MINISTERUL EDUCAȚIEI, CERCETĂRII, TINERETULUI ȘI SPORTULUI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

CLOUD BASED CROSS PLATFORM MOBILE APPLICATION FOR ORDERING TAXI

LICENSE THESIS

Graduate: **Razvan POPA**

Supervisor: **S. L. Ing. Cosmina IVAN**

2012

MINISTERUL EDUCAȚIEI, CERCETĂRII, TINERETULUI ȘI SPORTULUI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

VIZAT,
DEPARTMENT HEAD,
Prof. dr. ing. Rodica POTOLEA

DEAN,
Prof. dr. ing. Liviu MICLEA

Graduate: **Razvan POPA**

CLOUD BASED CROSS PLATFORM MOBILE APPLICATION FOR ORDERING TAXI

1. **Project proposal:** The application allows ordering a taxi direcly from a mobile phone and provides an operator the ability to respond to orders through a secure web interface all supported by a Cloud infrastructure.

2. **Project contents:** Presentation page, advisor's evaluation, introduction, project objectives and specification, bibliographic research, analysis and theoretical foundation, detailed design and implementation, testing and validation, user manual, conclusions, further development and appendices.

3. **Place of documentation**: Technical University of Cluj Napoca

4. **Consultants**:

5. **Date of issue of the proposal**: November 1, 2011

6. **Date of delivery**: June 28, 2012

Graduate: _____

Supervisor: _____

1 Table of Contents

1 Introduction

1.1 Project context

The era of mobile devices is outpacing both the PC revolution of the 1980s and the Internet Boom of the 1990s in terms of customer adoption. By the end of 2012 it is estimated that the cumulative number of iOS and Android devices activated will surge past 1 billion. To put it into perspective over 800 million PCs were sold between 1981 and 2000, making the rate of iOS and Android smart device adoption more than four times faster than that of personal computers.

The Internet grew to 495 million users by the end of 2001 while beginning its commercial ramp in 1996. Smartphone devices will see double the number of device activations during its first five years compared to the number of Internet users reached during its first five years.

On top of this massively growing smartphone installed base, more than 40 billion applications have already been downloaded for these devices. More than ever, consumers are splitting their time accessing services on the Internet from PCs versus doing so on mobile devices from apps.

As there have been many definitions for mobile devices, we explicitly state what these terms refer to in this thesis. Modern mobile devices are composed of two large categories: smartphones and tablets.

A *smartphone* is a mobile phone with more advanced computing ability and connectivity. A smartphone combines the functions of mobile phones, personal digital assistant (PDA), portable media players, compact digital cameras, and GPS navigation units.

A *tablet* is similar to a smartphone with the difference that tablets usually do not have a phone-calling ability. They offer a larger display and are designed for content consumption such as watching videos, reading newspapers or Internet navigation.

The line of demarcation between modern mobile devices and older devices having advanced capabilities (also called feature phones) is not an exact one. We will consider a modern mobile device as being composed of three components: hardware, operating system and a market for applications.

Figure 1.1: The components of a modern mobile device

Figure 1.1 represents the main components of a modern mobile device. The combination of these three components together with the associated service dedicated to mobile devices is referred to as the mobile ecosystem. Further we will refer to smartphones and tablets as mobile devices, unless specifically mentioning that we mean the largest category including all mobile devices that offer connectivity such as regular phones or also referred to as feature phones.

1.2 Taxi ordering apps for mobile

With the rising adoption of smartphones by the consumer base, a large variety of applications meant at simplifying users daily life and enhancing productivity are appearing. Taxi ordering applications have first become popular in western countries, as this region offered a higher income level that determined a larger smartphone penetration.

Taxi ordering applications have become popular for two major reasons. The first is a more mobile-centric usage pattern, where users relate to apps for fulfilling different needs that they have. In this respect, a taxi ordering application for mobile phones is perceived as a modern and more interesting way of ordering a taxi.

The second reason has to do with the advantages that this method has over traditional taxi dispatching systems, such as being able to see the taxi approaching on a map, being able to rate a taxi and the most important being spurring customer retention. The taxi business tends to be a medium to lower class fidelity service, where users mostly consider the closest taxi as opposed to quality, which is regarded as being at a relatively consistent level among taxi drivers and there is no way for the client to determine the quality when ordering. The taxi ordering application allows for a more engaged user base and offers fidelity rewards that are used to enhance the customer retention level. In a business world which is shifting from a consumer-centric marketing to a human-centric model, this apps allows taxi businesses to better interact with their clients in a more direct and responsible manner.

Also this distributed system helps taxi drivers, which can receive orders directly from their clients and are able to benefit from offering superior services as they can get higher ratings.

1.3 Project content and organization

This project is organized according to the following structure:

Chapter 1 contains general information regarding mobile devices and explains why we developed a mobile taxi ordering application

Chapter 2 details the requirements of the system and the objectives of this project.

Chapter 3 describes the research we conducted in order to specify the system requirements, the previous work we are basing our project on and also a comparison with similar services

Chapter 4 describes the patterns and architecture decisions we have taken and explains why we have chosen to implement them.

Chapter 5 contains the implementation details for Order Taxi application. We detail the implementation of the cross platform mobile application, the integration between the client mobile application and the Cloud services through a REST API developed using WCF, the integration with external services and also the frameworks used for signaling and push notifications. It also details the Operator panel that is used to respond to orders and explains the user interface.

Chapter 6 contains the testing and evaluation of the system, we exemplified how the system was tested and what results we obtained

Chapter 7 presents the installation manual for the system

Chapter 8 is a short summary of the project and lists future developments

2 Project Objectives and Specifications

2.1 Project objectives specification

As mobile devices penetration rate rises quickly having an App that caters for each user particular need has become the norm.

The specific App that we are developing allows clients (users) to order taxis directly from their mobile device without having to call trough an operator. Having an app for this need opens the doors for a wide verity of marketing strategies such as offering promotions, coupons, paying by credit card, offering taxi subscriptions. This allows users to have an alternative to the old method of ordering a taxi by calling, based on the powerful trend of mobile engagement.

The problem of	Ordering a taxi by calling is frustrating, uncool, few options for marketing campaigns, very low business-customer relationship
affects	Users fidelity level with a company, client and taxi driver see their ride as a one time event, no customer relationship management available
the impact of which is	Taxi companies do not engage with their customers to retain them, Users find it frustrating to order a taxi
a successful solution would be	A taxi ordering application for mobile devices.

Table 2.1 Taxi Ordering application

In order to better understand the taxi ordering application we are developing, we have created the following product statement which offers a summer but descriptive image of the system.

For	Smartphone users
Who	interested in ordering a taxi
The Taxi ordering Application	intends to create a customer-company lasting relationship and a new and cool way of ordering a taxi
That	Opens the doors to marketing practices unavailable in the current business model
Unlike	Calling a taxi operator for ordering a taxi
Our product	Is a modern approach that offers value added capabilities that offers new business practices to the industry

Table 2.2 Product statement for Taxi Ordering application

We have identified the following stakeholders for the Order Taxi applications, which we present in Table 2.3 together with their respective responsibilities. These stakeholders will be represented as actors in Chapter 4, where use cases for each of them will be detailed.

Name	Description	Responsibilities
Taxi Company	Company operating a fleet of taxis, interested in obtaining a higher user fidelity and increase market share	Accept the implementation of the system inside the company
Taxi ordering clients	Is interested in having an easy to use product that he can use to order taxi in a more engaging and modern way	Is responsible for inputting orders and viewing the order status.
Taxi drivers	The person who will be driving the taxi ordered by the customer.	is responsible with confirming the order and providing the service
Operator	The person managing clients orders which do not go directly to the driver	Monitors incoming orders and responds to orders in a timely manner by assigning a response containing a taxi id to each order
Manager	The person managing taxi operators and drivers	Is concerned with updating the operational area of the company and creating accounts for drivers and operators.

Table 2.3 Stakeholders of the project

2.2 Non Functional System requirements

Software is becoming critical in driving the information-based economy. Time-to-market, robustness, and quality are important factors for measuring the success of systems in competitive economies and environments. There is also a need to evolve and adapt rapidly and smoothly in an environment of continuous changes of business requirements. Therefore, there are stronger needs for standards and formalizations of the processes by which we specify and capture these requirements. Requirements capture is one of the initial phases to undertake in the system development process.

Non-functional requirements are properties and qualities the software system must possess while providing its intended functional requirements or services. These types of requirements have to be considered while developing the functional counterparts. They

greatly affect the design and implementation choices a developer may make. They also affect the acceptability of the developed software by its intended users.

In the following, we briefly describe the three categories of non-functional requirements that may be imposed on a software system.

2.2.1 System-related NFRs

These types of requirements impose some criteria related to the internal qualities of the system under development and the hardware/software context in which this system will operate.

Operational requirements: The Taxi Ordering Application will be running on iOS and Android phones, and will be using Cloud services for data synchronization and storage.

Performance requirements: For our system these requirements are considering the response time of data synchronization. As the response time to an order is dependent of the other user inserting the response manually (the operator or the driver being a human not a system that can respond automatically). Data synchronization must be assured to take place bellow 2 seconds.

Maintainability requirements: The system must be designed in such a way that it allows a modifiable, component based system that allows future modifications and functionality extensions without breaking the present working version. This requirement is especially valid for our mobile application, as different mobile application versions installed on smartphone must communicate effectively with the Cloud-based API. In order to achieve maintainability we relied on API versioning for this specific requirement.

Portability requirements: The system must work on a variety of mobile OS, of which iOS and Android are mandatory. A cross platform solution must be implemented to allow maximum code reuse with little adaptation.

Security requirements: The application must use the .NET security framework for assuring authentication and authorization security constraints, and must also allow for SSL communication.

2.2.2 Process and Project-related NFRs

These types of NFRs impose criteria to be followed while developing the project.

Conformance to standards requirements: The standard developing pattern for Titanium applications is MVC, which should be followed in order to assure maximum code reusability and maintainability.

Development time and cost requirements: The project needs to be completed and fully functional by 25 June 2012.

Development process requirements: The project needs to be developed in an iterative method, allowing for partial system components development and testing.

Testing requirements: The API for the Cloud infrastructure needs to be tested using a unit testing approach, while the mobile taxi application need to be tested using use-case testing.

Installation and deployment requirements: The mobile applications need to be deployed to the AppStore and GooglePlay accordingly.

2.2.3 Human-related NFRs

These types of requirements deal with constraints related to the stakeholders and the social and societal context in which the system is deployed.

Usability requirements: The application needs to be easy to understand and use by new users. It must also feature a demo the first time the application is launched.

Look and Feel requirements: The mobile application needs to adhere to the usual UI components and not define custom components that might reduce the learnability of the system.

Legal requirements: The application should not allow for malicious uses of personal data required by the application such as mobile phone numbers. This data must be stored on the Cloud and should be accessible only for specific users, if needed.

2.2.4 Device-related NFRs

GPS: The device must feature GPS capable services for approximate user position identification.

Internet: The device must allow for Internet connection for sending order, receiving responses and for the overall interconnection with the Cloud services.

Operating system: The device must work with any iOS or Android version.

Screen resolution: The application must work properly on any screen resolution. The application must be compatible with both the Retina display and the lower resolution display screens.

2.3 Functional Requirements

The functional requirements have been decomposed into low-level functional specifications that are depicted in Table. This specifications have been assigned priorities which are consistent with the iterative approach of development that we are following, according to the Non functional requirements presented previously.

FR1	**Ordering of taxi**	High
FR1.1	Ordering based on GPS location	High
FR1.2	Add precise location	High
FR1.3	Remember used locations for suggestions	High
FR1.4	Display map with user location	High
FR1.5	Allow dragging the position marker on the map	Medium
FR1.6	Get GPS location based on wireless	High
FR1.7	Handle cases when GPS location cannot be obtained by allowing only exact address input	High
FR2	**Dispatching of taxi**	High
FR2.1	View active orders	High
FR2.2	Respond to active order	High
FR2.2.1	Respond with car id and time until it arrives	High
FR2.2.2	Mark initial car position on a map	Medium
FR2.3	Blink and sound alert on new order	High
FR2.4	Call phone for order not processed within 1 minute	Low
FR3	**Order confirmation**	High
FR3.1	Display taxi id and time until it arrives	High
FR3.2	Allow order canceling	High
FR3.3	Allow taxi check-in	Medium
FR3.4	Allow push notification if app is closed	High
FR4	**Authentication**	High
FR4.1	Allow Facebook authentication	High
FR4.2	Allow authentication with username, password and phone number	High
FR4.3	Verify phone number	Low
FR5	**App promotion trough social networks**	High
FR5.1	Allow Facebook sharing of app use	High
FR5.2	Allow sharing of taxi check in	Low
FR5.3	Prompt for app review after 1 successful ride	High
FR6	**Feedback**	High
FR6.1	Allow feedback from button in tab bar	High
FR6.2	Allow taxi check-out	Low
FR7	User post-ride functionalities	High
FR7.1	Allow distance displayed on check out	Low
FR7.2	Prompt user for driver rating on check out	Low

Table 2.4 Functional Requirements

3 Bibliographic research

3.1 Bibliographic research for Taxi ordering applications

Having a device that has permanent Internet access and can be carried around easily by the population opens the doors to new approaches of solving frustrating problems. Taxi ordering has been perceived as a frustrating activity especially in big cities, and at pick hours.

According to [9] ordering a taxi using a smartphone is a better alternative for major cities such as San Francisco. And even though Taxi applications on the App Store are a dime a dozen, but for the most part they're just glorified phone directories that don't really make it any easier to call a taxi. These services just displayed a list of phone numbers from different taxi companies.

One of the first services that offered the ability to order a taxi without calling was *TaxiMagic* using which ordering a taxi is fairly simple: after launching Taxi Magic, your iPhone will use GPS or triangulation to determine your general location, and will present a list of nearby cab services (some listings are only phone numbers, while others fully support Taxi Magic's ordering system). After choosing a supported cab company, the application will ask for your exact street address and will then take you to a status screen that will alert you once your ride is dispatched (wait times can vary depending on the time of day and location). The status screen also allows you to view how far away the cab is and the driver's name.

While *TaxiMagic* operates by simply displacing the telephone as the ordering channel with an order sent from the smartphone, other services have appeared which work differently. According to [16] services such as Uber or GetTaxi operate by having both a driver app and a client app. When the client orders a taxi, the order is submitted to the closest driver, instead of going through an operator. By having a mobile app, the driver can be tracked by the client so the client can see when his taxi arrives.

After analyzing the currently available systems and the functionalities they offer, we started analyzing the infrastructure that should back our system. According to [1] we needed a Cloud-based service to offer the infrastructure for data synchronization between the apps and the operator panel. We also learned the best practices of developing mobile applications by migrating data and processing power inside the Cloud.

Using [18] we have identified the non-functional system requirements which we have listed in the previous chapter. This requirements serve as the base for the system architecture we have chosen which we are detailing in Chapter 4.

After learning how we can develop the aspects of a theoretical integration between the service and the Cloud and the benefits of this approach, we analyzed the means of implementing web services for fulfilling this goal. Using [4] we developed the API for our service which would be consumed inside the mobile application. We decided on using REST web services after reading [14], where it argues that by being lighter, REST services are preferred when working with mobile applications.

We needed the application to be available on as many mobile phone operating systems as possible, so we continued by analyzing cross platform mobile development

solutions. After reading [3] and comparing Titanium development with other development frameworks we decided on using this platform because it offered a series of advantages such as compiling natively and being able to be extended in order to support more advanced functionality available only for iOS or Android.

One important requirement was that the cross platform development supports *GPS*. Using [3] we learned how to implement *GPS* into our application and how to determine *GPS* coordinates with various precisions. Using [17] we learned that we can still determine the approximate user position when *GPS* is not available, such as when the user is located inside a building, which is a very common use case for our system. The way to identify the position of the user in such condition is by *GPS* triangulation. Even without a *GPS* receiver, the cell phone can provide information about your location. A computer can determine the location based on measurements of your signal, such as:color

- Its angle of approach to the cell towers
- How long it takes the signal to travel to multiple towers
- The strength of your signal when it reaches the towers

Since obstacles like trees and buildings can affect how long it takes the signal to travel to a tower, this method is often less accurate than a *GPS* measurement.

3.2 Similar applications

Other similar services in terms of functionality are already available on the market. We present here the most important services by functionality and then in Table 3.1 using a comparison matrix. We have inspired form these systems in determining the specifications of the system we are building.

	Call a Taxi	*Goo Taxi*	*Go Taxi*	*Hailo*	*Taxi Magic*	*Uber*	*Cabify*	*myTaxi*	*MY SYSTEM*
iOS available	✓	✓	✓	✓	✓	✓	✓	✓	✓
Android available	✗	✓	✓	✓	✓	✓	✓	✓	✓
Get position using GPS	✓	✓	✓	✓	✓	✓	✓	✓	✓
Show client position on map	✓	✓	✓	✓	✓	✓	✓	✓	✓
Exact client location input	✗	✓	✓	✓	✓	✓	✓	✓	✓
Free to use	✗	✓	✓	✓	✓	✓	✓	✓	✓

Order without calling	✗	✓	✓	✓	✓	✓	✓	✓	✓
Show nearby taxis	✓	✗	✗	✓	✗	✓	✓	✓	✓
Order submitted directly to taxi driver	✗	✗	✓	✓	✗	✓	✓	✓	✓
Order submitted to dispatcher	✗	✓	✓	✗	✓	✗	✗	✗	✓
taxi live GPS tracking	✗	✓	✓	✓	✓	✓	✓	✓	✓
arrival time estimation	✗	✓	✓	✓	✓	✓	✓	✓	✓
Taxi rating	✗	✗	✓	✓	✗	✓	✓	✓	✓
SMS alerts	✗	✗	✓	✗	✗	✗	✗	✗	✗
Pay inside application	✗	✗	✓	✓	✓	✓	✓	✓	✗

Table 3.1 Similar services comparison matrix by functionality

Call a Taxi app shows the nearby taxis and then allows clients to call the closest taxi company. Is just like a phone directory of taxi companies for each city that uses your location to show a list of the companies that are present in that area.

*Goo Taxi is a S*panish company, product based on idea that the dispatcher implemented a fleet tracking solution that might allow live tracking of the taxi as it arrives. Also offers an interesting SMS alert when the taxi arrives.

GetTaxi is set to disrupt public transportation in Europe. It's already live in UK, Russia and Israel and rolling out in France, Spain and New York City. Allows paying in app. Driver has 5 seconds to respond to request or it will go through a dispatcher. It provides a very clean looking interface from which we have inspired in developing our Taxi ordering system.

Hailo is an app exclusively for London licensed taxi drivers, but they extended the functionality to taxi drivers from more cities including Dublin and New York. They have both a driver and a client app similar to how our system is designed, as can be seen in table 3.1.

Taxi Magic – Using Taxi Magic the booking goes trough dispatcher. Allows booking from multiple companies by selecting the desired one. Figure 3.1 presents the application Taxi Magic together with its most important functionalities.

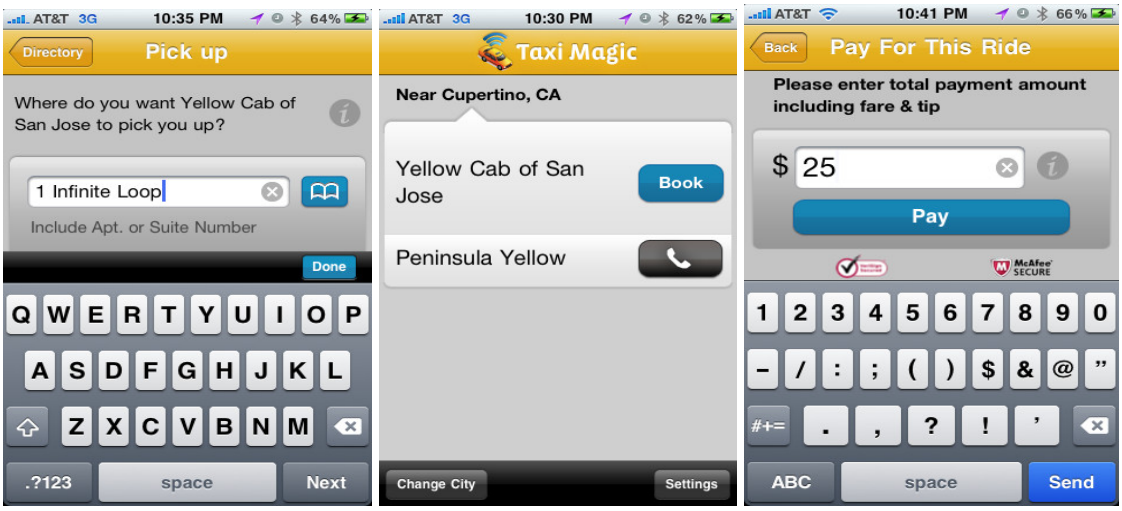

Figure 3.1 Taxi Magic - from left to right: order taxi, choose a taxi, pay using credit card

Uber - Luxury transportation in sleek black car, is based on the business model of offering private drivers as taxi. It is available in largest US cities and Paris. The app offers functionalities of ordering a taxi by getting the users GPS positions, it allows tracking the driver as he approaches and when the ride is over it allows for payment using credit card. In Figure 3.2 we presented a set of functionalities offered by the Uber system. In the left is shown the ordering View, displaying the nearby taxis, in the middle is displayed the tracking system and in right the in app payment functionality.

Figure 3.2 Uber – from left to right: Order a taxi, track the taxi, pay inside app

*myTaxi is d*eveloped by Inteligent Apps, a German company. The software was launched in Hamburg in late March 2010. So far, about 5000 taxi drivers are already using myTaxi in Germany and Vienna. They generated over 350.000 downloads for their app the last time this figure was checked.

As we presented in Table 1, out system is a close competitor in terms of functionalities with myTaxi, but it offers more advanced functionality by also allowing responding to orders by an operator, similar to Taxi Magic.

In the following section we are analyzing the previous work in terms of the two major components of our system: the client mobile application and the service.

3.3 Mobile development

As taxi ordering applications should be available on as many mobile platforms as possible, we tried to identify a cross platform solution. Analyzing the other services that are already available on the market lead us to considering Titanium as the cross platform application development platform.

We observed that some of the existing services are not using a cross platform approach and have relied on developing a separate application for each mobile OS. *Uber* is an example of application that has been designed for different mobile OS, as by the time the application was developed no mature cross platform approach was available.

HTML5 is regarded as being the future of Internet and also most of the apps, which are now built natively, are expected to be available as *HTML5*. However *HTML5* specifications, which are developed by World Wide Web Consortium, are not available yet and the release target for specifications has been delayed until 2014. While there are already a number of apps available which work based on a subset of *HTML5* specifications, the majority of top apps are still native, especially those which require a high integration with the platform. *PhoneGap* is a cross platform framework that rivals Titanium but operates differently. While the result of building an app with *PhoneGap* is a native app, this app is basically a wrapper around an *HTML5* page, which exposes a JavaScript interface for interacting with specific device features such as camera. We recommended using Titanium as a cross platform solution based on the large number of apps that have been built on the platform (over 40000), the native capabilities which go deeper than *PhoneGap* as it doesn't rely on a *WebView* and also the possibility to target both native mobile and *HTML5* releases.

We have decided on using Titanium as it offered true native cross platform development as opposed to hybrid development as in the case of PhoneGap. The decision was also based on the success stories featured for the Titanium framework.

3.4 Service development

InfoWorld [2] has designated *NodeJS* as being the Technology of the year 2012. *NodeJS* is a software system designed for writing *WebServers*. The main difference *Node.js* brings is that it is event based and not thread-based. *NodeJS* has been especially credited for allowing easy communication between the client browser and the server.

Socket.IO is a client side *JavaScript* library that talks to *Node.js*. *Nowjs* is a library that lets you call the client from the server. All these and SignalR are similar and related, but different perspectives on the same concepts. *SignalR* is a complete client and server-side solution with *JS* (JavaScript) on client and *ASP.NET* on the back end to create these kinds of applications. *SignalR* brings the possibility on developing using *C#* on the server, which we considered to be a benefit at the present time as the *.NET* framework is more mature in terms of development environment, productivity, security and libraries. *NodeJS* has the advantage of flattening the development stack as it uses *JavaScript* on the server side, and thus it is possible to develop a full *Mobile Cloud Computing* system using only *JavaScript*.

We have decided on using SignalR as we were able to integrate it with the .NET which is a more mature framework for implementing the server component.

4 Analysis and theoretical foundation

4.1 Theoretical overview

4.1.1 Titanium cross platform mobile application development

In Q1 of 2012, Android had 50.6% market share, while iOS had 23.8% of new sales. This means that their combined market share is approximately ¾ of all new smartphone sales.

But even if the two players dominate the market, developing for these two frameworks is time consuming as very little work can be reused, the programming language and development environment for *Android* and *iOS* being very different. Also if another *OS* must be targeted in the future, this would require rewriting the application.

In order to increase productivity a cross platform approach can be used. There is currently a wide range of cross platform solutions available including *Appcelerator Titanium*, *RhoMobile* and *PhoneGap*. *Appcelerator Titanium* is an open source project and has integrated most common functionalities of *iOS* and *Android*. Developing with Titanium compared to other *Cross Platform* solutions is that it allows for native development. For example when using *PhoneGap* you are actually developing a web app that run in a WebView UI component and has access to active services provided by the platform. When using *Titanium* you are programming in *JavaScript* and the framework translates the code into native *Java* or *Objectual-C*.

Titanium currently supports two operating systems: Android and iOS. It also plans on fully supporting *BlackBerry*, which is currently in Beta. One disadvantage is the lack of support for *Windows Phone* at the moment, if this OS needs to be targeted.

The development environment for Titanium is using its own *IDE* based on *Eclipse*, *Titanium Studio*), and it uses the JavaScript development language. *Titanium* can also be extended using module in order to offer complete native functionality, which is not supported by the Titanium platform.

Using *JavaScript* as a development language offers the benefit of low entry level compared to *Objectual-C or Java*. One difference when developing with *JavaScript* is that it requires understanding code organization when tackling large projects in the context of a language that was previously used for writing small code blocks in browsers. The recommended way of development in Titanium is using a *MVC* pattern. Also applying *OOP* best practices as encapsulation is the recommended approach (weather JavaScript is or isn't object oriented is highly debated).

Leveraging the *MVC* pattern will help produce cleaner code and promote reusability. Applying the *MVC* pattern within *Titanium* is relatively easy.

4.1.2 WCF

Windows Communication Foundation (*WCF*) is a framework for building service-oriented applications. Using *WCF*, applications can send data [13] as asynchronous messages from one service endpoint to another. An endpoint can be a client of a service that requests data from a service endpoint. The messages can be as simple as a single character or word sent as XML, or as complex as a stream of binary data.

In WCF all the communication details are handled by channel, it is a stack of channel components that all messages pass through during runtime processing. When are calling WCF service through a proxy class on the client side, it will send message (request soap message mainly includes some parameter values of method) and first the message will go through protocol Channels which mainly supports for security, transactions and reliable messaging, and second the message will go through encoder which convert messages into an array of bytes for transport, finally, the encoder message will go through the bottom transport channel which are responsible for transporting raw message bytes, WCF provides a number of transport protocols, including HTTP, TCP, MSMQ, peer-to-peer, and named pipes.

While creating such applications was possible prior to the existence of *WCF*, *WCF* makes the development of endpoints easier than ever. In summary, *WCF* is designed to offer a manageable approach to creating Web services and Web service clients.

4.1.3 REST

REST is a term coined by Roy Fielding in his Ph.D. dissertation [1] to describe an *architecture style* of networked systems. *REST* is an acronym standing for Representational State Transfer.

The Web is comprised of resources. A resource is any item of interest. For example, the Boeing Aircraft Corp may define a 747 resource. Clients may access that resource with this URL: http://www.boeing.com/aircraft/747 A *representation* of the resource is returned (e.g., Boeing747.html). The representation places the client application in a *state*. The result of the client traversing a hyperlink in Boeing747.html is another resource is accessed. The new representation places the client application into yet another state. Thus, the client application changes (*transfers*) state with each resource representation resulting a Representational State Transfer.

Roy Fielding's explanation [15] of the meaning of Representational State Transfer is "Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

The motivation for *REST* was to capture the characteristics of the Web which made the Web successful [5]. Subsequently these characteristics are being used to guide the evolution of the Web. *REST* is not a standard. There is no W3C *REST* specification. *REST* is just an architectural style. We can't bottle up that style. We can only understand

it, and design your Web services in that style. (Analogous to the client-server architectural style. There is no client-server standard.)

While *REST* is not a standard, it does use standards:

- HTTP
- URL
- XML/HTML/GIF/JPEG/etc (Resource Representations)
  text/xml, text/html, image/gif, image/jpeg, etc (MIME Types)

The Web is a *REST* system. Many of those Web services that we have been using these many years - book-ordering services, search services, online dictionary services - are *REST*-based Web services. *REST* is concerned with the "big picture" of the Web. It does not deal with implementation details (e.g., using Java servlets or WCF to implement a Web service).

4.1.4 Entity framework

The Microsoft ADO.NET Entity Framework is an Object/Relational Mapping (ORM) framework that enables developers to work with relational data as domain-specific objects, eliminating the need for most of the data access plumbing code that developers usually need to write. Using the Entity Framework, developers issue queries using LINQ, then retrieve and manipulate data as strongly typed objects. The Entity Framework's ORM implementation provides services like change tracking, identity resolution, lazy loading, and query translation so that developers can focus on their application-specific business logic rather than the data access fundamentals

Using the Entity Framework [13] to write data-oriented applications provides the following benefits:

- Reduced development time: the framework provides the core data access capabilities so developers can concentrate on application logic.
- Developers can work in terms of a more application-centric object model, including types with inheritance, complex members, and relationships. In .NET Framework 4, the Entity Framework also supports Persistence Ignorance through Plain Old CLR Objects (POCO) entities.
- Applications are freed from hard-coded dependencies on a particular data engine or storage schema by supporting a conceptual model that is independent of the physical/storage model.
- Mappings between the object model and the storage-specific schema can change without changing the application code.
- Language-Integrated Query support (called LINQ to Entities) provides IntelliSense and compile-time syntax validation for writing queries against a conceptual model.

4.1.5 SignalR

While sending data is fairly easy, listening for incoming information requires new approaches given that users might be charged additionally by the carrier and to avoid increased battery drainage.

Traditionally web applications used a polling technique for getting updates. For example a stock ticker knows that there are updates every 10 seconds, so it makes sense to implement a timer that gets the updates every 10 seconds from the server.

For the Order Taxi application this is not feasible as the polling interval is unknown. For example the driver could wait for hours without receiving an order, so he would poll thousands of times to receive one favorable result. Also the polling interval would have to be below 2 seconds (to offer a response time as low as possible), which would mean a high demand on the server and low scalability.

SiganlR is an Asynchronous library for .NET to help build real-time, multi-user interactive applications and is based on long polling techniques. Long polling is a variation of the traditional polling technique and allows emulation of an information push from a server to a client. With long polling, the client requests information from the server in a similar way to a normal poll. However, if the server does not have any information available for the client, instead of sending an empty response, the server holds the request and waits for some information to be available. Figure 4.1 illustrates this functionality by comparing time based polling and long pooling. Once the information becomes available (or after a suitable timeout), a complete response is sent to the client. The client will immediately re-request information from the server, so that the server will always have an available waiting request that it can use to deliver data in response to an event.

Long polling is itself not a push code, but can be used under circumstances where a real push is not possible. Using long polling in the client mobile application gets notified to update it’s data, and thus to synchronize with the server, exactly as the new data is available. Using a long polling approach there is a very low demand on the server which translates into a high scalability, as new users and drivers are added to the distributed taxi ordering system.

In the .NET framework SignalR represents an implementation of long polling. However SignalR does not rely only on the long polling technique. SignalR has a concept of transports, each transport decides how data is sent/recieved and how it connects and disconnects.

Transports build into SignalR are: WebSockets, Server Sent Events, Forever Frame, Long polling. SignalR tries to choose the "best" connection supported by server and client (a desired connection can also be specified implicitly). However WebSockets and Server Sent Events are still not widely supported at present.

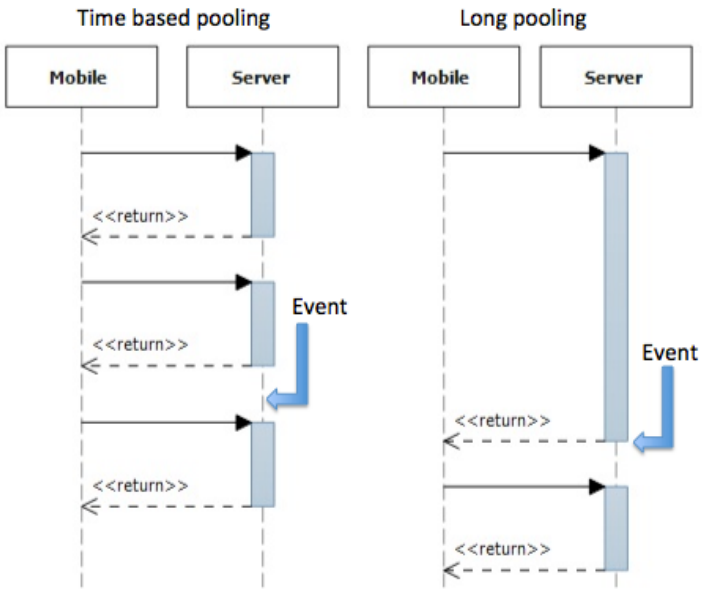

Figure 4.1: Long Pooling vs. Time-based polling

The implementation of *SignalR* in Titanium in Order Taxi app is by using a *WebView*, which triggers events once they arrive from the server. *SignalR* requires *jQuery* on the client side. On the server side it uses two concepts: *Hubs* and *persistent connections*. Taxi Ordering App uses *Hubs*. Extending the *Hub* class does implementation of a hub on the server. Each method defined inside this class is mapped to a *JavaScript* implementation at compile time. The *JavaScript* document is saved on the project root folder and the client must reference it. It is possible to have broadcast events, which alert all the clients or only one in particular. It is also possible to group client and inform a whole group at a time.

4.1.6 Push Notifications

Mobile apps are allowed to perform only very specific activities in the background, so battery life is conserved.

But there is required to be a way to alert the user of interesting things that are happening even if the user is not currently inside the app.

For example, maybe the user received a new tweet or their favorite team won the game. Since the app isn't currently running, it cannot check for these events. *Mobile OS* have provided a solution to this. Instead of the app continuously checking for events or doing work in the background, a server-side component for doing this can be written.

When an event of interest occurs, the server-side component can send the app a push notification, which is intended for signaling the app that there are event pending on the server. *Push Notifications* are not intended for transmitting data to the client app, but

are used as a signaling mechanism when the app is not running. *Push Notifications* for *Android* are called *C2DM* (*Cloud* to device messaging).

Push Notifications makes no guarantee about delivery or the order of messages. So, for example, while you might use this feature to alert an instant messaging application that the user has new messages, you probably would not use it to pass the actual messages.

Figure 4.2 represents how push notifications work. Apps need to register with the *Notification Service* that is *APNS* (Apple *Push Notifications Service*) for Apple and *C2DN* for Android. This assigns a specific id called device token that the app can send to the server. If the server needs to perform a push notification, it uses this device token to specify the app it want to send the notification to. Device tokens change so it is advisable to perform device token updates on the mobile preferably every time the app is opened. Users can also opt out of push notifications for the app or delete the app. In this scenario, push notification will not reach the user. The Notification service offers an *API* that the server can poll periodically to determine what device tokens are still active. Because communication between the server and the notification Service is not trivial, 3rd party services are available such as *Urban Airship*. Other operating systems also provide push notification, such as *Windows Phone* or B*lackberry*, the mechanism being the same.

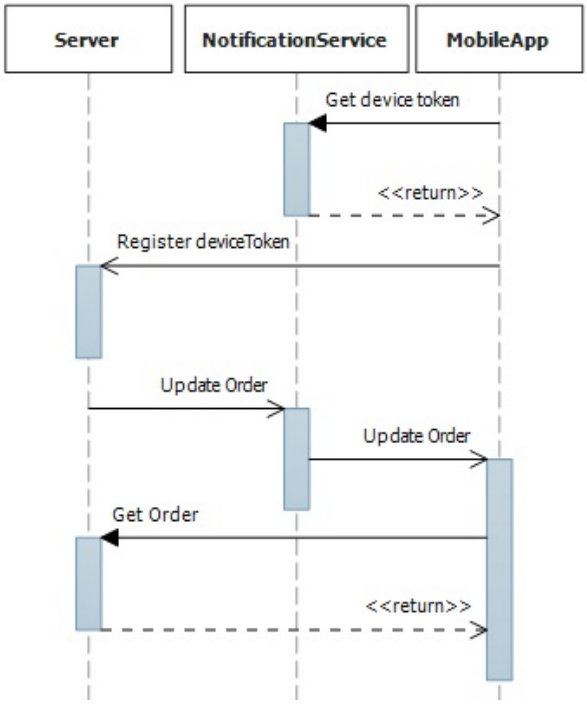

Figure 4. 2 Notifications mechanism

4.1.7 MVC

The purpose of many computer systems is to retrieve data from a data store and display it for the user. After the user changes the data, the system stores the updates in the data store. Because the key flow of information is between the data store and the user interface, you might be inclined to tie these two pieces together to reduce the amount of coding and to improve application performance. However, this seemingly natural approach has some significant problems. One problem is that the user interface tends to change much more frequently than the data storage system. Another problem with coupling the data and user interface pieces is that business applications tend to incorporate business logic that goes far beyond data transmission.

The *Model-View-Controller (MVC)* pattern separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes as presented in Figure 4.3.

Model. The model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).

View. The view manages the display of information.

Controller. The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

Figure 4.3 depicts the structural relationship between the three objects.

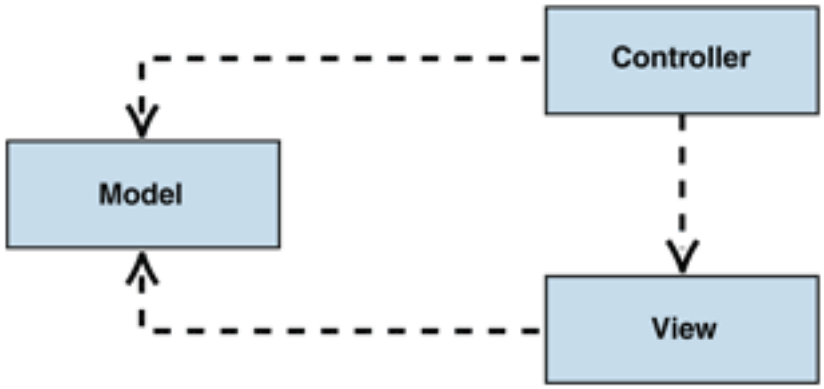

Figure 4.3: MVC class structure

Model-View-Controller is a fundamental design pattern for the separation of user interface logic from business logic. Unfortunately, the popularity of the pattern has resulted in a number of faulty descriptions. In particular, the term "controller" has been used to mean different things in different contexts. Fortunately, the advent of Web applications has helped resolve some of the ambiguity because the separation between the view and the controller is so apparent.

Advantages

- Allows multiple views to be constructed on the same data model. Because the view is separated from the model and there is no direct dependency from the model to the view, the user interface can display multiple views of the same data at the same time. For example, multiple pages in a Web application may use the same model objects. Another example is a Web application that allows the user to change the appearance of the pages. These pages display the same data from the shared model, but show it in a different way.
- Accommodates change. User interface requirements tend to change more rapidly than business rules. Users may prefer different colors, fonts, screen layouts, and levels of support for new devices such as cell phones or PDAs. Because the model does not depend on the views, adding new types of views to the system generally does not affect the model. As a result, the scope of change is confined to the view.

Disadvantages

- Complexity. The *MVC* pattern introduces new levels of indirection and therefore increases the complexity of the solution slightly. It also increases the event-driven nature of the user-interface code, which can become more difficult to debug.
- Cost of frequent updates. Decoupling the model from the view does not mean that developers of the model can ignore the nature of the views. For example, if the model undergoes frequent changes, it could flood the views with update requests. Some views, such as graphical displays, may take some time to render. As a result, the view may fall behind update requests. Therefore, it is important to keep the view in mind when coding the model. For example, the model could batch multiple updates into a single notification to the view.

4.2 Use case specification

There are four types of actors that the system supports, each having specific capabilities. We will analyze each actor together with the associated use cases.

4.2.1 User actor

The client use case diagram is depicted in Figure 4.4. The client actor is the user that has the Order Taxi application installed on his mobile phone. He uses the application to order taxis by sending his location to the server that consists of the *Cloud*. This is the main functionality of the system, which we will be presenting later on in a more detailed fashion. We note that this use case is composed of two elements: getting the *GPS* position of the user, which is used by the system to identify the address of the user, and it also offers the possibility of fix tuning this location to specify it more precisely, as this location will be sent to the operator or the driver.

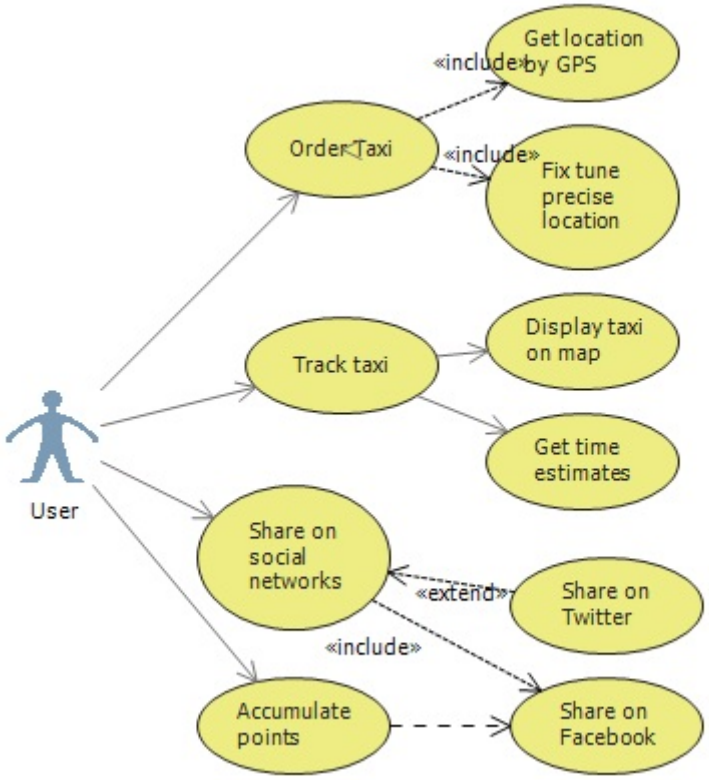

Figure 4.4 Client User

The track taxi functionality allows the user to see the taxi approaching on a map. In addition to this time estimation will be displayed together with a distance approximation. In case the response to an order comes from the operator, this functionality will not be available, as the taxi will not have GPS capabilities that allow tracking.

The Share on Social networks use case allows the user to share the current ride with his friends on social media. This is an element that enhances the marketing potential for the app and a key component in attracting new users. There are two types of services which users can use with the app: Facebook and Twitter.

One of the main benefits of using this app is the possibility of accumulating points, as a rewards program envisioned to spur app usage and attract new users. Each time the user will be making orders using the app, he will accumulate points which can be converted into free apps.

4.2.2 Driver actor

The Driver is responsible for responding to incoming orders. It does this by accepting or declining an order using a simple two buttons interface. The orders are assigned to drivers based on the proximity of the client to the driver.

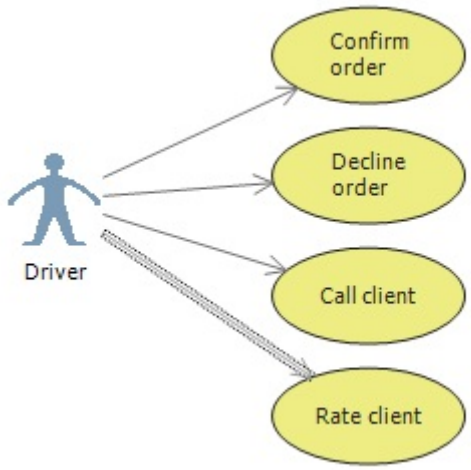

Figure 4.5 Driver Use Case

In addition to these functionalities, the Driver can also call the client in case he cannot see him at the specified pick up location or if he needs additional information. The driver cannot see the clients phone number, but only a call button.

The rate client functionality allows the driver to rate the client based on his behavior. The rating is based on considerations such as: was the client present at the meeting point, did he leave a tip, was he loudly etc. The rating is based on a 5 star system subjective to the driver. Future orders placed by the client will show the overall rating that this client has.

4.2.3 Operator actor

The operator is responsible for responding to orders that were not answered by drivers directly. This allows companies to use the traditional mechanism for finding a taxi and send a response to the client. Using this mechanism, the client always gets a response to his order, even if no driver is available in his proximity that is active.

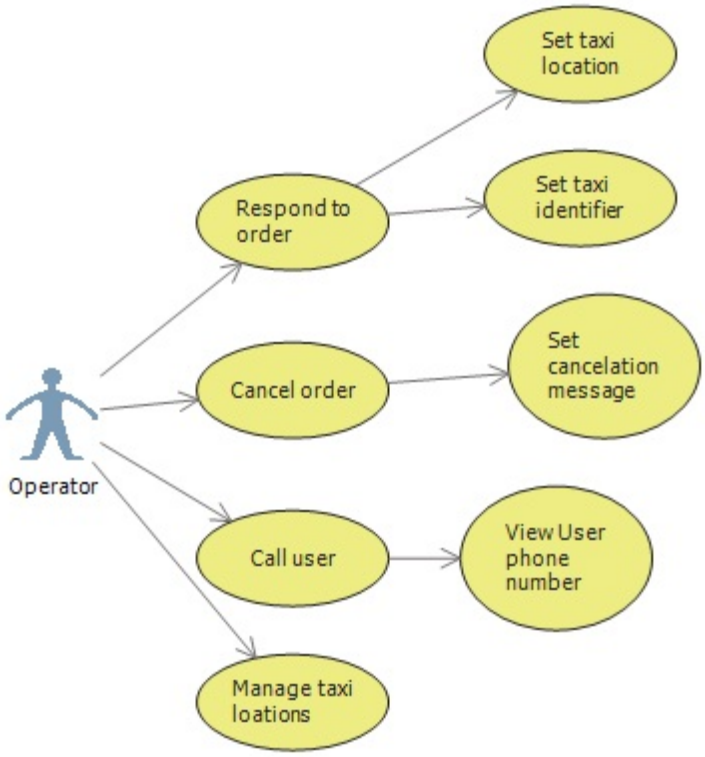

Figure 4.6 Operator Use Case

Responding to a client involves two things. First, an identifier for the taxi must be specified. This consists of the car number or some other way for the client to identify the taxi. Also the operator must specify the approximate location where the taxi is at the moment, by selecting it from a map. This creates a more visual context for the client and creates a sense of consistency between orders answered by the driver and the operator.

Similar to the driver, the operator can call the client to ask him more details. Distinctly from the driver, the operator can see the users number.

As we earlier specified, when a response to an order is set, the operator selects a address from the map which will display the initial taxi position to the client.It is possible for the operator to cancel an order from the client, in which case e can write a cancelation message explaining the decision.

4.2.4 Manager actor

The Manager actor is responsible with registering taxi companies into the system, setting the areas in which they operate, adding drivers and creating operator accounts. The manager is a role comparable with the administrator of the system. It has the power to change accounts, activate or deactivate them.

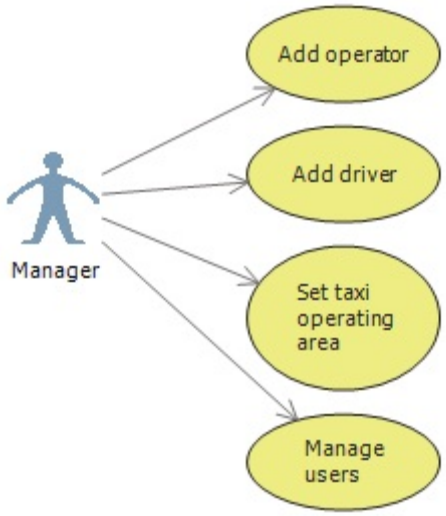

Figure 4.7 Manager use cases

4.2.5 Order Taxi use case detailed

We will present the Use Case for Order Taxi, which is the base activity of the system. Figure 4.8 depicts the flow of activities constituting this use case implementation. Alternative flows are also described below together with preconditions and post conditions.

Scalability is mentioned as one of the important requirements of this use case as sending multiple orders concurrently can affect the server response time, which has to remain within the bounds specified by the non-functional system requirements in chapter 2.

Use case description

Use-Case Start

The User must choose to send a new order to the system

1. The User inputs the address.
2. The App prompts for order confirmation.
3. The App verifies whether the user is registered and has a validated phone
4. The App encrypts the data, and sends it to the server. The user is displayed with a response pending window
5. The system send a response back to the user with the taxi details which are displayed on the mobile device.

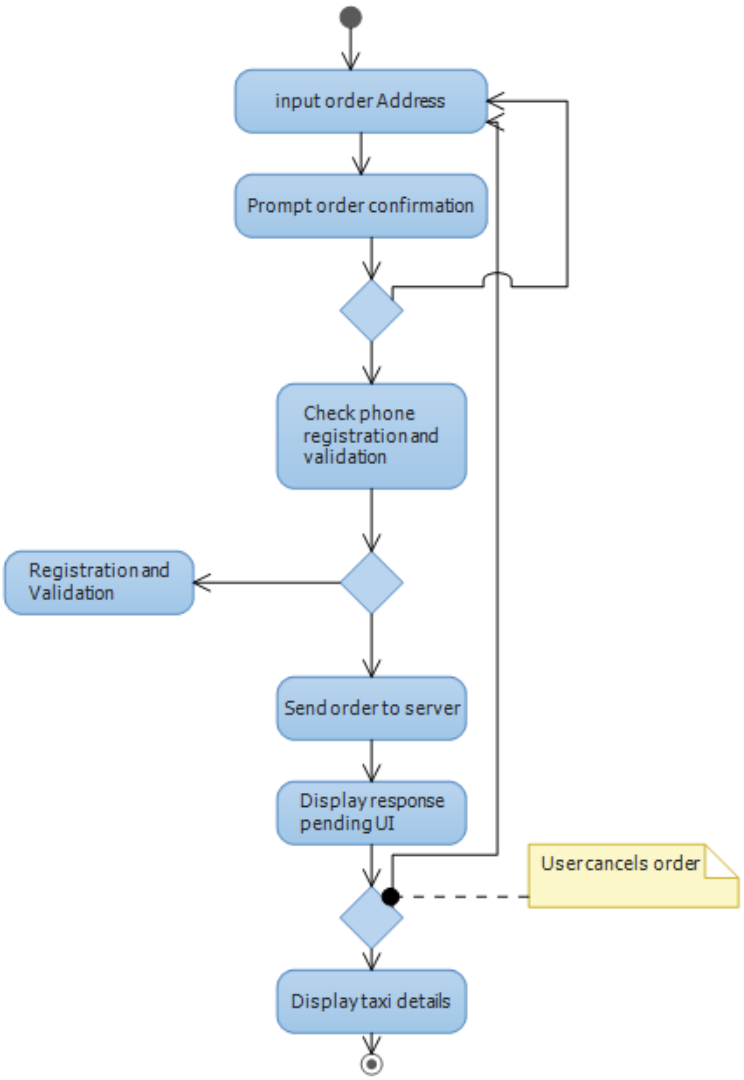

Figure 4.8: Flow of events for Order Taxi usecase

Alternative Flows

- The user is not registered or the phone is not validated

The User didn't register his phone. Before the user can place his order, he will be prompted to the phone registration and confirmation window.

1. The user fills his name and phone information
2. The system send the user an SMS with the registration code
3. The user confirms the registration code

- Operator cancels the order. The user is waiting for a response but operator cancels his order. In this case
    1. The App displayed a dialog notifying the user
    2. The user gets displayed the order taxi view
- Order not responded .The Operator fails to respond to the user in a specific period of time
    1. The User is displayed a call button for immediately calling the operator
    2. Clicking the button automatically dials the operator phone number

Special Requirements

- Scalability

The scalability is assured by implementing SignalR framework for long polling mechanism. This allows for persistent connections between the distributed system components with very low overhead.

Preconditions

- The presented scenario is a result of the User clicking the Order button
- There is a connection between the distributed system components.

Postconditions

- There is a persistent connection between the mobile and the server

4.3 Conceptual design

The Client and the Driver apps are applications for mobile devices. These applications communicate with the Cloud service using an API exposed by the service. The service is responsible for acting as a data synchronization mechanism, being the center of the system, similar to a *Hub*, as can be seen in Figure 4.9

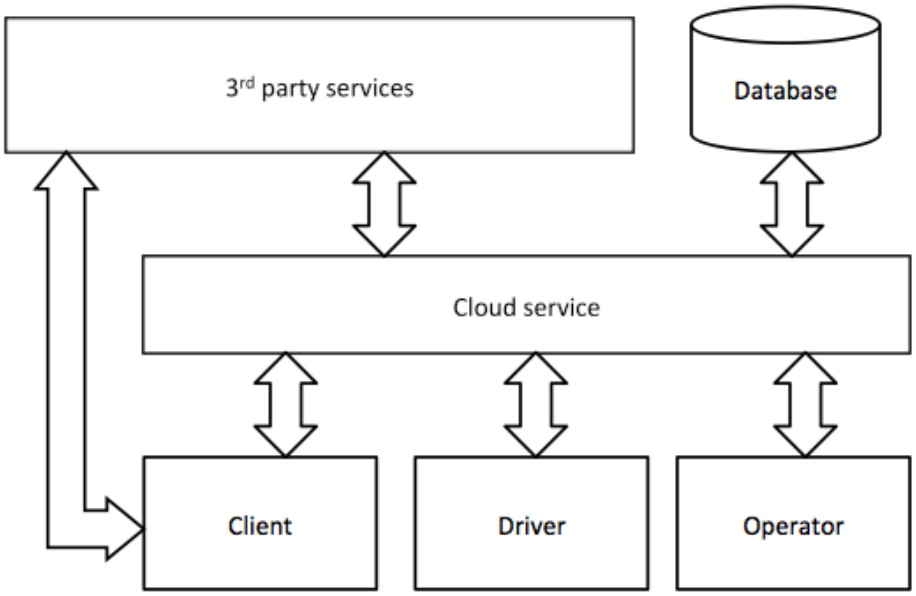

Figure 4.9: The block diagram of the system

Client and Driver apps and the Operator web interface are the consumers of the *API* exposed by the Cloud service. The Cloud Service acts as the intermediary between all communication that takes place between these components and handles the data synchronization. The Cloud service relies on its functionality on using a database for persisting information such as orders, responses, users etc. The Cloud service also interacts with external services for performing specific tasks. Examples of this include sending push notifications or reverse geocoding. The client mobile applications can also access external services. For example the Client app accesses Facebook and analytics services.

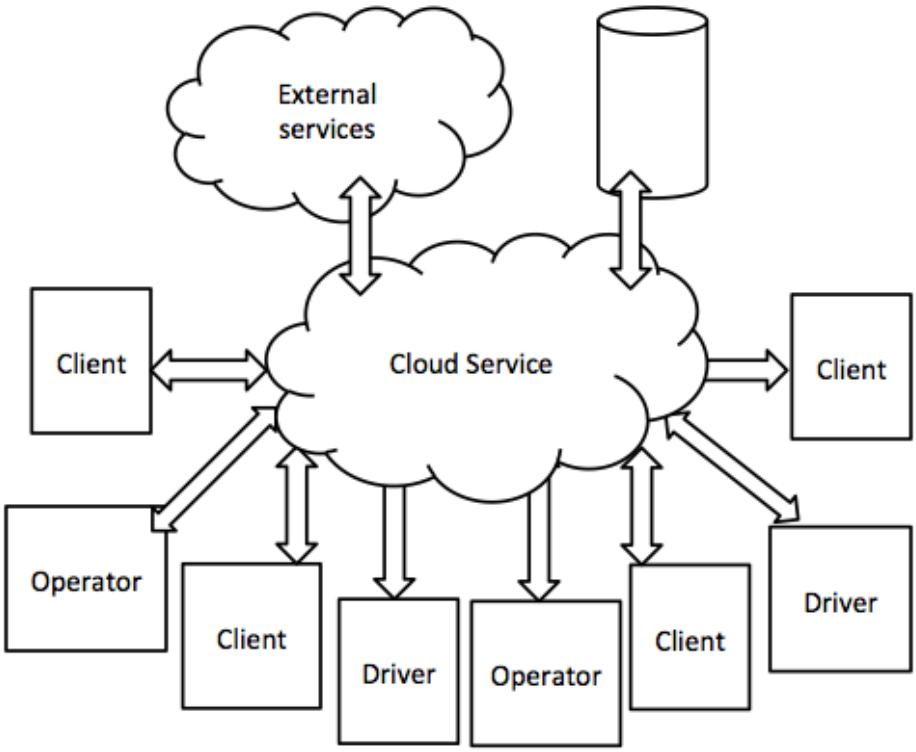

Figure 4.10 The Cloud service acting as a hub intermediating all communications

Figure 4.10 shows the Cloud service acting as a hub and intermediating all communication between the clients. While for a client ordering a taxi seems as a *P2P* operation, it actually constitutes a client-server operation[8]. As illustrated, numerous operators, divers and Clients can connect to the Cloud, new ones can be added, or some can be removed at any time. For example a taxi driver that terminates his work schedule for the day, can disconnect from the Cloud service and will not receive any new orders until he checks in again with the service.

All data is stored into a proprietary database, and specific tasks that have been delegated to external services are managed by the Cloud service. When a client orders a taxi, the Cloud service checks trough the fleet of taxis that are available on that area at that moment and selects one for being dispatched to the client. In case no taxi is received, the Cloud service sends the order to a specific taxi company in order to be processed.

Not all mobile applications are required to have a server component. One example is an application that is used to show the battery drain level. Though most applications have a service backend. If the service backend logic is fairly simple, for example if it is used solely for storing data, the developer can opt in for a 3rd party backend server such as *Parse.com* if the business logic is simple enough to allow this.

However applications that require a higher degree of complexity need to create their own model and an API for exposing the services.

Mobile Cloud Computing (MCC) [1] refers to an infrastructure where both the data storage and the data processing happen outside of the mobile device. Mobile Cloud

applications move the computing power and data storage away from mobile phones and into the Cloud, bringing applications and mobile computing to not just smartphone users but also a much broader range of mobile subscribers.

Cloud computing is known to be a promising solution for mobile computing due to reasons including mobility, communication, and portability, reliability, security. In the following, we describe how the Cloud can be used to overcome obstacles in mobile computing, thereby pointing out advantages of *MCC*[6].

Extending battery lifetime

Battery is one of the main concerns for mobile devices. Several solutions have been proposed to enhance the *CPU* performance, and to manage the disk and screen in an intelligent manner to reduce power consumption. However, these solutions require changes in the structure of mobile devices, or they require a new hardware that results in an increase of cost and may not be feasible for all mobile devices. Computation offloading technique is proposed with the objective to migrate the large computations and complex processing from resource-limited devices (i.e., mobile devices) to resourceful machines (i.e., server in Cloud). This avoids taking a long application execution time on mobile devices, which results in large amount of power consumption.

Studies evaluating large-scale numerical computations shown that up to 45% of energy consumption can be reduced for large matrix calculation. As a consequence computationally intensive tasks should be delegated to the Cloud as CPU intensive work can drain the battery level significantly.

Improving data storage capacity

Storage capacity is also a constraint for mobile devices. *MCC* is developed to enable mobile users to store and access large data in the Cloud. Examples of Cloud services for doing this are: Azure Storage and Amazon S3. Storing data in the Cloud also allows for data synchronization along multiple devices. *iCloud* is a service that allows work that is began on an iPad to be continued on a Mac for example. Also large database services are available which offer better scalability. These databases include rational databases (*SQL Azure*), highly scalable Table Storage databases and *NoSql* databases such as *MongoDB*.

Improving reliability

Storing data or running applications on Clouds improves reliability since they are backed up on a number of computers. This reduces the chance of data and application lost on the mobile devices. *CDN* can be used to provision data as close to the customer as possible to improve access time. In addition, *MCC* [10] can be designed as a comprehensive data security model for both service providers and users. The *Cloud* can be used to protect copyrighted digital from abuse and unauthorized distribution Also, the *Cloud* can remotely provide mobile users with security services such as virus scanning, malicious code detection, and authentication . Also, such Cloud-based security services can make efficient use of the collected record from different users to improve the effectiveness of the services.

When considering mobile applications, REST services are preferred over *SOAP* as they are lighter and easier to implement. Among the *REST* services, there are two common types: *JSON* and *XML* based. *JSON* is especially preferred when developed with Titanium, as *JSON* objects are pure *JavaScript* objects. Also *JSON* representations are less verbose. *JSON REST APIs* are offered by services like Google, Facebook, Twitter, Flickr etc.

We have analyzed the theoretical approaches and the frameworks required for implementing the proposed application. In the following chapter we are detailing how we implemented the Taxi Ordering application.

5 Detailed Design and Implementation

5.1 Specific implementation

In this chapter we are implementing the described application based on the theoretical analyses described in the previous chapter.

Figure 5.1 depicts the system general architecture as implemented in the Order Taxi application. While the functionalities for the building blocks have been described previously, we now concentrate on more specific implementations of various features.

5.1.1 General Implementation Architecture

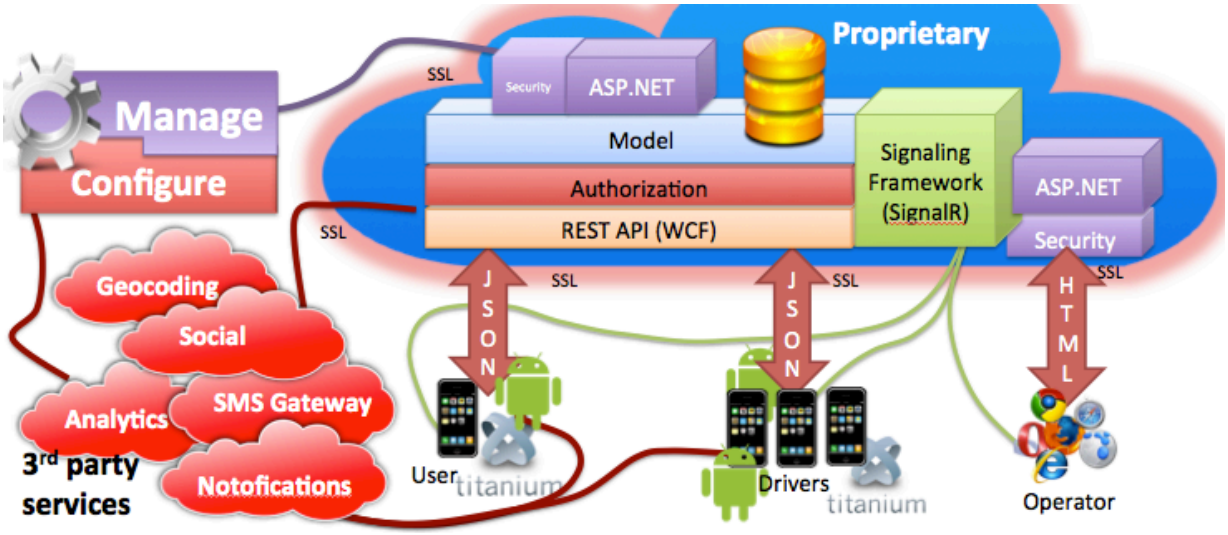

Figure 5.1: General Implementation Architecture

There can be identified two types of components. The first type includes the proprietary components. This need to be developed by the developers, are based on specific frameworks and include: the components constituting the Cloud service (designating the service built by the developers), the mobile application, the browser client and the management interface. The second type of components are represented by the *3rd party services* (distinctly from proprietary services, *3rd party services* or *external services* are built by other organizations and are made available to the public for use) are exposed trough certain *API*. In this example we are using 5 types of external services:

The Notification Service – used for sending *Push Notifications* (a way of alerting a mobile device), social services that offer integration with *Facebook* and *Twitter*, *Analytics* services for tracking usage patterns, *SMS* gateway service for sending the user an *SMS* with the activation code (required by the business rules), and a geocoding and reverse geocoding service for converting *GPS* position into street address. All of these components will be explained together with their implementation technologies and internal working.

For understanding how the whole system works together, let's examine the most important functionality of our *Order Taxi* application: sending an order by the client.

When a user orders a taxi from it's mobile phone, a request is serialized as *JSON*, signed using the user key and sent to the server trough the server *API*. The server determines the nearest driver to the user, by interrogating the database based on the business logic. Once the nearest driver is determined, the *Signaling Framework* (used to alert the mobile application) is used to determine the selected driver's app to download or update the order sent by the client. After the driver confirms the order, the server uses the same approach to inform the user that his order was responded.

All requests sent to the proprietary server components need to use SSL for proving security to the system.

The Order Taxi application exposes *JSON REST* services using *WCF* Framework. Working with *JSON* in the context of *WCF* is easy as the framework automatically serializes *JSON* request and responses and performs data binding between *JavaScript* and *CLR* objects.

Once the data is received on the server, the server performs some security check on the request. This usually implies authentication and authorization of the request. As there is no password required for the Taxi Ordering application, signing the data using a key, which is known only by the client and the server, and is unique for each client mobile application, does this.

Figure 5.2 depicts the integration between the client mobile application and the server. The use case described in the sequence diagram is the one for Send Order functionality. On The client we have the *MVC* pattern. The view sends an event to the controller, which calls the send order operation on the *Model*. The model component responsible for sending the request (implemented in Titanium using an *HTTP Socket*), serializes the data and sends them to the specified service endpoint exposed by the API. Once the request arrives on the server, the reverse process occurs as the JS data objects are translated into *CLR* objects by the *WCF* framework. The security component is responsible for checking the signature of the user in the case of Order Taxi applications. In other applications a username and password approach can be used for obtaining the same result. It is essential to observe tat once the request passes the security verification, it is recorded and the registration key for the order is returned to the client. Further on the client uses this registration key for referring to his order. At this point there is no response available for the client yet.

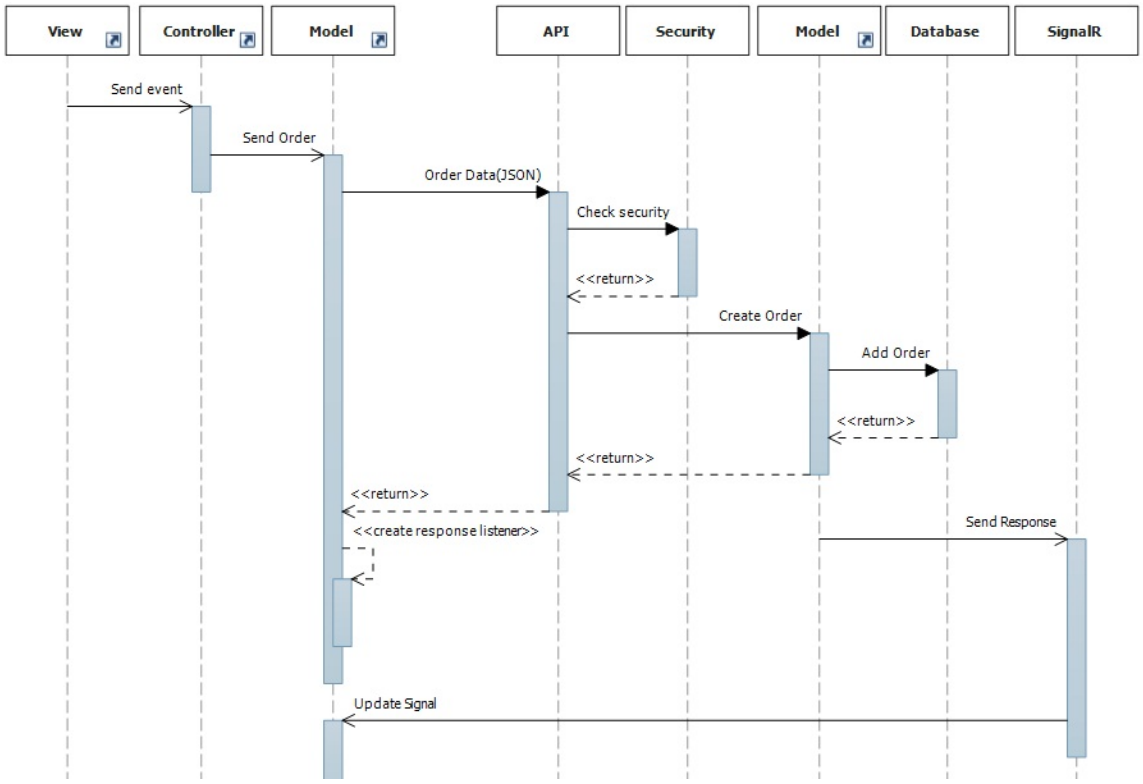

Figure 5.2: Integration between the client and the server

The order placed by the user in the system is assigned to the nearest driver. When the driver confirms the order, a response is sent to the client mobile application. This response can come after a variable amount of time which can be anywhere between a few seconds and 3 minutes. In consequence *SignalR* is used to inform the client when the response is ready. SignalR is not used to pass the response directly to the user, but rather the user applications download the response once it receives the update signal.

An important characteristic of the service is that it should offer scalability. All the components depicted can be hosted in the *Cloud*. This allows an elastic hosting environment, which can be used easily. For Order Taxi application, we are using *Windows Azure*. Using *Windows Azure* we can easily deploy the server component as it was built completely on the *.NET* framework. Though *Windows Azure* can also be used with many other development languages.

5.1.2 *Cross platform mobile application design*

Leveraging the *MVC* pattern will help produce cleaner code and promote reusability. Applying the *MVC* pattern within *Titanium* is relatively easy.

Considering the Order Taxi App, we will decompose it into its three components: *Model*, *View*, and *Controller*.

The *view* is responsible for creating the window and adding the necessary *UI* components. Views may optionally specify styling properties (color, font, and positioning). The recommended practice is to extract all styling properties into external *JSS* files, which promotes easy style modifications and customization. Figure 5.3 displays the view for the taxi ordering application. It is composed of a Window, on which are added a map, a textbox and a button. When the button is pressed it fires an event, which is handled by the controller.

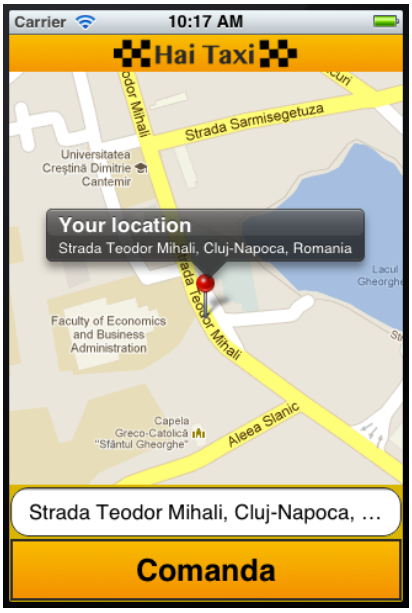

Figure 5.3: The view of Taxi Ordering App

The *controller* is responsible for managing events (button taps) and navigation. For example when the tap event is registered for the Order button, a notification is sent o the model together with the data the user inputted in the textbox. Some developers use event handlers inside the controller to communicate with the server. While this practice is convenient for simple views, it leads to unorganized code when there are multiple services to be accessed. In such a scenario it is advisable to move the server interaction on the model where it can be encapsulated and offer better reusability.

The *model* is typically a *JSON* payload retrieved from a *Restful* service. For example, the *handleResponseEvent* will return response information from the server.

The advantages of using *MVC* in *Titanium* include better code organization and readability, promotes reusability and maintainability.

5.1.3 GPS component

Titanium offers the possibility of interacting with device capabilities such as using the *GPS*. *Titanium* allows for specific parameter to be set on the *GPS* such as the precision of positioning, which must be a balance between time and required accuracy.

When the app starts the coordinates are the ones cached on the device, in other words the ones detected at the last time you used the *geolocation* on your device. Detecting the current position takes time so if you have an app based on *geolocation* you might want to take this in account to improve the user experience and avoid to get false results.

Titanium.Geolocation provides the needed methods to manage geolocation.

In general the most used method is *getCurrentPosition* that gives the current position and fires once and the *location* event that triggers repeatedly on the location change. To be more clear think you want to log a route the user follow: when the app starts you get its start position then while walking you can get the points of the route with a adjustable granularity.

There are some other methods that can be used like *reverseGeocoder*, *forwardGeocoder*, and *getCurrentHeading* – for compass. Because not all the devices have the *GPS* location (like iPod) or the compass and because the user can have the location services disabled you have to check if it exists, if it's enabled and then let the user know about this.

5.1.4 Server model implementation

The server model is responsible for implementing the business logic of the system. The service model is composed of a set of classes representing the implementation of the domain model. Figure 5.4 represents the most important objects that constitute the domain model. Because *Entity Framework* was used for mapping *CLR* objects to database records, these entities have a corresponding implementation in the *SQL* server database.

We are outlining the most important entities present in our system:

Order – represents an order placed by the client into the system. The Order has a set of attributes representing:

- State of the order: *IsResolved*, *IsCanceled*, *IsPendingResponse*. This attributes are used for identifying the current status of the order.
- Modification time: Each modification of the state of the order also setts a time stamp on the object, which is used by the system. For example when an order is created, it is displayed to the operator together with a label showing how much time has passed since the order was submitted.
- Data associated with an order: represent information such as the rating of the taxi driver or the feedback of the client following the ride.
- Associated data such as the pick up location or the owner is represented as references to the specific entities.

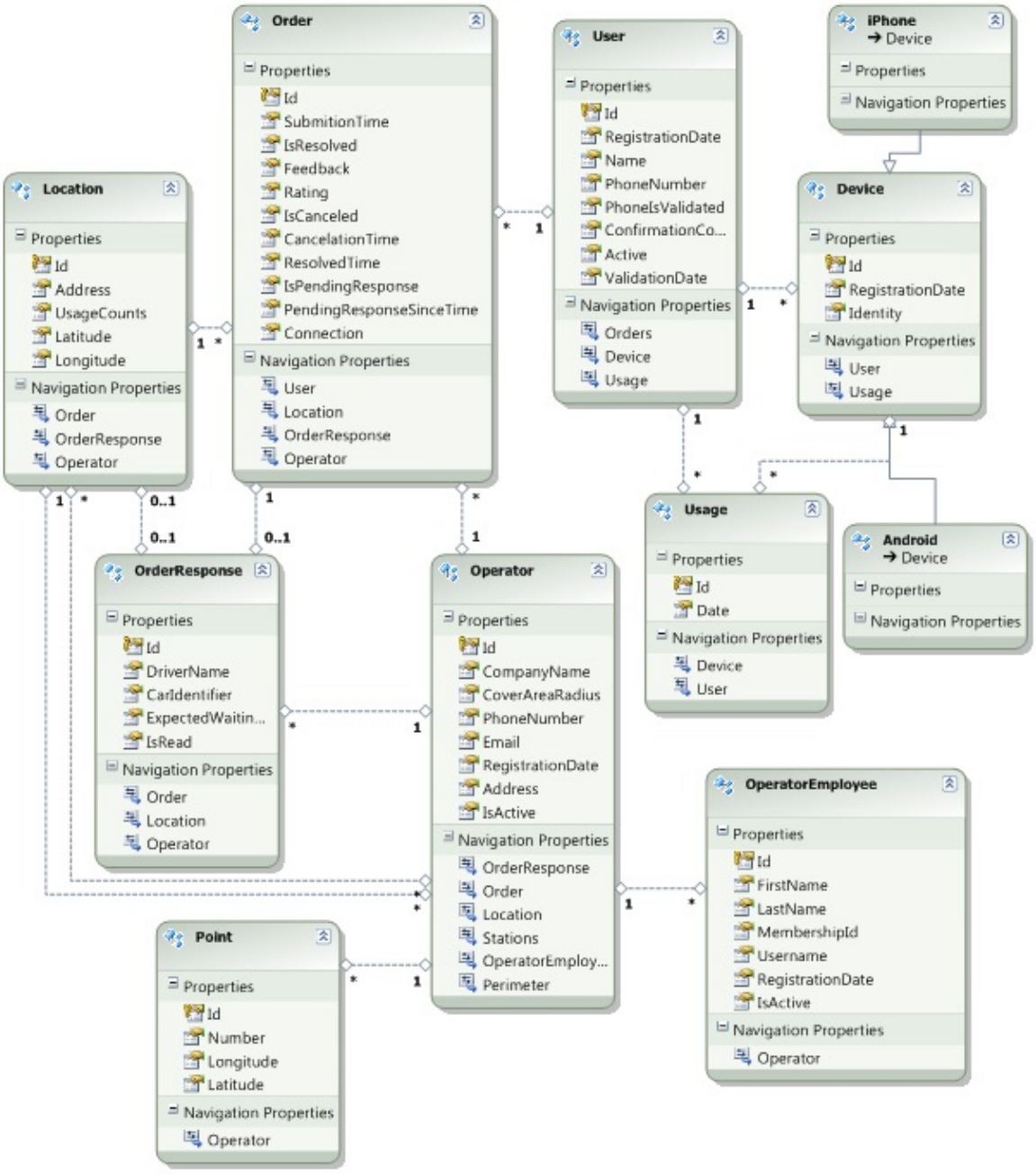

Figure 5.4 The entities composing the system model

User – represents the client who is placing orders into the system. The user has the following attributes associated with him:

- Name – represents the name chosen by the person submitting orders.
- Registration date – the date when this user registered by submitting his phone number and name
- Phone number is used for contacting the person in case he doesn't show up at the pick up location

- Phone is validated is used to determine whether the specific user was validated by the validation code sent trough the *SMS*.
- The user must be marked as active in order to be able to place orders into the system. A user state can be changed to inactive in case abuse has been observed for this particular user.
- The confirmation code is the activation code sent to the user trough *SMS* that is used to determine if the phone number is valid.

Location: Represents the position where an order is set. It contains the *GPS* coordinates for the location together with an optional name for it. There is also a property used for counting how often that location is used. The more frequent locations will be displayed first in the location suggestion list.

Operator: This entity represents an operator, which might me understood as a taxi company or a collection of taxi companies unified as a group. This entity presents the following properties:

- Activation date and time, when this operator has been registered
- A perimeter that represents the are where the operator is active, represented by a collection of points.
- Stations represent predefined locations on the map where taxis are expected to be present.

OperatorEmployee: This class represents the operator that responds to orders.

- The name is the real name of the operator
- *IsActive* property is set to true for those *Operators* which are active, meaning having permission to access the system, permission set b the manager.

OrderResponse: is created by the *OperatorEmployee* and represents the response the user receives for his order. It contains the necessary information required by the client such as the car number, the name of the driver, the expected waiting time for the user. It also has a *IsRead* property which is used for determining whether the response has been read by the user application. If the Response is not read in a certain amount of time, this means that that the client application is closed and as a consequence the response needs to be delivered using push notifications.

Point: This class is used for defining perimeters in which an operator activates. Ay order placed within this perimeter might come to the operator. The number represents the count for the point, so that given a collection of points and their orders, the system can check whether a specific point is inside that perimeter or outside of it.

The entities represented in Figure 5.4 are mapped by Entity Framework into the CLR classes represented in Figure 5.5. These classes are further extended in functionality by means of partial classes, by adding functionality as for example in the Operator class.

- *GetOperator* method is an extension to the Operator class which allows for obtaining a list of operators that are available for a specific GPS position, which is used when a user places an order. When a user places a new order,

this method is called and the order is assigned to one of the operators returned by this method based on a predefined logic.

- *GetCentroid* method obtains the center of the polygon which defines the operating area for the operator
-

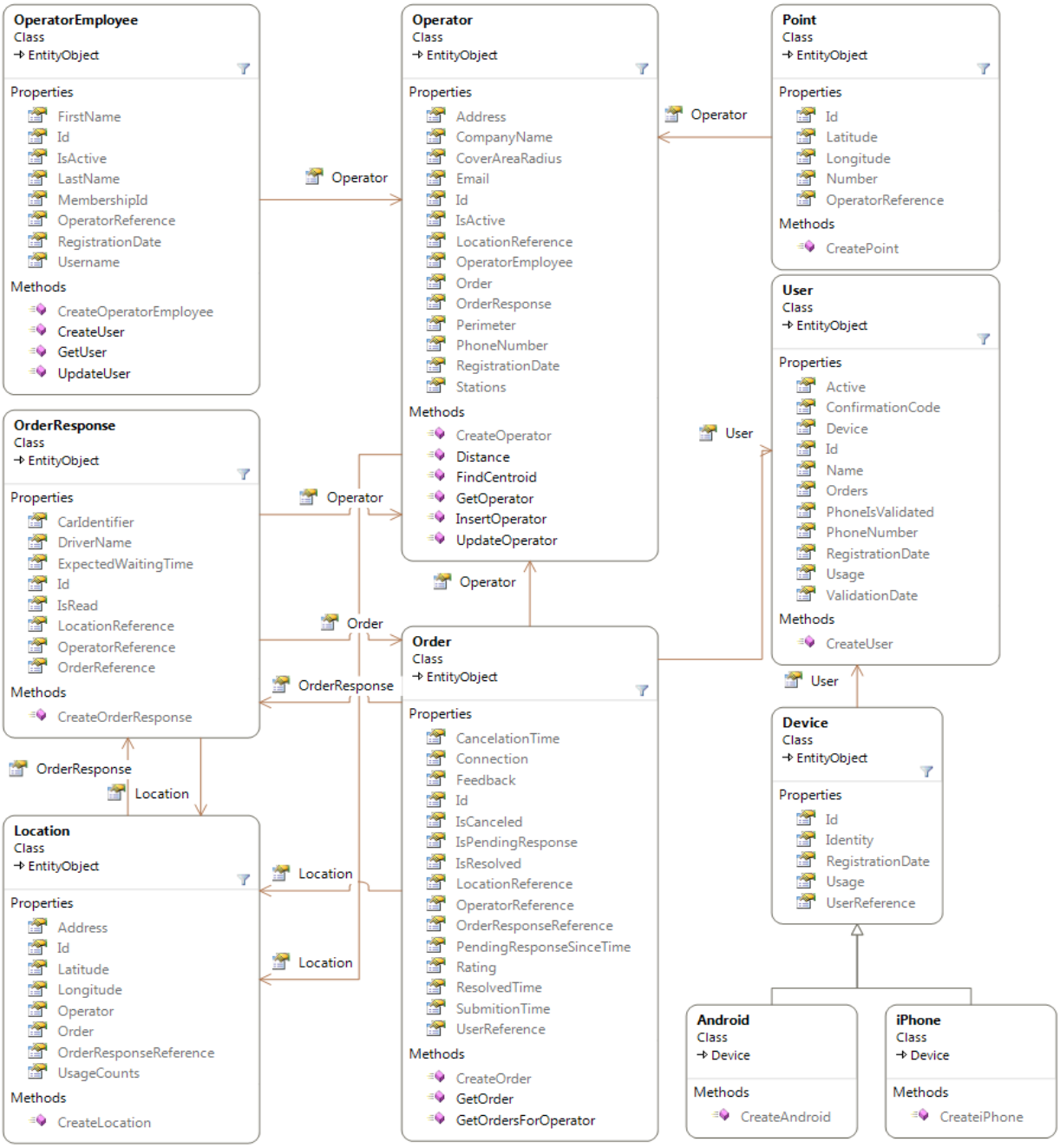

Figure 5.5 The class diagram of the system model

5.1.5 *Service database*

The service database is created using *Entity Framework* and is depicted in Figure 5.6. We can observe that the *Entity Framework* entities are mapped to these tables, providing the same properties as columns in the rational database.

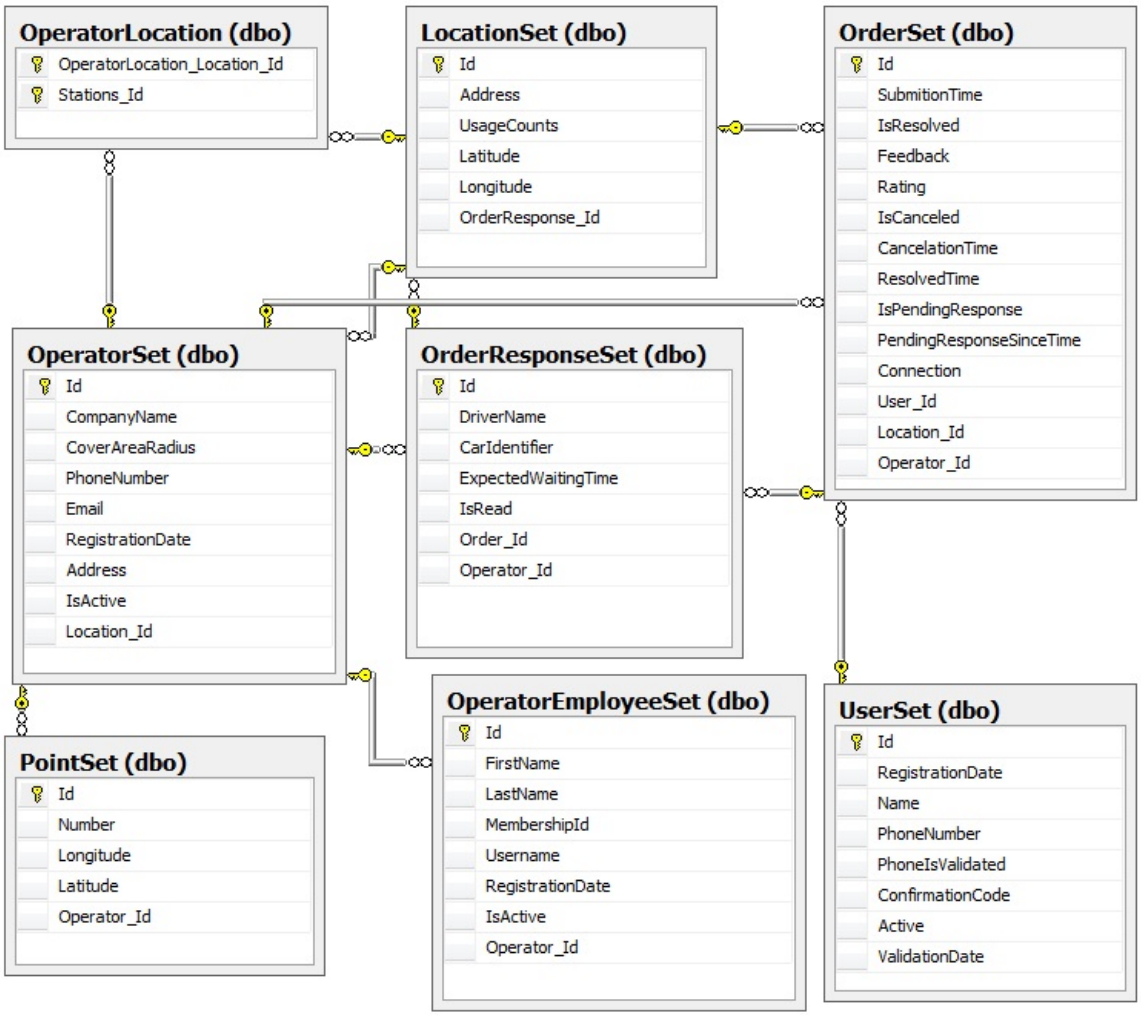

Figure 5.6 The database tables corresponding to the system entities

We are describing the most important tables used for persisting the application data and which have been generated based on the entities described earlier in this chapter.

Table *OrderSet* contains the columns representing the information described by the order placed by the client into the system. The state of the order in of type nVarChar an its value is constrained by the implementation to three valid values: *IsResolved*, *IsCanceled*, *IsPendingResponse*. These attributes are used for identifying the current status of the order and is defined not to allow nulls. During each modification of the state

of the order also sets a time stamp on the object, which is used by the system. For example when an order is created, it is displayed to the operator together with a label showing how much time has passed since the order was submitted. These columns are defined as DATETIME types.

The *UserSet* table represents the client who is placing orders into the system. The most important columns of this table include the name which represents the name chosen by the person submitting orders, the registration date – the date when this user registered by submitting his phone number and name. The phone number is used for contacting the person in case he doesn't show up at the pick up location, and is represented as a nvarchar. We used this representation especially because phone number formats are very different for distinct regions. *IsPhoneValidated* column is used to determine whether the specific user was validated by the validation code sent trough SMS. The user must be marked as active in order to be able to place orders into the system, this column being represented as a non-nullable Boolean type. A user state can be changed to inactive in case abuse has been observed for this particular user. the confirmation code is the activation code sent to the user trough SMS that is used to determine if the phone number is valid. This column is also represented as an nVarChar.

The table *OrderSet* has the foreign key represented by *User_id* that specifies the User which placed the order inside the system. *OrderSet* and *OrderResponseSet* have a one to one relationship between them. We decided to separate this two tables, even though there is a synonymy relationship between them as there can be order, which do not have a response, as for example an order that was canceled.

We can observe that there was no case for splitting entities in our model. In this mapping scenario, properties from a single entity in the conceptual model are mapped to columns in two or more underlying tables. In this scenario, the tables must share a common primary key. In our mapping scenario each entity in the conceptual model is mapped to a single table in the storage model. This is the default mapping generated by Entity Data Model tools.

Each operator can define multiple stations which represent positions on the map which an associated name (records in the *LocationsSet* Table) that are by the operator to specify the initial taxi position easily, by clicking the marker on the map associated to that specific location. The mapping of a list of stations (defined as actually being of type location) to the *OperatorSet* is done by *EntityFramework* by introducing a new table called *OperatorLocation*, as can be observed in Figure 5.6.

We are using this database model representing the conceptual model of the system inside *SQL Server 2008*. We have decided on implementing our database using Entity Framework generate Database from Model approach, as it offered a productive environment where changes in the model could be easily translated into updates of the database. We also notice that there is a balance between the number of read and write operations as every record inserted into the database needs to be read usually just one time, processed and than archived. For example an Order is created, added to the system, it is answered by the operator, the response is sent back to the client and after this there response is no longer used actively and constitutes solely a history used for creating suggestions as for example suggesting pick up locations.

5.1.6 Service REST *API*

The interaction between the client mobile and the server is performed trough the means of a *REST* API that has the methods depicted in Figure 5.7. We implemented the *REST* API trough WCF.

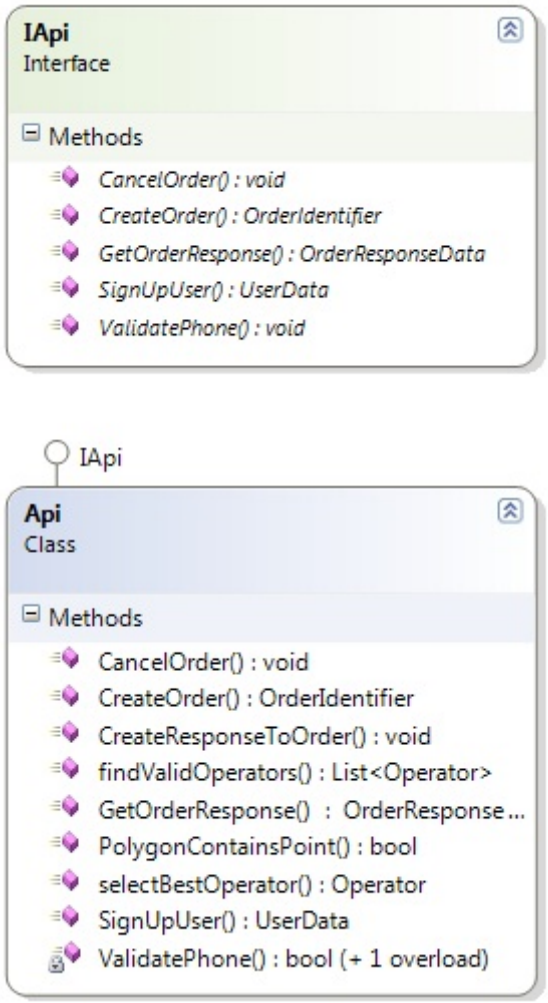

Figure 5.7 The service API using the interface IApi and its implementation in the class Api

IApi is the interface which is implemented by the API class. IApi exposes the following methods: CancelOrder, CreateOrder, GetOrderResponse, SignUpUser, ValidateUser.

SignUpUser – this method is exposed as a resource using the verb POST and is available by invoking the service at the location /Api/signupuser. The data passed to the method are encoded as JSON and consist of the User name and phone number. Once this data are received on the server the system sends the user an SMS message at the specified

phone number. At this moment the user status is set as invalidated. In order to validate the phone number the user needs to send the confirmation code sent to him trough SMS by using the next method described, ValidatePhone.

ValidatePhone – this method is used to validate the users phone number by sending using the POST verb the code sent to him by SMS. This rest resource is available at /Api/validatephone. The data needs to be submitted in the JSON format, as with all API calls to our service. When the code reaches the server, it is checked whether the code matches the one in the system. If the code is correct, the user is allowed to submit the order.

CreateOrder – this method allows for creating a new order inside the system. This is performed by sending the information required b the order to the address /Api/order using the POST verb. Once an order is received on the server, it is recorded inside the database; it gets displayed on a specific operator panel. Once the operator responds to the order, an OrderResponse entry is created and an update signal is sen to the client. Among the attributes in a create order request the most important are:

- The location where the order was sent for. This consists of the GPS position.
- The user who submitted the request
- The date and time the order was created
- The device information

GetOrderResponse – allows the client to read the response to his order. Accessible using the GET verb at the address /Api/order. The request contains the identity of the order. The response is encoded as JSON and contains the information for the client such as:

- The car number which was assigned for the order
- The information regarding the driver
- The estimated arrival time of the driver

CancelOrder – allows an order to be canceled by the user. The cancelation is performed by the client, and is accessible at /Api/cancelorder. The request verb is POST, as the submitted information contains the identity of the order that should be canceled. We note that we decided on not implementing the DELETE verb combined with the call at /api/order as some Api best practices recommend as in our case there is no associated delete operation on the server. When an order is canceled what happens is that the IsCanceled property is set for the order, and an update notice is sent to the subscribers, such as the operator.

We implemented Api versioning in order to manage successive updates to our Api without affecting previous versions of the application.

5.1.7 Security

The security is implemented using ASP.NET membership as described in the database diagram in Figure 5.8.

By implementing ASP.NET Membership Framework for handling the authentication and authorization logic of the applications we were able to integrate this services with our application easily. The benefit from a robust and throughout tested framework offers much better security than self-implemented frameworks. Figure 5.8 depicts the tables that constitute the implementation of the database behind ASP.NET Membership. *ASP.NET_User* table is mapped to the *UserSet* table inside the system database trough the use of the Membership attribute. As a consequence, the Security framework was not extended and was allowed to operate as it is. By not extending or close integrating any services of the Membership framework with our system, we decoupled the two subsystems and allowed for further changes.

The configuration for the security framework is done in the *Web.config* file. The authentication method chosen is Forms authentication, and *requireSSL* was set to true in order to use the self signed certificate which we created.

```
<authentication mode="Forms">
   <forms loginUrl="Login.aspx"
        protection="All"
        timeout="30"
        name="AppNameCookie"
        path="/FormsAuth"
        requireSSL="true"
        slidingExpiration="true"
        defaultUrl="default.aspx"
        cookieless="UseCookies"
        enableCrossAppRedirects="false"/>
</authentication>
```

The configurations are explained below:

- **loginUrl** points to the login page.
- **protection** is set to **"All"** to specify privacy and integrity for the forms authentication ticket.
- **timeout** is used to specify a limited session lifetime.
- **name** and **path** are set to unique values for the current application.
- **requireSSL** is set to **"true"**. This configuration means that authentication cookie can not be transmitted over channels that are not SSL-protected
- **slidingExpiration** is set to **"true"** to enforce a sliding session lifetime. This means that the **timeout** is reset after each request to your application.
- **defaultUrl** is set to the Default.aspx page for the application.
- **cookieless** is set to **"UseCookies"** to specify that the application uses cookies to send the authentication ticket to the client.
- **enableCrossAppRedirects** is set to **"false"** to indicate that the application cannot redirect requests outside the application scope.

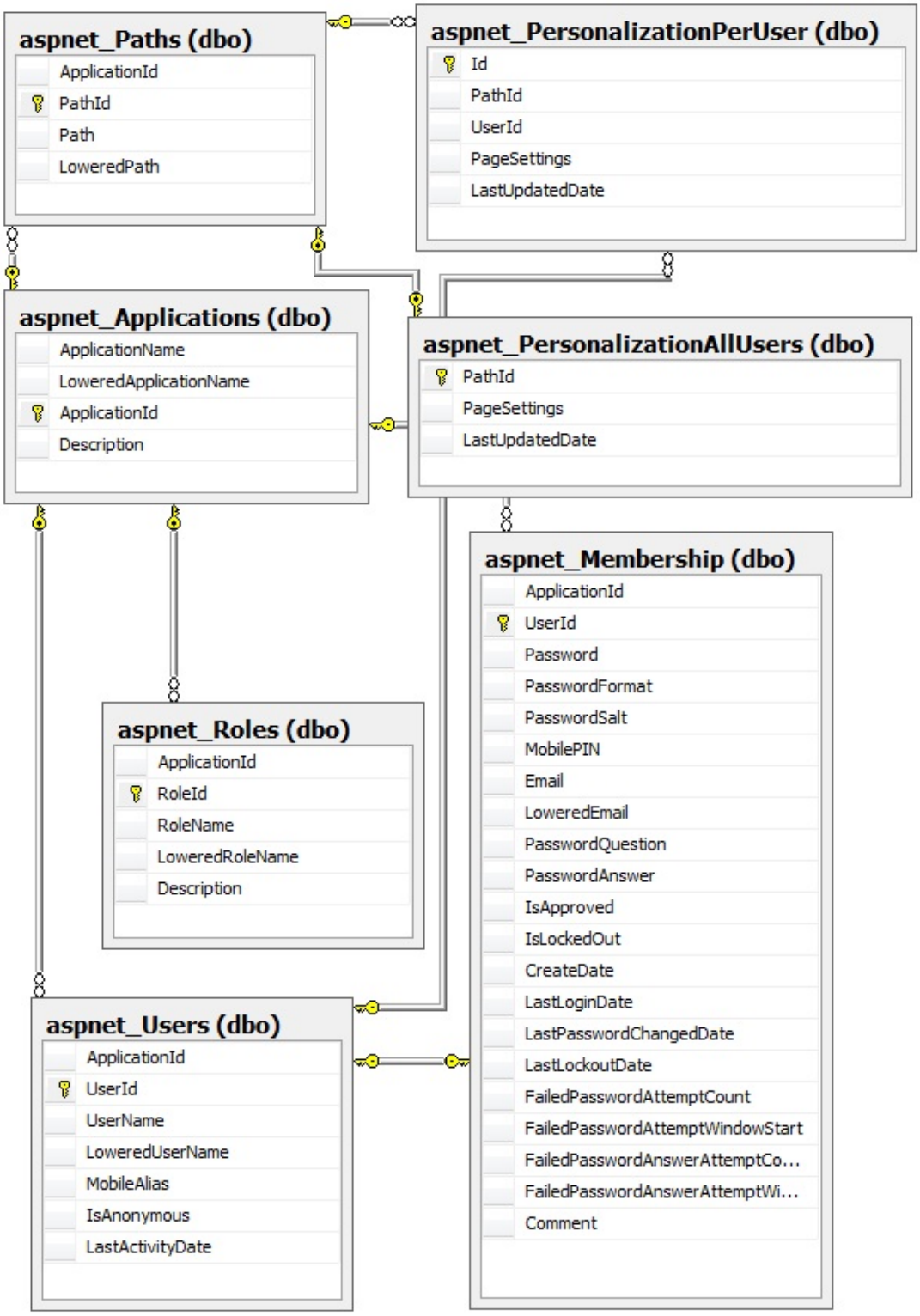

Figure 5.8 The implementation of the Membership API in the database

5.1.8 SignalR

Using SignalR we were able to implement the client listening to the server by using a long polling technique.

Initially we implemented this functionality using time-based periodic calls from the client to the server. However we observed that this approach has a negative impact of fast battery drainage and increases the volume of data passed over the Internet which is a major issue when using data services instead of Wi-Fi.

Also for Order Taxi application this is not feasible as the polling interval is unknown. For example the driver could wait for hours without receiving an order, so he would poll thousands of times to receive one favorable result. Also the polling interval would have to be below 2 seconds (to offer a response time as low as possible), which would mean a high demand on the server and low scalability.

In the .NET framework SignalR represents an implementation of long polling. However SignalR does not rely only on the long polling technique. SignalR has a concept of transports, each transport decides how data is sent/received and how it connects and disconnects.

On the server side SignalR uses two concepts: *Hubs* and *persistent connections*. Taxi Ordering App uses *Hubs*. Extending the *Hub* class represents the implementation of a hub on the server. Each method defined inside this class is mapped to a *JavaScript* implementation at compile time. The *JavaScript* document is saved on the project root folder and the client must reference it. It is possible to have broadcast events, which alert all the clients or only one in particular. It is also possible to group client and inform a whole group at a time.

When the mobile application registers using SignalR, on the server a session ID called client ID is given to it. We are storing this client ID for each order. When a response is available and needs to be sent from the server to the client, we are sending an update request to the mobile app, using the client ID stored earlier.

There are three components of the system that need to be updated from the server. This are the client app, the driver app and the operator panel. The update scenario for the driver app is similar to the one for the client app. We note that the client app and the driver app are components that need to be independently updated, so having groups in this scenario is unnecessarily. However for the operator panel we need multiple operators to be able to respond to an incoming order. In this scenario it is more feasible to group the operators in groups, a group representing one operating company. When a new order is registered into the system the first operator who sees the new order will be able to respond to it.

5.1.9 Push Notifications

Server push notification means that you can send users push notifications even when the app is not running on the device.

Although Appcelerator allows server push, many developers find it difficult to set it up. In order to send push notifications a certificate must be obtained from Apple, which is used for authentication purposes.

Once the certificate is obtained it must be registered with the push notification service, and after this the developer will receive access to the push notification services. We must mention that although it is possible to communicate directly with apple services in order to send push notifications, this task requires implementing a complex communication mechanism, so the majority of apps rely on 3rd party services which encapsulate this complexity. Sending push notifications by using 3rd party services is much easier as it comes down to accessing the service API.

The flow of events that lead to sending push notification is:

1. You connect to the APNS using your unique SSL certificate
2. Cycle through the messages you want to send (or just send 1 if you only have 1)
3. Construct the payload for each message
4. Disconnect from APNS

The flow of remote-notification data is one-way. The provider composes a notification package that includes the device token for a client application and the payload. The provider sends the notification to APNS that in turn pushes the notification to the device. This process is presented in Figure 5.9.

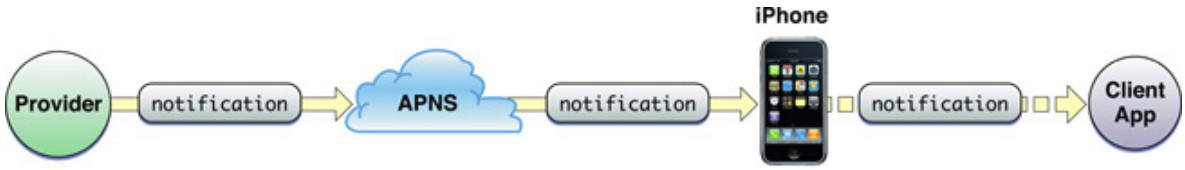

Figure 5.9 Push Notifications chain of execution

The implementation of push notifications is done by using the Urban Airship 3rd party service. Using Urban Airship we can implement a notification service for both iPhone and Android.

Before being able to send push notifications to iOS devices, we must register our *Push SSL Certificate* to communicate with Apple's push notification servers. The private key resides securely on Urban Airship servers, and Apple uses the public key to authenticate Urban Airship on our behalf.

The Push SSL certificate needs to be downloaded from the Apple server. There are two types of certificates: Development and Production certificate, corresponding to the stage of the application.

Once the certificate is downloaded it has to be uploaded on the Urban Airship server. After uploading the certificate, the library offered by Urban Airship is used to register for push notifications.

When the device is opened it registers for push notifications. If the registration completes successfully, the device receives a registration device token. Device tokens should be represented as an uppercase string, 64 characters long, without spaces or other separators and are assigned by the Apple server. The registration lets urban Airship service know that the device token is active, and should happen every time the application is opened to ensure that the list of device tokens remains up-to-date. A successful registration returns HTTP 201 Created for first registrations and 200 OK for any updates. Urban Airship queries Apple's feedback service, marking any device tokens Apple tells are inactive so we don't accidentally send anything to them in the future. The registration call tells that the device token is valid as of this time, so if a user turns push notifications back on for our application they can receive them successfully again.

The system can use this registration token in order to send push notifications to the device.

When registering for push notifications, the app can optionally, include a JSON payload to specify an alias, tag, badge value, or quiet time setting for this device token.

Sending a push notification to one or more users is done by calling the *REST* API using *POST* `/api/push/` An example of a notification is:

```
{
      "device_tokens": [
         "some device token",
         "another device token"
      ],
      "aliases": [
         "user1",
         "user2"
      ],
      "tags": [
         "tag1",
         "tag2"
      ],
      "schedule_for": [
         "2010-07-27 22:48:00",
         "2010-07-28 22:48:00"
      ],
      "aps": {
          "badge": 10,
          "alert": "Hello from Urban Airship!",
          "sound": "cat.caf"
      }
  }
```

With the APS notification payload, you can include the following parameters: badge, alert, and sound. The payload can only be 256 bytes.

Badge Numbers - The badge is the number within a red circle that appears on an application's home screen icon.

Apple requires an integer be sent as the badge value in the APS payload. However, you can have Urban Airship keep track of what badge value the user should have. This provides the advantage of being able to use advanced badge values, which Urban Airship will translate into the proper badge for you. This is called an autobadge.

There are three types of autobadge that Urban Airship supports: auto, increment, and decrement.

`auto` we will take the current stored badge value and insert that.

`+1` : we will take the badge value from the database, increment by the number after the + sign, and also increment our stored badge by that number.

`-1 :` we decrement instead of increment.

Quiet times -Time when no push notifications for the app will be delivered to that device token. If the start time is greater than the end time (e.g., 20:30 and 6:30), then the quiet time is overnight. This is the most common case, but quiet times during the day (e.g., 9:00 and 17:00) work as well.

When a push is sent during quiet time, the alert is *not* re-sent after the quiet time has elapsed. If a tz value is not included in the JSON data when a quiet time value is specified, you will get an HTTP 400 error during device registration.

Push notifications containing a badge update *will* still be sent during quiet time, but the *alert* and sound will be removed. By default, push notifications are silent, seen but not heard. To specify an alert sound to be played when the push notification is received, add a `sound`key to aps.

Custom sound - by default, push notifications are silent, seen but not heard. To specify an alert sound to be played when the push notification is received, a `sound`key needs to be added to `aps`. It is also possible to supply custom sounds that can be played when an app is downloaded. We use custom sounds in with our push notifications in order to alert both the client and the driver.

As push notifications are sent from the server in our example, we need to send the registration token to our server together with the identification of the user. If the system needs to send a push notification to one specific user, it uses the users registration token.

5.1.10 Operator panel

The operator panel is constructed as a web interface intended for responding to orders. The webpage is developed using *ASP.NET* and is secured using *.NET Membership Framework Forms Authentication*. A logged in user will be displayed a panel consisting of the current orders. Color codes are used for different order statuses. Brown is used for canceled orders, Green for orders that were responded successfully. No color is used for orders which have no been responded yet.

The webpage uses AJAX techniques for receiving new orders and sending them without requiring complete page reloads.

When a new order is inserted into the system, *SignalR* is used to notify the browser webpage to reload its orders panel. Partial page reloads are designed using the *ASP.NET UpdatePanel* and the *AJAX* functionality is built on JQuery.

Using the Operator panel website the operator is able to respond to client orders. Whenever a new order that is destined for that operator is inserted into the system, SignalR that the orders grid needs to be updated notifies the webpage. Using an update panel control, as we specified earlier, does the update.

We implemented the operator panel in such a way to allow multiple operators to work on the same set of orders without worrying of responding to the same order. Whenever an operator views the content of an order, the order is marked as pending. When this happens a message is sent to the user notifying him that the operator viewed his order. Also another operator is announced that his colleague is already processing the specific order. If he tries to process the same order he will get an alert.

5.1.11 Analytics

Analytics provide the means of measuring user experience inside the app and becomes a key component of the mobile ecosystem. Analytic provide the means of observing how users are interacting with the app and this provides the information required for further updates. Traditionally two analytics were used: *daily active users* and *user session time*. However it has been observed that user engagement falls sharply few days after the app first installation. As a consequence another key metric has become the user retention level after one day, one week and one month. Unless the app uses analytics it cannot understand its users base accordingly.

It is possible to devise ones own analytics system. However his is a time consuming and also challenging activity. For example analytics data need to be stored and only uploaded to the server when the user has an available Internet connection. However there are free services that provide analytics available. One of these services is *Flurry*, which provides a REST *API* for interaction. Activation of the API requires preregistration for their service for obtaining an *API* key. Titanium provides a free module, which can be used to encapsulate communication with the *Flurry* server.

Analytics provide information regarding usage statistics such as new users, active users (which combines new and returning users into one stat), and sessions (how many times your app is used over a time period. You can grow this stat several ways such as

acquiring new users, getting existing users to use more often or retaining users longer), session length (Session Length is a key engagement metric. Although different kinds of apps have different average session lengths, you'd like this stat to improve over time with each additional update), frequency of use (This stat reveals the intensity with which your app is used), retention level, and app versions.

It is also possible to track custom events such as the user clicking a menu button or completing a game level. We are tracking the address textbox editing feature, the map marker move, the amount the map marker is moved, the amount of time spent editing the address textbox, the amount of time until the order is placed.

5.1.12 Social media integration

Social networking services are used in mobile applications in two ways. First they provide a mean of authentication using *OAuth.* For example if the app requires the users email address it can prompt the user to authenticate in the app using his *Facebook* account. If the user agrees to allow *Facebook* to share the email address with the app, the app is authorized to retrieve the users email from the Facebook service using the *Facebook* API. *OAuth* is an open standard for authorization. It allows users to share their private resources stored on one site with another site without having to hand out their credentials, typically supplying username and password tokens instead.

The other common use of *social services* integration is for marketing purposes. Many apps allow users to post messages to *Facebook* or to post on the users behalf. The posted messages should be based on users intention to share them, but they also often serve as a marketing strategy inside the users social network. *Order Taxi* application uses *social services* for marketing purposes by sharing the users ride information with the users consent.

In order to use social services, as with most third party *APIs* an *API* key must be obtained by registering on the providers website. For *Facebook* this is available in the developers section of their website. Titanium offers modules for integrating *Facebook* and *Twitter* services and encapsulating the communication with the REST *API*.

In our Taxi Ordering Application, we integrating Facebook in order to allow people to post messages on their wall to inform their friend they are using Out Taxi ordering application. They can also invite friends to join the application and earn points. The points create a gamification strategy meant at spurring app usage by giving users badges.

Facebook currently provides to APIs, one called Graph API (the one which we are using), and the old one called *REST* API. The first step of the integration is authentication and authorization. This is done by showing official Facebook dialog for logging in the user and prompting the user to approve your requested permissions. We implemented an event listener for the login even in order to determine whether the event was successful.

In order to create requests, the method requestWithGraphPath is used. This method encapsulates the http socket connection with the Facebook server and allows us to easily communicate with the Graph API. Once we determine that the user logged in successfully, calls are made to the service with predefined parameters including: the

address of the resource (ex 'me/events'), the data serialized as JSON, according to the API specification, the verb associated with the request (ex: 'POST').

The application uses the GRAPH API for posting messages on users behalf to his Facebook account and for inviting friends to use the app. In order to be able to complete these requirements, the app needs to ask the user for these permissions. Asking for permissions is implemented in Titanium by setting Titanium.Facebook.permissions property, and this permission will be displayed when the user logs in.

5.1.13 Management and configuration

There are three types of components that need to be managed: The proprietary server, the mobile client and the 3rd party services. Managing the 3rd party services is usually done by a management panel on the providers website. A management panel for changes that have a high occurrence rate manages the proprietary server. For example new drivers are added manually to the system from this management panel. The hardest component to be managed is the mobile client application as there is no direct control over it. If major changes to the system are required for implementing new functionalities and this requires a change in the API it is important that API versioning is used to secure reliability for old clients that do not install the last updates. Backward compatibility of the system must always be respected. If the changes made to the system do not require any change in the API, but rater a change in the address, this can be resolved if the client was designed to adapt to such changes. Otherwise an update of the client app is required.

The management panel is depicted in Figure 5.10. On the left side there is a grid that displays the Taxi companies that are registered into the system. On the right there is a map on which I defined the perimeter where the taxi company is active. This operator might process any order that is received inside this perimeter. However if there are multiple operators available for the same perimeter the order is assigned to one of them, based on specific business logic inside the model.

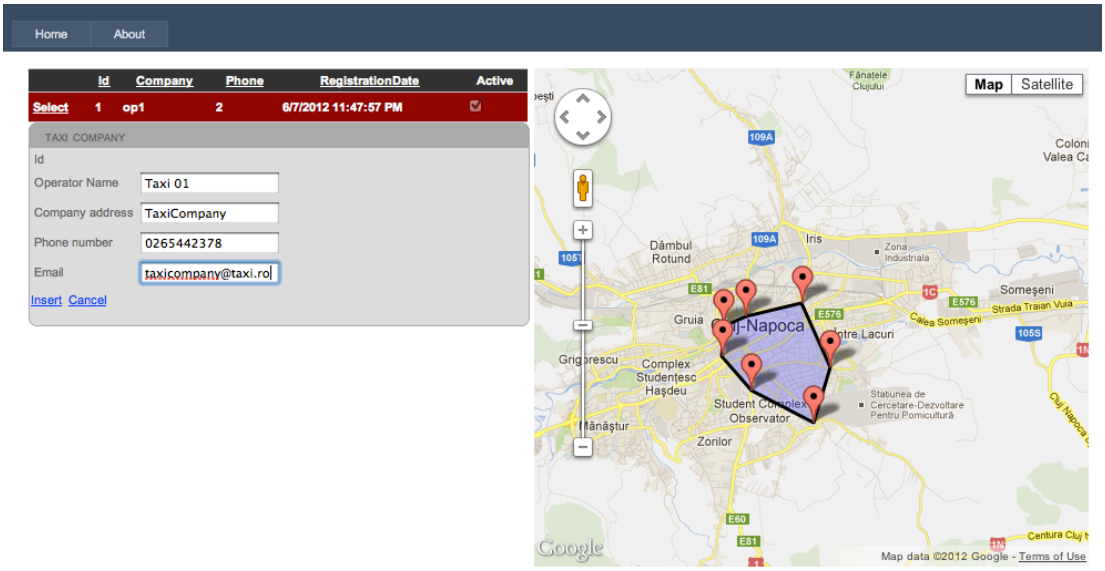

Figure 5.10 The management panel allows setting the perimeter as shown in the right

5.2 Conclusions

We have implemented the system based on the requirements presented in the previous chapter and we have identified the most important components of the system together with how they have been integrated.

We explained how a cross platform approach can be used to developing mobile application and we have proved the native capacities of the system by using the GPS API. We also presented how the mobile application can be integrated with the Cloud and how the Cloud can act as a data synchronization mechanism between the client mobile application, the driver application and the operator panel.

In the next chapter we are presenting the testing phase of the system and the approaches we have taken to validating that the components work according to the specifications and that the whole system implements the functional and non functional requirements.

6 Testing and validation

In order to assure that the system works according to the specifications and that all functional and nonfunctional requirements are met, we have implemented specific tests in order to validate the implementation.

6.1 WCF REST service validation

The WCF *REST* service is the entry gate to the Cloud service that glues together the client mobile application, the driver and the operator panel. The correct working of the WCF service is also a means of validating the model of the system server component and the response to various test cases.

The validation of the service was performed by adding a new project to the solution and creating *Unit Test* classes. Inside this class specific methods for testing functionalities have been defined.

As sending JSON requests required a series of step including creating the JSON request data from the CLR object, sending the request using a selected verb, and handling the HTTP specific errors, we defined a method called SendRequest. Using this method we specified the resource we wanted to test, the verb and the data.

The data object was defined as dynamic to allow for easier operation with encoding and decoding JSON data. We present the SendRequest method in Appendix together with the JSON data serialization approach.

We designed the system to return specific error codes for specific errors such as duplicate telephone number. When designing the Unit Test we have tested if this error codes are returned appropriately. For example when creating a new user we tested different combinations f telephone and user name.

The tested scenario inside the Unit Test for creating a new user is:

- We created a new user but with no phone number in order to test if the server returns the predefined error
- We created a new user with a phone number which was already registered
- We created a new user but without a specified name
- We created a set of 3 valid users

Besides checking if custom errors ere returned properly, we also checked if the system behaves according to expectations for typical usage flows such as:

- We created a valid set of users
- We added orders for this users
- We tested if the orders which were outside of areas where operators are active were responded using the specific error code
- We tested if the valid orders were registered inside the system

As we added new functionality to the system we expended the test scenarios and we rerun the test periodically to ensure that the system remains valid after performing modifications.

6.2 Validation of third party services

SMS Gateway

We used the SMS Gateway in order to send SMS messages to the user. This service was a middleman between the telephone operator and us. The integration of the SMS Gateway was trough a *REST* API they provided. The validation was performed by submitting a number of 10-15 messages that were delivered successfully.

Google reverse geocoding

Google reverse geocoding was used for converting GPS coordinates into street address data. We have tested the implementation of the service inside the application by performing a series of manual tests for various locations. We have also tested the accuracy for different positions and we identified that for Romania only for very few coordinates we were able to obtain the block number. We decided on displaying the user the most complete address we could obtain and allow him to complete it if necessary.

Facebook integration

The Facebook platform was integrated in order to allow users to post messages on their account by using our application. The system was tested by registering a group of accounts with the application and posting messages.

6.3 Mobile application testing

Usability testing of software applications developed for mobile devices faces a variety of challenges due to their unique features, such as limited bandwidth, unreliability of wireless networks, as well as the changing context. For assuring the system correctness Use Case Driven Testing of the functionalities of the system must be addressed in various environmental contexts such as no Internet access, no GPS signal etc.

Order Taxi app was designed with these considerations in mind and the testing approach was by using Use Case Driven testing in various conditions. The following issues have been tested:

Mobile context

It can be defined as “any information that characterizes a situation related to the interaction between users, applications, and the surrounding environment [2].” For our Order Taxi app we have tested how the app was performing in different times and places, especially how GPS was acting inside buildings and how well wireless networks aided the positioning based on the accuracy of position determination. It is very difficult to select a methodology that can include all possibilities of mobile context in a single usability test[2]. If the user position cannot be obtained and the GPS was turned off, we asked the user to turn it on or input the address manually.

Connectivity

Lack of connectivity or slow Internet speed is common with mobile applications. We tested whether the app can identify this situation and alert it to the user, notifying him that the app can not work properly and he has to either turn on internet (if this is not turn on already), or just report that the app can not work in the absence of an internet connection.

Screen size and resolution fragmentation

These concerns affect mobile applications. We tested the applications on multiple devices and also used UI elements that can adapt proportionally to screen sizes. We have observed that for some small screens the application was not displayed properly because we have implemented some minimum size attributes in order to display the text completely. In order to resolve the issue we added test cases for small screen displays, and if the screen width was bellow a specific value, we used a smaller font and minimum width, which solved the problem.

Lack of storage and computing power

These requirements must be tested especially if the work is not mitigated to the Cloud. For our app this was not a concern as we both outsourced the processing and storage to the Cloud. The order history and user data are all stored in the Cloud, and is consumed using the service API.

In conclusion we have tested the various components of the system and we are affirming that the system is valid and works according to the functional and non-functional requirements. In the next chapter we are presenting a usage manual for the system.

7 User's manual

7.1 Mobile client application

Client Application main screen

In this chapter we are presenting instructions on how the system is used and we are explaining the installation and registration procedures.

The mobile client application can be installed on iOS and Android Operating Systems. For iOS devices, the application will be available on the AppStore while for Android it will be on GooglePlay. After the user installs the app on his device the first screen he will see consists of a map, an exact address input and a order button. The screen is displayed in Figure 7.1. The left image displays the alert that informs the user that the application will need to retrieve his GPS coordinate. If the user does not allow this, he can still use the app by manually inserting the order address.

Figure 7.1 The first view of the application

The ordering functionality

In Figure 7.2 we represented the ordering functionality of the system. When the address textbox is clicked, an animation starts which displays the nearby addresses. The user can select any of these addresses by clicking them, which will automatically move the marker to that position. The option of selecting other nearby addresses is useful as GPS coordinates are not accurate enough when the user is located inside building, as the positioning is based on GSM tower triangulation as explained in Chapter 5.

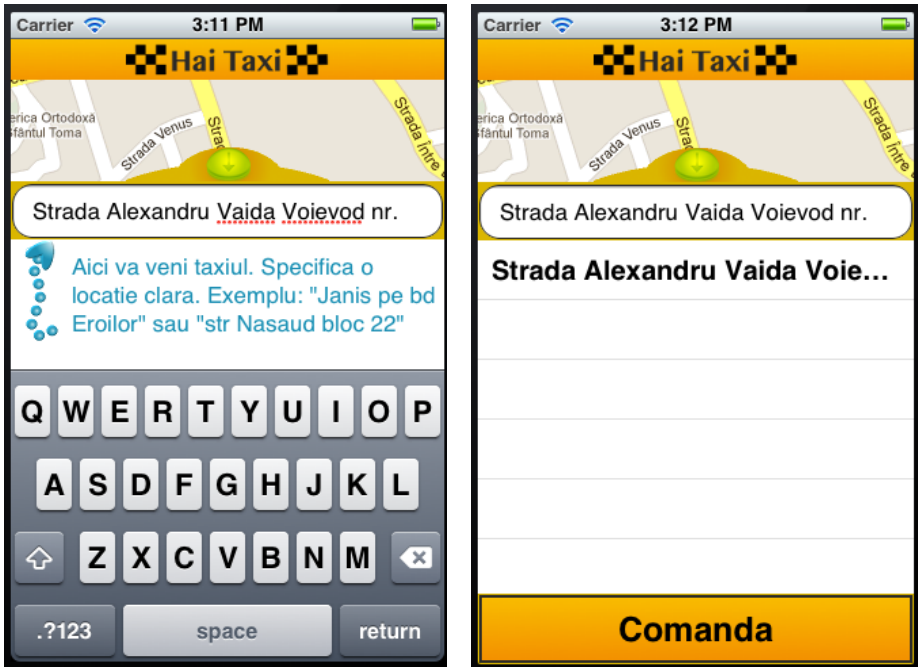

Figure 7.2 Ordering functionality of the application

Client registration

If the user is not yet registered a registration form will be displayed the first time he submits an order. The registration form is very simple, just requiring a name and a telephone number as displayed in Figure 7.3. The code received by SMS must be used to validate the user registration.

Figure 7.3 The registration form(left) and the SMS validation (right)

Pending response functionalities

After submitting the order, a new view is displayed showing a decreasing counter and a radar animation. This is presented in figure 7.4. If the user opts to cancel the order, he can click the cancel button in the top right corner.

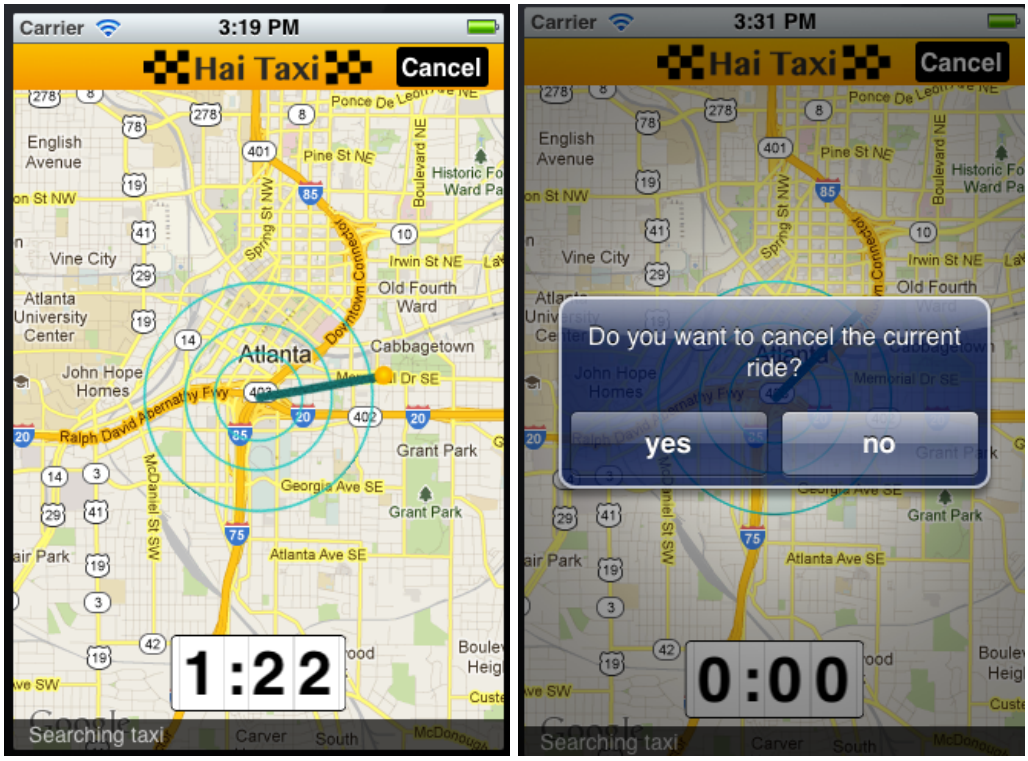

Figure 7.4 The view displayed while waiting for response (left) and the canceling confirm dialog(right)

7.2 The operator

On the operator side the first step is for the management to create the Taxi Company account. This is illustrated in Figure 7.5. When creating a new taxi company clicking on the map in the right sets a perimeter.

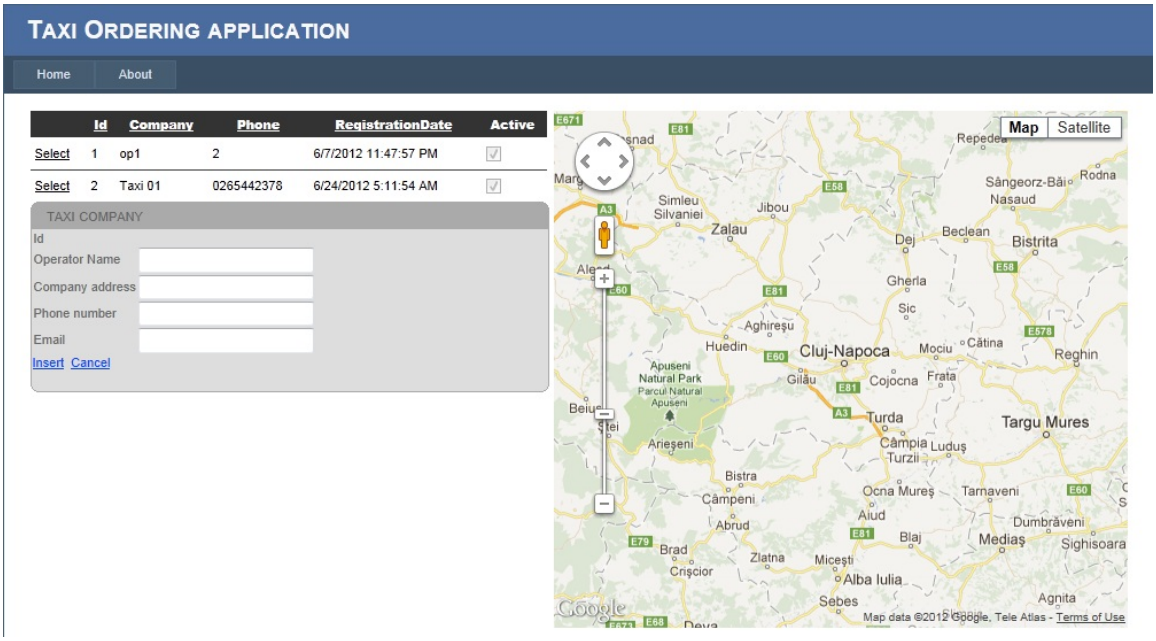

Figure 7.5 The management panel

After creating a taxi company operators can be defined which are able to respond to orders. Creating a new operator is displayed in Figure 7.7. In the left we have displayed the taxi company information and the perimeter where the taxi is active. In the right we can see how a new Operator is created. Using the username and the password inserted here, the Operator will be able to log in and respond to client orders. The orders have a color symbolism as presented in Figure 7.6. Red is the current order processed by the operator, green are the orders processed successfully and brown is used for orders which have been canceled.

	OrderId	UserName	Address	WaitingFor
Select	57	2	Strada Alexandru Vaida Voievod, Cluj-Napoca, Romania	3.00:19:08.8838000
Select	58	3	Strada Alexandru Vaida Voievod nr. 3	00:20:26.0668000
Select	59	3	Nasaud 22	00:01:09.0138000
Select	60	3	Baritiu 22	00:00:34.8038000

Figure 7.6 Color symbolism for orders displayed to the operator

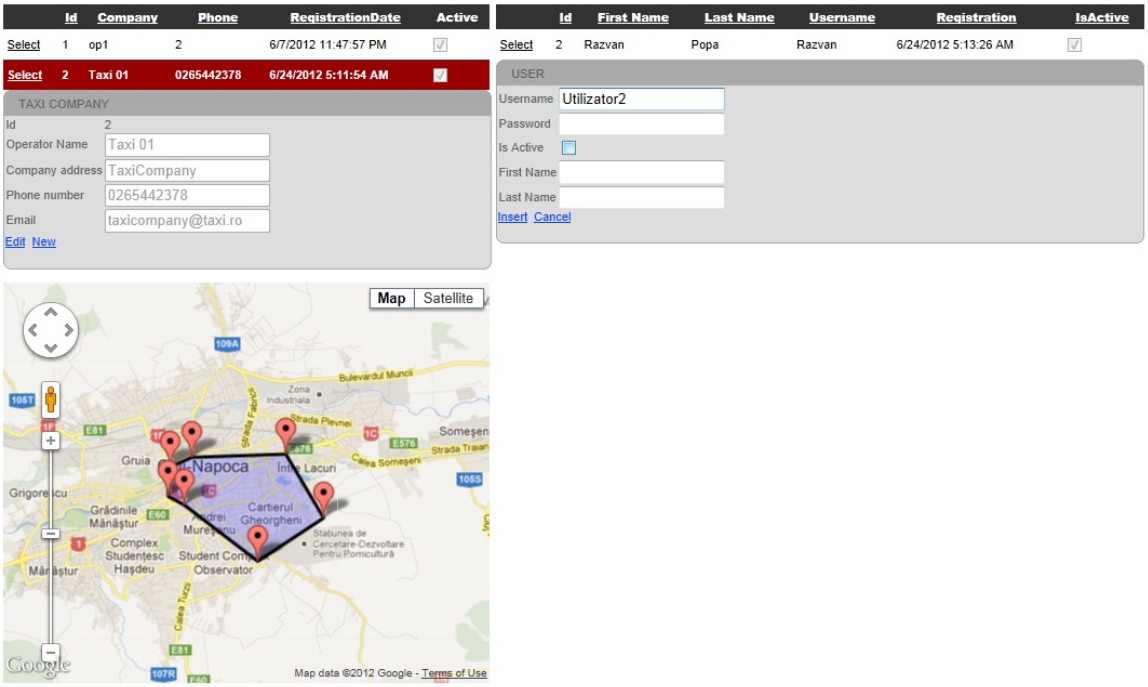

Figure 7.7 Creating an operator account

In this chapter we have presented the most important functionalities of the system and how they have been implemented. We also presented screenshots taken from the running system together with information regarding how the system should be used correctly.

8 Conclusions and further developments

We have implemented the system successfully and according to the specifications. We started by analyzing the requirements for our system by analyzing similar applications and creating a comparison matrix to help us identify the most important features. We than stated the functionalities which the system was designed to offer and we created the use cases for our actors. Before implementing the system we needed to identify the conceptual model and the suitable technologies necessarily for our requirements. We identified Titanium as the best solution for developing cross platform mobile applications.

For integrating the mobile with the service we needed to construct an API. After considering both *REST* and SOAP we decided on implementing a *REST* Api as it lighter [4]. JSON was chosen as the data encoding representation. Also we tried to externalize some of the requirements by using 3rd party services such as Analytics. The implementation of the system was performed using the technologies identified during the analysis phase, and the most important technical considerations were presented during this chapter. The implementation was performed in order to demonstrate that the analysis and design phases were correct. The validation of the system was performed in order to prove the correctness of the implementation, were we also presented how the mobile application was tested according to the general testing methodology for mobile devices.

After implementing and validating our system we affirm that the project was completed successfully and the proposed objectives have been met. However there is no perfect system so further work is needed in order to develop more advanced functionalities and to improve the overall system.

Third party services are appearing every day and their capabilities become even more impressive. In our example application we can further integrate other *3rd party services* and expand its functionality. It is important to monitor new entrants to the market in order to offer the user the most complete and updated system in terms of desired functionality.

*In terms of security c*onsidering our implementation of the security on the *API* layer, we can observe that using a hash function algorithm on a key, which should ideally be known by only the client app and the server, does the user authorization. However this key is sent trough *SMS* which cannot be considered a secure channel. As a solution the key could be sent encrypted or part of the key could be used using another mechanism and that combined on the client. Sending the key encrypted trough *SMS* diminishes the usability of the system, as a 4-digit code can get very complicated. As a solution the SMS could be read inside the app where it could be decrypted. SSL should be used for all communication between the client and the server, using a self-signed certificate.

Refferences

[1]	AEPONA Ltd., "Mobile Cloud Computing Solution Brief", November 2010, available at: http://www.aepona.com/documents_attached/Solution-Brief-Mobile-Cloud-Computing.pdf. , last retrieved 26 June 2012
[2]	Anind K. Dey and Gregory D. Abowd, "Towards a Better Understanding of Context and Context-Awareness", 2001, available at available at: http://smartech.gatech.edu/jspui/bitstream/1853/3389/1/99-22.pdf, last retrieved 26 June 2012
[3]	Appcelerator, "Titanium development platform" available at http://www.appcelerator.com/platform/titanium-sdk
[4]	C. Pautasso, Olaf Zimmermann, Frank Leymann. "RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision", 2010. available at: http://www.jopera.org/files/www2008-restws-pautasso-zimmermann-leymann.pdf, last retrieved 26 June 2012
[5]	G. Alonso, F. Casati, H. Kuno, and V. Machiraju. "Web Services: Concepts, Architectures, Applications". Springer, 2004.
[6]	H. T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. "A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches", Wiley, 2011, available at: http://www.eecis.udel.edu/~cshen/859/papers/survey_MCC.pdf, last retrieved 26 June 2012
[7]	J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn "Virtualized in-Cloud security services for mobile devices," in Proceedings of the 1st Workshop on Virtualization in Mobile Computing (MobiVirt),June 2008, p1-9, last retrieved 26 June 2012
[8]	J. Tyree and A. Akerman, "Architecture decisions: Demystifying architecture". IEEE Software, 2005, available at: http://www.cin.ufpe.br/~jbfan/arquitetura de software/aritgo decis%F5es.pdf. , last retrieved 26 June 2012
[9]	Jason Kinkaid, "Taxi Magic: Hail A Cab From Your iPhone At The Push Of A Button", 2008, available at http://techcrunch.com/2008/12/16/taxi-magic-hail-a-cab-from-your-iphone-at-the-push-of-a-button/, last retrieved 26 June 2012
[10]	L. Bass, P. Clements, and R. Kazman. "Software Architecture in Practice". Addison Wesley, 2003.
[11]	MarketWatch, "Node.js Selected by InfoWorld for 2012 Technology of the Year Award", January 11, 2012, available at http://www.marketwatch.com/story/nodejs-selected-by-infoworld-for-2012-technology-of-the-year-award-2012-01-11 Retrieved January 26, 2012.
[12]	Microsoft "ADO.NET Entity Framework At-a-Glance" available at http://msdn.microsoft.com/en-us/data/aa937709, last retrieved 26 June 2012
[13]	Microsoft, "WCF Feature Details" available at http://msdn.microsoft.com/en-us/library/ms733103.aspx last retrieved 26.06.2012
[14]	Roger L. Costello "Building Web Services the REST Way" available at http://www.xfront.com/REST-Web-Services.html, last retrieved 26 June 2012
[15]	Roy Thomas Fielding, "Architectural Styles and the Design of Network-based Software Architectures", available at http://www.ics.uci.edu/~fielding/ pubs/dissertation/top.htm, last retrieved 26 June 2012

[16]	Sarah Lacy, "Get Taxi Launching in London. This Is Way Beyond Uber", 2011, available at http://techcrunch.com/2011/07/19/get-taxi-launching-in-london-this-is-way-beyond-uber/, last retrieved 26 June 2012
[17]	Tracy V. Wilson, "How GPS Phones Work",2011, available at http://www.howstuffworks.com/gps-phone.htm, last retrieved 26 June 2012
[18]	Zarouni and K. Saleh, "Capturing non-functional user requirements", available at http://jucmnav.softwareengineering.ca/ucm/pub/UCM/VirLibIIT04/IIT04.pdf, last accessed 26.06.0212

Appendix 1 List of Tables and Figures

Appendix 2 Glossary

API – Application Programmable Interface

CDN – Content Delivery Network

GPS – Global Positioning System

IDE - Integrated Development Environment

IIS – Internet Information Services

JSON – JavaScript Simple Object Notation

MVC – Model View Controller

ORM - Object/Relational Mapping

POCO – Plain Old CLR Objects

POI - Points of Interest

REST - REpresentational State Transfer

SDK - Software Development Kit

SOAP - Simple Object Access Protocol

WCF - Windows Communication Foundation

XML - Extensible Markup Language

Appendix 3 SendRequest method

This method is used for easily testing the JSON API

```
public static dynamic SendRequest(string method, string _api, dynamic addRequest)
    {
      var WebServiceUrl = "http://localhost/TaxiDispacher/Api.svc";
      var url = WebServiceUrl + "" + _api;
      var request = (HttpWebRequest)WebRequest.Create(url);
      request.Method = method;
      request.ContentType = "application/json";//; charset=utf-8";
      var jsSerializer = new JavaScriptSerializer();
      var jsonAddRequest = jsSerializer.Serialize(addRequest);
      var writer = new StreamWriter(request.GetRequestStream());

      writer.Write(jsonAddRequest);
      writer.Close();
      var result = new ServiceResponse();
        try
        {
         var httpResponse = (HttpWebResponse)(request.GetResponse());
         using (Stream data = httpResponse.GetResponseStream())
         {
            string text = new StreamReader(data).ReadToEnd();
            dynamic resp = jsSerializer.Deserialize<dynamic>(text);
            result.response = resp;
            result.code = httpResponse.StatusCode;
         }
        }

      catch (WebException e)
      {
         using (WebResponse response = e.Response)
         {
                  HttpWebResponse httpResponse = (HttpWebResponse)response;
                  using (Stream data = response.GetResponseStream())
                  {
                           string text = new StreamReader(data).ReadToEnd();
                           dynamic resp = jsSerializer.Deserialize<dynamic>(text);
                           result.response = resp;
                           result.code = httpResponse.StatusCode;
                  }

         }
        }

      return result.response;
}
```