Tampere University of Applied Sciences

Nordic Gourmet Mobile Application

Darío Yuste Tirados

BACHELOR'S THESIS
April 2019

ICT Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
ICT Engineering

AUTHOR: Darío Yuste Tirados
Nordic Gourmet Mobile Application

Bachelor's thesis 51 pages, appendices 20 pages
April 2019

Many companies have their own mobile application in order to perform things faster and easier. Nordic Gourmet is a Finnish enterprise which is one of the largest providers of event and festival food services. The problem they want to tackle is to improve the ways of tracking employee's working hours.

As a solution for Nordic Gourmet's problem, a mobile application with these features has been requested: it needs to let the user start a job, select where the employee is going to work (festival and food truck); it needs to store the information of every job performed by the users; an admin must be able to see all the information and add new selectable options (events and food trucks) to the app; it needs to have users on it, being able to log in and out of their accounts.

With all these features an app has been created with React Native, a non-stop growing framework used for creating dynamic mobile applications using Javascript as the main language, and the React (web technology) basics. Thanks to React Native both, iOS and Android apps, can be developed at the same time and with the same design. Furthermore, as a backend server support, the app uses Firebase as the authentication system, custom database, backend functions server, and push notification sender.

The app has three main screen stacks: Authentication, User and Admin stack. Each one has their own screens on them. Authentication stack is the one in charge of the creation of new user accounts, and letting the users log into the system. The User stack is the one used for Nordic Gourmet workers, in which they are able to start a new job, track it while working on it (seeing time worked and where) and then end it. Employees are also able to see their own profile, access to their job history, and consult the events already programmed. The third main stack is the Admin Stack which is the one providing the management view of the app to supervisors, in which administrators are able to accept (or reject) new user accounts, see users actually working, add new events and work stations, and take a look to every user job history.

Key words: nordic, gourmet, mobile, app, react native, javascript, ios, android

CONTENTS

1. INTRODUCTION 6
1.1. NFC solution 7
1.2. Own tracker solution 8
2. APP SCREENS & WORKING DIAGRAM 9
2.1. Loading Screen 11
2.2. Authentication Screen 11
2.3. Outlaw Users Screen 12
2.4. User Main Screen 13
2.5. Working Screen 15
2.6. User Job History Screen 16
2.7. Profile Screen 18
2.8. User Events Screen 19
2.9. Admin Main Screen 21
2.10. New Users Screen 22
2.11. Working Now Screen 22
2.12. Admin Events Screen 23
2.13. Work Stations Screen 25
2.14. Management Screen 26
2.15. Admin Job History Screen 27
3. CODING FRAMEWORK: REACT NATIVE 29
3.1. Framework comparison 29
3.2. React Native basic operation 30
3.2.1. Render method 30
3.2.2. Props & State 30
3.2.3. Component lifecycle methods 31
3.2.4. Style 31
3.3. Common Components 31
3.3.1. View 31
3.3.2. Text and TextInput 32
3.3.3. Button and Touchable 32
3.3.4. FlatList 32
3.3.5. Modal 33
3.3.6. Icon 33
3.4. Stack Navigator 33
3.5. Local Storage 34
3.6. Listeners. EventRegister 35
3.7. Responsive screens 35

4. BACKEND SERVER: FIREBASE 37
5. AUTHENTICATION 39
6. DATABASE 41
6.1. Database design 42
6.2. Users 43
6.3. Work Stations 43
6.4. Events 44
6.5. Job History 44
6.6. Working Now 45
7. CLOUD FUNCTIONS & PUSH NOTIFICATIONS 46
7.1. Notification type 46
7.2. Cloud Functions 47
8. MULTIPLE LANGUAGES 48
REFERENCES 49
APPENDICES 51
Appendix 1. AppDatabase management methods 51
Appendix 1.1. User 51
Appendix 1.2. Work Stations 54
Appendix 1.3. Events 55
Appendix 1.4. Job History 57
Appendix 1.5. Working Now 63
Appendix 2. Cloud Functions 65
Appendix 3. Push Notifications handle 68
Appendix 4. Multiple language example 70

ABBREVIATIONS AND TERMS

TAMK	Tampere University of Applied Sciences
NFC	Near-Field Communication
UI	User Interface
IDE	Integrated Development Environment
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation
SDK	Software Development Kit
RNF	React Native Firebase
SQL	Structured Query Language
FCM	Firebase Cloud Messaging

1. INTRODUCTION

Today, the importance of mobile applications has significantly grown in various environments such as at work, home, entertainment, etcetera. For instance, people can use them for playing games, turn on the lights, or to control your house alarms. They have started to become a primary need in multiple companies. For example, nowadays every bank has its own mobile app to see your bank account, movements, etc. People are now using apps every day because you can interact with them, they are agile, they are easy to use. Thus, applications are being an extension of us. And that is what will be explaining in this document, talking about the solution made for a company thanks to a mobile application.

Nordic Gourmet, a Finnish enterprise, is the largest provider of event and festival food services. They have been organizing catering services for large and small events for 15 years. Nordic Gourmet has international food, flexible services and years of experience, which ensure them successful food and event catering. [1] But as with many other Finish enterprises, they use an old method to manage the hours worked by employees. They use a platform called MaraPlan, which can work perfectly for more conventional enterprises, those which employees work in an office or those which doesn't have many employees. However, Nordic Gourmet is not like others. In this company, the workers work independently in work stations inside the event or festival, and these events can be small, so you can take control of every worker, or big enough to be a mess trying to control any employee. Therefore, every worker must write down in a notebook or mobile phone their worked hours, start time and end time, and then send those hours to the supervisor. After that, the supervisor will introduce that data inside the MaraPlan, but sometimes the supervisor must ask other employees to ensure that the hours have been worked by the worker. All of this leads to a huge loss of time for everyone, employers and employees.

What can be done to solve this problem? Everybody could think that the answer is easy: all time the employee spend working must be tracked. In fact, that will

be the correct answer, and the best solution. Nevertheless, this answer lead us to another question: how can we track the employee working time?

There were two possible solutions: using NFCs tags, or making the employee to manage his own working time using specific software. Both solutions would have an application interface.

1.1. NFC solution

A possible solution could be installing in every work station at the event/festival a NFC tag. A NFC tag can store information, and it can be read/written by the phone (or another specific device, but we are focusing on mobile phones). This would be great due to the fact that supervisors can be sure that the employee was in the work station since the time they stated, because the NFC tag is located in a known place.

The problem in this approach could be employees without a NFC reader in their mobile phone. And supervisors cannot oblige workers to have a phone with it. Moreover, driving to the warehouse, or the journey to the event, can be included in the working time, which make the installation of NFCs more difficult.

1.2. Own tracker solution

In this solution, the worker would have a specific software (mobile application) to monitor their own worked hours. The application user will be able to choose event and work station, and start the tracking, storing all the data. Next, the employee will be able to stop the monitoring, so the start and stop time will be stored. At the end of the day, supervisors will be able to check this software and see every employee working time easily. This approach can solve the problem of having to write down the start and stop time, to then send later that information to the supervisor.

However, there are some problems controlling workers real position while starting to work, so the supervisor will need to keep asking for the worker position sometimes.

At the end, this has been the solution taken.

In the upcoming sections we will go through: the coding language used for this application, explain why the hybrid application type was picked; the server picked for the little backend of the app; the authentication used, simple and easy thanks to Firebase; the database implemented for the users, events, work stations and the users job history; the server functions that are working in the background to update information with push notifications; the screens made for the user interface; and last, the utility used for multiple languages.

2. APP SCREENS & WORKING DIAGRAM

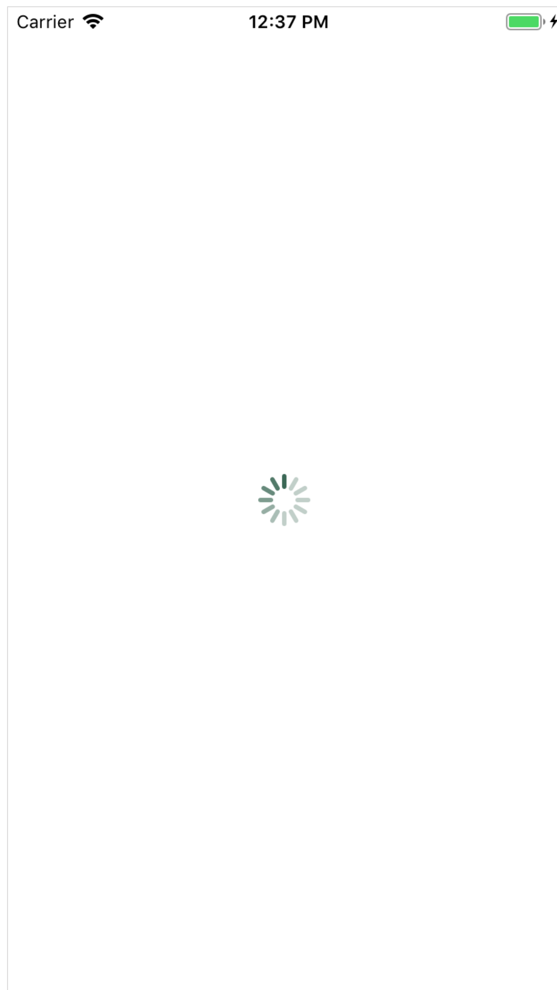
PICTURE 1. Screens Flow Diagram

The screens are divided into four groups: Authentication Screens, Admin Screens, User Screens and Others. The first three groups will be selected depending on the user logged in the app, and the group of other screen has helpful screens for special cases out of the rest of interfaces.

Before a user is identified in the system, the stack of screens will be the Authentication Screens, in which the employee will be able to both, create a new account or login into their already existing account. Immediately after one of the options, the user interface will change to a different one:

- If the user is new (the worker create a new account) → Other Screens Stack will be shown.

- If the user is logged in (the worker already had an account) → The flow can go in two directions, depending on the user role: admin or normal user, which mean heading to Admin Screens Stack or User Screens Stack respectively.



PICTURE 2. Loading Screen

2.1. Loading Screen

(PICTURE 2) As its name says, this screen is only for waiting purposes. There is only a blank screen with a spinner in the middle of it, which will make the user think that something is happening in the background, which is the case.

In the background of this screen, we are creating listeners to listen for push notifications (they will be explained later), but more importantly, we are listening for the “AuthState” of the user, which will define if there is, or not, a user logged in the system and who is it. This is possible thanks to the authentication method we have, Firebase Auth, which can save the last user that was logged in the system, so the next time the worker open the app he will be still logged. After knowing the user identity, some or none, the app will jump to the next screen.

2.2. Authentication Screen

(PICTURE 3) After the Loading Screen, if no user was found in the system, this screen will be the next jump. In this particular screen, the user has two tabs to press: “Already has an account” (default one) and “Create new account”, which head the user to the possibility of login into the system, or to create a new account.

As a user interface, it is simple, containing a bunch of TextInputs (boxes where the user can write text) for the user information, such as email & password for login, or name, family name, etc for new account; and a button at the end of the screen for submitting the information provided. If any error takes place during the process, it will be displayed between the TextInputs and the button, in a red colour.

The background of this screen is not as complex as the one before, from Loading Screen, in this one we only take the data from the user. Then we check it in case it is empty, and we send the data to the server functions.

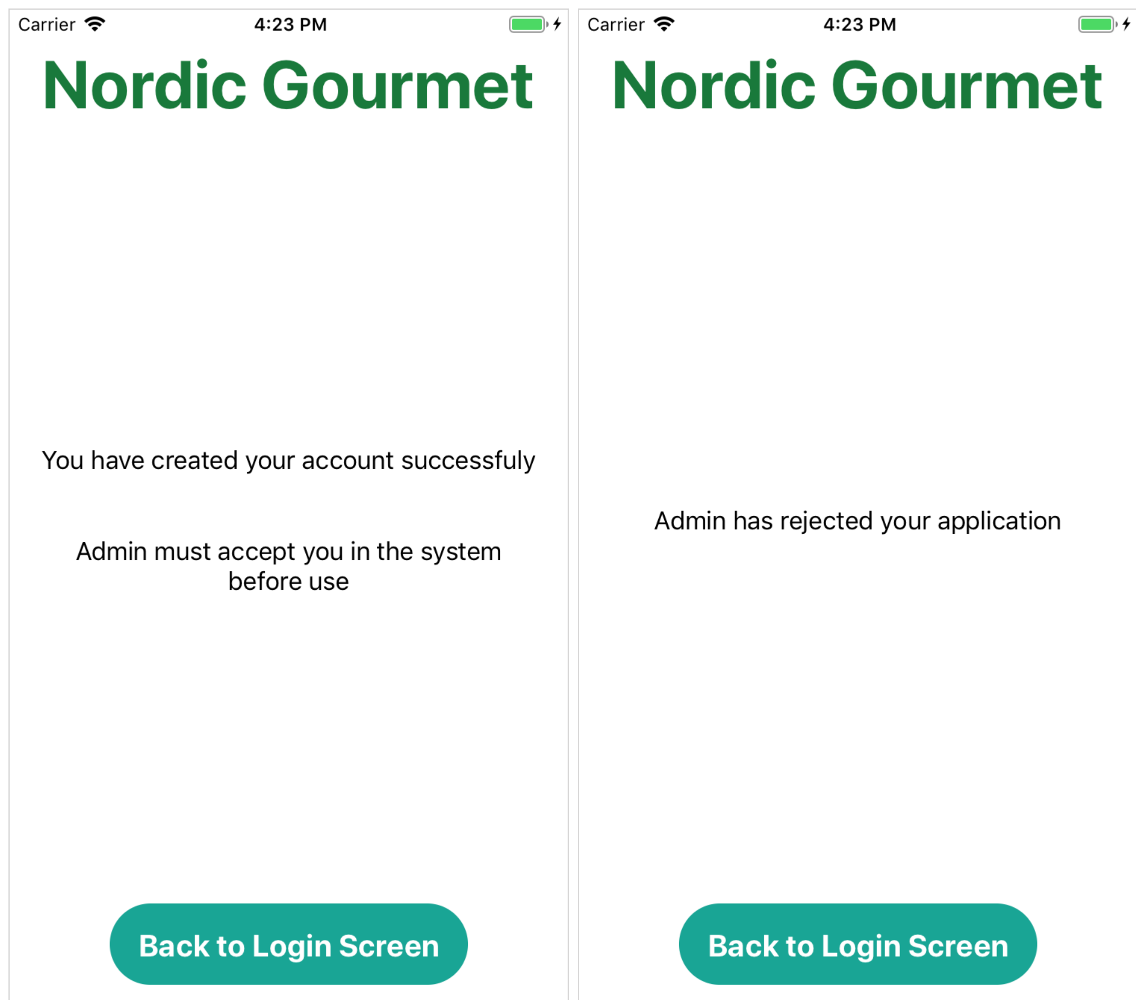
PICTURE 3. Authentication Screen: Login (left) and Create Account (right)

2.3. Outlaw Users Screen

(PICTURE 4) Other Stack Screen only have two screens, and this one is one of them. Outlaw Users Screen is made for that users which still don’t have the “active” state in the system. These users can be both, waiting for an admin reply for their application (“waiting” state), or already rejected by the admin (“rejected” state).

In this case, the UI is even simpler. It will only display a message explaining the situation and a button for going back to the Authentication Screen.

There is no background to explain for this screen.



PICTURE 4. Outlaw Screen: waiting user (left) and rejected user (right).

2.4. User Main Screen

(PICTURE 5) This screen is the most important one for the employee. In here, they will be able to start their new job and navigate through all the remaining screens a normal user can have. This screen is very easy to use.

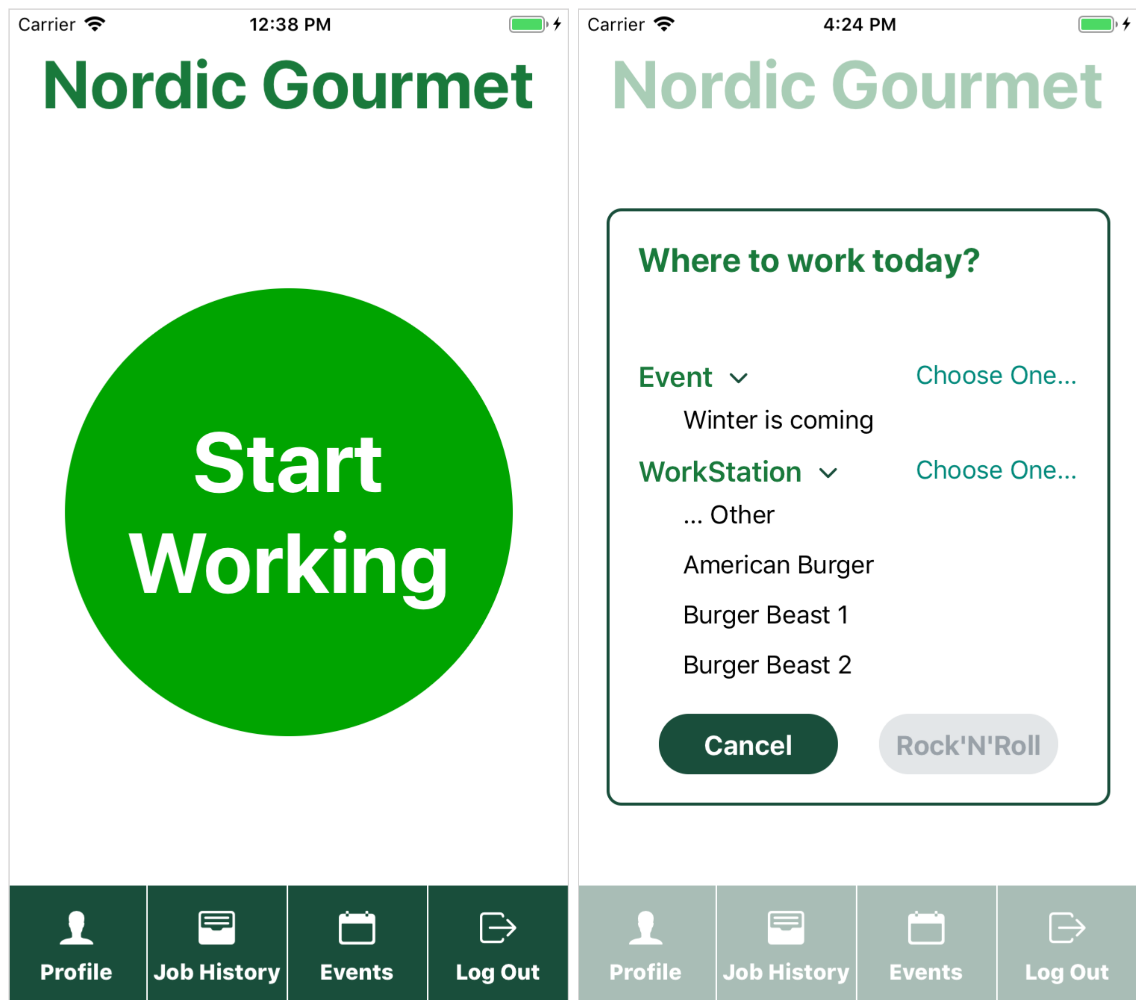
As user interface, the User Main Screen has a very big button in the middle of the screen, and four more button at the bottom of it. The big button will say “Start Working”, and as everyone can imagine, it will be the start for the process to choose where to work (event and work station). The start button will trigger a modal (pop up window in the middle of the screen) in where the user will be able to pick up from a list the event in which to work, and the work station from a different list. After completing the process and pressing “Rock’N’Roll”, the

screen will jump into the Working Screen, meaning that the employee is already working.

The four buttons in the bottom of the screen will lead the user to a different screen (Profile, Job History or Events), depending on which button is pressed. However, the last button, Log Out, will make the app to return to Authentication Screen, because no user will be logged in the system.

Talking about the background in this screen, we create four listeners to listen for inside app events, which will be created after a push notification is detected for the creation/removing of events and/or work stations. This listeners will trigger some functions to read the data provided, analyse it, and afterwards update the data in the screen and in the local storage.

Furthermore, in order to make things faster while using the app, in this screen the app will request many of the data the user can use to the server asynchronously, and after getting it, the data will be saved in local storage.
At last, whenever the user starts a new job, the background code will call some server functions to create that job in the server database (Job History), and a new “working now” entry (temporal database for currently working users).



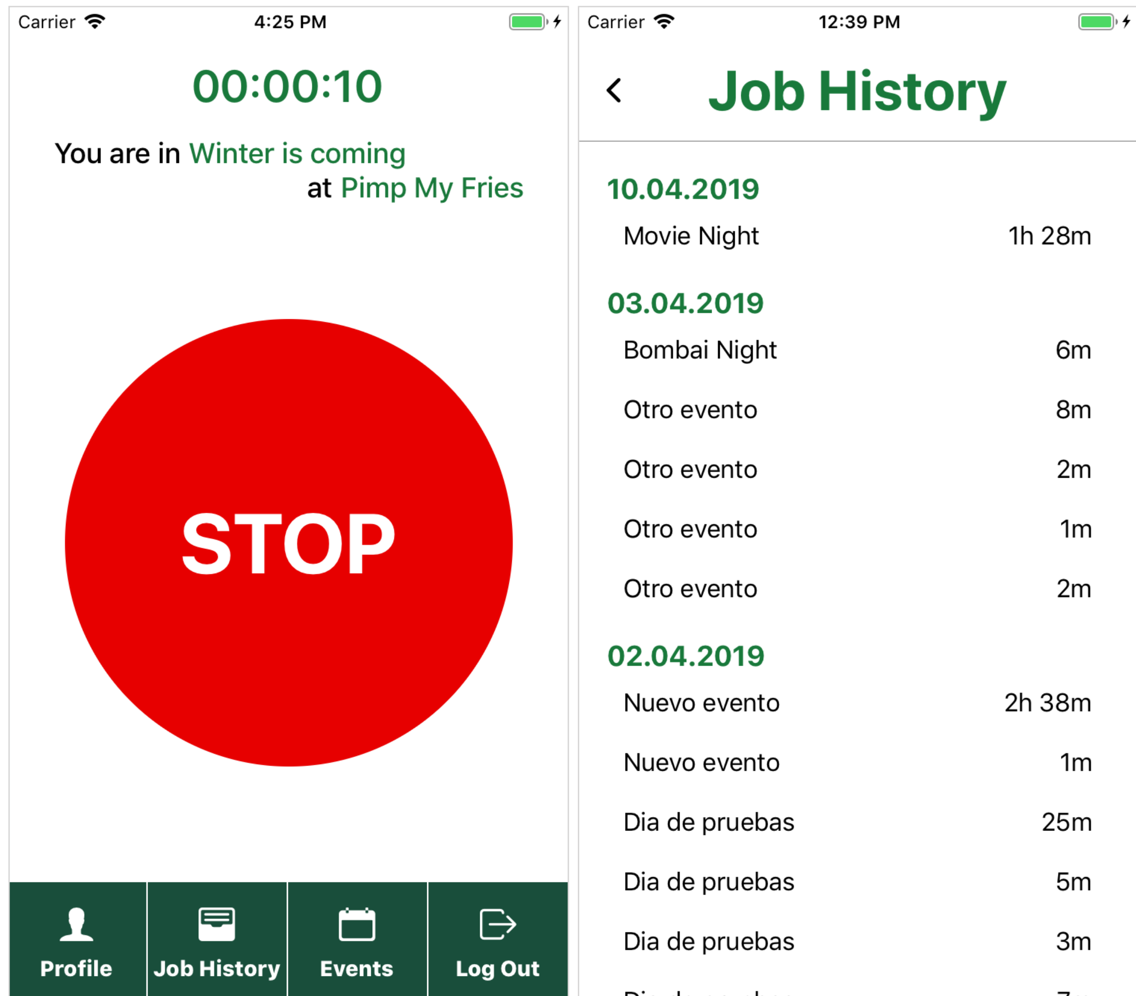
PICTURE 5. User Main Screen: normal view (left) and starting job view (right).

2.5. Working Screen

(PICTURE 6) This isn't actually a real screen in the code, but we can define it as a different one for a better understanding. If this screen is being showed, it means that the user is working. Thus, the worker will be able to see a timer in the top of the screen (time worked by then), and where is he working. In addition, the user will be able to stop the current job, which will trigger the change to User Main Screen again.

Similar to User Main Screen, the UI in this screen will have a big button in the middle of the screen, and four more buttons on the bottom (the same ones as before). Nevertheless, this screen have the timer on top as we explained recently. The big button this time will say "Stop", and when pressed it will trigger a pop up window to confirm your decision of stopping working.

The background of this screen is simpler than the User Main Screen one. It will only send information to server functions in order to update the job created in Job History while starting to work, and delete the "working now" also previously created.



PICTURE 6. Working Screen (left) and User Job History Screen list only (right).

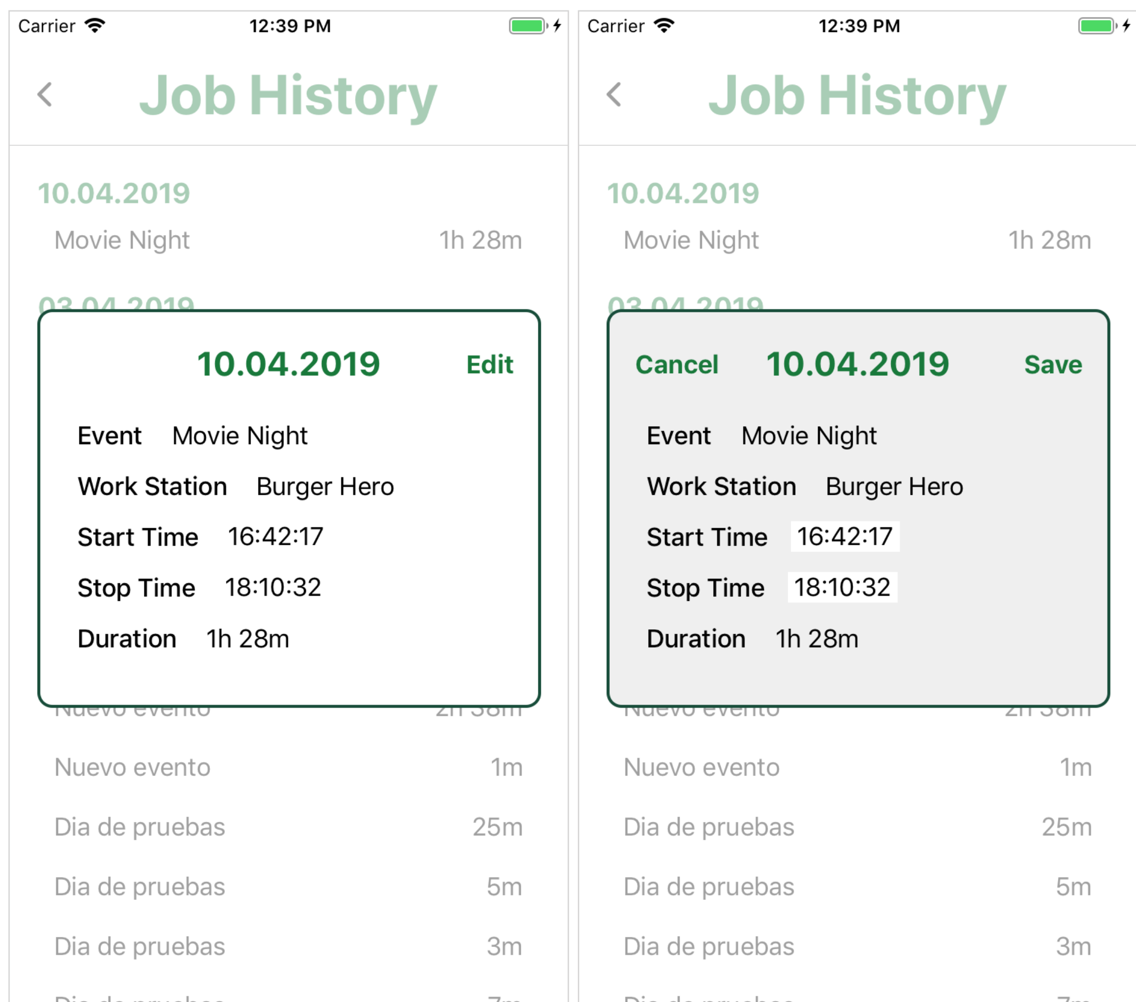
2.6. User Job History Screen

(PICTURES 6 & 7) Job History Screen for users is the screen in which the employees will be able to have a look to all their jobs completed. The jobs will be ordered by date, so that the most recent job done will be the first one (top of the screen). These list of jobs is only a summary of them. To see the information of every job individually, the user need to touch each job in the list (one by one).

User Interface is similar to User Events Screen, a FlatList component that has all the different dates among all the jobs, showing them as a subtitle, and inside

them, every job completed on that date (could be more than one on the same day). Moreover, as explained before, each job in the list is a TouchableOpacity (similar to a button) that the user can press in order to see the information of that specific job. The information will be shown in a modal in the middle of the screen. This new modal will have an edit mode as Profile Screen has.

The background in this screen is basically one thing: getting the job list from the server database. Once the client app has the data, it will order the data to make two different lists: *datesArray* and *jobHistoryData*. The first one will have all the different dates among the jobs, and the other one will have all the real data about the jobs (a small list of jobs for every different date). Also, when performing any change in some job (saving changes in “Edit” mode), it will send the changes to the server functions.



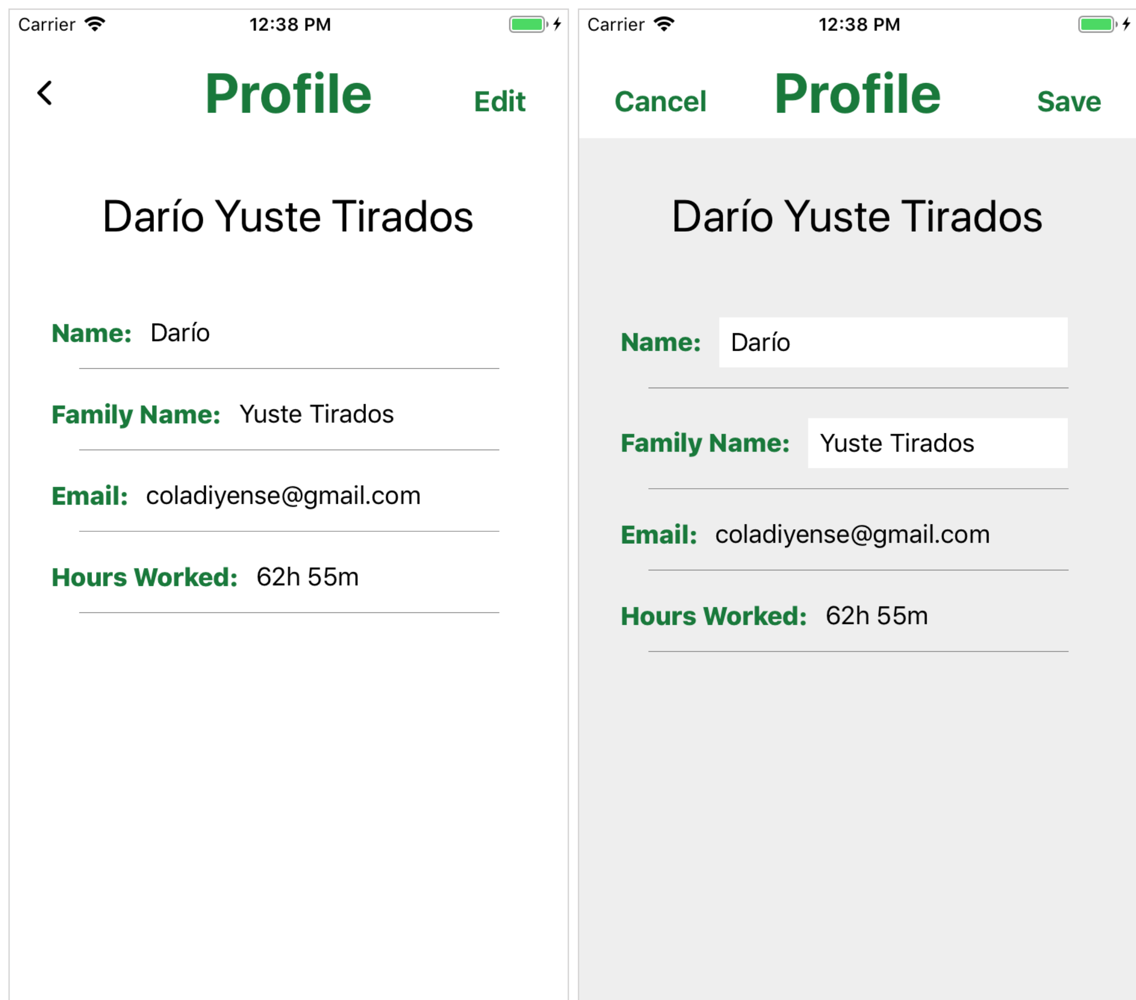
PICTURE 7. User Job History Screen: modal (left) and editing modal (right).

2.7. Profile Screen

(PICTURE 8) This is the screen where the user can see their profile and statistics. Furthermore, if the user needs to change their name or family name due to an error at the moment of account creation, they will be able to do it.

There is not much to appreciate as UI in the Profile Screen, just some Texts components showing all the data and a text button at the top (“Edit”). However, this button will change the interface shown. After pressing it, the screen will change its color to light gray, and only two white boxes will remain in the screen, the both editable properties, name and family name. It changes the Text components for TextInput components, so the user will be able to write in it. Also, the text button at the top-right of the screen will change to “Save” instead of “Edit”. Pressing this new button will reverse the screen to the previous screen, “no-edit mode”.

About the background code located behind this screen, it will change the mode between “edit mode” and “no-edit mode” when pressing the text buttons (as explained before). After touching “Save”, it will also analyze the changes made to the properties and compare them with the original text. If changes suppose a real change in the stored data, it will send the new data to the server (thanks to server functions), updating the data in there and, of course, it will update the data which was being showed in the screen.



PICTURE 8. Profile Screen: normal view (left) and editing view (right).

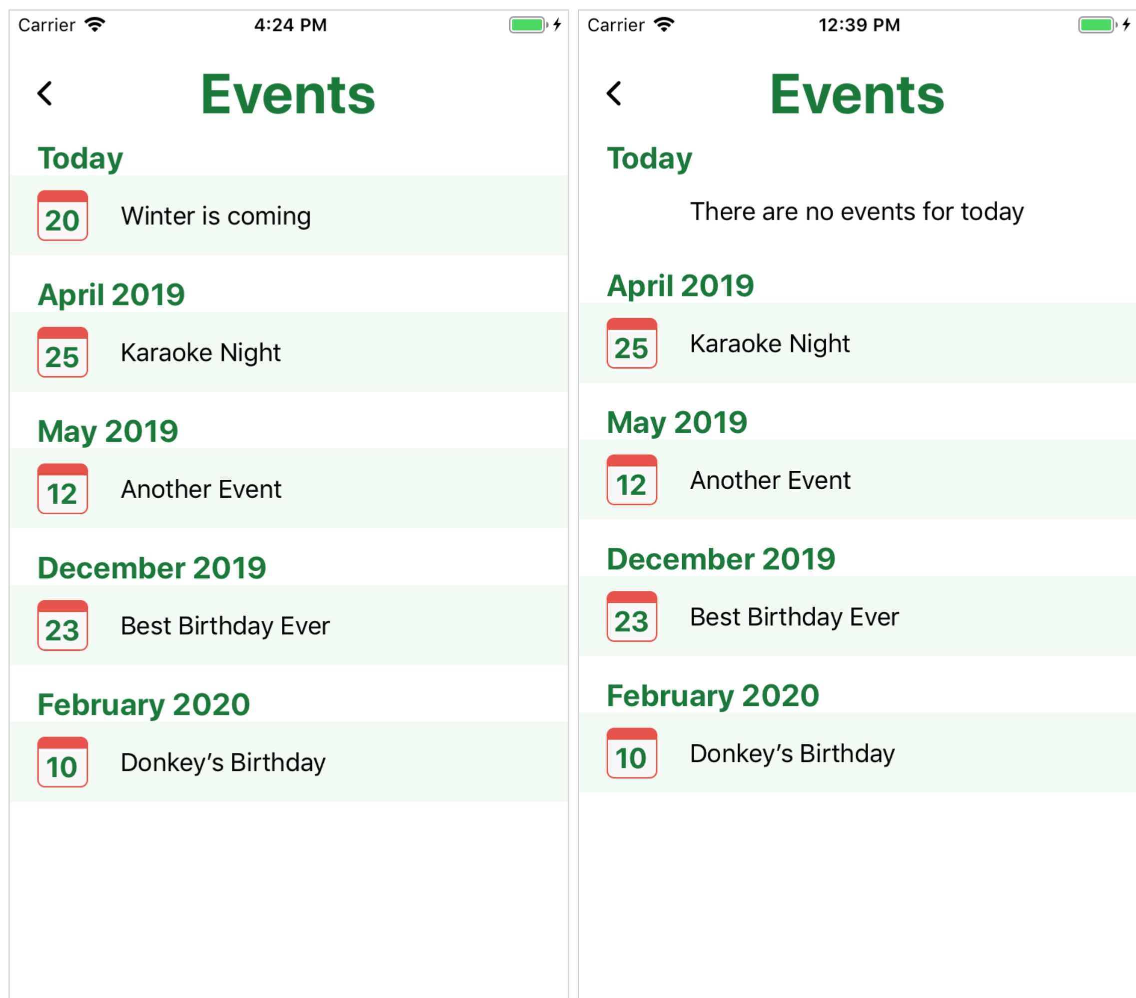
2.8. User Events Screen

(PICTURE 9) This screen is meant to be an informative list. In it the user will only be able to see the programmed events, today and future ones, but not already past events.

The UI is built with two different FlatList components (a list of items filled with an array of data), one for today events, and another one for the upcoming events. Both should be sorted primarily by date, and secondarily by name (on the same date). Why is “should be sorted” instead of “is sorted”? Despite it working perfectly every time the screen is opened, if the screen is already opened and a push notification with events data arrives, the data will be analysed and added to one of the lists, as expected for iOS, the new item will be shown correctly

sorted in the list, but in Android phones, it doesn't use to work the secondary sorting, the item just goes to the last position of the displayed list.

As it happened in the User Main Screen, this screen will also have listeners to listen to any data about festival events sent by inner app events, which would have been sent by the push notification listener allocated in the Loading Screen as we explained above. Also, at the opening of the screen, the data shown is recovered from the local storage (it was stored before in the User Main Screen).



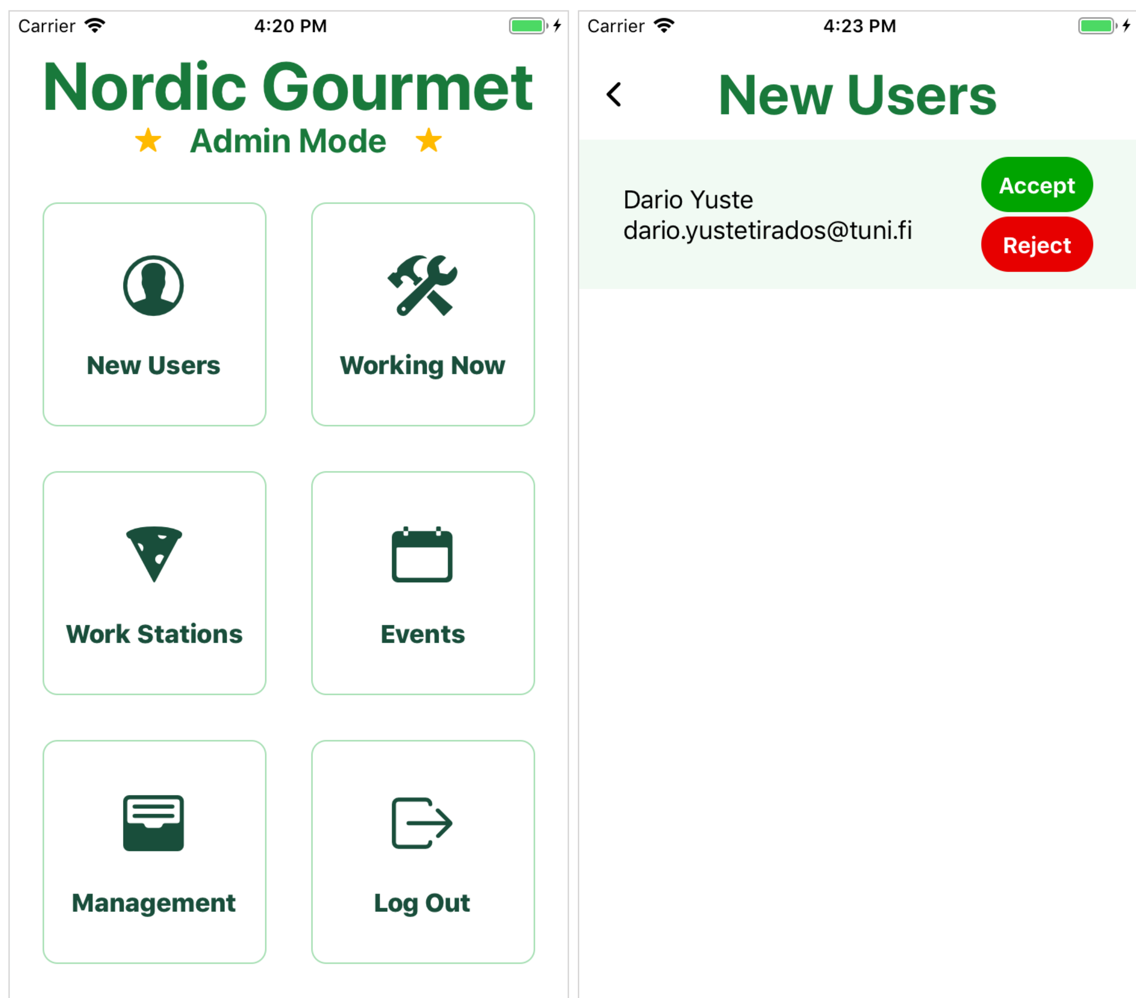
PICTURE 9. User Events Screen: normal view (left) and view with empty today events (right).

2.9. Admin Main Screen

(PICTURE 10) This is the first screen that administrators will see after logging into the system. Similar to the User Main Screen, this screen is in charge of showing all the possible options that an admin has in the different buttons.

The UI has a precise difference in the header, which now says “Admin mode”, in order to notice the user. However, that is not the important feature of the screen. It has six big buttons filling all the display space with the available options: New Users, Working Now, Work Stations, Events, Management and Log Out. Every button will make the admin to jump into another screen.

We have no background in this screen, just the possibility to change between screens with the 6 buttons explained before.



PICTURE 10. Admin Main Screen (left) and New Users Screen (right).

2.10. New Users Screen

(PICTURE 10) Whenever a user creates a new account, they must be accepted or rejected by an administrator. This screen is the one in charge of that. The admin will be able to see any new user account and accept or reject it from here.

As UI it has a FlatList component with every new user name, family name, and email. Each row, except from this information, is followed by two option buttons: a green button saying “Accept”, and a red button saying “Reject”.

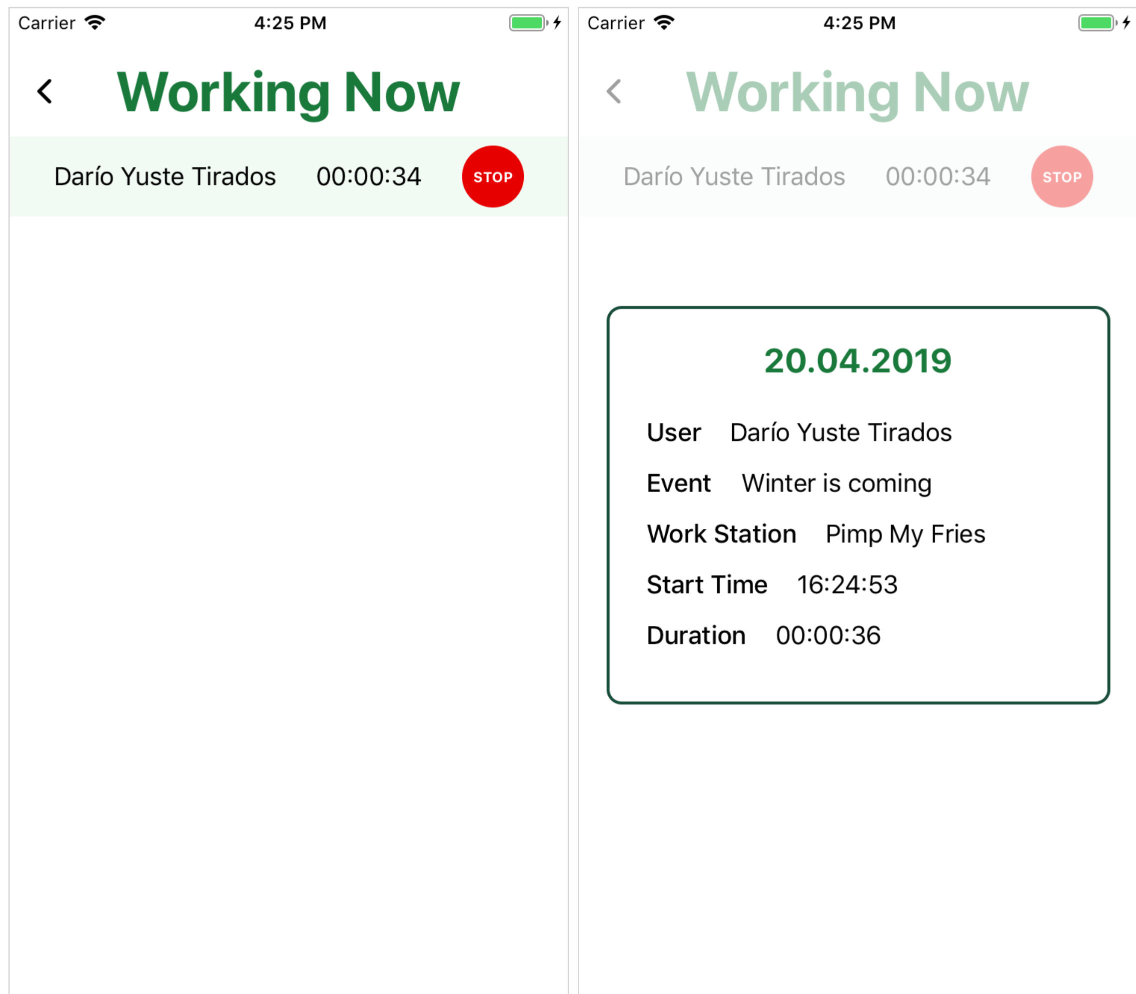
The background primarily is about getting the new users list from the server database. And secondary, update the user information in the system after selecting one of the options for them: accept or reject. In addition to this, the screen has two listeners with the same mission as the ones in Working Now Screen. These listeners manage every change in the new user documents so the screen can be updating according to the changes (addition or elimination of a new user).

2.11. Working Now Screen

(PICTURE 11) The admin will be able to see who is working in a specific moment in this screen. It shows the name of the worker, and how long they have been working on the job. To know any other information about some job, such as where is the employee working, the admin must press the entries of the list. Furthermore, administrators have the possibility to stop the employee job at any moment.

Again, the UI is based on a FlatList component filling the screen. This list has all the users currently working in it, and the users worked time. All the other information explained above will appear together after pressing any entry of the list (each row is a TouchableOpacity) in a modal. The option to stop the job manually is a red button in every row of the list.

This screen has listeners to listen to any event related with the creation/removal of any job, so that it will be able to add a new item to the list, or removed one from it, and update the screen. Also, as any other screen with lists on it, it request the server database data whenever the screen is being mounted.



PICTURE 11. Working Now Screen: user list (left) and job complete data (right).

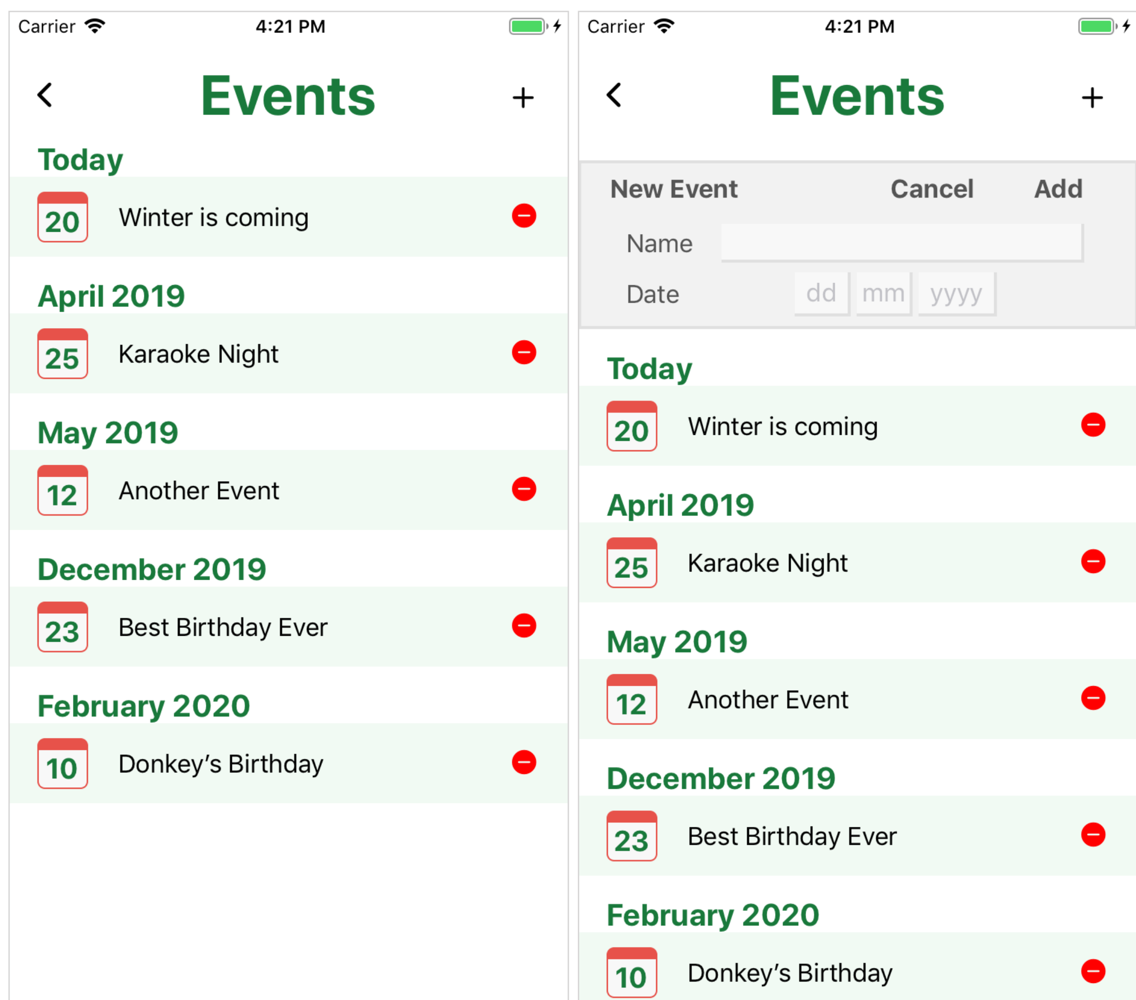
2.12. Admin Events Screen

(PICTURE 12) This screen is almost the same as the Events Screen from User Stack, it is used to see all the programmed events (today and further events). However, there is a big difference as the Admin Events Screen has the possibility to add new Events to the server database or remove them from it.

The UI is almost the same, a FlatList component with all the Events and a red button in each row for the possibility to remove one specific event. But as said

above, the difference comes up after pressing the add button on the top-right of the screen. After it, the screen will change and two white boxes will appear. These TextInputs, the white boxes, are the space where the admin can write the name and date of the new event.

Getting the data from the server database is the first background action performed in this screen. Lately, only actions related to the creation or removal of events will be attended, updating the screen with every new change, and also the database allocated in Firebase.



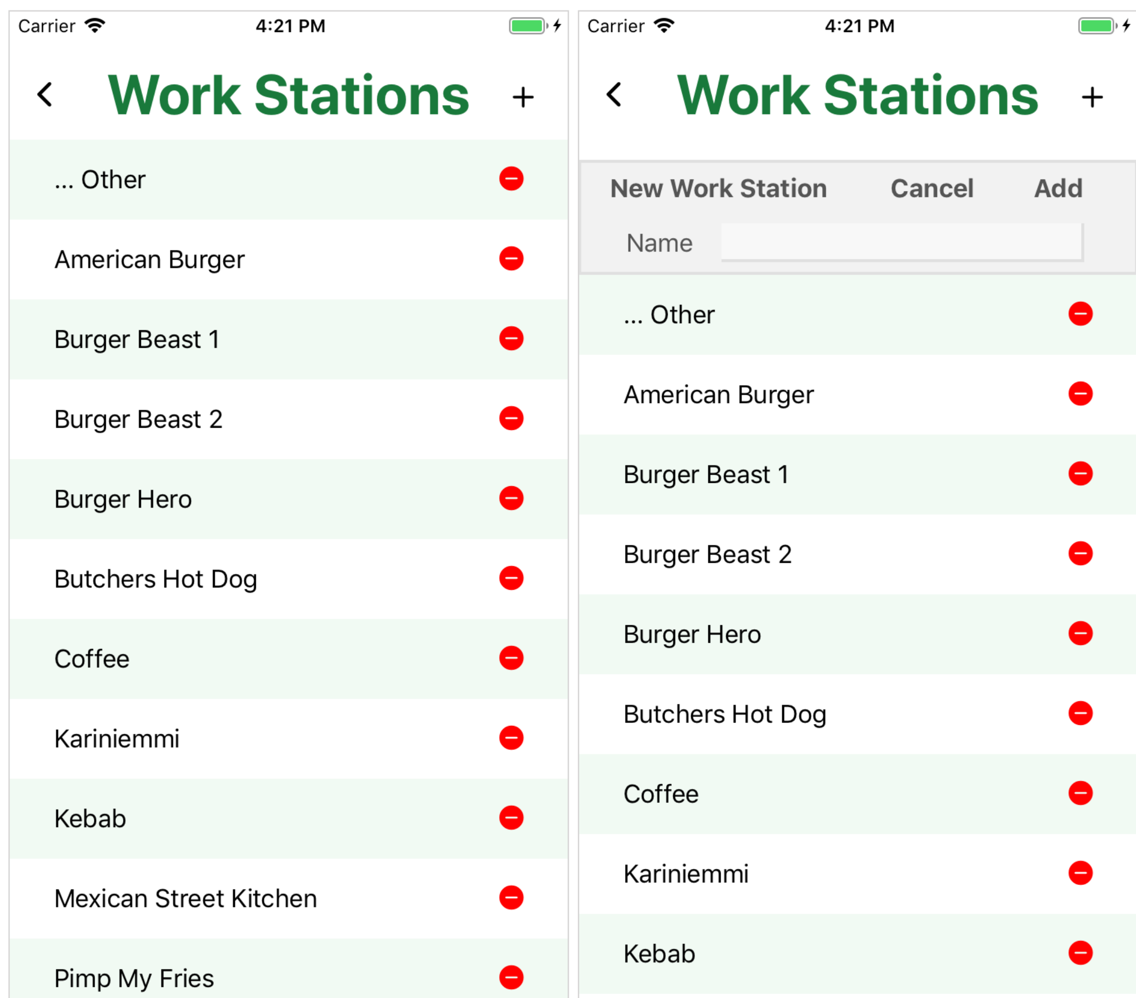
PICTURE 12. Admin Events Screen: normal view (left) and add view (right).

2.13. Work Stations Screen

(PICTURE 13) Work Stations Screen is the same type of screen as the last one seen, Admin Events Screen. In here, the admin can see the actual work stations in the system, as well as this they can add new ones or removed others as well.

A FlatList component is the main part if the UI, showing all the different work stations names in it, and the red button for the delete option. The add button is on the top-right of the screen, it will trigger a TextInput so that the admin will be able to write the name of a new work station, similarly to Admin Events Screen.

The background is exactly the same as Admin Events Screen. It is getting the work stations list at the screen opening, and updating the screen and database after any change (addition or elimination of a work station).



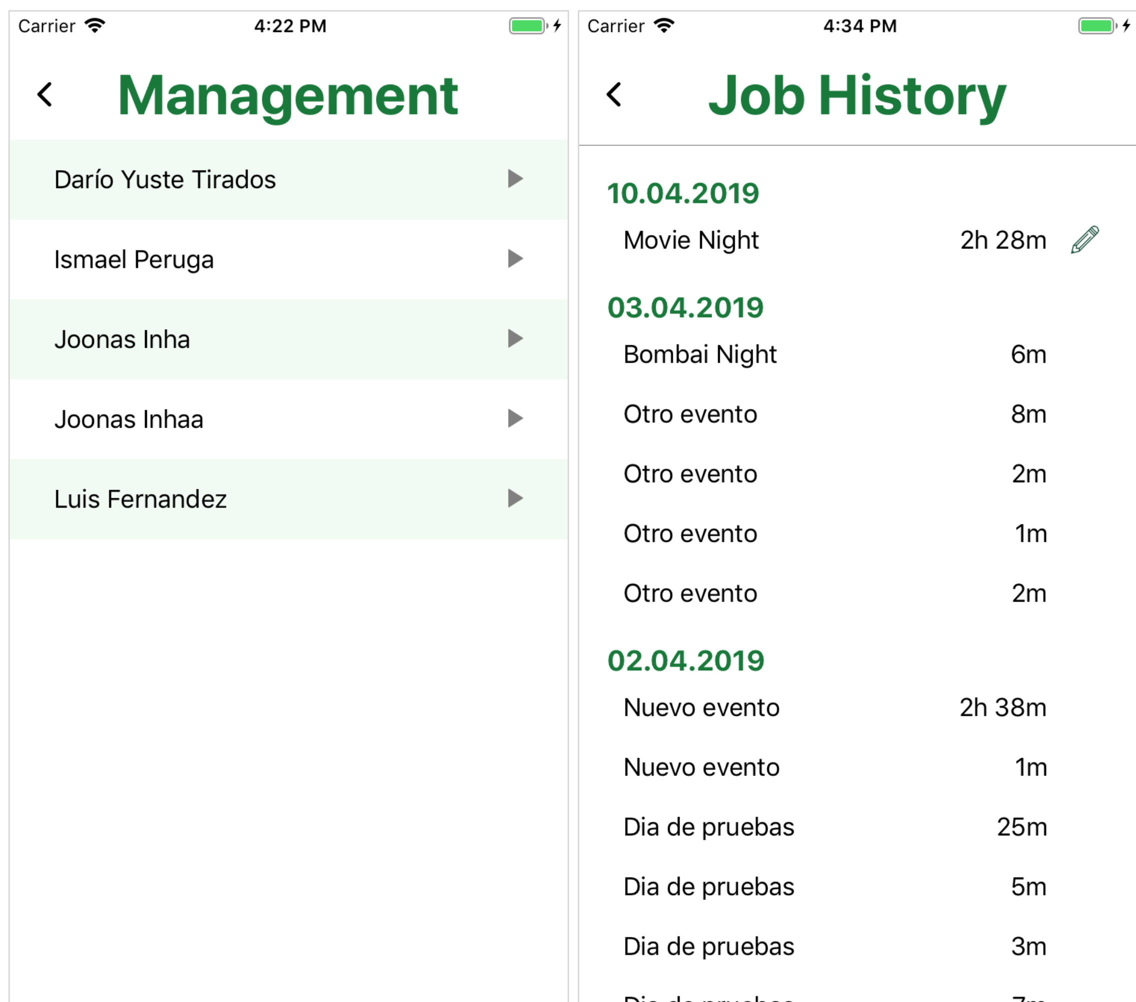
PICTURE 13. Work Stations Screen: normal view (left) and add view (right).

2.14. Management Screen

(PICTURE 14) The screen is where the supervisor can take a look of every worker in the system. In this screen only user names (with family names as well) are shown. Nevertheless, after pressing in any user name, it will jump into all the employee job history.

Once again, the main component in the user interface is a list with all the user first names and family names. Each user (FlatList's row) is a TouchableOpacity which will lead the admin to each user job history.

The only task this background must to do is to get the user names from the server database once the screen is mounted.



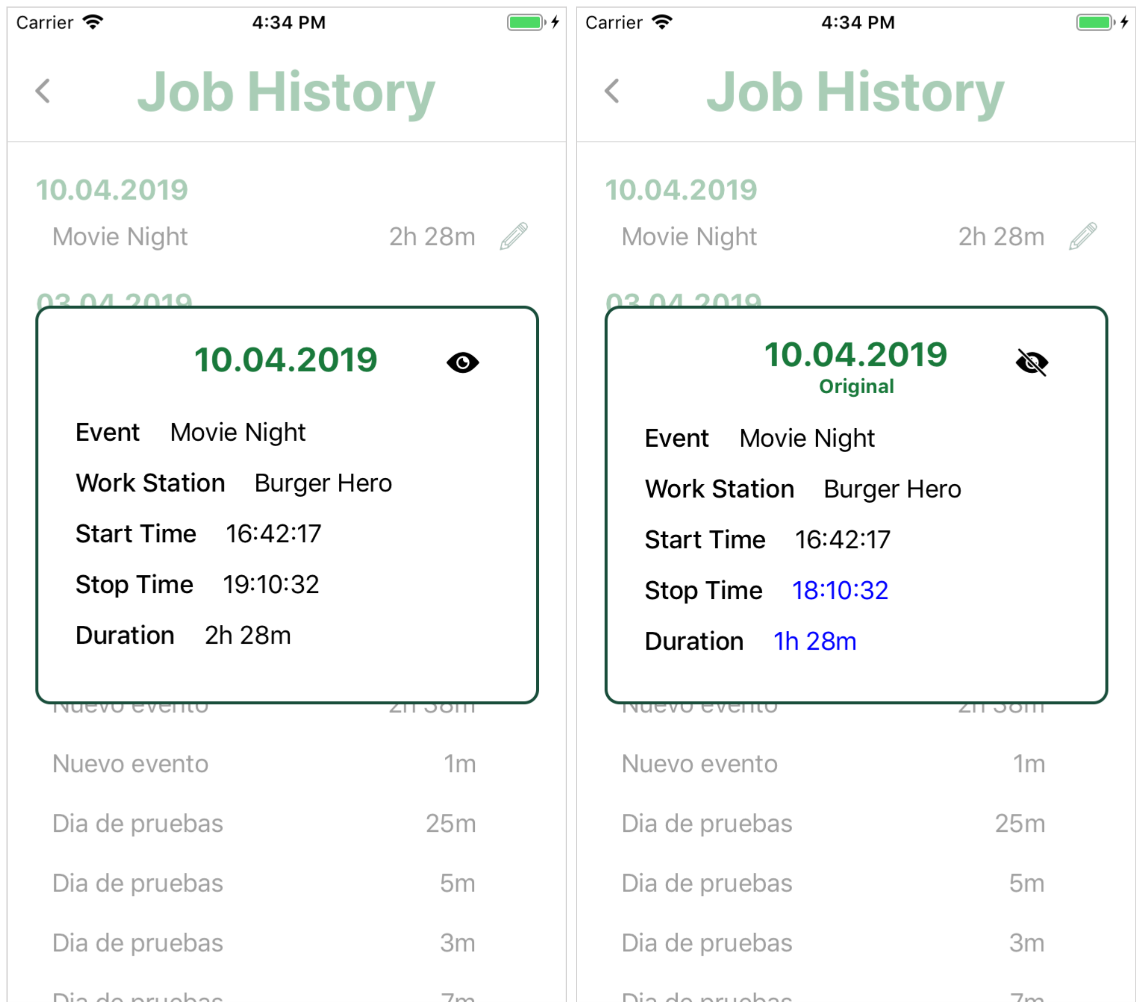
PICTURE 14. Management Screen (left) and Job History Screen list only (right).

2.15. Admin Job History Screen

(PICTURES 14 & 15) Similar to User Job History Screen, this one shows a summary of all the jobs performed by the employee selected in the previous screen. After touching one of the jobs it will show a modal with the rest of the job information. Furthermore, the admin is able to see every change made by the worker in the job data.

The UI has a list in all the screens, showing all the different job dates, and all the jobs in them. Every job entry is a TouchableOpacity, so that the admin can press it and trigger a modal with the information lacked before. This modal differs from the one in User Job History Screen. This new one doesn't have the edit mode that the other one has, but it has the option to see the original data, in case the employee had changed any information from its own screen.

The background is the same as in Job History Screen from User Stack, except for the lack of edit mode. The only action added is the possibility to show the original data. In order to perform this feature, the list of data got from the server database is analyze for showing by default the edited-name (e.g. *time_Start_Edited*) parameters if existing, if not, it will show the normal-name data.



PICTURE 15. Admin Job History Screen: modal opened with default data (left) and modal opened with original data(right).

3. CODING FRAMEWORK: REACT NATIVE

Nordic Gourmet wanted to support the two most common platforms on their app: Android and iOS, so we needed to choose between native languages, Java and Swift, which will need to be programmed in completely different codes; or choosing other coding languages that permit to create both operating systems at the same time, such as Ionic, Xamarin or React Native.
We decided to go with the second option for this, because it is easier in terms of complexity, and this specific application in not as big to need to be in native code.

3.1. Framework comparison

Xamarin is a C# framework, contrary to React Native and Ionic, which are JavaScript-based languages, JavaScript and TypeScript respectively. The main difference in the coding language is one of the reasons that we use to discard Xamarin. The other main reason was the fact that Xamarin is a fixed language as Microsoft is the owner of the language, and the only one releasing new versions (it is on its third version nowadays). This is as good as bad, because it means that Microsoft checks it before releasing a new version, but the problem at the same time is that new updates could take too long because of it. Furthermore, Xamarin needs its own development IDE (Integrated Development Environment), Xamarin Studio. In combination with all these facts we have discard Xamarin.

Ionic, can be developed in whatever IDE, such as Visual Studio Code (the one we are using), and uses TypeScript as language, which is an advantage. Nevertheless, Ionic doesn’t use native modules or views, it is all programmed in HTML (web language and components) that mimics the native widgets design. This feature would be perfect for a web app, but not that good for a mobile phone app. Hence, we have discarded Ionic too.

React Native, has been the framework used to program this application. React Native uses native widgets and, equally to Ionic, can be developed in whatever IDE, in our case, Visual Studio Code. Furthermore, React Native was created by Facebook in 2013, but it has been developed and updated by the community, which made React Native a non-stop-developing framework. Another good feature of React Native is that you are able to “go deeper” in the code to write native code if needed. [2]

3.2. React Native basic operation

React Native lets you build mobile apps using only JavaScript. It uses the same design as React, letting you compose a rich mobile UI using declarative components, which are all things user see in the screen. The apps built with React Native aren't mobile web apps because React Native uses the same fundamental UI building blocks as regular iOS and Android apps. Instead of using Swift or Java, you are putting those building blocks together using JavaScript and React. [3]

3.2.1. Render method

While coding components in React Native there is always one rule: every component must have a render method. This render method is the one that shows whatever it needs in the screen, from the simplest View component, to the most complex one. Render must return something at the end of the method, otherwise, you will get “the red screen” (fatal error screen). [4]

3.2.1. Props & State

Every component has two different types of parameters: props (properties) and state. Props are set whenever the component is created, so it is said that props are parameters that the component gets from its father (it inherits them). You can access the component props from code like *this.props*. Props are static and they not change after the component creation.

State on the other hand, are the parameters that are meant to change during the app execution. You should initialize *state* in the constructor, and then call *setState* when you want to change it. Additionally, any change on the state will trigger a re-render on the component affected. [4]

3.2.2. Component lifecycle methods

Each component has several “lifecycle methods” that you can override to perform some actions at some specific times, for example, running some code whenever the component is created. There are many lifecycle methods, but the most important ones are: *constructor( )*, *componentDidMount( )*, *componentDidUpdate( )* and *componentWillUnmount( )*. [5]

3.2.3. Style

With React Native, you don't use a special language or syntax for defining styles. You just style your application using JavaScript. All of the core components accept a prop named *style*. The style names and values usually match how CSS works on the web. In addition, there is a component called StyleSheet that help to make more than one style on it, and allow these styles to be used in more than one component at the same time. You create a StyleSheet with *StyleSheet.create( )*. [4]

3.3. Common Components

3.3.1. View

The most fundamental component for building a UI, View is a container that supports layout with flexbox, style, some touch handling, and accessibility controls. View is designed to be nested inside other views and can have zero to many children of any type. It is the most used component in the entire app. [4]

3.3.2. Text and TextInput

Text is the component for showing text, and it supports nesting, styling and handling. TextInput can show text as well, but it uses to be used for handling text input from the user via keyboard. Its props provide configurability for several features, such as editing availability, auto-correction, auto-capitalization, placeholder text, and different keyboard types, such as a numeric keypad. However, the most important prop is *onChangeText*, which will trigger an event with the new text written in the component. [4]

3.3.3. Button and Touchable

Buttons are one of the most basic components of a React Native application. They offer a very basic configuration in term of styling, so we will be using react-native-elements Buttons instead of the basic ones from React Native. Still, both types of buttons support the most important prop, *onPress*, which will trigger whatever action you demand after pressing the button. Also, you can make touchable views. They have the same prop, *onPress*, having the advantage that a whole View component, and its nested components, can be a whole button. Two examples of touchable views are: TouchableOpacity and TouchableWithoutFeedback. [4]

3.3.4. FlatList

FlatList is the component we are using to show any kind of list in Nordic Gourmet. This component will receive inside its props an array of data, *data*, from the one you will need to extract a key for every datum on the array, using *keyExtractor*. Every datum of the data array will trigger a render method for itself, *renderItem* prop method. This method can render a single View and a Text for each datum (for a simple display), or even another FlatList on it (for more complex display). Moreover, FlatList component has the *extraData* prop that allows you to re-render the FlatList every time that data in extraData changes. Otherwise the FlatList doesn’t know when to re-render itself.

3.3.5. Modal

The Modal component is a simple way to present content above an enclosing view. It could also be known as a pop-up, because these kinds of components used to appear after pressing a button or triggering some new feature. In our particular case, we use modals in order to give the user more customized alerts, to make a pop-up-menu for selecting options, or to show some information. [4]

3.3.1. Icon

In this case, the icons showed in the app are not from a React Native basic component. They are from another package: react-native-ionicons, which takes the design of the Ionic icons. These component accept 3 basic props: *name*, that identifies one icon among all of them, *size* and *color*. [6]

3.4. Stack Navigator

In most of the mobile applications, the user can see more than one screen, and not only that, the user themselves is able to move between them, pressing one button to navigate to a new screen from the main screen, and probably, pressing a back button for coming back to the main screen (as a basic example). React Native doesn't have that idea built in, and that is why we are using React Navigation.

React Navigation's stack navigator provides a way for your app to transition between screens and manage navigation history. Firstly, you need to create the stack with *createStackNavigator( )*, adding all the screens you need on it, and after that you will have one more component to use, but a bit more complex than others. This component admits many configurations, such as header on the screens, the main screen of the stack, or the way to make transitions between screens. To move between screens, we use a prop that every screen inside the stack will have, *navigation*, which have the method *navigate( )* (you access it like *this.props.navigation.navigate( )*). In this method, you write the

name of the screen you want to navigate to. Additionally, you are able to pass parameters through screens in case you need to. [7]

You can have one or more stack navigators in your app, personally we are using two different stack navigators: User Stack and Admin Stack.

3.5. Local Storage

Sometimes, you might need to store some information locally on the user device, so you can read that data later. For these occasions, we are using AsyncStorage, a React Native class, a simple, unencrypted, asynchronous, persistent, key-value storage system that is global to the app. With it you can read, write, update (merge) or remove data from the local storage of the user device. As an example, you can store a value: “Dario is a student in TAMK”, with a custom key: “Dario”. Once it is stored you will be able to read the value by searching the key inside the storage. [4]

It is important to say that all the data saved in this type of storage must be formatted before keeping it, and it will be necessary to parse (re-format the data to its original state) after reading it from the storage. In order to do it we use JSON (JavaScript Object Notation) methods: stringify (*JSON.stringify( )*) and parse (*JSON.parse( )*). The first one will convert the entry to a whole string, and the second one will revert that changes, converting the string into its original type.

In our case, we have created a class called LocalStorage in order to get rid of any usage of AsyncStorage throughout the code (this is recommended by React Native Docs). In this class, we have created some methods of AsyncStorage, but only read, write and remove.

3.6. Listeners. EventRegister

Inside a mobile application you might want to change the screen, or update data on it, because an action had been performed (event), and this can be done with the component state. However, these changes can be caused from different sources than the state. We will need something listening for that specific changes. These are called listeners.

We use the package react-native-event-listeners and its class EventRegister, which allow us to create new event listeners with a name as identifier and emit events to those identifiers. Also, listeners should be removed at the end of the life of the component which have them (inside *componentWillUnmount( )* lifecycle method), otherwise they can produce memory leaks. Every listener created with this package will have a callback that will be triggered every time that listener receives a new event.

Listeners are very useful in combination with push notifications, so whenever a notification arrives, after analysing the data, an event can be sent throughout the app, and the current mounted component has a listener for that type of event, it will trigger the callback of it, performing whatever action in that screen, commonly, updating it with new data.

3.7. Responsive screens

Due to the amount of different mobile phones sizes, not all the screens have the same amount of pixels, so in order to make the app compatible with every screen size, we need to make the UI responsive. This means that the measurements used for widths, heights, font sizes, etc. cannot be absolute, they must depend on the device screen size, as a percentage. The problem is that not all the React Native style properties accept the percentage value, they need dp (independent pixels) as a value, which would be absolute. We need then to develop something to calculate the amount of dps needed from a percentage given.

This kind of thing could be tedious to develop that by ourselves, but thanks to the react-native-responsive-screen we don’t need to. This package has two main methods: *widthPercentageToDP( )* and *heightPercentageToDP( )*, which allows us to create responsive sizes in our StyleSheets, so that every view or text will look the same on different phones with non-identical screen size. [8]

4. BACKEND SERVER: FIREBASE

As with many other mobile applications, we need some backend running in order to make everything work. Nordic Gourmet mobile app connects users and admins (employees and employers). All the data that the admin creates, such events or work stations, must be seen from the user side of the app. On the other hand, all the data the users produce, such new accounts or jobs done, must be seen from the admin flank.

For this application we decided to use Firebase for all of this support. Firebase will be the one running our databases on the cloud, sending push notifications whenever they are needed, or managing the authentication of the employees/ employers. Firebase is built on Google infrastructure and scales automatically, for even the largest apps. [9] Firebase came up because of its free fee for normal applications (not so big apps) and because of all its capabilities. Although, that was not all. React Native and Firebase had been merged thanks to React Native Firebase, a community which have created a module able to implement all the functionality and capabilities of Firebase for React Native apps. [10]

Thanks to Firebase, we have been able to solve the authentication problem at once. This is because security is serious issue nowadays, and implementing an authentication system by yourself could be a huge problem. With Firebase you simply choose the method to authenticate users in the app, implement it in the code, and ask the user for credentials (create accounts and login into them).
Similar to authentication, we have our database allocated in Firebase, which allows us to have all the data interconnected between users and synced. Furthermore, we don’t need to worry about database scaling, Google does it for us.

Another feature used is the cloud functions which are functions that can be run in the Firebase servers, completely like a custom server, on our own backend. These functions can be called from code, or can be triggered due to some change inside Firebase app (database, new user, etc.).

Finally, we are using Firebase push notifications in order to update some information in the app, just in case the worker has the app already open, so that he cannot see new data until close it and reopen it (the data is get from the server at the start of it, or at the opening of some components). For this reason, we will send push notifications without alerting the user, just to change the data, receiving that notification “under the hood”.

5. AUTHENTICATION

As we have already seen in the Authentication Screen section, the worker will be able to create a new account, and login to it after that. Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, email, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more. [11] We are using email authentication in our app, in combination with a password.

This method authenticate users with their email addresses and passwords. The Firebase Authentication SDK provides methods to create and manage users that use their email addresses and passwords to sign in. Firebase Authentication also handles sending password reset emails. [11]

After the account creation, the user will be created in the Firebase servers, and more importantly, it will create a unique identifier for the user (uid) in your app, which is very useful when creating database entries such as the user profile document, or any ID for the user inside the database. After a successful sign in, we are able to access the user's basic profile information and control the user's access to data stored in other Firebase products (e.g. restrict his access to some collections in the database). [11]

The implementation of this feature has been really easy thanks to React Native Firebase (RNF) [10]. For this we have three methods:

- *singUpUser( )*, associated with RNF method *createUserWithEmailAndPassword( )*, which needs two parameters to create the account: an email and a password. Email doesn’t need to be a validated one, but if it is not, when the supervisor tries to send any information to the employee, the communication will fail. Password, on the contrary, has some restrictions, including at least 6 characters, 1 number and 1 capital letter. Moreover, our method also creates a new entry on the database, a new User document, with all the worker information given while creating the account (name, family name and email).

- *logInUser( )*, associated with RNF method *signInWithEmailAndPassword( )*, which needs email and password as well, it is to check credential and see whether user is register in the system or not.

- *logOutUser( )*, associated with RNF method *singOut( )*, which just logs the user out.

6. DATABASE

As we have explained before, we are using Firebase as our app database too. For more details, we are using *Cloud Firestore*, their next generation of *Realtime Database*. This new version has the advantage to automatically scale and offer more powerful queries.

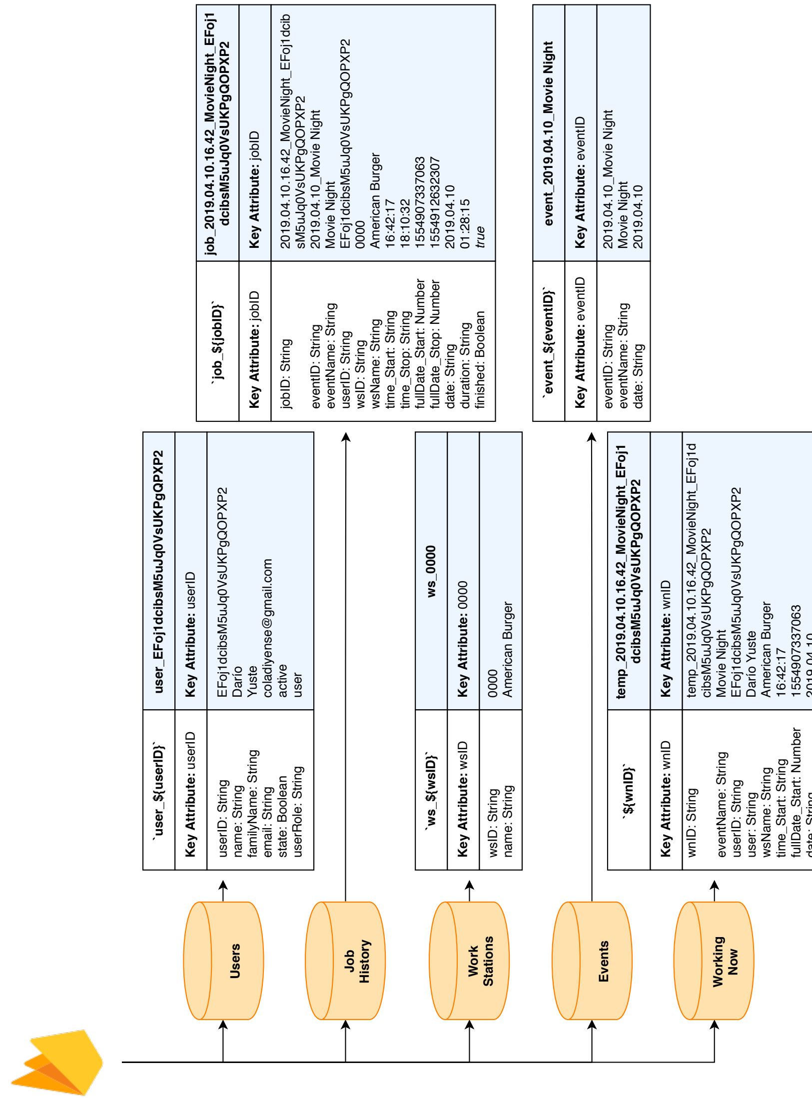
Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud Platform. Like Firebase Realtime Database, it keeps your data in sync across client apps through realtime listeners and offers offline support for mobile and web so you can build responsive apps that work regardless of network latency or Internet connectivity. Cloud Firestore also offers seamless integration with other Firebase and Google Cloud Platform products, including Cloud Functions. [12]

Could Firestore is a NoSQL type database, and it is organized in Collections (directories of a normal computer), and Documents inside them (files inside the directories). Furthermore, each Document could have another Collections inside it and so on. Documents must have a unique name in the collection, otherwise, the creation of a document with the same name as other already created will cause the removal of the previous one. Collections and Documents can be created, updated (merge information), and deleted.

The database can be accessed from code as a document path in a computer file system. The start of the database is the root directory “/”, and from it starts the rest of the database. For example, to find a document called “Dario” inside the collection “Users”, we will need to look into “/Users/Dario” from the application code.

Code examples of every collection can be seen in the Appendix 1. Database management methods.

6.1. Database design



PICTURE 15. Server database design and examples.

Our database will have five different types of data, called collections in Firebase Could Firestore argot: Users, Job History, Work Stations, Events and Working Now. Each collection will have information stored about our users, events, etc. in different documents. In the picture 15 we can see the database model: the five collections, the structure of each document on them and an example with real data on it. Every document has name (the first text box), a key attribute to identify that specific item among the whole database (second text box), and the rest of attributes (third text box).

6.2. Users

File Name → It is made with a prefix, “user_”, and the userID.

Key Attribute → The userID is automatically generated by Firebase, and it is made by alphanumeric characters.

Attributes → It stores the name, family name and email as user information, which has been filled out by the user themselves. Also, it stores the user role, which can be “user” or “admin”; and the user state, that can be “active” (for user that can already use the app), “waiting” (for that employees that are waiting for an admin response), and “rejected” (for those users that had already receive the admin response, but it was the “reject user” response).

The functions associated to this collection are able to retrieve any user property, using an ID passed through method parameters, or taking the current logged user ID (from the *auth* module). We have some methods to get all the different types of users (*active*, *waiting* or *rejected*), using queries inside the database. Also, we have setters for the collection, being able to set a whole new user from scratch or set a user property (it needs an ID, from parameters or from the current logged user by default).

6.3. Work Stations

File Name → It is made with a prefix, “ws_”, and the wsID (Work Station ID).

Key Attribute → The wsID is made with 4 numbers, depending on the last ID in the database. In order to do it you get the largest ID and then it adds +1. It is only 4 numbers because there will not be many work stations in the system.

Attributes → In this particular case we only store the name of the work station.

The methods for this collection are very basic, being able to get all the work stations, create a new one, or remove an existing one.

6.4. Events

File Name → It is made with a prefix, “event_”, and the eventID.

Key Attribute → The eventID is made in combination of the event name and the date it takes place. First the date, separated by dots each parameter of the date (*year.month.day*), and then the event name. Both attributes separated by an underscore.

Attributes → The only two attributes stored are the name of the event and the date of it.

Similar to the Work Station methods, these methods are able to create and remove an event, but they has three different ways to get events: get all the events, get the event happening on the current date, or get the further events (date above current date).

6.5. Job History

File Name → It is made with a prefix, “job_”, and the jobID.

Key Attribute → The jobID is the complex of the IDs. It is created with event information (date and name), user information (userID) and the current time. First the date of the event is separated by dots. After that, the current time is separated by dots as well, and in between date and time there is another dot. Then the rest of parameters separated by underscores, which are event name and userID at the end.

Attributes → Job History documents are the heaviest ones due to the amount of information they store. It stores event and work station information, ID and name, in order to identify the job location. It stores the userID to know who did it. In addition to this, it stores all the information related to the job duration: start

time and its full date (the value in milliseconds of a Date object), the stop time and its full date, the date, and the total duration of the working time. At last, it stored a boolean called finished that indicates whether the job is the one running (the employee is working on it at the moment) or the rest of jobs already finished.

Job History functions are the most complex, because it is the only collection that supports updating information while keeping the original in order to compare the data in some occasions. There are getters to get all the job history of a worker (user and admin mode, they are different), the total amount of hours worked by an employee, or get a single job. The rest of the methods can create a new job from scratch (setter), finish the current job the employee is working on, and update an existing job.

6.6. Working Now

This collection is a bit different from the others, all documents in it are temporary, and they only exist while the jobs are running (*finished* boolean equal to *false*). They are created when a job starts, and they get deleted whenever the user or admin finishes that job.

File Name → It is made only with wnID (Working Now ID).

Key Attribute → The wnID is made with a prefix, “temp_”, and the jobID that is being identified.

Attributes → The attributes stored in these documents are part of the job that is being identified. Event and work station names (we don’t need the ID here because they will only be used as a temporary collection, only for showing data, we don’t use it for other operations) as job location information. The userID and the user name is for identifying the worker working on the job. And, of course, the job information that exists till now: start time and its full date, and the date.

Similarly to Events and Work Stations, the methods associated to this collection can create and remove a new document, and they can get all the jobs running, or just one from a particular user.

7. CLOUD FUNCTIONS & PUSH NOTIFICATIONS

In many mobile applications that use data from a server, such as our database, sometimes they have outdated information because the database has changed in the server, but the app didn't request this new information, it is running with the old one. In these cases, we have two different solutions (apart from closing and reopening the app). Making the app to request data from server periodically, which can be useful if the data is constantly changing. However, this will not happen in our app. Changes will be frequent but not constantly. Thus, we are using a second option, paying attention for changing in the database and send those changes to the user via push notifications. To achieve this, we will use a combination between Firebase Cloud Functions, which lets you automatically run backend code in response to events triggered by Firebase features (database changes in our case) and HTTPS requests [13]; and Firebase Cloud Messaging (FCM), which is a cross-platform messaging solution that lets you reliably deliver messages at no cost. [14]

7.1. Notification type

While talking about notifications, people uses to think only in the banners or messages they can get in their screens. Nevertheless, there are more than those notifications. Inside FCM we can distinguish two types of notifications: notifications messages or data messages. On the one hand, the first ones are the most common ones among people thoughts, they alert the user with an on-screen message. On the Other hand, the data messages cannot be seeing by the user, they occur under the hood of the app. With these notifications, the server can send new data in order to do some action according to it: refresh the screen, add new data to a list, delete existing data, etc.

In our app we will be using the second type of FCMs, the data messages. Yet, before using FCMs in the app, we will first need to create a certificate for iOS devices, an APN (Apple Push Notifications) Certificate specifically [15] and load it into the Firebase Console [16].

7.2. Cloud Functions

Using Cloud Functions we can detect changes in our database: a new document has been created, updated or deleted. Furthermore, there is another method that combines all of them: *onWrite( )*. This method is the one used in almost every server functions.

In general terms, we are listening to any new creation of a document in Users collection, which means that a new user has created an account. In Events and Work Stations collections we are listening for both, creation and elimination, of any document, which means a new item has been added to the list (or deleted from it). Lastly, we are listening for every new creation or elimination at Working Now collection, which means that an employee has started, or stopped, a job. (This functions can be seen on the Appendix 2. Cloud Functions)

After the creation or removing of the document the server itself will send a notification to a specific topic (group of devices connected that will receive the same notification sent, it works like a subscription to a newspaper). The notification will be a data message, as we previously explained, with the new document data (or the previous data if the document has been removed) as a message payload. Finally, notifications must be handled in the client application, which can be seen how to handle them in Appendix 3. Push Notification handle.
All in all, we have managed to update the worker or the administrator screen while they have the app open.

8. MULTIPLE LANGUAGES

Nordic Gourmet, with as many other mobile applications, supports more than one language. In fact, it supports 3 languages: English as the default one, Finnish because it is for a company in Finland, and Spanish because I (Darío) am Spanish.

In order to support this feature we have used a package called react-native-localization. Thanks to it we are able to make an object of objects, in which the main object is the one we will call in the code, and the rest of the objects are the languages used (e.g. en, es, fi). Each language will have the same keys of the key-value pairs, but have different values. (e.g. *title_Profile: "Profile"*). Therefore, you will refer the string with its key while coding (in combination with the main object name: <mainName>.<key>, e.g. *Strings.title_Profile*), and depending on which language the user is using, it will show different strings (values).

An example can be seen in Appendix 4. Multiple languages example.

9. REFERENCES

[1] Nordic Gourmet, 2018. https://nordicgourmet.fi

[2] Cruxlab. *Xamarin vs Ionic vs React Native: differences under the hood*, 2018. https://cruxlab.com/blog/reactnative-vs-xamarin/

[3] Facebook Inc. *React Native (v 0.59), Build native mobile apps using JavaScript and React*, 2019. https://facebook.github.io/react-native/

[4] Facebook Inc. *React Native (v 0.59), Docs*, 2019. https://facebook.github.io/react-native/docs/tutorial

[5] Facebook Inc. *React (v 16.8.6), React.Component*, 2019 https://reactjs.org/docs/react-component.html

[6] Ionic. *Ionicons (v 4.5.5), Beautifully crafted open source icons*, 2019. https://ionicons.com

[7] React Navigation Core Team, *React Navigation (v 3.0), Docs*, November 17, 2018. https://reactnavigation.org/docs/en/hello-react-navigation.html

[8] Tasos Maroudas. *Build responsive React Native views for any device and support orientation change* , June 13, 2018. https://medium.com/react-native-training/build-responsive-react-native-views-for-any-device-and-support-orientation-change-1c8beba5bc23

[9] Firebase Team. *Firebase helps mobile app teams succeed*, October 29, 2018. https://firebase.google.com

[10] React Native Firebase Team. *React Native Firebase, Simple Firebase integration for React Native*, 2019. https://rnfirebase.io/docs/v5.x.x/getting-started

[11] Firebase Team. *Firebase Docs, Firebase Authentication*, March 27, 2019. https://firebase.google.com/docs/auth/

[12] Firebase Team. *Firebase Docs, Could Firestore*, January 31, 2019. https://firebase.google.com/docs/firestore/

[13] Firebase Team. *Firebase Docs, Cloud Functions for Firebase*, January 23, 2019. https://firebase.google.com/docs/functions/

[14] Firebase Team. *Firebase Docs, Firebase Could Messaging*, March 29, 2019. https://firebase.google.com/docs/cloud-messaging/

[15] Ankush Aggarwal. *Generate APNs certificate for iOS Push Notifications*, July 18, 2016. https://medium.com/@ankushaggarwal/generate-apns-certificate-for-ios-push-notifications-85e4a917d522

[16] Anum Amin. *React Native: Integrating Push Notifications using FCM*, July 15, 2018. https://medium.com/@anum.amin/react-native-integrating-push-notifications-using-fcm-349fff071591

APPENDICES

Appendix 1. Database management methods

Appendix 1.1. User

```
import firebase from "react-native-firebase";

export default class User {
  /* GETTERS */
  getUserID = async () => {
    return await firebase.auth().currentUser.uid;
  }

  getUser = async () => {
    const uid = await this.getUserID();
    return await firebase.firestore().collection('Users')
      .doc(`user_${uid}`)
      .get().then((snapshot) => {
        return snapshot.data();
      }).catch(error => {
        alert(error.message);
      });
  }

  getUserDisplayName = async () => {
    return await firebase.auth().currentUser.displayName;
  }

  getUserProperty = async (property, userID) => {
    if (!userID) {
      userID = await this.getUserID();
    }
    return await firebase.firestore().collection('Users')
      .doc(`user_${userID}`)
      .get().then((snapshot) => {
        return snapshot.get(property);
      }).catch(error => {
        alert(error.message);
      });
  }

 getNewUsers = async () => {
    return await firebase.firestore().collection("Users")
      .where("state", "==", "waiting")
```

```
    .get()
    .then(data => {
      var users = [];
      data.docs.forEach(doc => {
        users.push(doc.data());
      });
      return users;
    })
    .catch(error => {
      alert(error.message);
    })
}

getRejectedUsers = async () => {
  return await firebase.firestore().collection("Users")
    .where("state", "==", "rejected")
    .get()
    .then(data => {
      var users = [];
      data.docs.forEach(doc => {
        users.push(doc.data());
      });
      return users;
    })
    .catch(error => {
      alert(error.message);
    })
}

getActiveUsers = async () => {
  return await firebase.firestore().collection("Users")
    .where("state", "==", "active")
    .where("userRole", "==", "user")
    .get()
    .then(data => {
      var users = [];
      data.docs.forEach(doc => {
        users.push(doc.data());
      });
      users.sort((a, b) => a.name.localeCompare(b.name))
      return users;
    })
    .catch(error => {
      alert(error.message);
    })
}
```

```
/* SETTERS */
  setNewUser = async (data) => {
    const uid = await this.getUserID();
    const ref = firebase.firestore().collection('Users')
      .doc(`user_${uid}`);
    this.setUserDisplayName(data.name);
    firebase.firestore().runTransaction(async t => {
      t.set(ref,
        {
          userID: uid,
          name: data.name,
          familyName: data.familyName,
          email: data.email,
          state: "waiting",
          userRole: "user"
        });
    }).catch(error => {
      alert(error.message);
    });
  }

  setUserProperty = async (property, value, userID) => {
    if (!userID) {
      userID = await this.getUserID();
    }
    const ref = firebase.firestore().collection('Users')
      .doc(`user_${userID}`);
    if (property === "name") {
      this.setUserDisplayName(value);
    }

    firebase.firestore().runTransaction(async t => {
      t.update(ref,
        {
          [property]: value,
        });
    }).catch(error => {
      alert(error.message);
    });
  }

  setUserDisplayName = (name) => {
    firebase.auth().currentUser.updateProfile({
      displayName: name,
    })
  }
}
```

Appendix 1.2. Work Stations

```
import firebase from "react-native-firebase";

export default class WorkStations {
  /* GETTERS */
  getAll = async () => {
    const documents = await firebase.firestore()
      .collection("WorkStations").orderBy("name").get();

    var workStations = [];
    documents.forEach((job) => {
      workStations.push(job.data());
    });

    return workStations;
  }

  /* SETTERS */
  setNew = async (data) => {
    // Get the last ID in the system.
    const collectionRef = firebase.firestore().collection("WorkStations");
    const lastDoc = await collectionRef
      .orderBy("wsID", "desc").limit(1).get();
    var lastID = null;
    if (!lastDoc.empty) {
      lastID = lastDoc.docs[0].data().wsID;
    }

    // Create new ID. If none before = 0. Else lastID + 1
    var id = (lastID === null ? 0 : parseInt(lastID) + 1).toString();
    // Complete ID with 0s till 4 digits.
    if (id.length < 4) {
      for (i = id.length; i < 4; i++) {
        id = '0' + id;
      }
    }

    data["wsID"] = id;
    const ref = collectionRef.doc(`ws_${id}`);
    return await ref.set(data)
      .then(() => {
        return data;
      })
      .catch(error => {
        alert(error.message);
      }); }
```

```
  /* REMOVERS */
  remove = async (id) => {
    const ref = await firebase.firestore()
      .collection("WorkStations").doc(`ws_${id}`);
    ref.delete()
      .catch(error => {
        alert(error.message);
      });
  }
}
```

Appendix 1.3. Events

```
import firebase from "react-native-firebase";
import Dates from "../AppUtilities/Dates";

export default class Events {
  /* GETTERS */
  getAll = async () => {
    const documents = await firebase.firestore()
      .collection("Events").orderBy("date").get();
    var events = [];
    documents.forEach((event) => {
      events.push(event.data());
    });

    if (events.length == 0) {
      return null;
    }
    else {
      return events;
    }
  }

  getTodayEvents = async () => {
    const today = new Dates().getCurrenDate().dateReverse;
    const documents = await firebase.firestore()
      .collection("Events")
      .where("date", "==", today).get();

    var todayEvents = [];
    documents.forEach((event) => {
      todayEvents.push(event.data());
    });
    return todayEvents;
  }
```

```
getFurtherEvents = async () => {
  const today = new Dates().getCurrenDate().dateReverse;

  const documents = await firebase.firestore()
    .collection("Events")
    .where("date", ">", today).get();

  var furtherEvents = {};
  documents.forEach((event) => {
    const eventData = event.data();
    const eventDataDate = eventData.date.split(".");
    const key = `${eventDataDate[0]}.${eventDataDate[1]}`;
    if (!furtherEvents.hasOwnProperty(key)) {
      furtherEvents[key] = {
        dateString: new Dates().getDateString(key),
        events: []
      };
    }
    furtherEvents[key].events.push(eventData);
  });

  return furtherEvents;
}

/* SETTERS */
setNew = async (data) => {
  // Get the last ID in the system.
  const collectionRef = firebase.firestore().collection("Events");

  const dates = new Dates().formatDateForStorage(
    data.date.day, data.date.month, data.date.year);
  if (dates == null) {
    return null;
  }
  // Create ID
  const id = `${dates.dateReverse}_${data.name}`;
  data["eventID"] = id;
  data["date"] = dates.dateReverse;
  data["dateFull"] = dates.dateFull;
  const ref = collectionRef.doc(`event_${id}`);
  return await ref.set(data)
    .then(() => {
      return data;
    })
    .catch(error => {
      alert(error.message);
    });
}
```

```
/* UPDATERS */
  update = async (dataToUpdate, id) => {
    const ref = await firebase.firestore()
      .collection("Events").doc(`event_${id}`);
    ref.update(dataToUpdate)
      .catch(error => {
        alert(error.message);
      });
  }

  /* REMOVERS */
  remove = async (id) => {
    const ref = await firebase.firestore()
      .collection("Events").doc(`event_${id}`);
    ref.delete()
      .catch(error => {
        alert(error.message);
      });
  }
}
```

Appendix 1.4. Job History

```
import { EventRegister } from "react-native-event-listeners";
import firebase from "react-native-firebase";
import LocalStorage from "../AppUtilities/LocalStorage";
import Dates from "../AppUtilities/Dates";
import User from "./User";
import WorkingNow from "./WorkingNow";

export default class JobHistory {
  /* GETTERS */
  getUserJobHistory = async (userID) => {
    var uid = userID;
    if (!userID) {
      uid = await new User().getUserID();
    }

    return await firebase.firestore().collection("JobHistory")
      .where("userID", "==", uid)
      .get()
      .then(data => {
        var jobs = [];
        data.docs.forEach((doc) => {
          if (doc.get("finished")) {
            // I need to ask for the changes files
```

```
          var jobData = doc.data();
          Object.keys(jobData).map((key) => {
           // Test if exist ay Edited property.
            const regex = /_Edited$/;
            if (regex.test(key)) {
              const originalKey = key.slice(0, key.indexOf("_Edited"));
              jobData[originalKey] = jobData[key];
              delete jobData[key];
            }
          });

          jobs.push(jobData);
        }
      });
      return jobs;
    }).catch(error => {
      alert(error.message);
    });
}

getUserJobHistory_Admin = async (userID) => {
  var uid = userID;
  if (!userID) {
    uid = await new User().getUserID();
  }

  return await firebase.firestore().collection("JobHistory")
    .where("userID", "==", uid)
    .get()
    .then(data => {
      var jobs = [];
      data.docs.forEach((doc) => {
        if (doc.get("finished")) {
          jobs.push(doc.data());
        }
      });
      return jobs;
    }).catch(error => {
      alert(error.message);
    });
}

getJob = async (jobID) => {
  const documents = await firebase.firestore()
    .collection("JobHistory")
    .where("jobID", "==", jobID)
    .get();
```

```
  if (!documents.empty) {
    return documents.docs[0].data();
  }
  else {
    return null;
  }
}

getUserHoursWorked = async () => {
  const uid = await new User().getUserID();
  return await firebase.firestore().collection("JobHistory")
    .where("userID", "==", uid)
    .get()
    .then(data => {
      var hours = 0;
      var minutes = 0;
      var seconds = 0;

      data.docs.forEach((doc) => {
        if (doc.get("finished")) {
          var splitTime = doc.get("duration").split(':');
          hours += parseInt(splitTime[0]);
          minutes += parseInt(splitTime[1]);
          hours = hours + minutes / 60;
          minutes = minutes % 60;
          seconds += parseInt(splitTime[2]);
          minutes = minutes + seconds / 60;
          seconds = seconds % 60;
        }
      });
      hours = new Dates().zeroComplete(hours.toFixed(0));
      minutes = new Dates().zeroComplete(minutes.toFixed(0));
      seconds = new Dates().zeroComplete(seconds.toFixed(0));
      return `${hours}:${minutes}:${seconds}`;
    })
    .catch(error => {
      alert(error.message);
    })
}

/* SETTERS */
setNewJob = async (jobData) => {
  const uid = await new User().getUserID();
  const date = await new Dates().getCurrenDate();

  // WorkStation data and Event data already in jobData
  jobData.userID = uid;
  jobData.date = date.dateReverse;
```

```
  jobData.time_Start = date.timeFull;
  jobData.fullDate_Start = date.dateFull;
  jobData.finished = false;
  jobData.jobID = `${date.dateReverse}.${date.time}` +
    `_${jobData.eventName}_${jobData.userID}`;

  // Storage locally the start time too
  new LocalStorage().setLocalStorage("currentJob", jobData)
    .then(() => {
      EventRegister.emitEvent("changeToStopScreen", jobData);
    });

  const ref = firebase.firestore().collection("JobHistory")
    .doc(`job_${jobData.jobID}`);

  firebase.firestore().runTransaction(async t => {
    t.set(ref, jobData);
  })
    .then(() => {
      new WorkingNow().setNew(jobData);
    })
    .catch(error => {
      alert(error.message);
    });
}

/* UPDATERS */
finishActualJob = async (jobID) => {
  var jobData;
  if (!jobID) {
    jobData = await new LocalStorage().getLocalStorage("currentJob");
  }
  else {
    jobData = await firebase.firestore().collection("JobHistory")
      .doc(`job_${jobID}`).get().then((job) => {
        return job.data();
      });
  }
  const date = new Dates().getCurrenDate();

  // Change Stop data, and change finished to "true"
  jobData.time_Stop = date.timeFull;
  jobData.fullDate_Stop = date.dateFull;
  jobData.finished = true;
  jobData.duration = new Dates().getTimer(jobData.fullDate_Start);

  /* Storage locally the start time too */
  new LocalStorage().removeLocalStorage("currentJob")
```

```
      .then(() => {
        EventRegister.emitEvent("changeToStartScreen");
      });
    const ref = firebase.firestore().collection("JobHistory")
      .doc(`job_${jobData.jobID}`);

    firebase.firestore().runTransaction(async t => {
      t.update(ref, jobData);
    })
      .then(() => {
        new WorkingNow().remove(jobData.jobID);
      })
      .catch(error => {
        alert(error.message);
      });
  }

updateJobHistory = async (dataToUpdate, jobID) => {
    if (dataToUpdate != null && dataToUpdate != "") {
      const ref = firebase.firestore().collection("JobHistory")
        .doc(`job_${jobID}`);

      // Get the data from the file
      const doc = await ref.get();
      const data = doc.data();
      var newJobData = data;
      var updateDuration = false;
      var deleteDuration = false;

      Object.entries(dataToUpdate).map(([key, value]) => {
        // If the new data is different from the original data.
        // Make new parameter: parameter_Edited
        if (data[key] != value) {
          const regex = /^time_/;
          if (regex.test(key)) {
            // Get "Start" or "Stop"
            const string = key.split("_")[1];
            var dateString =
              new Date(data[`fullDate_${string}`]).toString();
            dateString = dateString.replace(data[key], value);
            newJobData[`fullDate_${string}_Edited`] =
              new Date(dateString).valueOf();
            updateDuration = true;
          }

          newJobData[`${key}_Edited`] = value;
        }
```

```
          // If both datum are equal. Delete the _Edited parameter.
          else {
            const regex = /^time_/;
            if (regex.test(key)) {
              // Get "Start" or "Stop"
              const string = key.split("_")[1];
              delete newJobData[`fullDate_${string}_Edited`];
              deleteDuration = true;
            }
            delete newJobData[`${key}_Edited`];
          }
        });

        if (updateDuration) {
          // Get the duration. Take care, if it is negative => ERROR
          newJobData[`duration_Edited`] = new Dates().getDuration(
            newJobData[`fullDate_Start_Edited`] ?
              newJobData[`fullDate_Start_Edited`] :
              newJobData[`fullDate_Start`],
            newJobData[`fullDate_Stop_Edited`] ?
              newJobData[`fullDate_Stop_Edited`] :
              newJobData[`fullDate_Stop`]);
          if (newJobData[`duration_Edited`] == "-1") {
            throw new Error("Wrong-Duration");
          }
        }
        if (deleteDuration) {
          delete newJobData[`duration_Edited`];
        }

        // Update changes in server
        ref.set(newJobData);

        // Send new data to show in the UI
        Object.keys(newJobData).map((key) => {
          // Test if exists any Edited property.
          const regex = /_Edited$/;
          if (regex.test(key)) {
            const originalKey = key.slice(0, key.indexOf("_Edited"));
            newJobData[originalKey] = newJobData[key];
            delete newJobData[key];
          }
        });
        EventRegister.emitEvent("updateJobHistory", newJobData);
      }
    }
  }
```

Appendix 1.5. Working Now

```
import firebase from "react-native-firebase";
import User from "./User";
import JobHistory from "./JobHistory";

export default class WorkingNow {
  /* GETTERS */
  getAll = async () => {
    const documents = await firebase.firestore()
      .collection("WorkingNow").orderBy("time_Start").get();
    var workingNow = [];
    documents.forEach((job) => {
      workingNow.push(job.data());
    });

    return workingNow;
  }

  getUserActualJob = async (userID) => {
    if (!userID) {
      userID = await new User().getUserID();
    }
    const documents = await firebase.firestore()
      .collection("WorkingNow")
      .where("userID", "==", userID)
      .get();

    if(!documents.empty) {
      const jobID = documents.docs[0].data().wnID.replace("temp_", "");

      return await new JobHistory().getJob(jobID);
    }
    return null;
  }

  /* SETTERS */
  setNew = async (jobData) => {
    const collectionRef = firebase.firestore().collection("WorkingNow");

    // Get User name from database
    const userName =
      await new User().getUserProperty("name", jobData.userID);
    const userFamilyName =
      await new User().getUserProperty("familyName", jobData.userID);

    const data = {
```

```
      wnID: `temp_${jobData.jobID}`,
      userID: jobData.userID,
      user: userName + " " + userFamilyName,
      date: jobData.date,
      eventName: jobData.eventName,
      wsName: jobData.wsName,
      fullDate_Start: jobData.fullDate_Start,
      time_Start: jobData.time_Start
    }
    const ref = collectionRef.doc(data.wnID);
    ref.set(data)
      .catch(error => {
        alert(error.message);
      });
  }


  /* REMOVERS */
  remove = async (id) => {
    const ref = await firebase.firestore()
      .collection("WorkingNow").doc(`temp_${id}`);
    ref.delete()
      .catch(error => {
        alert(error.message);
      });
  }
}
```

Appendix 2. Cloud Functions

```
const functions = require("firebase-functions");
const admin = require("firebase-admin");

// initializes your application
admin.initializeApp(functions.config().firebase);

exports.send_WorkingNow = functions.firestore
  .document("WorkingNow/{some_document}").onWrite(change => {
    var wnData;
    var action;
    // If the file is being created
    if (!change.before.exists && change.after.exists) {
      wnData = change.after.data();
      action = "Create";
    }
    // If the file is being deleted
    else if (change.before.exists && !change.after.exists) {
      wnData = change.before.data();
      action = "Delete";
    }

    // the payload is what will be delivered to the device(s)
    let payload = {
      data: {
        title: "WorkingNow",
        action: action,
        data: JSON.stringify(wnData)
      }
    }
    admin.messaging().sendToTopic("NordicGourmet_admin", payload);
    admin.messaging().sendToTopic("NordicGourmet_user",
      { data: { title: "WorkingNow", action: action } });
    return null;
  });

exports.send_NewUsers = functions.firestore
  .document("Users/{some_document}").onCreate(snapshot => {
    // the payload is what will be delivered to the device(s)
    let payload = {
      data: {
        title: "NewUsers",
        action: "Create",
        data: JSON.stringify(snapshot.data()),
      }
    }
```

```
    admin.messaging().sendToTopic("NordicGourmet_admin", payload);
    return null;
  });

exports.send_Events = functions.firestore
  .document("Events/{some_document}").onWrite(change => {
    var eventData;
    var action;
    // If the file is being created
    if (!change.before.exists && change.after.exists) {
      eventData = change.after.data();
      action = "Create";
    }
    // If the file is being deleted
    else if (change.before.exists && !change.after.exists) {
      eventData = change.before.data();
      action = "Delete";
    }

    // the payload is what will be delivered to the device(s)
    let payload = {
      data: {
        title: "Events",
        action: action,
        data: JSON.stringify(eventData)
      }
    }

    admin.messaging().sendToTopic("NordicGourmet_user", payload);
    return null;
  });

exports.send_WorkStations = functions.firestore
  .document("WorkStations/{some_document}").onWrite(change => {
    var wsData;
    var action;
    // If the file is being created
    if (!change.before.exists && change.after.exists) {
      wsData = change.after.data();
      action = "Create";
    }
    // If the file is being deleted
    else if (change.before.exists && !change.after.exists) {
      wsData = change.before.data();
      action = "Delete";
    }

    // the payload is what will be delivered to the device(s)
```

```
  let payload = {
    data: {
      title: "WorkStations",
      action: action,
      data: JSON.stringify(wsData)
    }
  }

  admin.messaging().sendToTopic("NordicGourmet_user", payload);
  return null;
});
```

Appendix 3. Push Notifications handle

```
import firebase from "react-native-firebase";
import { EventRegister } from 'react-native-event-listeners';
[. . .]

export default class App extends Component {
[. . .]
  componentDidMount() {
    // Push Notification Listener
    this.messageListener = firebase.messaging().onMessage(msj => {
      // It need to emit a new event inside the app depending in:
      //  -> Title: "Events", "WorkStations", "NewUsers", "WorkingNow"
      //  -> Action: "Create", "Delete"
      // Data must be parsed.
      if (`${msj._data.title}_${msj._data.action}` === "WorkingNow_Delete") {
        EventRegister.emitEvent("changeToStartScreen", "");
      }
      else {
        EventRegister
          .emitEvent(`${msj._data.title}_${msj._data.action}`
            , JSON.parse(msj._data.data));
      }
    });
    [. . .]
    firebase.auth().onAuthStateChanged(user => {
      // Before login into an account, it unsubscribes from every topic.
      firebase.messaging().unsubscribeFromTopic("NordicGourmet_admin");
      firebase.messaging().unsubscribeFromTopic("NordicGourmet_user");
      if (user != null) {
        // Ask for permission and get token for FCM.
        firebase.messaging().hasPermission()
          .then((enabled) => {
            // If there are no permissions
            if (!enabled) {
              firebase.messaging().requestPermission()
                .then(() => {
                  // User has authorised. Get Token
                  firebase.messaging().getToken()
                    .catch((error) => {
                      alert("TOKEN ERROR: " + error.message);
                    })
                })
                .catch(error => {
                  // User has rejected permissions
                  alert(error.message);
                });
```

```
          }
          else {
            // User has authorised. Get Token
            firebase.messaging().getToken()
              .catch((error) => {
                alert("TOKEN ERROR: " + error.message);
              })
          }
        })

      new User().getUserProperty("userRole")
        .then((type) => {
          // After login into an account, it subscribes to the role topic.
          firebase.messaging().subscribeToTopic(`NordicGourmet_${type}`);
         [. . .]
        });
    }
    [. . .]

componentWillUnmount() {
  [. . .]
  this.messageListener();
}
```

Appendix 4. Multiple languages example

```
import LocalizedString from "react-native-localization";

export const Strings = new LocalizedString({
  // <------------------------------ ENGLISH STRINGS ------------------------------>
  en: {
    title_Profile: "Profile",
    title_Main: "Nordic Gourmet",
    title_JobHistory: "Job History",
    title_Working: "Working",
    title_Management: "Management",
    title_Events: "Events",
    title_WorkStations: "Work Stations",
    title_NewUsers: "New Users",
  },
  // <------------------------------ FINNISH STRINGS ------------------------------>
  fi: {
    title_Profile: "Profiili",
    title_Main: "Nordic Gourmet",
    title_JobHistory: "Työhistoria",
    title_Working: "Työskentely",
    title_Management: "Hallinto",
    title_Events: "Tapahtumat",
    title_WorkStations: "Työasemat",
    title_NewUsers: "Uudet käyttäjät",
  },
  // <------------------------------ SPANISH STRINGS ------------------------------>
  es: {
    title_Profile: "Perfil",
    title_Main: "Nordic Gourmet",
    title_JobHistory: "Historial",
    title_Working: "Trabajando",
    title_Management: "Gestión",
    title_Events: “Eventos",
    title_WorkStations: “Puesto de Trabajo",
    title_NewUsers: “Nuevos Usuarios",
  }
});
```