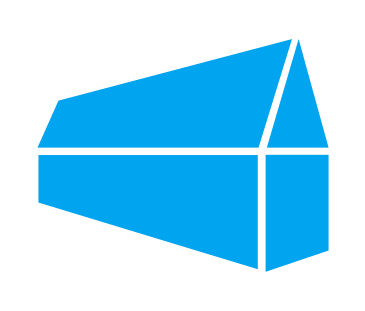
Microsoft House app

Design and Implementation of Mobile Applications

Filippo Pedrazzini, Emanuele Torelli

July 27, 2017



Contents

[1. Introduction 4](#_Toc494318044)

[2. Requirements analysis 4](#_Toc494318045)

[2.1. Microsoft guidelines 4](#_Toc494318046)

[2.2. Assumptions and requirements clarification 4](#_Toc494318047)

[2.2.1. Booking meeting rooms 4](#_Toc494318048)

[2.2.2. Parking reservations 5](#_Toc494318049)

[2.2.3. Shared calendar 5](#_Toc494318050)

[2.2.4. Login 5](#_Toc494318051)

[2.3. Use Cases 6](#_Toc494318052)

[3. Design Phase 7](#_Toc494318053)

[3.1. Architecture 7](#_Toc494318054)

[3.2. High-Level 7](#_Toc494318055)

[3.3. Low-Level 8](#_Toc494318056)

[3.3. MVVM pattern 8](#_Toc494318057)

[3.2. User Interface 9](#_Toc494318058)

[3.2.1. Mock-up 9](#_Toc494318059)

[4. Implementation 10](#_Toc494318060)

[4.1. Tools and Equipment 10](#_Toc494318061)

[4.1.1. Xamarin.Forms 10](#_Toc494318062)

[4.1.2. Azure Mobile Apps 11](#_Toc494318063)

[4.2. External Services 12](#_Toc494318064)

[4.2.1. Xamarin.Forms Maps 12](#_Toc494318065)

[4.2.2. Azure App Authentication Service 12](#_Toc494318066)

[4.2.3. Azure Notification Hub 13](#_Toc494318067)

[4.2.4. Implementation sample 14](#_Toc494318068)

[5. Issues 15](#_Toc494318069)

[5.1. Requirements 15](#_Toc494318070)

[5.2. OS Rules 15](#_Toc494318071)

[5.3. UI controls customization 15](#_Toc494318072)

[6. Testing 16](#_Toc494318073)

[6.1. Supported platforms 16](#_Toc494318074)

[6.2. Unit testing 18](#_Toc494318075)

[6.3. Integration testing 20](#_Toc494318076)

[6.4. Notification testing 21](#_Toc494318077)

[7. Conclusion 21](#_Toc494318078)

[8. Other info 22](#_Toc494318079)

# 1. Introduction

The purpose of the project was to develop a mobile application, following the guidelines that Microsoft gave us in the beginning of the semester. This document explains how we managed the work to create, just in few months, a professional application ready to be used by the employees working in Microsoft House.

# 2. Requirements analysis

## 2.1. Microsoft guidelines

Microsoft didn’t explain us in detail all the functional requirements that the app must meet; in fact, with the few slides provided (microsoft.pdf) and after exchanging some emails we had a summary idea of the main functionalities we had to implement. According to the information collected, the three key functionalities are:

* Booking meeting rooms: the app must give the possibility to the employees to see the details of the rooms inside Microsoft House and their availability, and eventually book them.
* Parking reservations: the app must show the available parking slots and allow the employees to reserve one of them.
* Shared calendar: the app must allow employees to share a calendar with all the events that will be taking place in Microsoft House.

Moreover, Microsoft asked to develop our app using Xamarin and Azure. This is not a functional requirement, but it’s an important aspect of the specifications because it influenced most of our future decisions during the implementation.

## 2.2. Assumptions and requirements clarification

In the beginning we couldn’t have a clear idea about how to meet the goals, therefore, before moving on with the development, we tried to give a clear interpretation of each main requirement, making the necessary assumptions in order to work in a consistent scenario. Here there is an explanation of the aspects we considered.

### 2.2.1. Booking meeting rooms

The problem with this first requirement was identifying which kind of users can book the various rooms, because we guess that in a working space like Microsoft House not all the employees have the same privileges. However, we decided to assume that all the employees can visualize the details of a room and, if available, book it, selecting the time and date of the reservation. Moreover, users can visualize a list of the reservations they made, and, for each room, the forthcoming reservations.

### 2.2.2. Parking reservations

There are many ways to implement this specification, some of them initially seemed reasonable but turned out to be infeasible (e.g. if we consider the GPS position to locate the user in order to automatically update the availability of the slots, then we don’t take into account employees that go to work by bike). After reasoning a lot, we reached the following solution: we assumed that at the entrance of the parking there is a QR code, which will be used by the employees. From the app, the user can see the number of available slots at any time, but he cannot reserve a slot until reaching the place. When he arrives, he can park his car or leave the park focusing the QR code by means of the app, and the number of available slots is updated accordingly.

### 2.2.3. Shared calendar

This specification was not clear whether every employee can create an event or not and which is the difference between private meetings and public events. We assumed that all the users can create a public event in one of the principal areas of the building, and such events are visible by everyone in the shared calendar. When a public event is created all the users are notified through the app. Users can also update or delete an event from the shared calendar, and a notification is sent as well.

### 2.2.4. Login

This is an implicit but fundamental requirement, since of course we don’t want to let everyone who has an installation of Microsoft House app on his device to be able to do all the things listed before. Therefore, users must be properly authenticated before exploiting the other functionalities of the app.

## 2.3. Use Cases

To explain better the functionalities in a visual way we show a use cases diagram. The application is mainly targeted to Microsoft employees, therefore, in the continuation, the word user refers to employees. However, considering the whole system behind the app, we expected that there must be also a superuser or administrator who, by means of a dedicated login (therefore not accessing directly the app), can exploit extra functionalities.

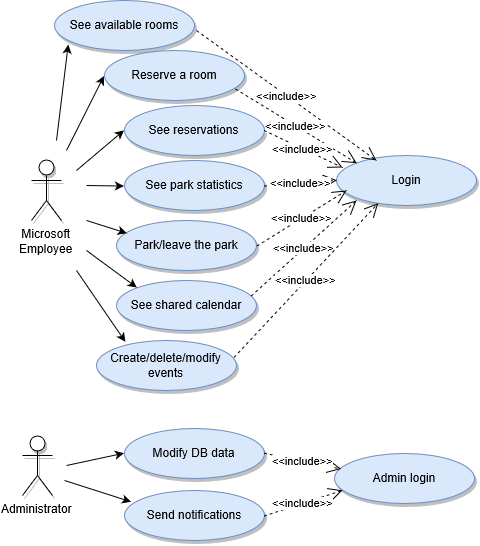


Figure 1: Use cases diagram

# 3. Design Phase

## 3.1. Architecture

The design and the choice of the architecture were influenced since we had to implement the application using Xamarin and Azure. Xamarin is a platform used for developing native cross platform applications targeting Android, iOS and Windows devices. Azure, indeed, is a cloud computing service necessary to build the app’s backend.

## 3.2. High-Level

The following diagram shows the high-level components of the system we are doing to develop:

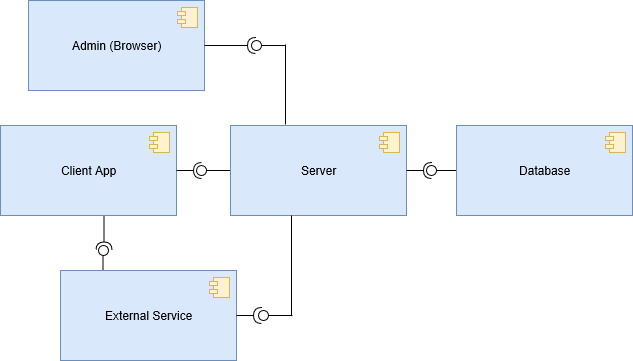


Figure 2: High-level component diagram

## 3.3. Low-Level

In this section, we provide is a description of the structure and the relationships between the two main components, which are the Client App and the Server.

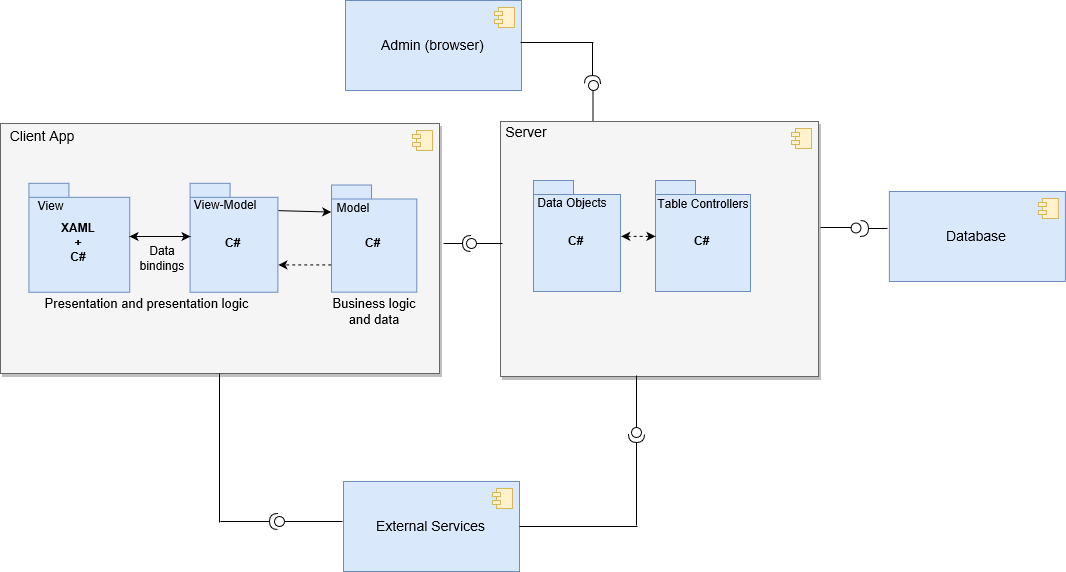


Figure 3: Server and Client App components

The client, which corresponds in this case to the mobile app, is composed by different software components:

* View: it corresponds to the classes needed to build the UI.
* View-Model: it establishes a communication between view and data objects through data bindings, in this way the data objects are exposed and presented in the view.
* Model: it represents the data objects.

The server includes all the methods and functionalities required to interact with the database.

* Controller: it contains the methods to update and modify the data objects.
* Model: it represents the data objects on the server, each one is mapped to a table on the database.

## 3.3. MVVM pattern

Xamarin documentation strongly suggests basing the implementation of the client on the model-view-viewmodel (MVVM) pattern, which is slightly different from the well-known MVC. MVVM facilitates a separation of development of the graphical user interface from development of the business logic or back-end logic (the data model), and the presentation model abstracts a view (creates a view-model) in a manner not dependent on a specific user-interface platform. That is why, using this pattern, it’s possible to create cross platform applications.

The view-model of MVVM is a value converter; meaning the view model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented. In this respect, the view-model handles most if not all of the view’s display logic. The view-model may organize the access to the back-end logic around the set of use cases supported by the view.

# 3.2. User Interface

## 3.2.1. Mock-up

Before starting the implementation phase, we thought it was useful to draw a mock-up to figure out what the aspect of the app would have been. For this purpose, we used a vector graphics editor called Sketch. To have an idea, here we show some screens taken from the mock-up.

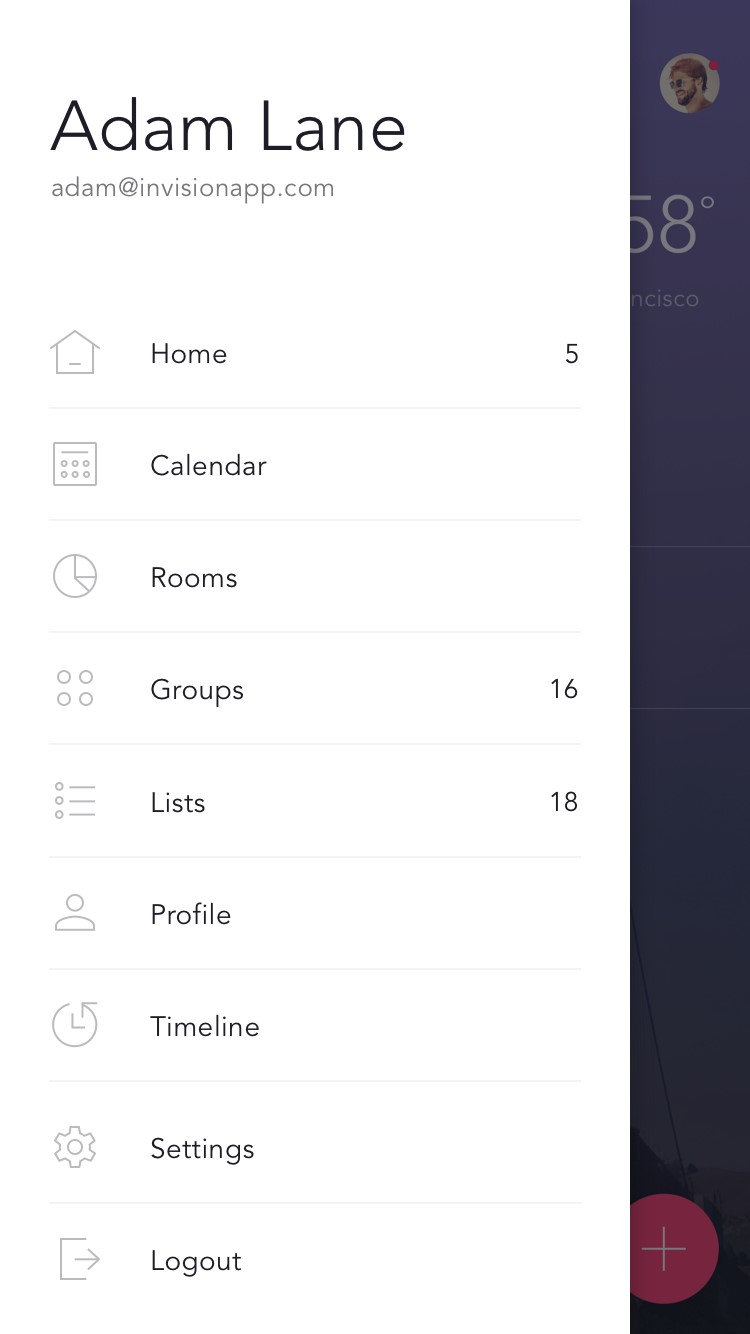
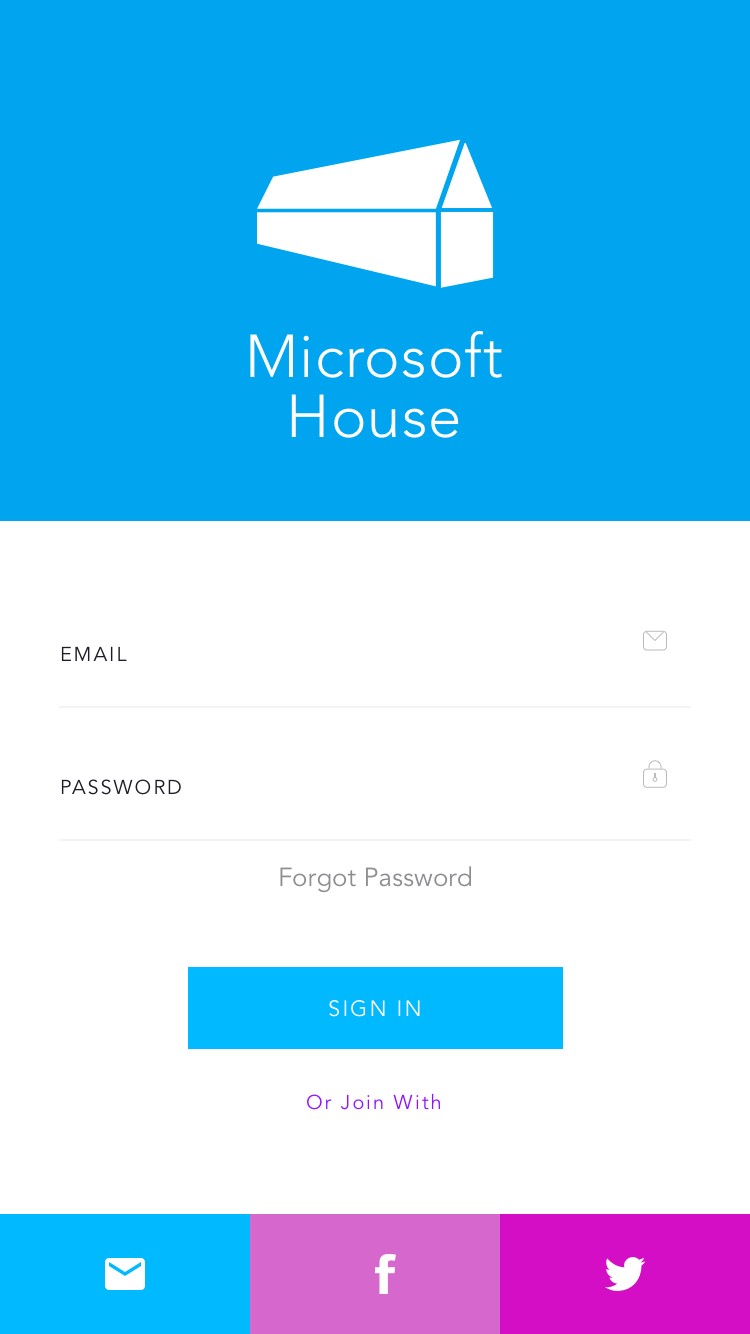
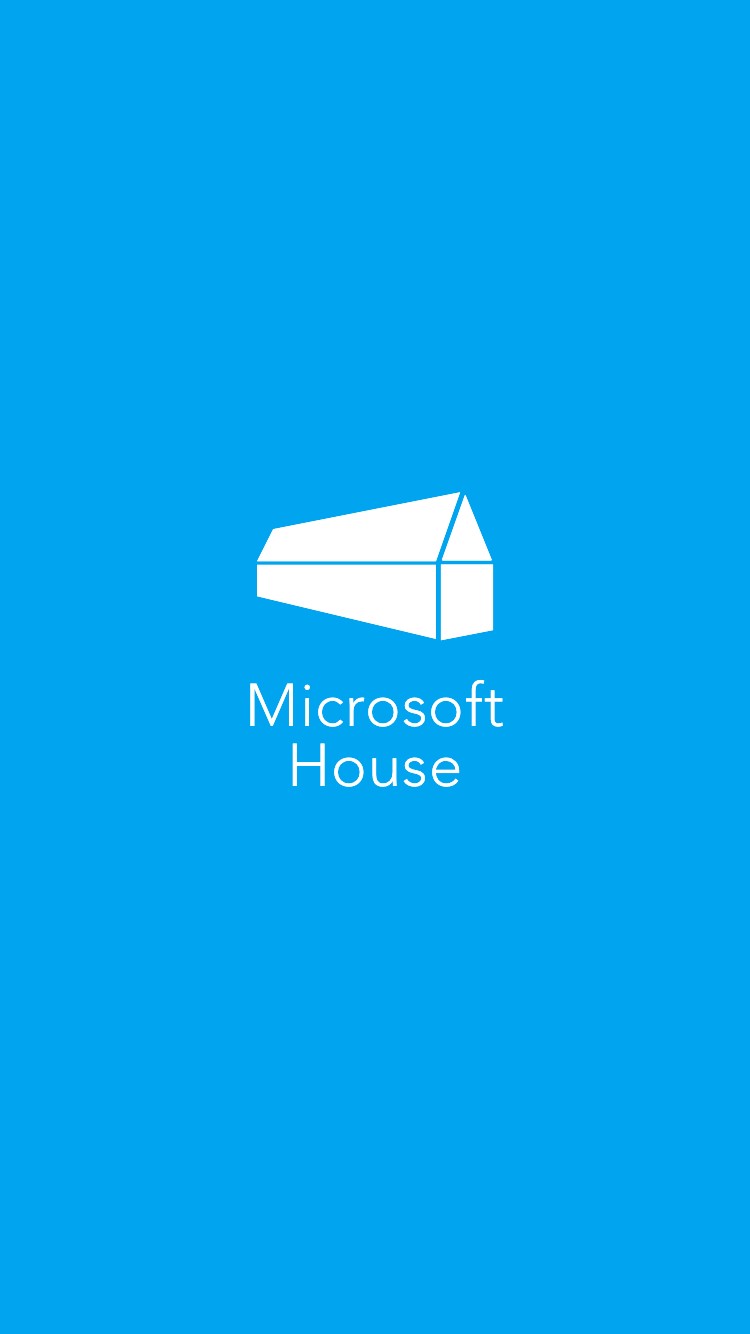


Figure 4 Mock-up: Splashscreen - Login - Hamburger menu

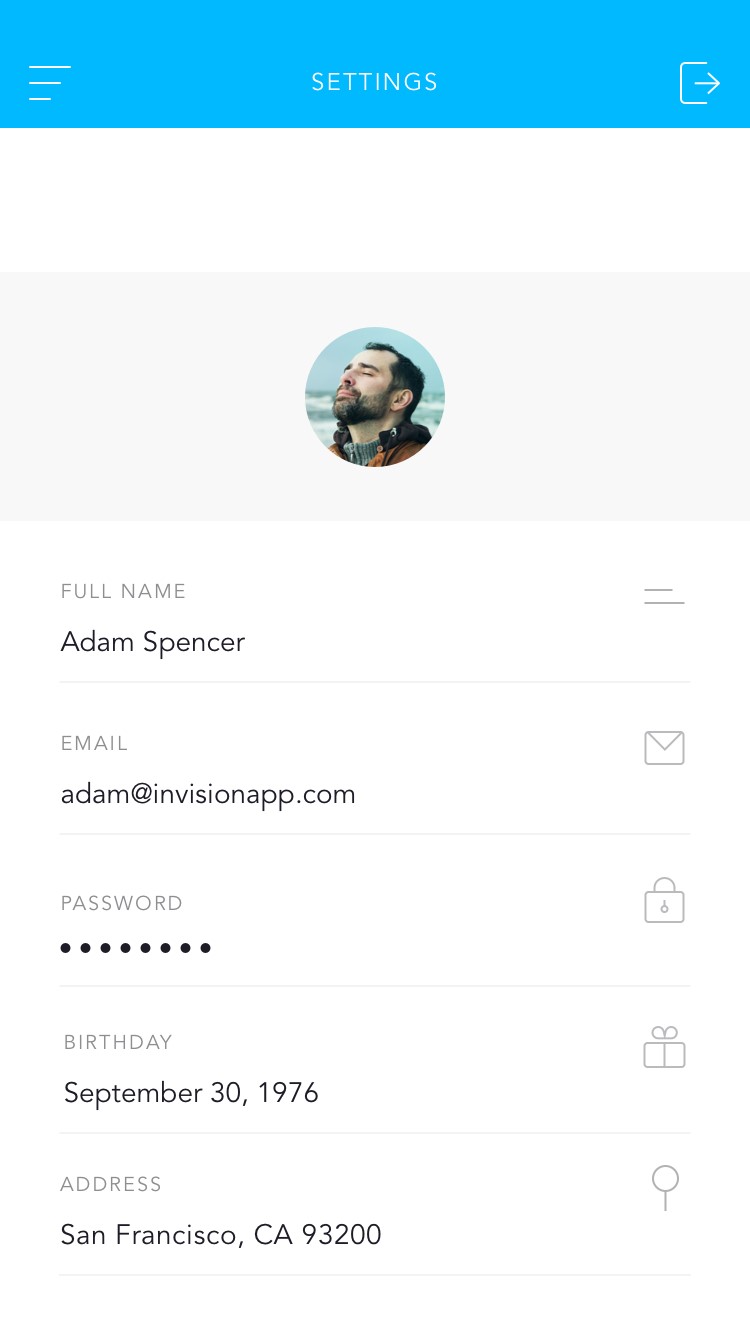
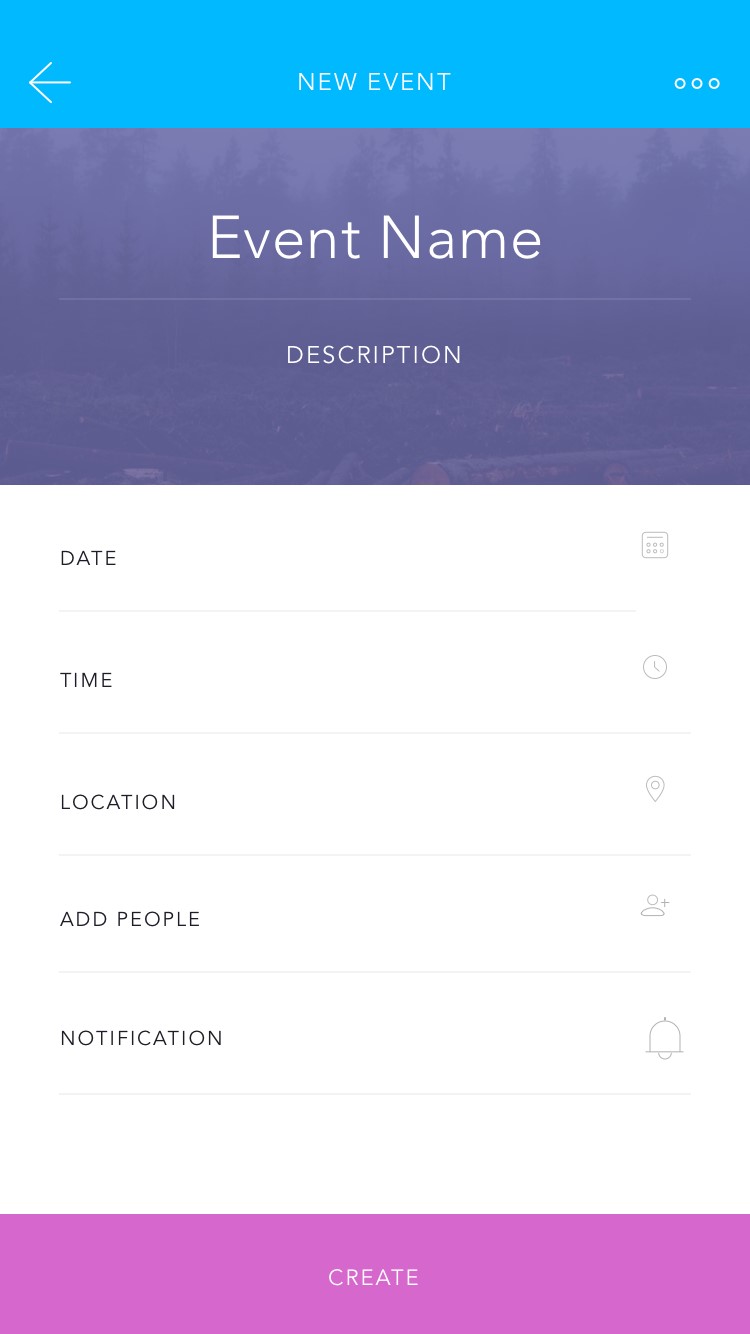
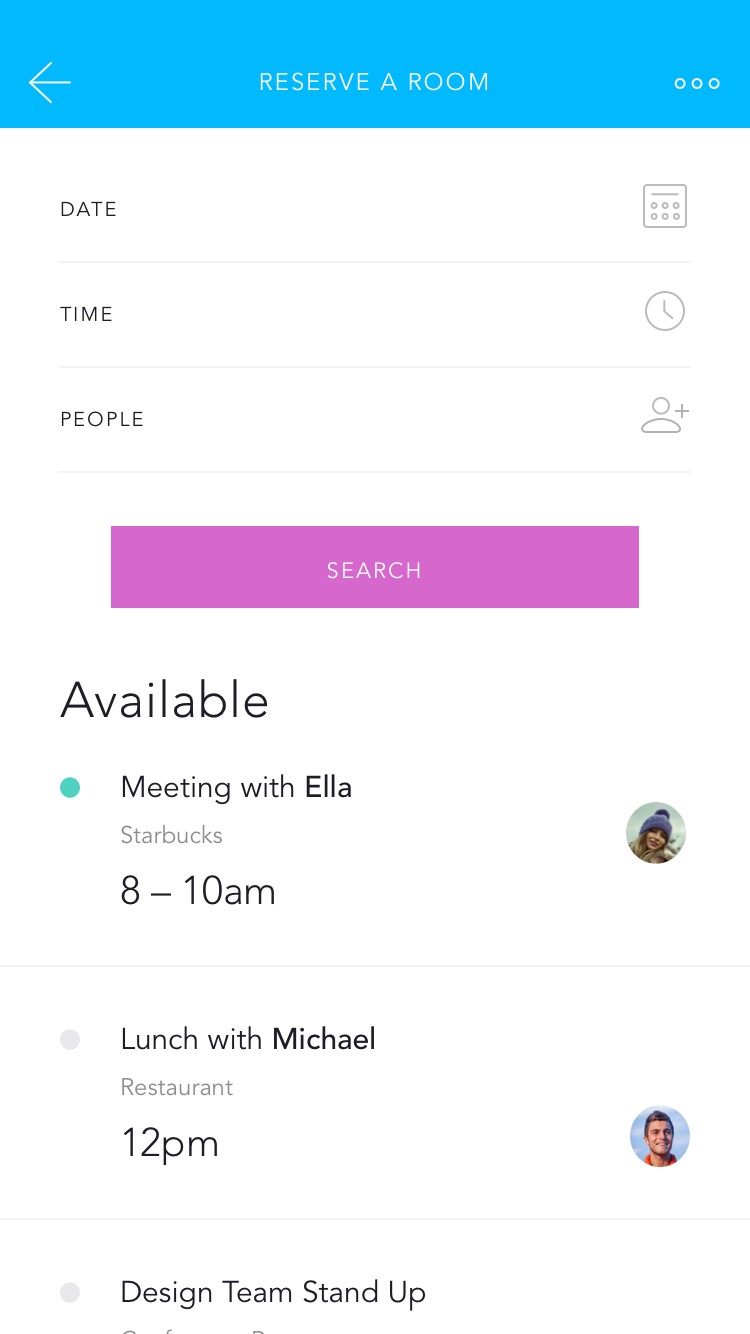


Figure 5 Mock-up: Search room - New event - User settings

# 4. Implementation

In this section we are going to explain the different tools and methodologies we used to implement the app. In the end (section 5), we will explain also the difficulties and the issues we faced during the development.

## 4.1. Tools and Equipment

### 4.1.1. Xamarin.Forms

In order to develop the application for each mobile OS we used Xamarin.Forms, which is a platform based on C# (and XAML) able to build cross-platform applications, targeting iOS, Android and Windows devices. Xamarin.Forms is one of the Xamarin products, and it comes integrated with Visual Studio, Microsoft’s main IDE.

The operating principle of Xamarin.Forms is as follows: by means of a PCL (portable class library) shared project it’s possible to share the up to the 96% of the lines of code among the different platforms. Some difficulties arise if the purpose is to create a professional product with advanced features (like integrations with external services) because there is the need to adjust and modify many things for each platform in order to make everything working properly.

The shared project configuration allows the developer to share most of the code between all the different platforms. The idea is to create four different subprojects (Shared, Droid, iOS and UWP), where the shared project represents all the view classes and the logic behind, while the others are used in case someone needs to handle a task of a specific platform, then he must implement it in three ways (one for each platform).

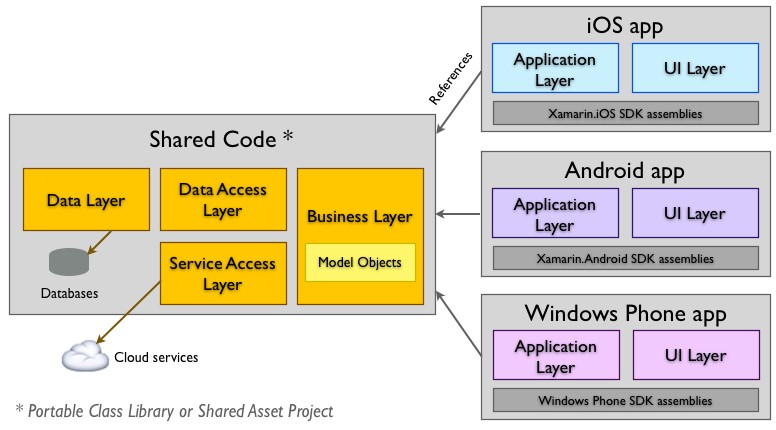


Figure 6: Xamarin.Forms platform

Xamarin.Forms libraries generally provide all the elements required to implement a complete navigation flow as well as the controls which compose the body of the UI. The advantage is that, while compiling the application, these elements are interpreted and instantiated as native elements according to the target platform. For example, an Entry element of the Xamarin.Forms library will be converted into:

* A UITextField, if you compile an iOS app.
* An EditText, if you compile an Android app.
* A TextBox, if you compile an UWP app.

This concept is visualized with a diagram in section 5.3., in relation to UI controls customization.

There are some aspects in which there’s the necessity to go beyond the default Xamarin.Forms libraries, therefore modifying the specific code for each platform. In this case, the approach is generally to create an interface in the PCL shared project and then implement the interface in the different platforms, using native instructions. An example in which we used this kind of approach was in the implementation of external services, as well as in the customization phase of some UI controls (TimePicker, DatePicker, Entry, etc.) See section 5.3 for more info.

## 4.1.2. Azure Mobile Apps

We decided to implement our app’s backend using Azure, and specifically the Azure Mobile Apps service. This choice has been taken for two reasons: first, because Microsoft specified it in the specification document; second, because the online documentation usually deals with Azure Mobile Apps together with Xamarin.Forms. In fact, both Azure and Xamarin are part of Microsoft and what the company is doing is establishing a synergy between the two platforms, in order to provide a complete environment for developing mobile applications. However, we had many troubles integrating Azure with our project on Visual Studio and Xamarin, because it’s a really delicate process and there are a lot of places where things can go wrong. Moreover, many of the services we dealt with were subject to frequent updates, and, due to this fact, the documentation was incomplete or obsolete.

## 4.2. External Services

To fulfil the requirements but also to enrich our application, we integrated some external services, in this section we will show the most significant ones.

### 4.2.1. Xamarin.Forms Maps

Xamarin.Forms Maps uses the native map APIs on each platform. This provides a fast, familiar maps experience for users, but means that some configuration steps are needed to adhere to each platforms specific API requirements. Once configured, the Map control works just like any other Xamarin.Forms element in the shared code project.

### 4.2.2. Azure App Authentication Service

To fulfil the login requirement, we used the Azure App Authentication Service, which implements a OAuth 2.0 authentication procedure, used to properly identify a user to the mobile backend. The OAuth protocol is used to route the authentication request to the right place and to verify that the authentication took place properly. There are three actors in the OAuth protocol:

* The Client is the application attempting to get access to the resource.
* The Resource is the mobile backend that the client is attempting to access.
* The Identity Provider is the service that is responsible for authenticating the client.

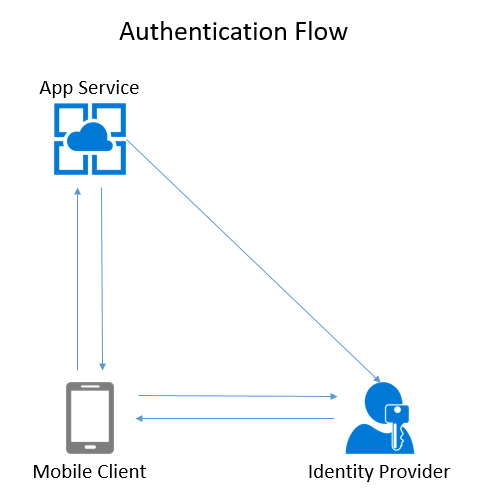


Figure 7: Authenthication flow

At the end of the process, a cryptographically signed token is stored in the app. This token is added to every request made by the client to the resource to securely identify the user.

We could have chosen any identity provider suitable for our purpose, e.g. Facebook, Twitter, Azure itself. However, to realize a scenario that simulates what a Microsoft employee must deal with, we decided to adopt as Identity Provider the Microsoft Account. This carried out another advantage, because we already had an account: we needed one for accessing Azure in general.

### 4.2.3. Azure Notification Hub

The main difficulty concerning push notifications is to provide a user experience as homogeneous as possible among the different platforms. Each platform provider provides their own push notification service (PNS). For example, iOS uses Apple Push Notification Service (APNS). Google uses Firebase Communications Manager (FCM), which is a rebranding of the older Google Cloud Messaging (GCM). Newer versions of Windows, including Universal Windows, use Windows Notification Service (WNS). For this purpose, we relied again on Azure, with the service called Notification Hub.

Notification Hub handles all the registration and bulk sending logic to allow you to send a single message to multiple recipients without having to worry about what platform they are on. The mobile device initiates the process, registering with the push notification service. It will receive a Registration ID in return. The registration ID is specific to an app running on a specific device. Once a device has the registration ID, it will pass that registration ID to Azure backend. The backend will use the registration ID when communicating with the PNS to send the push notifications.

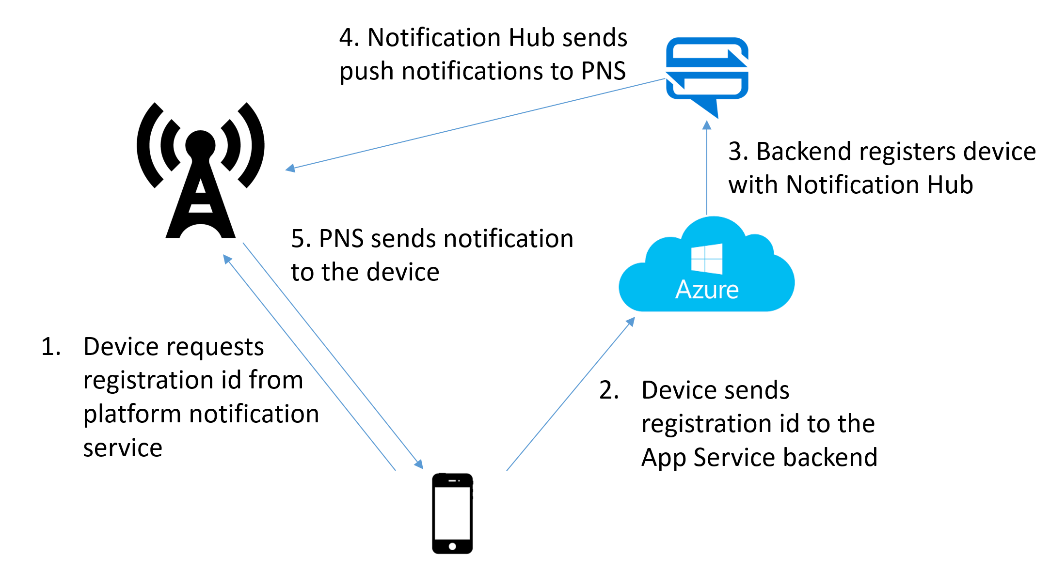


Figure 8: Notification Hub

### 4.2.4. Implementation sample

To provide an example of what we have been discussing in the previous sections, we show how the parts related to the login and the push notifications are structured in our project. Here the separation between shared code and platform-specific code is evident: in the shared project, we provided the interfaces that are effectively implemented in the platform-specific projects, and this is because we had to work completely outside the context of the shared libraries provided by Xamarin.Forms. The core functionalities are identified by the public methods LoginAsync and RegisterForPushNotifications, which handle the homonymous operations.

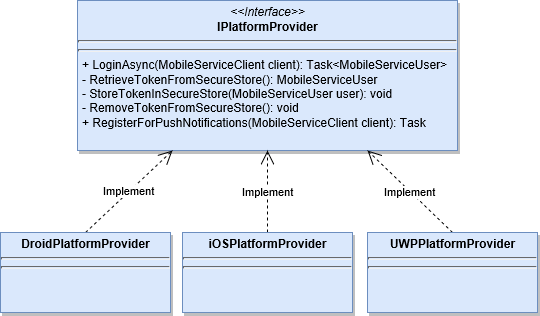


Figure 9: Implementation of login and registration to PNS

# 5. Issues

## 5.1. Requirements

It was not easy to maintain a high usability while meeting all the requirements; in fact, most of the applications usually have just one main functionality, while our application is much more complex and merging the different tasks while keeping a friendly user interface was our main issue. Therefore, our UI was subject to continuous adjustments and improvements; due to this fact, our first mock-up is completely different from the final result. Moreover, some of our initial ideas for what concerns the UI had to be revised due to technical constraints (e.g. our login page, which is substituted by the Microsoft Account’s one).

## 5.2. OS Rules

Another problem that arose during the implementation was due the fact that each mobile OS has its own user interface peculiarities, which means that even doing a mock-up in the beginning, it was impossible to follow it for the duration of the entire project. We were forced to have a different UI look and feel for each platform. In Figure 10 there is an example of how a navigation pattern (Tabbed Page) is displayed in the different platforms.



Figure 10: Tabbed Page visualization in each platform

## 5.3. UI controls customization

Xamarin.Forms apps are implemented using the native controls of the target platform, allowing to maintain the appropriate look and feel for each device. Custom renderers provide an approach for customizing the appearance and behaviour of standard Xamarin.Forms controls, because they let developers override the process in which a Xamarin.Forms control is “converted” into a native one. They can be used for small styling changes or sophisticated platform-specific layout and behaviour customization.

In our project, we aimed to provide the most similar user experience as possible among all platforms, therefore we implemented some renderers for controls that are different between the platforms. Here is for example a diagram that considers a renderer for the Xamarin.Forms Entry control. If we override some properties of the Entry control and the relative renderer for each platform, we will obtain a customized native control.

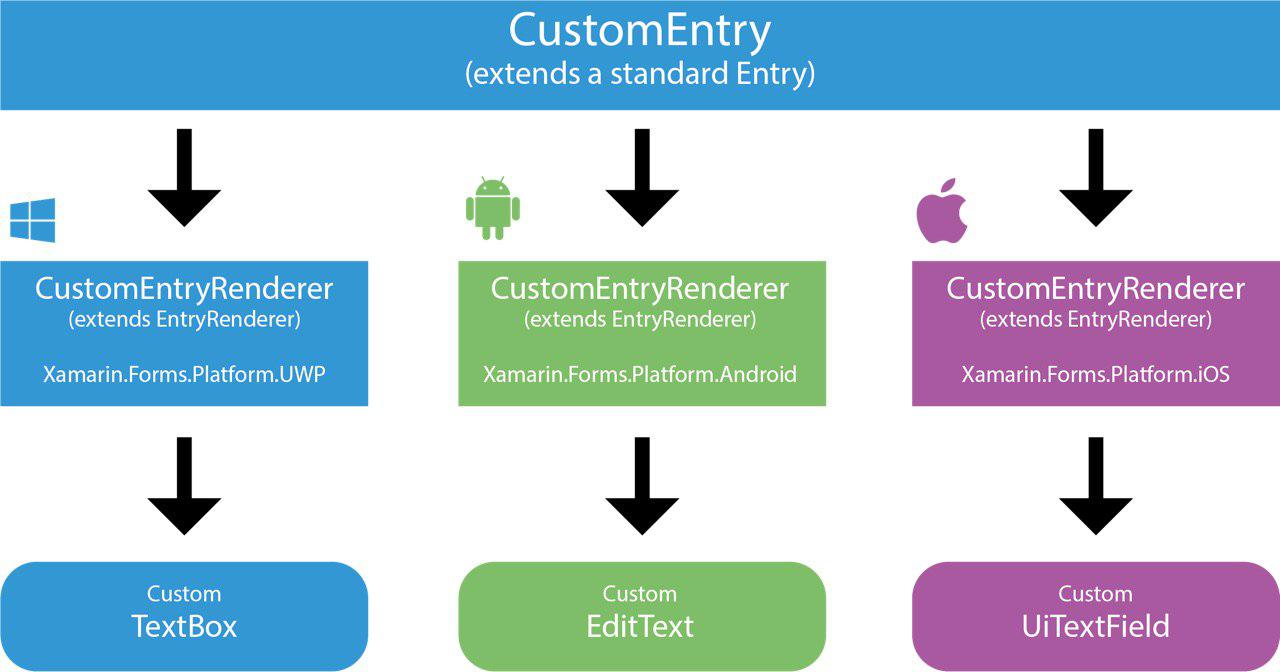


Figure 11: Schema for implementing a Custom Entry

However, the implementation of a custom renderer is a painfully hard task, because it needs a huge writing of lines of code and requires a deep knowledge of the native platform controls. Moreover, it’s better not to abuse of custom renderers, because the performance of the application will be affected negatively.

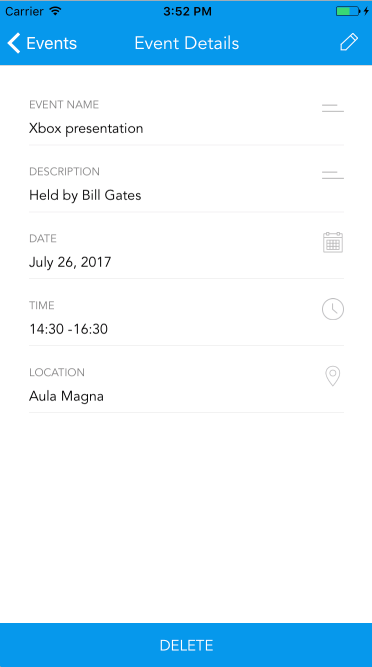
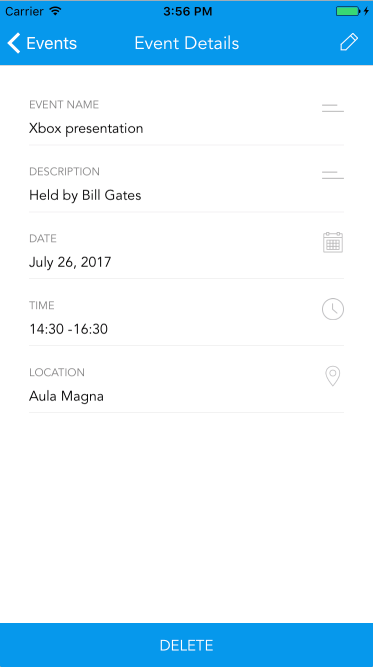
# 6. Testing

## 6.1. Supported platforms

Since the development of a cross platform application has the aim to deliver a product to many platforms and devices, we tested our app using various emulators and real devices, to be sure that it was rendered correctly and in a responsive way in all of them. Here are the requirements that the target device must ensure to run Microsoft House app properly:

* Android: 6.0 Marshmallow (API level 23) or higher
* iOS: 8.0 or higher
* UWP: Windows 10 (10.0, build 10586) or higher

We provide also some screenshots of how the app is rendered in various devices:



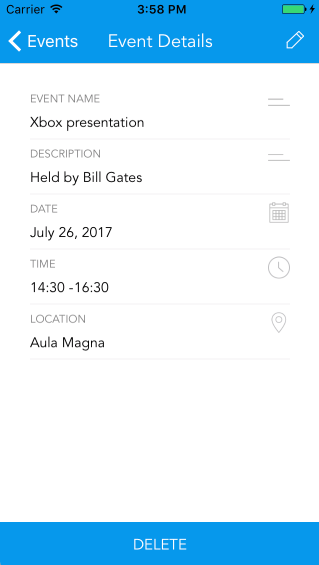
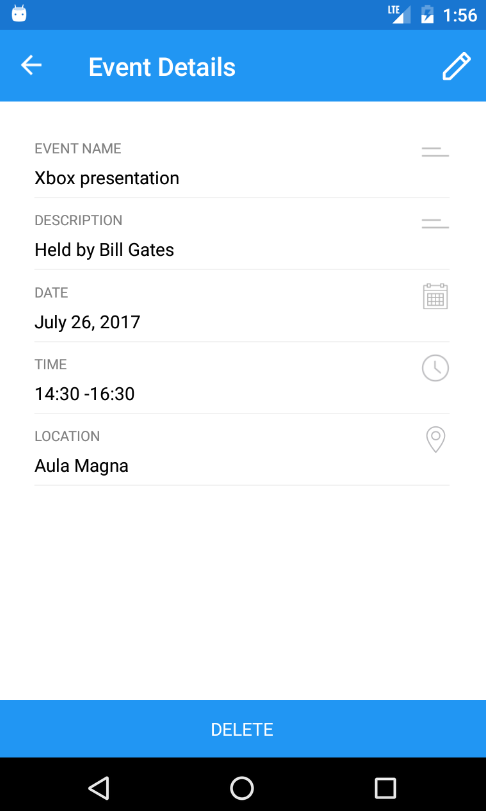
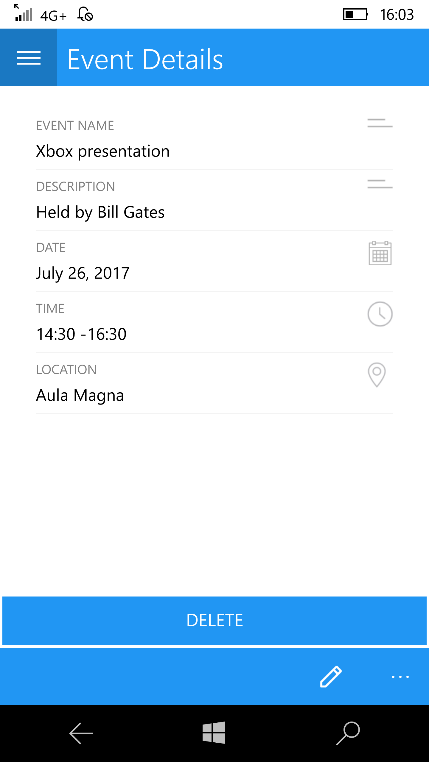
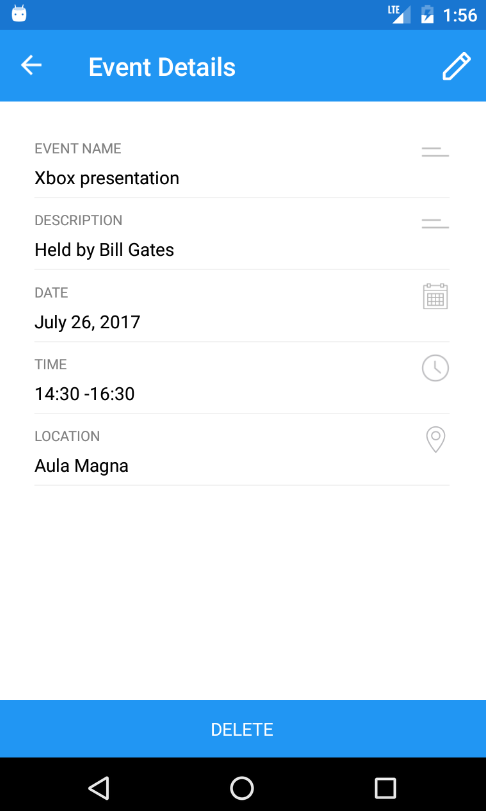


Figure 15: emulator with Android 6.0

Figure 17: Microsoft Lumia 950

Figure 16: Nexus 5

Figure 12: iPhone 7

Figure 14: iPhone 5

Figure 13: iPhone 6

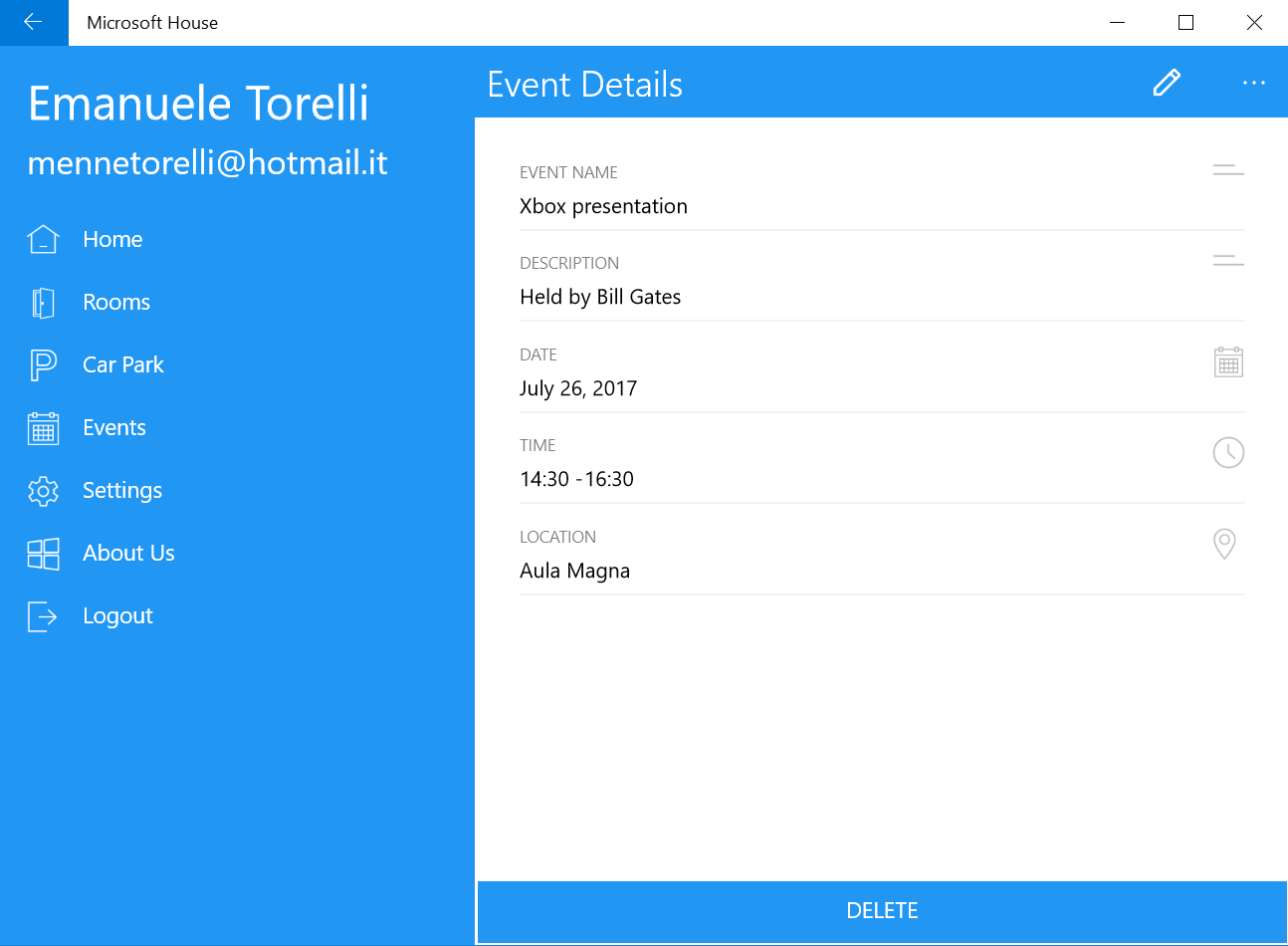


Figure 18: Dell XPS with Windows 10

## 6.2. Unit testing

The unit testing phase has been taking place since the start of the development of the front-end of the app, to verify that the user interface reacts to every user action. We provide few examples of unit test cases, just to have an idea of what we considered during the development.

|  |  |
| --- | --- |
| **Test case** | **User clicks on the menu icon** |
| Precondition | The user is on a generic master-detail page. |
| Action | The user clicks on the menu icon. |
| Outcome | The hamburger menu appears with all the available options. |

|  |  |
| --- | --- |
| **Test case** | **User clicks on a room** |
| Precondition | The user is on the room list page |
| Action | The user clicks on a room. |
| Outcome | A new page is created showing all the details of the room. |

|  |  |
| --- | --- |
| **Test case** | **User clicks on a date of the calendar** |
| Precondition | The user is on the page of the shared calendar. |
| Action | The user clicks on a date of the calendar. |
| Outcome | All the events of the selected date will be displayed below the calendar. |

|  |  |
| --- | --- |
| **Test case** | **User selects the date of an event** |
| Precondition | The user is on the new event page. |
| Action | The user clicks on the picker and chooses the date of the event. |
| Outcome | The user can see that the picker has saved the chosen date, the chosen date is bounded correctly to the view-model relative to the new event. |

|  |  |
| --- | --- |
| **Test case** | **User want to create a new event/reservation with inconsistent date or time** |
| Precondition | The user is attempting to create a new event/reservation and has inserted a date previous than today’s date or starting time greater than ending time. |
| Action | The user clicks on the reserve/create button. |
| Outcome | The user is alerted that he has inserted wrong date or time, and the event/reservation will not be created. |

## 6.3. Integration testing

Integration testing has been taking place since we started integrating the app with the Azure Mobile App backend. In particular, we had to perform many test cases to see if the database records were updated effectively according to the user actions. Below we show some examples of the test cases we investigated.

|  |  |
| --- | --- |
| **Test case** | **User logs in** |
| Precondition | The user is on the entry page, without being logged in. |
| Action | The user clicks on login button, and |
| Outcome | When the user clicks on the login button, he is redirected to the Microsoft Account login page. When the user inserts his credentials correctly, he is redirected to the home page and from now he will be successfully authenticated. |

|  |  |
| --- | --- |
| **Test case** | **User creates a new reservation** |
| Precondition | The user is on the new reservation page, and he has filled the pickers with date and time so that he can see a list of available rooms. |
| Action | The user clicks on the room he wants to reserve. |
| Outcome | A new reservation is properly stored in the server, the user can see an updated list of reservations. |

|  |  |
| --- | --- |
| **Test case** | **User creates a new event** |
| Precondition | The user is on the new event page, and he has filled the entries and the pickers with correct data. |
| Action | The user clicks on the “Create” button. |
| Outcome | A new event is properly stored in the server, the user can see an updated instance of the shared calendar, all the users receive a push notification on their device. |

|  |  |
| --- | --- |
| **Test case** | **User wants to park his car** |
| Precondition | The user has opened the QR code scan page, in the database there isn’t a record associated to the fact he has parked. |
| Action | The user focuses his device on the QR code. |
| Outcome | The user sees that the available slots in the car park are decreased of one unit, the database creates a record associated to the fact he has parked. |

|  |  |
| --- | --- |
| **Test case** | **User wants to leave the car park** |
| Precondition | The user has opened the QR code scan page, in the database there isn’t a record associated to the fact he has parked. |
| Action | The user focuses his device on the QR code. |
| Outcome | The user sees that the available slots in the car park are increased of one unit, the database deletes the record associated to the fact he has parked. |

## 6.4. Notification testing

Push notifications occur when a user modifies the shared calendar. To check the reliability of notifications, even if we cannot guarantee an availability of 100% since they rely on the dedicated push notification service, we tested them in the sequent scenarios:

* The app is running and in the foreground.
* The app is running, but in the background.
* The app is not running at all.

# 7. Conclusion

We can say that the choice of Xamarin and Azure was worthy, because they allowed us to create a single product that is compatible practically with all the mobile platforms available on the market. On the other hand, we think it's excessive to say that realizing a cross platform app requires the same effort with respect to a traditional one, and this is because of the inherent difficulty of integrating external services, and the fact that if you want to level out completely the UI among all the platforms, you have to modify many platform-specific aspects, losing the principle of “code once and run everywhere”.

# 8. Other info

All the info about the project and the source code are available at <https://github.com/Menne/Microsoft-House>.