My preferred cities weather

Integrated Masters in Informatic Engineering and Computation

Mobile Computation

Edgar Carneiro - up201503784
Sérgio Salgado - up201406136

Introduction

This report has as its main objective to explain in detail the implementation of the application “My preferred cities weather”, a mobile app for weather data consultation.

The application allows the user to obtain detailed information regarding the current weather of a Portuguese district capital city. The user is also able to manage his selected cities of interest, by being able to, at any time, add new cities or remove already featured ones. Additionally, the user can also request for forecasts regarding its selected cities, thus obtaining the same detailed information for the next few hours, or even days.

The weather information presented to the user includes details such as respecting temperatures, pressure, precipitation, wind and humidity. Furthermore, images resembling the weather are also used, in order to facilitate data comprehension and make the application more enjoyable.

For obtaining the weather related information the OpenWeather API was used.

This project was created in the scope of the curricular unit Mobile Computation, which belongs to the Integrated Masters in Informatic Engineering and Computation.

Architecture

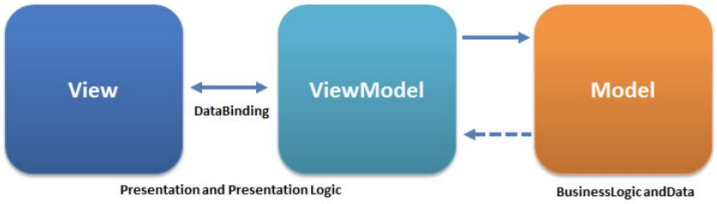
For the mobile application architecture we opted for using the **Model-View-ViewModel (MVVM)** pattern. The main purpose of this architectural pattern is to separate the development of the graphical interface from the development of the business logic.

The **Model,** in the MVVM pattern, is responsible for representing and storing the data, as well as implementing any existent business logic. The Model should also be agnostic to any concern regarding the graphical representation of the data.

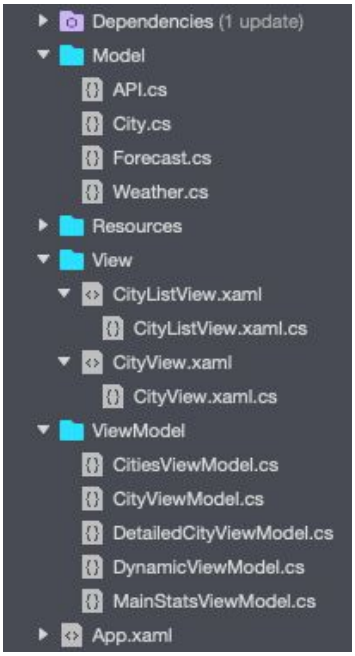
The **View,** in the MVVM pattern, is responsible for the management of the graphical interface that is presented to the end-user. Furthermore, it should also handle any given user inputs. The View should be agnostic to business logic, hence only being accountable for visually presenting the data it receives.

The **ViewModel,** in the MVVM pattern, is responsible for connecting the two previously presented architecture components. As such, it functions as state representation of the Model, that is then provided to the View through **data binding**.

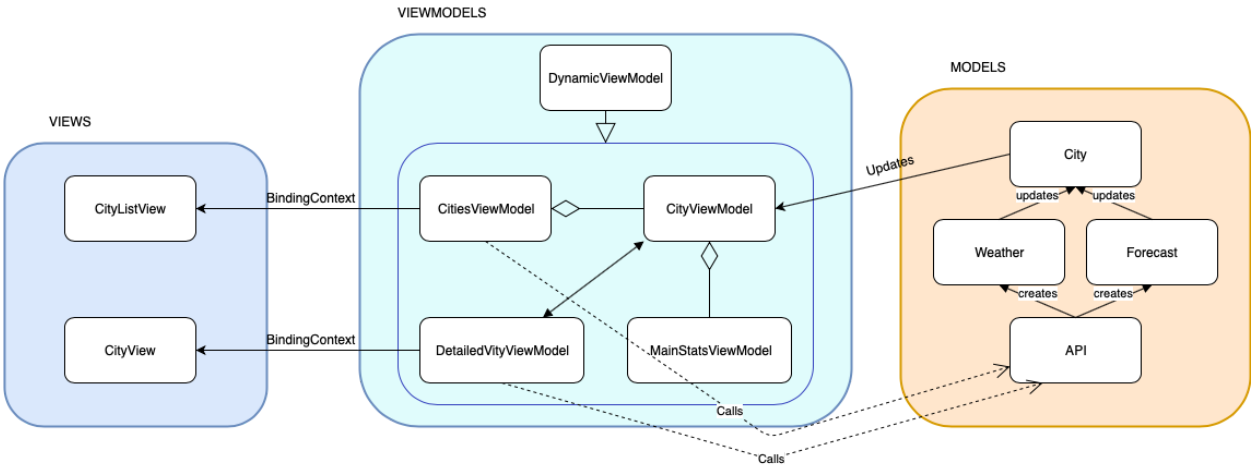
A visual illustration of this architectural pattern is presented below.



Considering the project at hand, our codebase was divided in the following manner:



A more in-depth analysis of each of the customer application's packages and respective files is presented next. Additionally, a graphic representation of our architecture is presented below:



Models

Package responsible for storing and handling the logic associated with the data used in the context of the application, using adequate representations.

Notice the natural mapping existent between the model package's class and the different requests made to the Weather API. The *Weather* class represents a data structure for deserializing the request that obtains the current weather information of a Portguese city. The *Forecast* class represents a data structure for deserializing the request that obtains the forecasts for a Portuguese city, with the three hours difference, for the next five days. The *API* class is responsible for storing data necessary for making requests to the OpenWeather API, such as the URI formats. Finally, the *City* class, as the main data aggregation structure where all the weather information regarding a City is stored, be it the current weather or the forecasts. Additionally, the *City* class also has a reference to a *ViewModel* class, so that it can notify it upon necessary updates. Thus, the MVVM model is promptly implemented.

Views

Package containing all the View classes used throughout the application. A view represents a single screen with a user graphical interface. As such, as expected, there will be an equal number of views equivalent to the number of different screens the customer can visualize.

The *CityListView* view is responsible for showing the user its cities of interest. It also allows the end-user to quickly know the current temperature and weather status for the city. Additionally, the user can also add new cities to its list of interest, or remove cities featuring in this list. The *CityView* view is responsible for showing the User the detailed weather information of a city, as well as the obtained forecasts for this city. Besides showing the city name and the current temperature, it also presents the user the main statistics - temperature difference, pressure, precipitation, humidity and wind velocity - regarding the current weather. Furthermore, it also shows the temperature and rain variations for the next eight hours, using a line chart. By scrolling down, the user can also consume the same statistics for the forecasts made in three hours intervals.

The visual aspect of all the Views can be seen further ahead in section *Functionalities*.

ViewModels

Package containing all the classes responsible for connecting the Views with the Models and for realising the actual requests to the Open Weather API. The ViewModels also handle the binding between the business logic data and the views displayed data. Furthermore, techniques for ensuring that changes in the model's data are perceived in the views data are also defined in this package.

The *DynamicViewModel* presents an abstract class responsible for notifying when changes to the binded data occur, thus invoking the necessary methods for reflecting those changes in the graphical interface. The *CitiesViewModel* is the class responsible for managing the data that is provided to the *CityListView* model. Additionally, it also contains information regarding all the possible selected cities. The request for getting a City's current weather is also its responsibility. The *DetailedCityViewModel* handles the detailed information of a city. Furthermore, it is also responsible for requesting the Forecast data to the API. The *CityViewModel* is arguably the core ViewModel for our application, since it reflects all the properties of a city that are used by the *Views* package. As such, it presents the City model with the functionality of passing it the City's current data state, so it can re-render the *Views.* The *MainStatsViewModel* represents an auxiliary data structure for storing main weather information regarding an event. An event is, for example, the current weather, or the forecast at eight in the morning of the next day. The data stored by a *MainStatsViewModel* instance includes information such as temperature difference, pressure, precipitation, humidity, wind velocity and API representation Icon. The *MainStatsViewModel* serves as a helper ViewModel to the *CityViewModel* class.

Resources

Package containing the resources used by the graphical interface. Includes all the icons used to help identify the weather statistic presented in the *Views*.

Functionalities

In this section, an overview regarding the requested and extra implemented features is presented. Notice that the extra features are marked as such and that some extra steps were taken in order to improve the already requested features.

Favorite Cities

On the first execution of the application, an empty list is shown to the user with a picker and an "Add" button besides it. When the user presses the picker, a pop-up shows up, presenting the user with all 18 district capital cities of Continental Portugal. Once the user picks one of these cities and presses the adjacent button, a new entry on the list shows up.

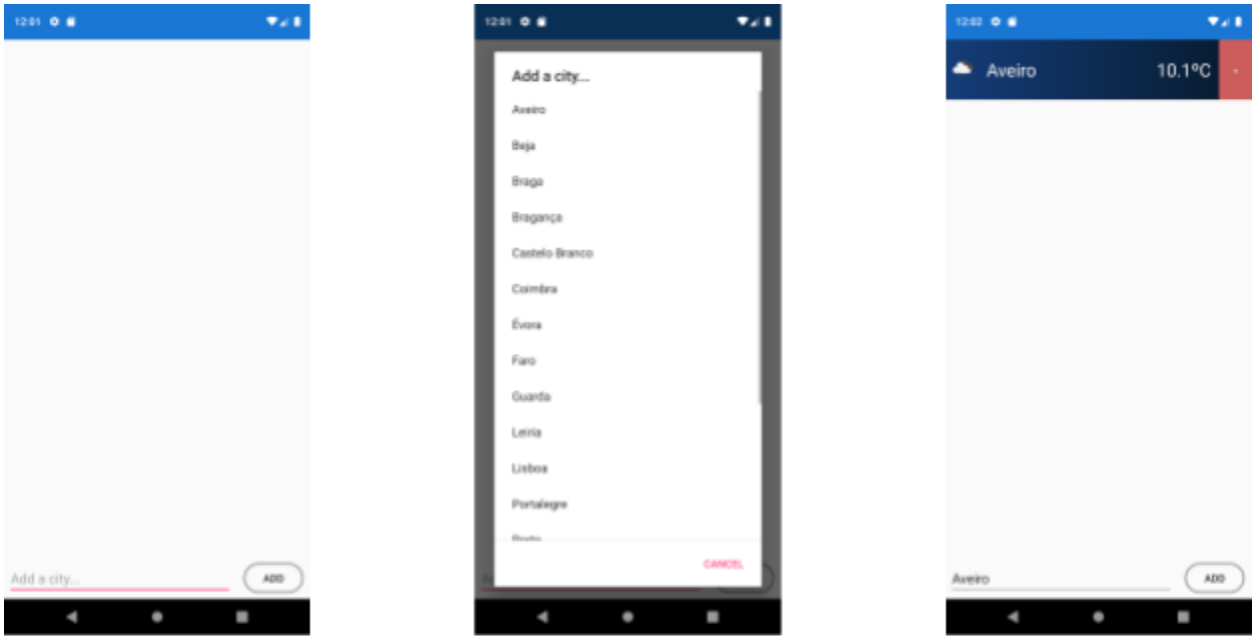
After requesting information from the OpenWeather API, this entry informs the user about quick and basic weather information on that city, such as current temperature and current weather conditions in the form of an icon retrieved from the API, for example, if it is currently raining or there's a cloudy sky. Notice that the background dynamically updates, thus adapting itself to a more adequate color that matches the City's current weather.

If the user presses the city entry, the app then navigates to the specific city screen, where more detailed information is shown to the user, giving information about a wider range of weather metrics and weather predictions for that city during the next 24 hours.

On the right side of this entry, there exists a button labeled with a "-" sign. When the user presses this button, this city entry is then removed from the favorites' list and the user can no longer consult both the quick weather details nor the more specific weather conditions (including predictions) about that same city.

Background Gradient (Extra Feature)

In order for the user to receive extra visual information about the weather in the favorite city list, a color gradient, besides serving as background, also reflects upon the current weather conditions. For example, if it is currently snowing in one of the cities, the background gradient changes its color to a clear blue to white gradient or if a thunderstorm is happening, the gradient transforms to a dark purple to black gradient.



Favorite City List screen. On the left, an empty list, with the picker highlighted at the bottom. On the middle screenshot, once the picker is pressed, a pop-up shows up allowing the user to select a city to add to the favorites. On the last image, a favorite city entry on the list showing basic information about the current weather conditions is presented with a button on the right to remove the entry from the list.

Favorite City List screen presented on our second platform, Universal Windows Platform. Screenshots show the same functionalities as the ones taken in Android and in the same order.

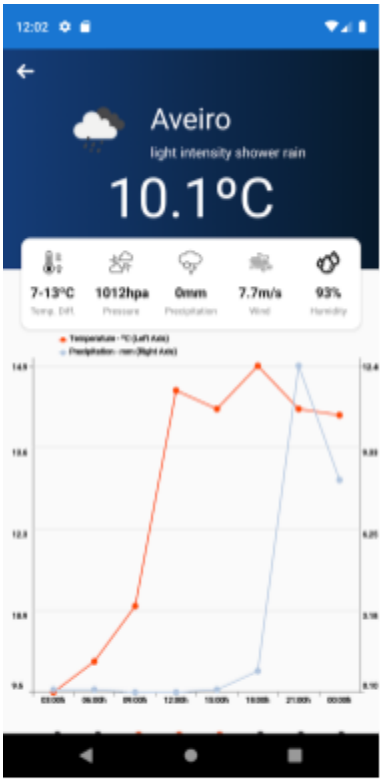
City Screen

As specified before, the city screen grants information to the user about a wider range of weather metrics. This screen can then be divided into three parts: the current day conditions, a line graph showing the temperature and precipitation predictions for that city during the next 24 hours and a more detailed list about those predictions.

The first section shows weather information for the current day, including minimum and maximum temperatures for the current day, pressure, precipitation, wind speed and humidity. Above these metrics, the current temperature in the area is shown, bundled with an icon showing the current weather conditions and a small weather description. Consistently, the background once again matches the City's current weather, having the same purpose as the one presented in the previous sub section.

On the next section of this screen, a line chart is presented to the user, presenting a more visual way of comparing the variations of temperature and precipitation metrics predicted for the next 24 hours. This chart is created with the usage of the *SkiaSharp* library, whose implementation details are addressed in the respective section of this report. Notice that the weather icon corresponding to each chart timestamp is presented below the respective timestamp. We chose to present them below the timestamp instead of next to the point in the plot, since we also have the precipitations in the chart and as such, by adding them next to the plot, we would be adding visual clutter and making it much harder to interpret the line chart.

During the last section of this screen, the user can consult detailed weather metrics similar to those contained in the first section but, instead of measurements for the current weather conditions in the city, the list contains metrics related to the weather predictions for the next 24 hours, with each entry referring to every 3-hour interval.



A city screen showing more detailed information about the current weather conditions in one of the favorite cities. On the bottom part of both screens, information about the next 24-hour predictions is displayed in both a line graph and a detailed list.

City Entry screen on UWP.

Implementation Details

In this section, the user can get knowledge regarding Implementation details. Aspects such as the line chart drawing or the API request handling are highlighted in this section.

Property Changes

For handling property changes an abstract *ViewModel* class was developed, named *DynamicViewModel*. The purpose of this class is to ensure that properties that are defined and changed in a certain manner, trigger updates of the graphical interface. This is possible since the *DynamicViewModel* defines two important methods:

```
        public event PropertyChangedEventHandler PropertyChanged;

        protected bool SetProperty<T>(ref T storage, T value, [CallerMemberName]
string propertyName = null)
        {
            if (Object.Equals(storage, value))
                return false;
            storage = value;
            OnPropertyChanged(propertyName);
            return true;
        }

        protected void OnPropertyChanged([CallerMemberName] string propertyName =
null)
        {
            PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
        }
```

For then defining properties that, when changed, automatically trigger changes in the visual interface, one simply has to respect the following structure (example using a string property):

```
        private string _FieldName;
        public string FieldName { get => _FieldName; set => SetProperty(ref
_FieldName, value); }
```

Notice that the class implementing the field must extend the *DynamicViewModel*. If a View has an element that is **binded** to the example *FieldName* property, using ***Xamarin* binding contexts**, if the property changes, the view will be automatically updated with the new value.

Requesting to the *API* & *JSON* handling

Exactly three requests are made to the OpenWeather API. Those are:

- http://api.openweathermap.org/data/2.5/weather?q={city},pt&units=metric&appid={key}
- http://api.openweathermap.org/data/2.5/forecast?q={city},pt&units=metric&appid={key}
- https://openweathermap.org/img/wn/{icon}@2x.png

Notice that the `{city}` parameter represents the Portuguese capital district, the `{key}` represents the API key and the `{icon}` represents an icon string (one can consult those strings here).

All the requests to the OpenWeather API respect the following structure (example of request for getting forecasts of a city):

```
            using (HttpClient client = new HttpClient())
                try
                {
                    HttpResponseMessage response = await
client.GetAsync(API.getForecastURL(city));
                    if (response.StatusCode == HttpStatusCode.OK)
                    {
                        Forecast apiForecast =
JsonConvert.DeserializeObject<Forecast>(
                            await response.Content.ReadAsStringAsync()
                        );

                        city.UpdateForecast(apiForecast);
                        if (view != null)
                            view.UpdateChart(); // Updating chart since it does not
work with binds

                    }
                }
                catch (Exception ex)
                {
                    Console.Error.Write(ex.StackTrace);
                }
```

Briefly explaining the code above:

- Firstly, a *HttpClient* instance is created;
- Secondly, the request to the API is made using the respective *URL*, and the response is saved. Note that in our project the *URL*s are managed by the API model.
- Next, if the response is valid, the response content, as a string, is deserialized into a *C#* object. For that extend, we use the **Newtonsoft.Json** package. It is important to state that the class used in serialization must exactly match the *JSON* structure of the received response. Two examples of deserialization classes used in our project are the Weather and Forecast models. After deserializing the response we can access it through the instance fields.
- Using the deserialized object instance, the remaining data structures are updated (in the example the City model).

SkiaSharp

The SkiaSharp library was used to draw the line chart shown to the user inside the city screen, informing the user about both temperature and precipitation variations throughout the next 24 hours, with a 3h interval between each point.

Using this library proved to be somewhat difficult at first, since it doesn't natively support graph drawing, meaning every single element of the chart (axis, guidelines, graph info, line chart and all other details) need to be drawn from the ground up, using only primitive functions offered by the library, such as drawing circles, lines and text.

In order to draw our graph, we only use two reference points, the bottom-right most point and the top-right most point of the area we want our chart to be drawn inside the created canvas. We then infer all other elements in the chart in relation to these two points.

Another important topic during the chart creation is that it scales accordingly to the minimum and maximum values of the 24 hour prediction, so that every point is contained inside the designed area. To keep the user informed about the scaling of the chart, four guidelines are shown, essentially dividing the Y-axis in four areas, which values are showing on the side of the respective axis (temperature or precipitation).

Performed Tests

The following section encompasses a detailed description of all the tests performed, namely the scenario description of the test and the respective scenario result. Any user using the application shall be able to replicate any of the tests described below.

Favorite Cities

Test Number	Scenario Description	Scenario Result
1	The user presses the picker located at the bottom of the screen.	A picker should pop up on the screen, showing a list of cities available to add to the list.
2	The user selects a city from the picker list.	After a brief moment, a new entry appears on the screen, referring to the city the user selected previously. Basic information about this city's weather is requested from the API and presented. The City's background is also correlated with the current weather.
3	The user selects a city from the picker list which has been selected before.	An alert pops up on the screen, informing the user that the selected city is already present on his favorites.
4	The user presses on a city entry added previously.	The app transfers the user to a new screen where weather information about the pressed city is requested and presented to the user.
5	The user presses the button labeled with "-" inside the city entry on the list.	That specific city is then removed from the favorite cities' list, removing the chance for quick weather preview and allowing that city to be picked again on the city picker.

City Screen

Test Number	Scenario Description	Scenario Result
1	The user wants to view current day weather metrics such as min. and max. temperatures, wind speed, pressure, etc.	On the top half of the screen, this information is displayed on a frame structure, highlighting the current day metrics. Above it, simpler information about the city status is also presented.
2	The user wants to view weather predictions about temperatures and precipitation for the next 24 hours, divided into 3h intervals and displayed in a line chart format.	A line chart with 3 axis is presented to the user, showing in detail and in a simple form the variations of temperature and precipitation that will be felt during the next 24 hours for that particular city.
3	The user wants to view weather predictions for the next 24 hours, divided into 3h intervals	A prediction list detailing each of the supported weather metrics is displayed under the line graph, in a format similar to how current weather conditions are displayed to the user, in a frame structure.
4	The user presses the left arrow icon presented at the top left corner of the page.	The user is then taken back to the favorite cities' list.

Setup & Way of Use

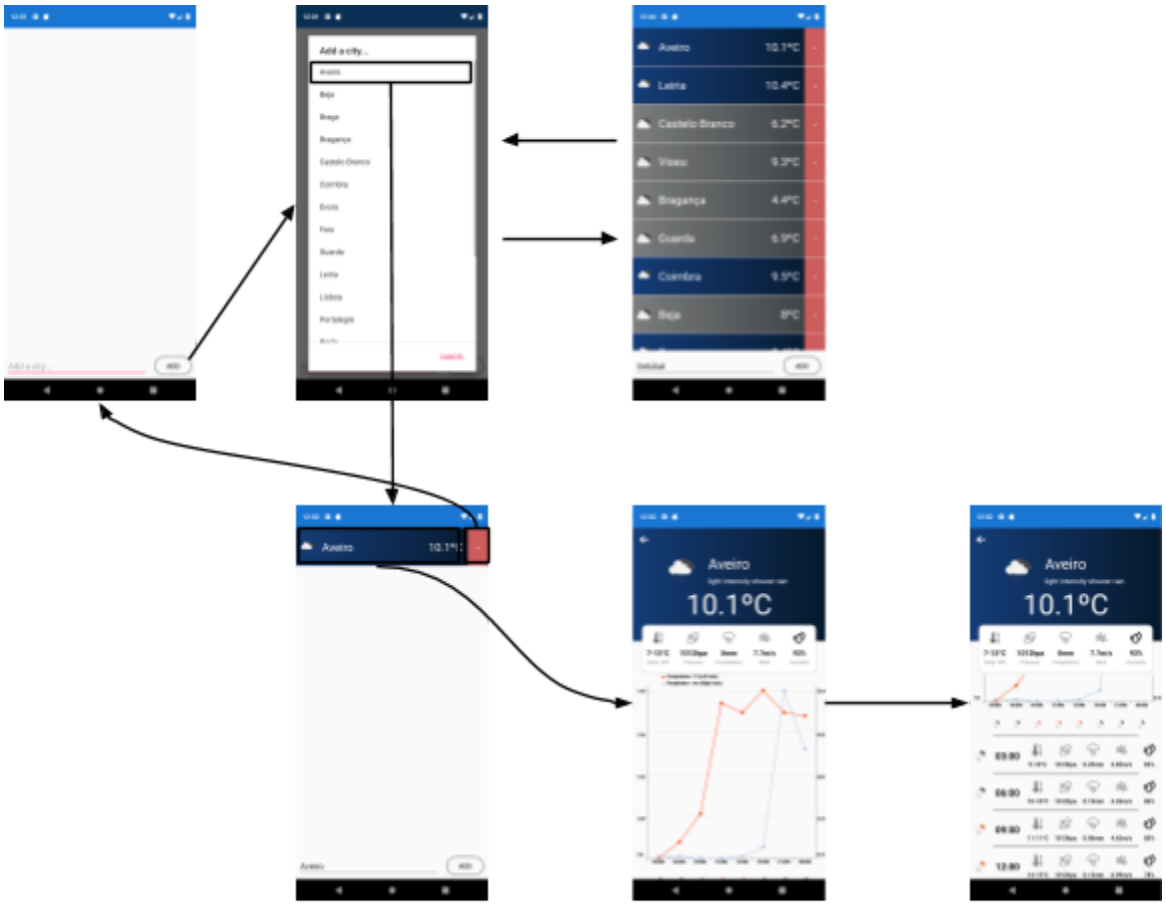
Setup

For running the mobile application one must:

1. Run the *Xamarin* project in *Visual Studio.*
2. Connect your mobile device to the machine running the editor.
3. Optionally, one can use its own OpenWeatherAPI key. To do so, change the appid value of the field *Key* in the `Model/API.cs` file.
4. Run the project in your mobile, as presented here.

Way of Use

To better understand how the application functions, a flow diagram is presented below.



Workflow of the application.

Conclusion

The group believes that the project allowed us to put into practice the skills learned in the Mobile Computation course practical classes, specially the ones regarding *Xamarin*. Additionally, skills in *C#* were obtained, namely how to properly develop an API client.

The group faced some difficulties when using *Xamarin*, mainly since we already had experience with other frameworks used in building mobile applications, such as *Flutter* and *React Native*. The difficulties emerged as a result of the advantages and disadvantages of the development in *Xamarin* over the referred frameworks.

Nonetheless, the obstacles were surpassed and the developed implementation succeeds to correctly perform on the thoroughly developed battery of test scenarios. Additionally, several extra features were developed and the group believes that those greatly improve the user experience.

To sum up, we believe our implementation of the proposed mobile application and its multiple features - both core and enhancements - were thoroughly explored and correctly developed. Hence, we believe our implementation of this project was successful.

Bibliography

In the development of this project a vast set of references were used, namely:

- Mobile Computing Page and Resources, https://paginas.fe.up.pt/~apm/CM/
- OpenWeather API, https://openweathermap.org/api
- Xamarin documentation, https://docs.microsoft.com/en-us/xamarin/
- Xamarin Forums for clarifying some doubts, https://forums.xamarin.com/categories/xamarin-forms
- NewtonSoft JSON documentation, https://www.newtonsoft.com
- Skiasharp documentation, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/graphics/skiasharp/