye on their own health matters.

Keeping in mind these possibilities, an important matter in both scenarios is the child vaccination. To control these vaccines only using a smartphone can be very handy, and if there was the possibility of saving this information in a server to preserve it, users could have a complete tool for controlling children’s vaccines.

This document collects the essential information about the different stages of the development of VacApp, a mobile application that meets these needs.
This thesis is dedicated to my partner Sara for all her support, help and motivation during this project; to my parents for giving me the opportunity to earn a master’s degree and prepare myself for the future; to my project director Juan for teaming up with me once again; and to Alex and Gerard for all their help in the development.
# INDEX

INTRODUCTION .................................................................................................................. 1

CHAPTER 1. OBJECTIVES ................................................................................................. 3

1.1 Context .......................................................................................................................... 3
   1.1.1 e-health .................................................................................................................... 3
   1.1.2 Mercy Corps ............................................................................................................ 3
   1.1.3 aaaida ...................................................................................................................... 4

1.2 Objectives of the project ............................................................................................. 4

CHAPTER 2. SPECIFICATIONS ......................................................................................... 7

2.1 Login screen ................................................................................................................... 7
2.2 Registration screen ........................................................................................................ 8
2.3 Add and see entries ....................................................................................................... 8
2.4 Settings screen ............................................................................................................. 10
2.5 Synchronize .................................................................................................................. 10

CHAPTER 3. ARCHITECTURE AND DESIGN ............................................................... 13

3.1 Technologies ................................................................................................................. 13
   3.1.1 Telerik AppBuilder ............................................................................................... 13
   3.1.2 Local storage .......................................................................................................... 14
   3.1.3 Node.js .................................................................................................................. 15
   3.1.4 Express .................................................................................................................. 15
   3.1.5 MongoDB ............................................................................................................ 15
   3.1.6 Mongoose .............................................................................................................. 16
   3.1.7 Route Injector ....................................................................................................... 16

3.2 Architecture .................................................................................................................. 17

3.3 Design .......................................................................................................................... 24
   3.3.1 Models ................................................................................................................... 24
CHAPTER 4. IMPLEMENTATION ................................................................. 29

4.1 HTML ........................................................................................................... 29
  4.1.1 Layouts .................................................................................................. 30

4.2 CSS .............................................................................................................. 31
  4.2.1 Conflicts with the Kendo UI CSS file ...................................................... 32
  4.2.2 Color and font ......................................................................................... 32
  4.2.3 Icons ..................................................................................................... 33

4.3 Application icon .......................................................................................... 34

4.4 Comparison with specifications ................................................................... 35
  4.4.1 Login screen ........................................................................................... 35
  4.4.2 Synchronize ............................................................................................ 36

4.5 Improvements .............................................................................................. 37

CHAPTER 5. CONCLUSIONS .......................................................................... 43

5.1 Environmental study .................................................................................... 43

5.2 Conclusions .................................................................................................. 44

5.3 Future guidelines .......................................................................................... 45

BIBLIOGRAPHY .............................................................................................. 47

ANNEX .............................................................................................................. 49
FIGURE INDEX

Fig. 2.1. Login process mockup................................................................. 7
Fig. 2.2. Registration process mockup. ..................................................... 8
Fig. 2.3. Adding entries and detailed view mockup. ................................. 9
Fig. 2.4. Settings screen mockup. ............................................................ 10
Fig. 2.5. Synchronize mockup. ............................................................... 11
Fig. 3.1. User edition screen from the Route Injector back office. .......... 17
Fig. 3.2. Prescription edition screen from the Route Injector back office. .... 17
Fig. 3.3. Architecture of the overall project. .......................................... 18
Fig. 3.4. Application diagram............................................................... 18
Fig. 3.5. Login sequence diagram ....................................................... 19
Fig. 3.6. Registration sequence diagram.............................................. 20
Fig. 3.7. Entry creation sequence diagram........................................... 21
Fig. 3.8. Detailed entry view sequence diagram.................................... 21
Fig. 3.9. Element edition sequence diagram. ....................................... 22
Fig. 3.10. Screens seen through the synchronization process. .............. 23
Fig. 3.11. User schema. ................................................................. 24
Fig. 3.12. Vaccine schema. ............................................................... 25
Fig. 3.13. Prescription schema. ........................................................... 26
Fig. 3.14. PHR schema. ................................................................. 27
Fig. 4.1. Layout of a page using semantic elements from HTML 5......... 30
Fig. 4.2. Application icons (left: Android, right: iOS). .......................... 35
Fig. 4.3. Login screen........................................................................ 36
Fig. 4.4. Screens seen through the synchronization process. .............. 36
Fig. 4.5. Actual record tab compared to chapter 2 PHRs tab. .............. 38
Fig. 4.6. Actual detailed views compared to chapter 2 specifications.......... 39
Fig. 4.7. Detailed views for vaccines and prescriptions in the record tab. ....... 40
Fig. 4.8. Vaccine creation view. ......................................................................... 41
Fig. 4.9. Settings view. .......................................................................................... 42
INTRODUCTION

Following the prediction of eMarketer, an independent market research company from New York whose clients include two-thirds of the 500 biggest companies in the United States, in 2014 the total number smartphone users would rise to 1,75 billion; and, following the figures of past years, this number will continue to increase in the future.

We have arrived at a point in which having a smartphone is a common thing, and with them, the usage of mobile applications has become a regular thing for consumers. This popularization of smartphones and mobile applications opens a way for different areas to expand through it, and health is one of them.

Combined with the possibilities offered by smartphones, there are many applications regarding health that reach lots of people. However, there are places where smartphones cannot be afforded and where humanitarian help is needed.

These workers could take advantage on something as usual as a smartphone, even their own ones, to keep record of their actuations easily using a mobile application; and, in the same way, that application could be used by people in richer areas to keep an eye on their own health matters.

Keeping in mind these possibilities, an important matter in both scenarios is the child vaccination. To control these vaccines only using a smartphone can be very handy, and if there was the possibility of saving this information in a server to preserve it, users could have a complete tool for controlling children’s vaccines.

This document collects the essential information about the different stages of the development of VacApp, a mobile application that meets these needs, and is divided following these phases.

The first chapter gives some context for the project and lists its objectives that should be fulfilled to consider the app finished.

The second chapter contains the specification of the app, including the different functionalities it has and mockups to define the looks of the app.

The third chapter talks about the architecture and the design of the app. It includes explanations about all the technologies involved in the project and the internal structure of the application.

In the fourth chapter we can find all the data about the implementation phase of the project: how the specifications have been applied, what has changed since then and how it changes or adds new functionalities to the project and general results of the app once it is finished.
Finally, the fifth chapter contains some personal conclusions about the project, along with possible improvements on the application and future guidelines for revisions. In addition, it also has a brief environmental study about the project.

After this chapter, a bibliography is included, containing books and websites that have been consulted during the development of the project.
CHAPTER 1. OBJECTIVES

Before explaining all the phases of the development of this project, it would be interesting to see the motivations, the context of the idea and what are the objectives that should be accomplished at the end of the project.

1.1 Context

1.1.1 e-health

The e-health is a very recent term (estimated to appear in the 1999) that refers to the healthcare practice that is complemented by the using of electronic processes and communications. Among its characteristics and objectives we can find an increase in the efficiency of actual health care, the empowerment of patients over their choices or a change in the relationship between the consumers and the health professionals, pointing it to a much more close relation. [1]

Thanks to the smartphone evolution and the actual scenario, where anybody can afford one of them and their use has spread widely, e-health has found a very good way to offer its benefits to the people, in the form of mobile applications that can be used on a daily basis to improve the control we have over our health and that of our related ones.

1.1.2 Mercy Corps

During the Mobile World Congress 2014, in the context of the WIPJam Hackathon, Intel hosted a workshop called “Solve Social Problems with Code through the Intel “Code for Good” Program”, which consisted on explaining that initiative, which aims to gather developers and promote social purposes for solving problems through software. This session also involved one of the chiefs of Mercy Corps, who talked about the projects of the agency and how they collaborate with Intel to develop them.

Mercy Corps focuses on environments under several difficulties, such as natural disasters, economical problems or armed conflicts. These targets arose during the speech when talking about the difficulties of maintaining a complete medical record in some Third World countries. If we add this inconvenient to the fact that in those places there is nearly no data coverage, not only keeping a record is difficult, sharing it with specialists on developed countries too.

With these handicaps in mind, and as an idea for the Hackathon, the concept of VacApp emerged: an easy to use application that keeps a record of taken and
pending vaccines, medical prescriptions and past interventions that keeps all that information in the phone memory, letting the user back it up whenever there is an internet connection available. Thanks to this app, specialists and volunteers operating in isolated areas could control the vaccination process easily and carry that information with them all the time.

1.1.3 aaaida

The other motivational aspect for the development of this project has been the existence of aaaida, an e-Health oriented social network developed by Alteraid, a spin-off of the Universitat Politècnica de Catalunya (from now on, UPC).

aaaida is not a social network like Facebook or Twitter, where users post their thoughts and share contents. It aims to offer an innovative way for families to look after their elder relatives in a non-intrusive way. In fact, not only the family is able to do so, but also friends, caregivers and doctors associated with that person, in order to give them the best treatment possible.

Taking this into account, there are other members in the families that need a special attention besides elderly, and those are the children. With this vaccine-record application linked to aaaida, people could watch an instant history about the vaccinations taken by their children, prescriptions given at that moment for treating any side effects and even personal records notated by their doctors. And linking this situation with the one mentioned before, international associations could take advantage on the possibilities offered by aaaida and this app in order to keep track of the results of their humanitarian missions.

1.2 Objectives of the project

So, taking in consideration the aforementioned concepts and ideas, the app to be developed should offer the following options:

- Create a mobile application compatible with the different mobile operating systems in the market (Android, iOS, Windows Phone).
- Add vaccines to a list for keeping a record on pending appointments.
- Add prescriptions for medication that may have to be taken, including information such as the treatment finalization date, side notes or the dose to be taken.
- Add Personal History Records (from now on, PHR) with notes, observations or advices from the user’s doctor.
- Let the user edit and delete these entries, so the lists are up to date.
- Provide a server facility, so all changes can be backed up when a connection is available.

These are basic points for the app to be completed, but as it can be seen in future chapters, there is room for improvement and new additions.
VacApp: mobile tool for the child vaccination control
Now that the main objectives have been settled, we can start designing the mobile application. The first step consists in giving the app a structure, deciding which are going to be the different screens that will form it and what is needed.

By doing so, the structure of the different pages is also specified, and serves as a base for the final stage of implementation.

2.1 Login screen

The first scheme shows the login process and all the screens involved in it:

As it can be seen, when the app loads, there are two possibilities depending on the connectivity of the device: if there is any kind of connection (wi-fi or cellular) and/or the user has not logged up yet, the login screen will appear, where the user can use its existing account or register if they do not have one.
2.2 Registration screen

In the case that the user has never logged in, they can register using their e-mail and entering a password (in fact, they have to do so in order to use the app, since all the data they create and change in the app should be uploaded to the server at one time or another). The following mockup shows the registration process:

![Registration process mockup](image)

The registration and login screens are very similar, they both have inputs for the e-mail to register and a password. After the user has registered, they are automatically logged in, and their credentials are stored for future occasions.

2.3 Add and see entries

This is the main action that the users will be doing: adding entries and seeing their details, editing or deleting them. Not all of them have the same fields (although they share some like a title or the date), which means that their creation screens must be separated and will have to be coded separately instead of defining one for all of them.

The following mockup shows the screens seen when a user adds a vaccine, a prescription or a PHR (process depicted using a solid arrow) and when, once they are created, they select them (the process depicted with a dashed arrow):
The first column shows the different tabs, with their lists of already created elements. When the user selects the first item of each one, the screens on the second column appear. Once the entry required fields are filled and the user submits it, they go back to the screen they were, but with the list updated with the last entry they created. If they select one of them, then the app shows the corresponding screen of the third column, where all the details of a vaccine, prescription or PHR are shown. In that view, the users can edit the fields, delete the entry or just go back.
2.4 Settings screen

The following figure shows the structure of the settings screen, accessible by touching the gear icon at the top left of the screen:

![Settings screen mockup](image)

*Fig. 2.4. Settings screen mockup.*

The settings screen shows the current user logged in and has 2 editable options: a check box to ask or not for confirmation when the user deletes an entry and a selector with the language of the app.

This page can only been accessed through the main view tabs, not when the user is creating an entry or viewing its details. It has been done this way to keep the screens simple and do not distract the user from the creation or edition of an entry.

2.5 Synchronize

There is another icon that can be found in the main screen, right on the opposite side of the settings one, and it is the synchronize button, which uploads any generated data to the server. This last mockup shows the synchronizing process, which in fact does not have a persistent screen:
As in the settings case, the synchronize icon can only be found in the main screen, and has a cloud to give the user the idea of cloud storage. When it is selected, any additions, editions and deletions done in the vaccine, prescription or PHR tabs are uploaded to the server. Since the user does not have to do anything more to do so, while the data is being sent the screen shows a loading animation, which will disappear once that all the changes have been backed up. After that, the screen will appear just as before.

In broad terms, these mockups summarize the contents of the different screens and how some of them are connected through the actions of the user, such as creating or deleting entries.

The use of these schemes can very helpful for later development stages, since we have a much more defined aspect for the app and it is easier to locate the elements on the screen instead of trying different options writing code directly.

These screen structures, however, should not be taken as final. During the implementation phase of the project some improvements could be done or even decisions taken now can appear to be counterproductive and need to be changed. All of this will be seen in the later chapter of implementation, where
the results of the development are shown and changes done from these basic guidelines are discussed and compared with the final decisions.
CHAPTER 3. ARCHITECTURE AND DESIGN

With the mockups done and the structure of the different pages stated, we should specify the design guidelines and the architecture of the mobile application. This chapter will show the technologies used in the project and the reasons behind those elections, in addition to the design of the different parts involved in the app.

3.1 Technologies

3.1.1 Telerik AppBuilder

One of the goals of the project is to design a mobile application suitable for all the operative systems (Android, iOS and Windows Phone) without having to code each version separately, because then each one should be done in a different programming language. To do so, it was decided to use the platform known as AppBuilder. [5]

AppBuilder (formerly Icenium) is a developing platform launched by Telerik, a company known for providing different solutions aiming to improve the development of software, such as UI controls, analysis and correction of bugs, generation of reports or even the preparation of tests for knowing the performance of an app in different environments (browsers, devices...).

The main point on the decision of using this framework is its use of HTML 5, JavaScript and CSS (Cascading Styles Sheets) as languages. As in the case of web development, the different screens are coded as web pages, stating their components through HTML, giving them a visual style using CSS and with the functions implemented using JavaScript. [2]

AppBuilder also offers Kendo UI, a framework developed from scratch by with the jQuery library as its base. It provides a collection of widgets, themes, effects and many more features to enrich the web (and so, all the mobile apps developed with AppBuilder). In AppBuilder, the user can start an empty project or a project based on Kendo UI. [8]

By doing this, it will start with a CSS style sheet that will apply the main theme as the visual style of the app. A good point about some themes of this framework is that they replicate the native looks of the different mobile operative systems, helping to give an overall aspect more integrated with the device.

Another point in favor of AppBuilder is the Apache Cordova plugin library. Since it is based on Apache Cordova to access the native functionalities of the mobile
devices, these plugins can be added to the project in order to offer extra features or to achieve an easy access to those native functionalities. [9]

And finally, but not least important, AppBuilder is cloud-based. There are several IDE options for the users to choose (like browser clients to code at any computer, Windows clients, Visual Studio extensions or even a command-line interface) and all the projects are stored on their servers, so the developer can work on its project at any place where they have access to a computer, only by logging in one of the clients.

Another objective is to keep the changes done in the app locally and give the user the opportunity to synchronize its app with a server whenever they have data connectivity, so their information is stored in the remote machine as a backup. To do this, there is a need of a server that takes the requests from the app and a database that stores all the information.

### 3.1.2 Local storage

Since the app uses HTML 5, there is a very useful feature new to this last version of the language suited for keeping changes locally until they are pushed to the server, and it is the local storage. [6]

In past versions of HTML, additional information on websites had to be stored in cookies, and these had to be included in every request sent to a server. The local storage, however, has a larger size limit (up to 5MB), and that information is never sent to the server when a request is done unless the developer loads it and attaches it on purpose.

The local storage object (referenced as `window.localStorage`) is very easy to use: information is saved in a key/value pair, and the user can set or get the value of a key with the lines `localStorage.setItem("key_name","value")` and `localStorage.getItem("key_name")`.

Since the stored data has no expiration date, the deletion of items in the `localStorage` object must be done programmatically (using `localStorage.removeItem("key_name")`). And in the case that the user tries to store 2 or more values using the same key, there will be no duplicates, just one key with the latest value given, since they are overwritten.

In the case of this project, the `localStorage` object will come in handy because the entry additions, deletions and changes can be stored there until they are uploaded, and since that information will have no expiration date, the user can exit the application without fear of losing those changes and send them to the server any time they want.
3.1.3 Node.js

Node.js is a framework based on JavaScript (using the V8 JavaScript engine, developed by Google) used to build network solutions with a high degree of scalability. It runs on the server layer and uses an event-driven, non-blocking I/O model, which means that the code continues running while requests are answered (they are taken one by one on a single thread). The main advantage of this method is that the CPU keeps doing operations and does not leave the thread idle, waiting for a response and occupying memory. [4]

This means that Node.js is perfect for taking a very large number of requests, but is not suitable for heavy computations. That would saturate the single thread and would cause problems to all the other users because their requests would remain blocked until all the computation is done.

A very useful tool provided by Node.js is NPM, or Node Package Manager, which allows the users to install components in a very easy way for Node.js from a wide repository. Users can specify the modules that their project requires in the file package.json, and by running the npm install command, all of them will be downloaded and installed without problem.

Node.js has become a very popular choice among the developer community, and it is a very good option to consider for the server side, since the operations needed for the mobile application do not need a very intensive use of the CPU and it would allow a very high number of simultaneous users to use it.

3.1.4 Express

Express is one of the most famous frameworks for Node.js and it is used to develop web applications. It is minimalist, flexible, robust and very easy to use, reasons that make it a very common choice among developers that use Node.js.

Express is inspired by Sinatra, a popular lightweight framework for Ruby, and it offers, among other features, complete routing management of HTTP requests, support for template engines or middleware usage. It can be installed using the aforementioned NPM tool from Node.js, which makes it extremely easy to integrate in the project.

3.1.5 MongoDB

When selecting a technology for the database, there is a commonly chosen one if Node.js is also being used, and that is MongoDB. The main reason is that both of them use JavaScript and JSON format, which makes it more or less easy to create a simple Node.js server with a MongoDB database.
MongoDB is the most known NoSQL database, a movement that tries to offer a modern look on the classical relational databases and their shortcomings. NoSQL databases can manage very large quantities of data (in fact, the MongoDB name comes from “humongous”) and offers some improvements over relational databases, such as no bottlenecks or horizontal scalability.

3.1.6 Mongoose

The use of both Node.js and MongoDB for the creation of a server and a database is commonly followed by the addition of a Node.js library known as Mongoose, which offers object data modeling (ODM) for our database.

In a few words, thanks to Mongoose, the database information is modeled into objects using previously created schemas and models. Schemas are just prototypes of what these objects should look like (in the case of a vaccine, for example, we would need to specify that they have a string field for the title, the date and the type of entry for internal recognition in the mobile app), and with that schema as a base, then we can create instances of the model Vaccine.

Thanks to this modeling, it is very easy to query all the documents created using a specific model and retrieve objects that the mobile application can directly work with, and the following module also requires it.

3.1.7 Route Injector

Route Injector is a module for Node.js developed by Juan López (professor at EETAC and co-founder of Alteraid), Álex Albalá and Gerard Solé (alumni from EETAC and developers at Alteraid). Its purpose is to create REST APIs from Mongoose models.

The module “injects” the different routes into the Express server (GET, POST, PUT and DELETE), so by just creating the needed models (in this case, Vaccine, Prescription and PHR) and thanks to Route Injector, we can add, change or delete these entries from the database, and even get all the vaccines, prescriptions or PHRs, since Route Injector determines the plural of the models and can work with the entire collections of the database.

Route Injector also provides a back office for the administrator of the server to manipulate the contents of the database in a very easy and visual way. Through the side menu we can select the collection of the database to show and then select an item to edit or delete it (in addition to create new entries).

In the user menu (this is, when a user is selected to be edited) we can assign to each one of them a vaccine, prescription or PHR already created, and create a new one directly from there, and rearrange the entries they have assigned with a very useful drag-and-drop system.
The following figures show some screens of the back office and how it looks:

Fig. 3.1. User edition screen from the Route Injector back office.

Fig. 3.2. Prescription edition screen from the Route Injector back office.

3.2 Architecture

This paragraph shows how all the technologies explained before combine, and serves to get a glimpse of the structure of the project. The next figure contains a diagram showing its different parts and how they are connected:
As it can be seen in the figure, the app (developed using HTML 5 and Kendo UI) connects through the Internet with the server (based on Node.js and that uses, among other modules, Express and Route Injector). It will communicate with the MongoDB database to store the entries sent from the app’s local storage, and as said before, thanks to the Route Injector back office we could edit, delete or even create new ones, in addition to register new users.

Taking into account the specifications from the past chapter we can create a diagram of the application and see what are their components. As the app is developed using HTML for defining the visual structure and JavaScript for the functions, the diagram can be divided in those different type of files and show their contents for a better understanding:

![Diagram of the application](image-url)

**Fig. 3.3. Architecture of the overall project.**

**Fig. 3.4. Application diagram.**
In that diagram, the arrows show which methods are called when something is done in one of the screens defined at the index.html file. The methods that do not have an arrow pointing them are called within other methods. Attending to the previous figure, these are the main functions of the application:

- **function(global):** this function is called right at the beginning, when the app is initialized. It is already declared when a project using Kendo UI is created in AppBuilder, and it can be very useful for setting values or initializing plugins before the user starts using the app.

- **login():** it will take the information entered by the user and send it to the server to check if they are registered. If so, the user will be able to continue to the main menu of the app and begin its usage. The following figure shows the communication sequence between the app and the server for this function:

![Login sequence diagram](image)

Fig. 3.5. Login sequence diagram.

If the server verifications end in success, the server response will contain the user’s entries that are already backed up, along with their user ID, which is needed for some requests like editing entries, and the user will be allowed to continue to the main menu and use the application.

In the case that the verification has encountered an error because some or all the user information is wrong or the user does not exist, the app will warn the user what is the problem thanks to the status code sent along with the server response.
- register(): the Route Injector back office lets us create new users very easily, but of course, normal users will not have access to it, so there has to be a way for them to register by their own. This function will take entered information and create a user instance in the database, and then the app will log in normally. The following diagram shows the sequence of this function:

![Registration sequence diagram](image)

In the case that the user already exists, the user will be warned about it, so there is no danger of creating duplicated users. On the other hand, if the user does not exist, it will be stored in the database and then the process will follow the same way as the login one, but of course, there will be no entries to be retrieved since it is a new user.

- addNew*(): there are 3 different methods for adding new entries, one for each type (vaccine, prescription or PHR), since each one has their own information content.

The process, however, is the same for all of them: by touching the “Add new...” entry in any of the tabs, the information for the new entry is collected from the specific page for that type of entry and used to create a JSON object. Then, this object is stored in the local storage and in an array for easier manipulation of that information (so the app does not have to load the localStorage content every time that is needed) along with a POST request that will be sent to the server when the user presses the synchronize button, saving it into the database.
The following diagram shows the sequence followed during the execution of this function, which can be applied to vaccines, prescriptions and PHRs:

![Diagram showing sequence of events]

- `viewDetailed*()`: as before, there is a method like this for each type of entry. It is called when a vaccine, prescription or PHR entry is selected, and it calls a new page that shows all the information about that element.

Except in the vaccine case, prescriptions and PHRs obtain information that is not showed in the corresponding item on the tabs (or maybe it is showed, but gets truncated due to the screen size), and thanks to this function the user can access the entire data of that element.

![Diagram showing detailed view]

**Fig. 3.7. Entry creation sequence diagram.**

**Fig. 3.8. Detailed entry view sequence diagram.**
• saveElementChanges(): in the previous case, when all the information of an entry is displayed, the user can edit it and save the changes in case that they made a mistake or, for example, a vaccine appointment got rearranged.

So, when a change is done in the detailed view, this function will be called and the new information will be stored, and instead of a POST request, it will create a PUT one with the new entry and its ID, so instead of aggregating a new item to the database it will be changed. The following figure shows the sequence of this function, which is very similar to the one in the addNew*() functions, but with the difference that first the previously created entry is loaded in a specific view:

![Sequence diagram](image)

**Fig. 3.9. Element edition sequence diagram.**

• update*List(): these functions are the ones that maintain the tabs updated whenever an entry is created. If, for example, a vaccine is added, this function will be called to take all the vaccines in the local storage, will create a div element for each one with their respective data and then they will be inserted in the list of the vaccines tab.

This process is the same for vaccines, prescriptions and PHRs, but of course, only the corresponding one is called upon the creation of an entry.

• saveSettings(): as it was defined in the chapter 2, the settings screen has 2 editable options regarding the language of the app and the
confirmation asked before deleting an entry. When the user touches the return button, the current state of those options will be stored.

- `deleteElement()`: when the user is looking at the detailed view of an entry, they have the option to delete it. By doing this, that entry will be deleted from the local storage and a DELETE request will also be stored for the later process of synchronization.

The sequence followed in this case is the same as when editing an entry, with the difference that instead of having the same array of entries with one changed, that one is deleted and it is not available anymore.

- `saveToLocalStorage()`: this function is called by most of the processes in the app. Basically, it will save a given object or variable in the localStorage object. This includes new entries, modified entries, deleted entries (they are saved in an array, the deleted one gets pushed and then that array is saved in the memory), pending requests and the app settings.

The sequence of this function can be seen in previous figures where data is stored in the localStorage object.

- `synchronize()`: when the user is in the main menu, they can call this function to upload to the server all the operations they have done in the app (this is, creating vaccines, prescriptions and/or PHRs, editing or deleting them).

Although changes and creations are not uploaded instantly, they are stored locally all the time, so it is not mandatory to call this function every time to manage the entries. The following figure illustrates the sequence followed by this function:

![Diagram](image)

Fig. 3.10. Screens seen through the synchronization process.
When all the requests have been handled, the application warns the user about it. This function, however, cannot be called if there is no internet connection, and to avoid errors the communications will not begin in that case; instead, the app will tell the user that there is no available connection at the moment.

### 3.3 Design

This section will show how the different aspects of the application have been designed and how they interact with the previously explained architecture elements.

#### 3.3.1 Models

The models, as explained in this chapter's section about Mongoose and Route Injector, are used to store in a database objects with a determined structure, based on a schema. Both server and application work with these models in their own way, using the information they contain. The models we can find in the project are the following:

- **User**: the mobile application does not use a “user” object internally, but of course needs to work with its components. The following figure shows the components of the User model, following the schema specified for Mongoose:

  ![User schema](image)

  **Fig. 3.11. User schema.**

  The user ID is assigned to the user when they are registered, and as explained before, is used to save changes to its properties. The email and password fields are used for the login function, and the remaining components (myVaccines, myPrescriptions and myPhrs) are arrays containing the IDs of the mentioned entries.

  These IDs are used on the login step to determine which ones a certain user owns from all the entries in the collections, in order to populate the tabs with them after the server has verified the login information of the
user. These IDs, however, are different from the field id, which is not depicted in the next figures and that corresponds to the ID assigned by the database once the entries are stored in there.

- **Vaccine**: in the case of vaccines, the application works with JSON objects that are nearly equal to the Mongoose schema because it is easier to store the data and retrieve it this way. The vaccine schema is the following:

![Vaccine schema](image)

The title should contain the name of the vaccine (for example, hepatitis B or chicken flu), but there is no restriction about it. The date field is contained as a Date value, and the Route Injector back office recognizes it and represents it in a calendar on the web view.

The time represents the hour at which the appointment for the vaccination has been scheduled, and the type field is used for internal purposes of the app (to detect the type of the entry in some functions, basically).

The vid is the ID created through the combination of some characters of the title, the date and the time, and it is also used internally by the app as mentioned before.

The creation of this own ID comes from the fact that the entries are not stored immediately in the database, so they would not receive a proper ID until then, and since some methods need some identification for the entries, they are assigned this special ID.

- **Prescription**: as in the case of vaccines, the app also stores prescriptions as JSON objects, and of course, their structure also follows the Mongoose schema:
In this case, the title is recommended to be the medicine to take to keep it simple, but it can contain a phrase without problems. Prescriptions also have a date, which should be the creation date of it, but also have a final date, which is when the treatment ends. Regarding that, they should have a field for the dose of the treatment, and here there are some differences between the Mongoose schema and how the app stores that information. While the schema has a “dose” field that contains a String, the application has 4 different fields for that information:

- doseTakes: it is the quantity of medicine that the user should take, depicted as a number.
- doseTakesMeasure: it defines the measure of the “doseTakes” value, and it can be “pill(s)” or “ml”, so it does not matter if the medicine format is based on tablets or a liquid.
- doseFrequency: it is also a number as “doseTakes”, but quantifies the interval of takes.
- doseFrequencyMeasure: it defines the measure of the “doseFrequency” value, and it can be “hour(s)” or “day(s)”.

When the prescriptions are uploaded to the server, these 4 values are concatenated into a string, and when they are retrieved, the application processes that string to extract the values and store them in the correct variables.

The text field of the prescriptions is the only variable that is optional. When users create a prescription, they can leave it blank in the app or use it to store notes about the medicine or secondary effects that may appear during the treatment.

- PHR: in the same way as vaccines and prescriptions, PHRs are stored in the app as objects that follow the corresponding Mongoose schema:
The PHRs, as it can be seen in the previous figure, contain a title and date fields as the vaccines and prescriptions, and they work in the same way. The text, unlike in the case of prescriptions, is mandatory, since it is the main part of the PHR. The type and special ID work as in the previous cases.

With this chapter we have seen the technologies involved in the project, how it is structured and an application diagram that shows its internal elements. Now that we know the contents that the app should have and how they would be structured, it is time to go to the last stage of the development.
VacApp: mobile tool for the child vaccination control
CHAPTER 4. IMPLEMENTATION

The first step consisted in defining the needs of the application and how they could be translated into a general design of how the app and its different screens are structured.

Then, the technologies and platforms needed to develop the app have been chosen, taking into account some of the project objectives and trying to use the latest solutions available.

Finally, this chapter will show the implementation of those specifications and architecture and design choices, the results of the process and how the application looks in a real device.

4.1 HTML

The first thing to do is to create all the screens (or pages or views, just how they are called in the framework). The The AppBuilder project has an index.html file where all the views are declared, which are nothing more than div elements with the data-role attribute set to “view”. The Kendo UI will instantiate a mobile View when it founds elements with that data-role.

```html
<div id="tabstrip-vaccines" data-role="view">...</div>
```

In order to change from one view to another, there are two ways to do so: specifying the destiny in the html file or programmatically, using JavaScript. The first way consists in setting the href attribute of an `<a>` tag (a hyperlink) to the desired page:

```html
<a data-icon="cross" href="#tabstrip-vaccines">Vaccines</a>
```

This example corresponds to the vaccine tab shown at the bottom of the screen. Whenever it is pressed, the application will navigate to the vaccines view, since its ID is specified in the href attribute.

The other option is to go to the desired page through JavaScript. This method is used when, before changing views, there are instructions that need to be executed, like when the user is logging in. In that case, the html part of the button (an `<a>` tag with its data-role attribute set to “button”) would be like this:

```html
<a data-role="button" onclick="login();">Sign in</a>
```

And the login() function, which is called when the button is pressed, would include a line like this:

```javascript
document.location.href = "#tabstrip-vaccines";
```
Both onclick and href attributes can be set, but in the case that the destination page is declared in the href, it will only work if the onclick function returns true, which means that the method has been processed.

This would cover the declaration of the views and how to navigate through them, but there is another element regarding the views that should be mentioned, the layout.

### 4.1.1 Layouts

A layout is, basically, the structure of a view. It describes how elements are placed in the screen by declaring `<div>` elements and, through CSS, placing them following the desired pattern. Another way to achieve this is to assign data-roles to the different `<div>` elements, like “header” or “footer”. Furthermore, HTML 5 introduced semantic elements in order to improve the definition of the page layout. Some of them include `<header>`, `<section>` and `<footer>` elements, and depending on which one is used, it will be placed like it is showed in the following figure:

![Layout Diagram](image)

Fig. 4.1. Layout of a page using semantic elements from HTML 5.

In the case of VacApp, only header and footer elements have been used. The header will usually contain the settings and synchronization buttons (along with the application name), and the footer changes depending on the current view: it can contain the tabs if the user is in the main menu, or it can contain editing, saving and deleting buttons if an entry has been pressed. Using `<section>` and `<aside>` elements may be useful for showing the details of the selected entry and having aside the complete list, so the user could select all the entries in a list without changing the view. This solution, however, would be very limited due to the size of a mobile screen, so those elements are not used at all.
The headers and footers can be declared in the same <div> that represents the view, but they can also be written apart and then assigned to the desired page by using the “data-layout” reference (and in that case, setting the “data-role” for each layout). This is very useful because allows the developer to create a layout and apply it to different views so they have the elements in the layout in common. In this case, the 3 tabs of the main menu can share the same layout, so there is no need to declare the settings and synchronization buttons in each view, but only once.

To assign a layout to a view we have to set the “layout” property. It can be done through the “Designer” view of the program or directly in the html file:

```html
<div id="tabstrip-vaccines" data-role="view" data-layout="Layout1"></div>
```

Note that it is not mandatory for all the views in the app to use the same layout, so for different sets of views we can create different layouts to show different options depending on the content shown.

### 4.2 CSS

With only the html file we have the views, but when the app is executed, it will not be very visually appealing. As any webpage, since this application is programmed using HTML, we will have to add CSS files that apply styles and visual effects to the elements on screen.

The CSS files are the ones that contain all the style instructions for the app. In order to apply styles to HTML elements, there are 3 different ways to do so:

- **Link an external style sheet and write all the code separated from the HTML file:**
  ```html
  <link href="styles/main.css" rel="stylesheet" />
  ```

- **Write the styling code inside the HTML file, in the <style> tag inside the <head> one:**
  ```html
  <head>
    <style>
      body{
        background-color: black;
      }
    </style>
  </head>
  ```
- Write the styling code inside each element:

```html
<h1 style="color:red;margin-left:20px">Heading 1</h1>
```

These three options can be combined, so it is possible to set some properties in an external sheet and other ones inside the elements or in the `<head>`. In that case, all of them will apply, and if there is a property declared in all three places, the priority would be, from high to low:

Inline style > Internal style sheet > External style sheet

For this project, all the style instructions have been written in external CSSs, in order to keep separated the core elements from the aesthetical aspects. By doing this, it is easier to find an specific part of the style sheet and just change it, or if there is a piece of code in the CSS that is applied to several elements, just changing one line can affect all of them. However, using the Kendo UI introduces some problems when trying to apply own CSS files.

### 4.2.1 Conflicts with the Kendo UI CSS file

In some cases, it has been found that a declaration stated in the CSS made for the app was not being applied. This happens when we try to apply a style to an element that is also styled after the CSS file created by the Kendo UI. The application gives more priority to this last file (called "kendo.mobile.all.min.css").

There is a way to keep the two CSS files intact and yet, give the maximum priority to the one that we have created, and is the usage of the "!important" declaration. In a CSS file, if we write "!important" next to an attribute and before the semicolon, the application will apply that style in any case. This is a technique often used in web CSSs for setting different parameters depending on the version of the browser. In this case, we can use it to override a style with higher priority without having to change the source code.

### 4.2.2 Color and font

Nowadays, a lot of mobile applications have a determined color associated. For example, both Twitter and Facebook use the blue color as their main one, with the difference that the Facebook choice is a darker blue. YouTube uses the red color, and Yahoo has always used purple as their signature. In the same way, VacApp should have a basic color to be recognized by. The color selected is a lighter variation of green. Usually, the color green is associated with health or healthy habits, so it is a good idea to use it as the main color of the app.

In the same way, it would be a good idea to choose typography to use at some parts of the app. From the wide collection available at the site
www.fontsquirrel.com (with hundreds of 100% free for commercial use fonts), the Quicksand font has been chosen. It uses a thin stroke for the words, which gives a clean and actual look. Since it is a custom font, we have to create a @font-face rule so it can be used. The @font-face rule is used in webpages to apply fonts that the user may have not installed in those sites.

To apply it, in the @font-face rule we have to specify a name for the font family (for example “VacAppFont”) and then write the URLs that point to the font archives. To work properly, there are 4 files that should be included in the @font-face rule: TTF, WOFF, EOT and SVG. This is how it is written in the CSS file:

```css
@font-face{
  font-family: 'VacAppFont';
  src: url('quicksand.eot');
  src: url('quicksand.woff') format('woff'),
       url('quicksand.ttf') format('truetype'),
       url('quicksand.svg#webfont') format('svg');
}
```

To apply the font, it is as easy as to specify it in the font-family attribute of a text CSS declaration (in this case, the title of a vaccine element from the vaccine tab list):

```css
.vaccine-title{
  font-family: VacAppFont;
  color:rgb(0,0,0);
  padding-top:10px;
  padding-left:15px;
  overflow:hidden;
  text-overflow:ellipsis;
}
```

### 4.2.3 Icons

Kendo UI offers a very complete icon font for its usage in the application. These icons can be applied in button objects, list items or, for example, tab items. Since these icons are already integrated in the project and they have a very clean look that suits the app general aesthetics, they will be used for specific cases.

To apply one of them, the only thing to do is to specify the data-icon attribute in a compatible tag. Some examples on how to assign these icons to an element would be the following:

```html
<a data-role="button"
   data-icon="reply" onclick="goBack();">Return</a>
```
The first case has been taken from a button. Leaving the inner HTML field blank would result in a button that only shows the selected icon. The second case is from the vaccine tab, and the third one is applied to a list item. The problem here is that, despite the icon font has more than 300 icons, only 34 are already defined and ready to use (and not all of them suit the app contents).

In order to have a wider variety of icons, some of them have been selected (the complete collection can be seen, with the Unicode code assigned to each one of them, in the Kendo UI documentation) and added to a CSS file with their definitions. This is how an icon from the Kendo UI font can be defined for its use:

```css
.km-cloud:after,
.km-cloud:before
{
    content: "\e08e";
}
```

This example is used for the synchronization button, which has a cloud image on it. The only thing needed is to specify the Unicode code (here, “\e08e”) and a name for referencing the icon (here, “cloud”).

### 4.3 Application icon

With the primary color of the app decided and a font chosen to have a specific look, we can now design the application icon. The current style in applications is the use of plain colors and simple silhouettes, which seems to have been popularized since its use in Windows 8 and that has been also applied by Apple in its mobile phone operative system iOS 7 (but including the use of gradients).

As the app uses plain colors in its menus, it is a good idea to keep that style for the application icon, since it is the first thing that users will see if they look for it at an app store. In the case of VacApp, there is not a distinctive icon for the app, but there is an object directly related with it: the syringes.

Of course, to avoid a strong look, the syringe would not be realistic, but designed using geometric shapes like rectangles, to make it look friendlier. And to relate it a little more with the app, instead of just having a syringe in the middle of the icon, it could be combined with the initial of the application:
As it can be seen in the previous figure, the iOS version of the app has an icon with a gradient, in order to make it look more integrated with the native app icons on iOS.

### 4.4 Comparison with specifications

As said in the chapter 2, the specifications given there would serve as guidelines for this stage of the development. The following paragraph will consist in a series of figures showing the final look of the application and comparing them to the first sketches of the app.

#### 4.4.1 Login screen

The login screen is the first view to compare with the initial specifications. Its design has not changed much, but there is a very important difference respect the definitions given in the second chapter of the report, and it is that the registration screen has been assimilated by the login view.

As it can be seen in the figure 2.2, the register and login screens have the same elements: 2 text inputs for entering the user e-mail and password and buttons for signing in or registering. If there were more information needed for registration, its screen would remain (and be necessary, in fact), but since the only needed is an e-mail and a password, it has been decided to keep it simple and have just one screen for both processes. The next figure shows how it looks:
4.4.2 Synchronize

As mentioned in previous chapters, there is no specific view for the synchronizing process, and the view from where this function can be called remains the same as in the initial specifications. The reason to show this button only in the main menu was to not confuse the user when they are creating or editing entries by showing them multiple buttons and options. The following figure shows the screens seen while the synchronization is being executed:

Fig. 4.3. Login screen.

Fig. 4.4. Screens seen through the synchronization process.
The synchronization button will only work if there are requests pending and if there is an internet connection (if not, the user will be shown an alert telling them that it is not possible to upload the changes, or that there are no changes to be uploaded). In the case that there are requests and internet connection, once the process is finished, an alert will warn the user that the changes have been uploaded correctly.

4.5 Improvements

During the implementation stage some improvements have appeared in order to add new features or to enhance some others. The views that have not been shown in the previous paragraph can be found there, including the mentioned additions. This paragraph summarizes them, explaining what they do or change:

- Unified login and registration view: from the login and register screen mockups at the chapter 2, we can see that the only information needed for both functions is an e-mail and a password. Therefore, instead of having separate views for each one of them, it would be easier to share the same for both functions.

By doing this, we have a login view with only login and registration buttons. When this last one is pressed, instead of heading the user to another view that would look quite similar to the login one, the app will take the information entered in the login input fields and register a user using that data.

- PHRs tab converts to record tab: in the specification chapter we have seen that the initial idea was to have a tab strip in the main menu, where each tab corresponded to a type of entry: vaccines, prescriptions or PHRs.

In this scenario, past vaccines and prescriptions had to options: to be deleted automatically when their appointment date (in the case of vaccines) or their final date (in the case of prescriptions) arrived, or to remain in their tabs until the users delete them. In any case, that information would be lost forever, and maybe in the future could be useful. For that reason, the PHRs tab has been renamed to record tab.

The record tab serves, as its name suggests, as a record, a file. This means that any past entry, be it a vaccine or a record, can be moved to it, and so, the information regarding them is not lost (until the user deletes them, which is still possible).

This can be very useful in the case that, for example, some medicine taken in the past had a bad reaction or effect in the user. They can write annotations about that in the text field of the prescription and file them apart for future situations.
PHRs are not discarded for the app; they are directly created in the record tab, which is the logical thing to do since they are Personal History Records. The next figure shows how the record tab would look like:

![Record Tab](image)

Fig. 4.5. Actual record tab compared to chapter 2 PHRs tab.

This change in one of the tabs and its purpose motivates the 2 next points, which are needed by that modification.

- "Move to record" button: as it has been said before, vaccines and prescriptions can now be stored in the record tab for later consultation or just to keep a history of taken vaccines or medicines.

To do so, a new button has been added to the detailed view of vaccines and prescriptions. By touching it, the selected entry will be moved from its original tab to the record. This is never done automatically, even if the vaccination or expiration date arrives, because in that way the users will have to be aware and helping not to forget them.

As PHRs are directly created in the record tab, their detailed view does not have this button.

With this addition, detailed views have this look, compared to the initial specifications:
“Restore” button: in the same way that a user can move a past vaccine or prescription to the record, maybe at some point they want to make them go back to the tab they came from. This can be due to a mistake on moving them or to reuse a past prescription for a treatment, because then they only have to change the dates instead of creating a new entry and entering all the information.

This is why a “restore” button has been added to the detailed view of vaccines and prescriptions that are stored in the record (as in the previous case, PHRs do not need this button because they are not moved to any other tab).

Vaccines and prescriptions in the record tab cannot be edited (it makes no sense to change them because the point of keeping them filed is to check their information), so the only way to change their fields and reuse them is to first push the “restore” button and then edit them in their respective tabs.

Fig. 4.6. Actual detailed views compared to chapter 2 specifications.
The detailed views that count with this button can be seen in the next figure:

![VacApp: mobile tool for the child vaccination control](image)

Fig. 4.7. Detailed views for vaccines and prescriptions in the record tab.

- **Notifications:** there is an aspect about vaccines that was not considered in the initial specifications, and it is the following. When a vaccine entry is created, the user has to enter a date and an hour. This information should refer to the day and hour when the vaccination takes place, but that hour can only be seen if the user enters the app. Taking this into account, it would be helpful if the app warned the user that they have a vaccination that day, and that is why notifications have been added.

For doing this, there is no a plugin in the Cordova Plugin Registry that has been added to the project. It allows us to show banner notifications very easily and customize them. When the app loads, the plugin creates the `window.plugin.notification.local` object, which can be called to add a notification like in this example:

```javascript
window.plugin.notification.local.add({
  id : vaccineId,
  title : "Vaccination today",
  message : "Vaccination for" + vaccineTitle + "at" + vaccineTime,
  date : dateObject,
  sound : "TYPE_NOTIFICATION"
});
```
The ID of the alarm is the same as the vaccine entry, which is used for searching purposes within the app, and can be used for the cancellation of the notification.

The title and the message are strings that can be modified, and the dateObject passed as the date field of the JSON object is formed using not only the date and hours entered by the user, but also a new field added for this new feature, which indicates the quantity of time in advance that the user wants to be reminded.

For example, if the hour of a vaccination is 11:30, the user can ask the app to warn them 15 minutes earlier, so the notification will appear at 11:15.

Although it is possible to show notifications, Cordova does not give access to developers to the native alarm functions of mobile phones, so it is not possible to set an alarm using the clock of the phone. The actual vaccine creation screen, with the notification time selector, looks like this:

![Vaccine creation view](image_url)

**Fig. 4.8. Vaccine creation view.**

- Sign out button: as it has been seen there is a login step before the user can use the application, but in the specifications there is not a button to sign out anywhere. The first idea was to having to sign in every time the app started, but doing this would not take advantage of the local storage offered by HTML 5.

Thanks to the local storage, save data can be kept without an expiration date, and for mobile applications developed with HTML that does not change.
That means that even if the application would need the user to sign in every time they launch it, if it is not programmed in the app to delete all the data in the local storage, that information would remain there. With this we can store the user credentials and let the user enter the app automatically.

Thanks to that, the user does not have to enter their login information every time, but most importantly, they can use the application even if there is no internet connection, which is one of the main objectives of the project, because anything they create, edit or delete will be assigned to the user saved in the local storage, and it can be uploaded altogether using the synchronize function when a connection is available.

This button has been placed in the settings screen, and has been given a red color instead of the default green of the application to remark it. The settings view, with this addition, looks like this:

![Settings view](image)

With this last paragraph, the implementation chapter concludes. The app views have been shown and not only the functions listed in the chapter 1, but some new features have been added to the final application. With the application finished we can conclude the development of the project and pass to the last part of the document.
CHAPTER 5. CONCLUSIONS

This last chapter contains the thoughts and conclusions reached through the development of this project, as well as an environmental study about it and some possible improvements and guidelines for future revisions.

5.1 Environmental study

It is very important to take into account the environmental impact of a project, since it is an important fact that could result in its rejection. To determine the impact of VacApp we have to take into account these points:

- Usage of the app: the amount of time that a user uses an app affects the environmental impact from the point of view that, while using it, the phone spends battery, and the more battery spent, the earlier that the device has to be charged, meaning a more frequent electric consumption.

  In the case of VacApp, however, there is an important point that should be considered, and it is that it is not a continued use application. Unlike social network or instant messaging apps, VacApp does not offer a service that would be accessed several times during the day: the user stores a vaccine appointment or a prescription and can consult them, but the most reasonable thing is that they do it once or twice a day, at most.

  In the case of humanitarian missions, the app would be used much more, but even in that case, as there are no background operations, the battery consumption would not be big. In addition, VacApp does not communicate all the time with the server, so it does not increase the data consumption of the phones.

  With this, we could say that the usage of a mobile phone depends on the user, but VacApp would have a very slight impact on the environment.

- e-health: going back to the first chapter, to the e-health definition, we find the increase in the efficiency of health care and the improvement in the user – doctor relationship as some of its objectives. Increasing the efficiency of health caring would result in fewer visits to the doctor and less complications in the treatments (and then, fewer displacements, less energy consumption…). Thanks to VacApp, users can store prescriptions about medicines and possible secondary effects or alternative products, which means that future problems can be avoided.

  Moreover, changes in the relationship between patients and specialists imply a better communication between them, information saved and personal treatment, and VacApp contributes to that change. Altogether, it means a better diagnosis, which can avoid problems in the future and
unnecessary waste of energy, movement or products whose manufacture has a considerable impact in the environment.

Calculating the environmental impact of a project like this can be complicated, since it is difficult to determine the specific impact of an application when each user can give a different use to it. But, if we take into account the mentioned aspects before as a main approach, the conclusion is that VacApp does not represent a threat to the environment.

5.2 Conclusions

This paragraph includes some personal thoughts and impressions about the project, which have emerged through its development and once it has been finished.

The first of all is how it has served to me as a way to learn about some topics which I have not seen or developed in depth in the master course. One of them is the HTML language. It is a language commonly known among developers, but in past courses it has never been explained, so to learn about it, it would be necessary to study it on my own.

Thanks to this project I have been able to learn about this language, but also about CSS and to deepen in JavaScript, which I have already seen in the career. I think it is not mandatory, but much recommended to use unknown tools in this kind of projects, because then we take advantage on the time needed to develop the project to learn something new that we may have not seen before.

There are also some technical conclusions I would like to explain. Regarding the framework, Telerik AppBuilder is a good platform for the development of mobile applications, as it offers different solutions that include cloud services, statistic analytics or integration with the different stores, among other features. In addition, it uses HTML 5, the most recent version of the markup language, along with JavaScript and CSS 3, which is also the latest version. If we add the Kendo UI with its collection of widgets or the Cordova plugin registry, with tons of plugins that add more functionalities, we have a very complete solution.

The negative side of this tool is that, due to its cloud oriented services, it is absolutely impossible to work without an internet connection, as the program requires the user to be logged in through their Telerik, Facebook or Google account; and to this, we have to add the fact that there is no local copy of the Project, so the user cannot work with it and upload it later. Nowadays it is not difficult to find a wireless connection even on public places, but having the possibility to work without it is always better than not having it.

By testing the project I have seen that Node.js works very fast, proving that it is a technology that has a very favorable future. Another proof of this is that there are big companies and websites like Yahoo, Mozilla or LinkedIn that are using
Node.js for their services. Of course, comparing the user influx for servers of these companies to the one of the server of this project is not valid, but the reason of their use of the technology remains the same: its efficiency and speed. Another good point about Node.js is its support for modules to complement it, by which I have been able to use Route Injector and its benefits as explained in the chapter 2.

And related to Node.js, Route Injector, the module for Node.js mentioned in the chapter 3 and developed by past students and a professor from EETAC, seems like a very useful module that can save a lot of time to developers when putting up a server. It is very easy to modify it to meet our needs and creates all the routing options that could be needed.

If we take the points suggested in the introduction, we can find that all of them have been completed, which leaves me with a good feeling about the project. Moreover, some of these objectives have been expended with new functionalities that were not included in the first idea, which means that there has been an evolution that implied a reconsideration of the base of the project as new requirements appeared.

Finally, about VacApp, the application result of this project, I think that it can be a very useful tool, as mentioned in the introduction, not only for regular smartphone users but also for its use in specific situations like in zones in need of humanitarian help. The vaccination control is a very important matter in some of those places, and doing it in a traditional way can result in the loss of data or miscommunications; with VacApp, all the information collected at the end of the day could be stored in a server, so all the humanitarian personnel could retrieve it and work with it.

But to prove its utility in a much closer situation, we can think about an elder person in a residence. If that person needs a treatment and receives some indications from their doctor, the family can receive that information through VacApp, so they know the health state of their relative, or all the workers at the residence can share the same prescriptions about that resident, so all of them know the medication and the dose needed for that person. VacApp can be very useful in both kind of situations, and it does not need of an expensive device to run it, just an ordinary smartphone.

5.3 Future guidelines

As in any project, there are points and aspects that can be improved, or some others that may have not been taken in consideration or could not be added in time.

One of them is the integration with the aaaida network. As explained in the context paragraph of the first chapter, VacApp could be combined with the possibilities of the aaaida network in order to offer a more complete service.
Instead of just creating entries from the app, the users could be able to log in to their aaaida accounts and save vaccination dates that could be downloaded to the app, or edit these web created entries from the mobile phone.

Basically, let the users create, modify or delete entries from both platforms, giving them different choices on how to manage them and implementing a real bidirectional communication path.

Another improvement that could be done consists in implementing a reminder service through e-mail. Since the users have to register using their e-mails, it could be useful to receive some note about pending vaccinations or treatments started, like a weekly newsletter or punctual mails.

The user could decide if they want to receive these mails from the settings view, so anybody has to receive them if they do not want. Or instead of registering entering their e-mail, let the users register using another account from a social network, like Facebook. With this it could also be possible to share information through it.
BIBLIOGRAPHY


Telerik official website for its AppBuilder developing platform. Program source, information and support documentation can be found there.
(Last access: Apr. 2014)

W3Schools is the reference website for web developing, including tutorials and basic information about HTML, CSS and JavaScript, among other languages.
(Last access: 10 Nov. 2014)

Telerik site with resources for developers regarding AppBuilder, including forums, tutorial videos, sample projects and a feedback portal, among other options.
(Last access: 29 Oct. 2014)

Documentation about Kendo UI and its usage.
(Last access: 11 Nov. 2014)

Library with hundreds of plugins to add to Apache Cordova projects, with tutorials on how to install and use them.
(Last access: Sep. 2014)
ANNEX

Mongoose models

There is an important part about the project that would be great to be integrally transcribed, and that is the Mongoose modeling.

The code of the application can be easily found at a public repository from GitHub for its complete revision (https://github.com/DrGrijando/VacApp), but the models mentioned in the chapter 3 are not uploaded on the Internet, and they can be useful as a reference for other people to know how they are structured.

The following code snippets contain the schemas on which the models are based, and there are 4: User, Vaccine, Prescription and PHR. For a detailed explanation on the different fields of the schemas, consult the chapter 3 (page 24) of this document.

User model (schema.js)

```javascript
var mongoose = require('mongoose')
 , Schema = mongoose.Schema
 , ObjectId = Schema.ObjectId;

var functions = require('./functions');

var jsonform = require('mongoose-jsonform');
var injector = require('mongoose-injector');

var schema = new Schema(

     { userId: {type: String},
       email: {type: String, required: true, unique: true},
       password: {type: String, required: true},
       myVaccines: [{type: ObjectId, ref: 'Vaccine'}],
       myPrescriptions: [{type: ObjectId, ref: 'Prescription'}],
       myPhrs: [{type: ObjectId, ref: 'Phr'}],
     profile: {
       name: String,
       surname: String,
       phones: [ { description: String,
                   number: Number }
     ]
   }, { id: false }));```
// Is used to enable refection in security middleware
schema.plugin(jsonform, {
  excludedPaths: ['_id', '__v'] // these paths are generally excluded
});
schema.plugin(injector, require('./injector'));

functions(schema);

exports.getSchema = function () {
  return schema;
};

Vaccine model (schema.js)

var mongoose = require('mongoose'),
    Schema = mongoose.Schema,
    ObjectId = Schema.ObjectId;

var functions = require('./functions');

var jsonform = require('mongoose-jsonform');
var injector = require('mongoose-injector');

var schema = new Schema({
  title: {type: String},
  date: Date,
  time: {type: String},
  type: {type: String, enum: ['vaccine', 'prescription', 'phr']},
  inRecord: {type: String, enum: ['true', 'false']},
  vid: {type: String}
}, { id: false });

// Is used to enable refection in security middleware
schema.plugin(jsonform, {
  excludedPaths: ['_id', '__v'] // these paths are generally excluded
});
schema.plugin(injector, require('./injector'));

functions(schema);

exports.getSchema = function () {
  return schema;
};
**Prescription model (schema.js)**

```javascript
var mongoose = require('mongoose'),
    Schema = mongoose.Schema,
    ObjectId = Schema.ObjectId;

var functions = require('functions');

var jsonform = require('mongoose-jsonform');
var injector = require('mongoose-injector');

var schema = new Schema({
    title: {type: String},
    date: Date,
    finalDate: Date,
    dose: String,
    text: String,
    type: {type: String, enum: ['vaccine', 'prescription', 'phr']},
    inRecord: {type: String, enum: ['true', 'false']},
    vid: {type: String}
}, { id: false });

//Is used to enable reflection in security middleware
schema.plugin(jsonform, {
    excludedPaths: ['_id', '__v'] //these paths are generally excluded
});

schema.plugin(injector, require('./injector'));

functions(schema);

exports.getSchema = function () {
    return schema;
};
```

**PHR model (schema.js)**

```javascript
var mongoose = require('mongoose'),
    Schema = mongoose.Schema,
    ObjectId = Schema.ObjectId;

var functions = require('functions');

var jsonform = require('mongoose-jsonform');
var injector = require('mongoose-injector');

var schema = new Schema({
```
title: {type: String},
date: {type: Date},
text: {type: String},
type: {type: String, enum: ['vaccine', 'prescription', 'phr']},
vid: {type: String}
}, { id: false });

// Is used to enable refection in security middleware
schema.plugin(jsonform, {
excludedPaths: ['_id', '__v'] // these paths are generally excluded
});

schema.plugin(injector, require('./injector'));

functions(schema);

exports.getSchema = function () {
  return schema;
};