MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

**FACULTY OF AUTOMATION AND COMPUTER SCIENCE**
**COMPUTER SCIENCE DEPARTMENT**

# M-TICKETING BASED ON NFC TECHNOLOGY

LICENSE THESIS

Graduate: **Roxana BALAG**

Supervisor: **Sen. lect. dipl. eng. Cosmina IVAN**

**2014**

MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

**FACULTY OF AUTOMATION AND COMPUTER SCIENCE**
**COMPUTER SCIENCE DEPARTMENT**

DEAN,
**Prof. dr. eng. Liviu MICLEA**

HEAD OF DEPARTMENT,
**Prof. dr. eng. Rodica POTOLEA**

Graduate: **Roxana BALAG**

# M-TICKETING BASED ON NFC TECHNOLOGY

1. **Project proposal:** *The purpose of this project is to define and build a system capable of integrating public transportation subscriptions and tickets in the customer's smartphone. The project's main objective is to offer a way to release travel tickets and subscriptions in an electronic format, directly on the user's mobile device and to allow the user to validate those tickets in an easy and intuitive way.*

2. **Project contents:** *Introduction, Project Objectives, Bibliographic Research, Analysis and Theoretical Foundation, Detailed Design and Implementation, Testing and Validation, User's Manual, Conclusions, Bibliography, Appendix.*

3. **Place of documentation**: *Technical University of Cluj-Napoca, Computer Science Department*

4. **Consultants**:

5. **Date of issue of the proposal:** November 1, 2013

6. **Date of delivery:** July 3, 2014

Graduate: _____

Supervisor: _____

## Table of Contents

# Chapter 1. Introduction

Public transport plays an important role in the daily lives of hundreds of millions of people. The limited national and local finances starve the public transport networks of funding, although greater usage would imply important environmental and financial benefits. The recent advances in mobile technologies and services could be harnessed to increase the attractiveness and efficiency of public transport systems.

## 1.1. Project context

It is safe to say that everyone has a horror transportation story. Waiting endlessly for a late bus, train or tram, getting stuck at an hour long ticket queue, last minute changes in services and schedules and many other annoyances are part of the transport's reality for the regular passengers.

A wave of new technologies is emerging in the public transit systems worldwide, offering hope of easing, or even eliminating, many of these issues. Increasingly flexible electronic ticketing, smart cards and real-time information feeds via smartphones are making transportation networks faster, cheaper and more efficient for both the passengers and operators.

Transport operators around the world have tried using mobile technologies and services to enhance the passenger experience. In some parts of Asia, such as Japan and South Korea, mobile handsets are already widely used to purchase and validate tickets using NFC. Similar systems are being deployed in parts of Europe and the Americas, fact which can be observed in Figure 1.1, which shows the scale and growth of pilots and deployments around the world up until 2013.

The adoption of mobile technologies and services by the transport sector is being driven by two major trends. Firstly, consumers increasingly rely on smartphones to manage their daily lives. The use of SMS and QR codes is now common in the transport industry. In developed markets, in particular, consumers have become accustomed to using social network, dedicated apps and mobile web sites to quickly access information and make purchases. More and more, when we need to complete a task, we take out our smartphone. As a result, both commuters and occasional passengers now expect to be able to user their smartphones to interact with the public transportation system.

Secondly, many transport operators are under pressure from transport authorities to cut costs, while at the same time making their services easier to use and more appealing for passengers, encouraging the switch away from the private car. This pressure reflects the desire to reduce both congestion on roads and the impact of private cars on the environment. At the same time, transport operators are typically required to keep ticket prices down to a level where public transport is affordable for the vast majority of the population.


Figure 1.1 The global scale and growth of mobile commerce transport pilots and deployment [2]

There are different kinds of values that mobile technologies can create for consumers, transport operators, mobile network operators and public transport authorities. However, the value created by mobile NFC, in combination with other technologies, could go beyond localized benefits as traveling can become smarter. These potential benefits spread across four different categories:

- Time savings

For example, due to the fact that tickets can be bought on the go, without the need to wait at a queue.

- Information availability

This benefit is relevant for all implied parties, not only for the customers who are more likely to use public transport when the relevant information is at hand, but also for the transport and mobile operators, who are in a better position to offer more valuable services for customers if they can be based on the customers' habits and journeys.

- Financial savings

This represents an essential aspect for transport operators, and could arise, for example, from fraud reduction or the cheaper distribution of tickets.

- Financial gains

New and innovative services can create new streams of revenue.

## 1.2. Project motivation

The problem this project intends on solving is the current need to carry travelling subscriptions and tickets in a paper format, making the passenger susceptible to forgetting them home or maybe even losing them. The mobile phone is a device which has become essential in daily activities, such that the chance of forgetting your mobile phone or lose it is considerably smaller than in the case of a small paper ticket. This way, if the tickets and subscriptions are on his/her phone, the consumer doesn't need to be concerned about them anymore, simplifying his/her life in a small measure.

Other aspects taken into consideration are the availability of tickets and the need to wait at queues for their release. The waiting time can increase to unacceptable periods of time and the fact that the tickets are only available at the operator's selling points is definitely a problem that must be surpassed.

In this project, the NFC standard for smartphones and similar devices will be used to establish a radio communication between devices by touching them or bringing them in close proximity. The part of the system that will act as a ticket validator will be an unpowered chip, which will perform certain actions on the passenger's tickets (for example, if the passenger had three remaining tickets on his/her phone, after validation, he/she will have only two remaining tickets; if the passenger has no more tickets, he/she will be alerted of this fact and asked if he/she wants to purchase tickets etc.). Also, the passenger will be able to permanently check the application installed on his/her smartphone for the current tickets/subscription status and other relevant information (like when the last ticket validation was performed, which bus was he/she in, the time of the travel etc.)

## 1.3. Project overview

In the first chapter, we have seen the motivation behind this project and the context in which the project will be developed.

In the second chapter, we will discuss the project objectives, and the requirements and constraints the NFC e-ticketing system shall satisfy.

In the third chapter, a bibliographic study is performed on the domain, and important and relevant issues are presented, like what e-ticketing is all about, different ways of implementing an e-ticketing application, fare payment options, NFC architecture, tags, data format, security, payment methods, similar systems, etc.

In the fourth chapter we will present an overview of the used technologies. We will start by presenting Windows Phone 8 OS and the applications that can be built for it, followed by WCF services. The chosen payment method, PayPal, will be detailed, showing the benefits it offers. Furthermore, we will define the use cases that the system must accomplish.

In the fifth chapter, the detailed implementation of the system will be described.

In the sixth chapter, we will shortly present how the system was tested.

The seventh chapter presents guidelines about how to install and use the application and the last chapter will underline the conclusions reached while developing the application and give some guidelines for future development.

# Chapter 2. Project Objectives

The purpose of this project is to define and build a system capable of integrating public transportation subscriptions and tickets in the passenger's smartphone.

Because of the paper nature of tickets and subscriptions, they are at high risk of being damaged and/or lost and a client wanting to use the transit company's services needs to travel to a vendor's facility to be able to buy tickets or subscribe, being inconvenient for the user and may decrease the interest of the user in the transport company. A successful solution would be able to reduce the risk of losing/damaging the tickets or subscriptions and offer the passenger an easy, convenient way of buying tickets and subscribing, in an environment friendly manner.

The manual release of bus tickets and subscription is becoming a money consuming area, due to the fact these tickets are made of paper and the transportation companies need to pay personnel to deal with selling tickets and subscriptions. Also, they need to sustain multiple locations close to the stations where the users can buy the previously mentioned tickets/subscriptions. The main disadvantage on the passenger's side is the fact that buying tickets cannot be done 24 hours per day, 7 days per week due to working hours and selling points locations, while the tickets are perishable and easy to lose.

For the public transportation companies, which need to provide accessibility and ease of user for their clients while minimizing their own expenses, this project is a solution that supports online payment and electronic format of tickets and subscriptions, which increase the user's interest towards public transportation due to its ease of use and convenience. Unlike manual release of paper tickets and subscription, this project has the advantage of releasing tickets and subscriptions in an electronic format, easy to acquire, without the need to go to a facility.

## 2.1. Project stakeholders

Before stating the objectives of this project, we must first identify the stakeholders of the system, such that their interests and responsibilities are covered and protected by the project's objectives.

The identified stakeholders are:

- Public transport authorities

This stakeholder is represented by authorities on local, regional and nationwide level, which are interested in an environment friendly transportation system. It is the project sponsor, responsible for finding ways to ensure a "green" transportation system.

- Public transport operators

The public transport organizations and their staff are interested in having a tool which simplifies their work, while minimizing costs. This is the main stakeholder, responsible for attracting customers while reducing operational costs and maximizing profit.

- Payment industry

Banks, credit cards companies and mobile phone operators are interested in integrating their services in transportation systems. They are responsible for delivering solutions for payment to their clients.

- Suppliers

Necessary suppliers of hardware, software and all kinds of consultants (market, financial) are interested in selling their services. This particular stakeholder is responsible for the development and maintenance of the system (mainly hardware – NFC tags and devices).

- Passengers

Transit services consumer, interested in buying tickets and subscriptions with a minimal effort. This stakeholder is responsible for using the system.

## 2.2. Project objectives

Having identified the main stakeholders and their interests, the project objectives can be stated more clearly, keeping in mind what each stakeholder is looking for in the system and what their responsibilities are.

The project's main objective is to offer a way to release travel tickets and subscriptions in an electronic format, directly on the customer's smartphone device and to validate those tickets in an easy and intuitive way. This ensures a "green" alternative to the paper tickets, a way of minimizing operational costs of public transport operators and a minimal effort from the passenger's side.

Among the secondary objectives, we can mention the following:

- Buying tickets and subscriptions anywhere and anytime, without the need to travel to a dedicated facility to make the purchase

This objective ensures that the payment industry's needs will be met (an alternate method of payment must be used such that the passenger is able to purchase tickets directly from his/her mobile device – online payment)

- Reducing susceptibility of losing and damaging the aforementioned paper tickets and subscriptions

The passengers keep their interest in the system and the risk of prejudice is minimized, while ensuring ease of usage.

- Keeping a history of the passenger's former travels

The public transport operators can customize and shape their services according to the passengers' needs and travel, providing an opportunity of financial gains and profit maximization.

- Keeping transit services related information at the reach of the passenger whenever he/she needs it

This objective could help maximize the transport operators' profits, by offering up-to-date information about the services they offer, thus increasing the attractiveness and efficiency of the system.

- Get people acquainted with Near Field Communication and increase their trust in this technology

The suppliers get to sell their services, while the public transportation operators increase their attractiveness towards the clients. The passengers are presented with an intuitive way of validating tickets by using their smartphones, in an already known manner.

## 2.3. System requirements

In defining requirements, there are four types of requirements that need to be defined: business requirements, stakeholder requirements, and solution (system) requirements.

Business requirements describe why some organization is taking the project, stating some of the benefits that the organization or its clients expect to receive as a result. They define the scope of the solution.

The stakeholder requirements (referred to as user needs or user requirements) describe the activities the user performs using the system, i.e. the activities the user must be able to perform.

Solution (system) requirements are the foundation used by developers to build the solution. These requirements state what the system "shall do". System requirements are classified as either functional or nonfunctional requirements. A functional requirement specifies something the developer needs to build in order to deliver the solution. Nonfunctional requirements specify all the remaining system requirements that are not specified by the functional requirements. They are referred to as quality of service requirements, as they define the overall qualities or attributes of the resulting system, by placing restrictions on the product being developed, the development process, and specify external constraints that the product must meet.

A successful solution must deliver features (functional requirements, the things the system should do), while exhibiting certain desirable properties (nonfunctional requirements, less visible characteristics) and meeting constraints (boundaries that the solution cannot exceed). In what follows, the system requirements of the system will be presented, both functional and nonfunctional.

### *2.3.1. Functional requirements*

Functional requirements specify what the system or a component must be able to perform. They specify specific behaviors or functions.

The functional requirements of the system are:

- Registration

The system shall be able to provide a way for a new user to use the system's services, by creating a personal, individual account from his smartphone device.

- Authentication

The system shall be able to provide the user a way to authenticate into the application, such that only the right user has access to the system and user's own private data.

- Ticket validation

The system shall provide a way for the user to validate a ticket located on his device.

- Validations tracking

The system shall provide a way for the user to see past ticket validations with their respective detailed information, so that the user can provide proof of his/her validation when requested to do so.

- Subscriptions tracking

The system shall provide the user a list of all his/her subscriptions, with detailed information about the validity of the subscription (the time interval in which the subscription can be used) and the bus lines supported by the subscription (on which buses can the user travel using a particular subscription).

- Tickets status tracking

The system shall provide the user information regarding the status of his/her tickets (number of tickets bought over time, number of remaining/usable tickets) and a brief overview of the last ticket validation.

- Ticket purchasing

The system shall provide a way for the user to buy more tickets, regardless of time and location, without the overhead of traveling to a kiosk.

- Bus lines consulting

The system shall provide the user a list of all the company's bus lines with their afferent information, such that the user can make an informed decision about which bus lines to travel on and which lines to subscribe to, such that his/her needs are met.

- Subscription purchasing

The system shall provide the user a way to subscribe to the organization's services, for a month's period, on the bus lines of his/her choice, starting from a chosen date.

- Logging out

The system shall provide the user a way to log out, such that at the next usage, his/her credentials will be required.

- Tag writing

The tag writing system shall provide the user a way to configure a tag (acting as ticket validator) with the information of the bus on which the tag will be located.

### *2.3.2. Nonfunctional requirements*

While features (functional requirements) are the simplest to identify, they are not what determines how successful a solution is. There are a multitude of possibilities that provide the necessary features but what makes one solution a better fit than another are the properties (nonfunctional requirements). Nonfunctional requirements describe how the system should behave.

The identified nonfunctional requirements of the system are:

- Availability

The system shall be able to deliver services whenever requested, being available 24 hours a day, 7 days a week, as long as the user has an active internet connection.

- Accessibility

The system shall be accessible from any location as long as an active internet connection exists.

- Reliability

The system shall be able to deliver services as specified, each time manifesting the same features and behaviors, i.e. to behave consistently in a user-acceptable manner within the environment for which it was intended.

- Security

The system shall be able to protect itself against accidental or deliberate intrusion of unauthorized parties, granting access and permission to use its services only to authenticated users.

- Usability

The system shall be easy to use and have the capability to enable the user to learn, to operate and control the application. It shall provide an intuitive and clear way of usage, determined by a user-friendly interface.

- Maintainability

The system shall provide a way for the application to be improved or adapted over time to changes in environment and specifications.

- Extendibility

The system shall provide an easy way to extend the system functionality without having to make major changes to the infrastructure of the system. The system shall conform to the open-closed principle.

- Performance

The system shall have a response time of under 5 seconds for all the operations it must perform.

# Chapter 3. Bibliographic Research

As stated by the authors in [10], paper-based tickets are still a reality in public transportation, although for many years the public transport market entities tried to replace paper tickets by electronic media or e-tickets. Despite its many benefits for the transport operators, e-ticketing has not yet been able to live up to the users expectations.

According to [7], public transport operators in many countries have implemented or are about to introduce e-ticketing systems, and according to [13] hundreds of millions of people are familiar with payment and ticketing cards that use contactless techniques because of the expansion of public transport payment systems such as those in Hong Kong, Tokyo and London since the late 1990s.

It is stated in [14] that Near Field Communication is a relatively new technology that, among other things, allows mobile phones to emulate smart cards such as the travel cards used in public transportation. Despite its promising services and optimistic predictions, NFC technology and mobile ticketing services based on it, have yet to take off. However, NFC holds much potential for mobile services and one of the most promising early applications of this technology is considered to be mobile ticketing.

Studies presented in [15] suggest that there is general consumer interest in the services mobile commerce provides, for example purchases on web sites, routine bank services, and electronic receipts and tickets. In the context of mobile payments, both technology trust and trust in transaction partners (mobile network operators, payment services providers) are likely to play a role in trust formation.

Each of these facts is going to be presented in more detail, along with the benefits of building a NFC-based mobile ticketing system.

## 3.1. E-ticketing

In [4], a ticket is defined as a contract between a user and a service provider. If the user can prove his/her ownership of the ticket, then he/she is granted the right to use the service under some terms and conditions. Usually, the ticket needs to be validated in order for the user to be able to use the service. Basically, ticketing represents a company's pricing policy, with the consideration of operational, commercial and social objectives. The ticketing system is the translation of fares into concrete means of payment (for the passenger) and fare collection (for the operator) [5]. The main steps in using a service with such a pricing policy are: ticket payment, issue and validation.

In the public transportation market, several types of tickets can be identified: single journey tickets, single-operator tickets, multi-journey tickets, weekly or monthly passes, group tickets, special event tickets [7]. Tickets can be delivered as paper tickets or electronically (e-tickets), but paper-based tickets usually imply relatively high maintenance costs.

To facilitate the use of public transport, cities aim at making the ticketing system as easy and as attractive as possible. Electronic ticketing is a form of electronic commerce for different kinds of tickets, having as main characteristic the fact that the tickets are sold and stored in an electronic device, such as smart cards or mobile phones [7].

Like paper tickets, electronic tickets should include some basic information for their practical use. Some of the information electronic tickets could contain are presented below [4]:

- Unique identifier for every ticket
- Issuer – the entity responsible for issuing the e-ticket (can be different from the service provider)
- Service provider – the entity who delivers services to the user
- User – information about the e-ticket owner
- Service – description of the service contract
- Terms and conditions of use
- Type of ticket
- Transferability
- Number of uses
- Destination
- Attributes – other information about the e-ticket
- Date of issue
- Issuer's digital signature – if issuer has a public key cryptosystem, being able to digitally sign the e-ticket
- Device identification

In [3], five different business models for public transport e-ticketing systems have been identified:

- Prepaid value model

It is currently the most common form of automated ticketing. The ticket is issued by the transport operator and based on the value stored on a card (value which can represent money, number of rides, time-interval).

- Enhanced payment card

It is represented by a contactless credit or debit card or a payment application on an NFC phone which is used to pay the fares. The user presents the card to a reader and the payment transaction is processed based on the public transport operator's fees.

- Postpaid model

It is based on smart cards or NFC-enabled phones. The fares are billed afterwards, according to the usage which was recorded (user's location and identification are required).

- Combined/enhanced collaborative models

In this case, a smart card or phone incorporates multiple applications (transit and payment).

- Embedded secure element/ (U)SIM

In this model, an intermediary, usually a trusted service manager, or a mobile network operator, or a handset manufacturer determines the business rules. This can be the case for all business models mentioned above.

### *3.1.1. Interoperability and multi-service*

According to [7], there are 26 European countries that operate smart card ticketing systems, some of them nationwide, but most countries have at least an e-ticketing system in their capital. A next step forward is to combine existing schemas to create national

solutions to finally come to cross-border interoperability. In Asia, for example, where the largest e-ticketing schemes exist, there is no true interoperability between neighboring territories or networks. In order to take advantage of the many technologies that exist for e-tourism, interoperability of the products is necessary. Until today there is no real interoperability for touristic e-services, resulting in high deployment costs and a lack in flexibility [7].

A customer should be able to download an application onto their preferred (and compatible) media (smart card or mobile phone), which can be recognized in all participating countries. When traveling abroad, users would be able to use that media and buy a transit pass for the duration of the stay. Interoperability in e-ticketing for public transport implies removing the obstacles for the customer to switching transport modes. All ticketing needs should be in one place and on their local transport smart ticketing media. It has already been shown that it is technically possible to adapt applications and download them onto multi-application cards/mediums.

Appart from its core application in the transport sector, e-ticketing could be extended to other areas, such as tourism and leisure activities. That way, users of e-ticket could make use of a service package that promises greater flexibility and convenience.

In Asia, for the largest systems, the integration of payment functionalities appears to be a key success factor.

According to the Urban ITS Expert Group [7], one of the major advantages of smart ticketing should certainly be to propose complementary services, other than transport payment services. They emphasize that smart ticketing does not necessarily mean to have one ticket for one journey, but one wallet with several tickets (which can easily be bought) and in the future possibly one wallet for several services. Integrating other services, such as the possibility to enter a museum with that ticket, represents the most advanced level of integration.

### *3.1.2. Fare payment*

Transit agencies have traditionally used cash fare systems, but cash is expensive to transport, count, and guard. For these reasons, many agencies have introduced automated fare media by expanding fare payment to electronic, magnetic stripe contact cards, and smart cards.

#### *3.1.2.1. Smart cards*

Smart cards have been around since the idea of combining plastic cards with microchips was patented in 1986. Later in 1992, financial institutions in France substituted their magnetic stripe cards with smart cards to reduce fraud. Many distinct sectors nowadays use smart cards: health care, banking, government, human resources and transportation. The purpose of these smart cards is to store information like banking data, transportation fares, or other data. Smart cards are until today the most common form of e-ticketing [7].

Smart cards are the size of a credit card; they are technologically simple and relatively cheap to produce. The microchip it contains stores, processes, and writes data in a secure environment. Some other details and additional services can be stores on the smart card. A smart card with a memory chip, but without a microprocessor, can store value on the card, but is not reprogrammable [7]. There are more types of smart cards, the

most important classification being being into reloadable and non-reloadable smart cards. The non-reloadable smart cards are usually purpose specific and are not being processed through a bank network, being in essence a medium for converting cash into an electronic transaction. The reloadable smart cards are usually issued by financial institutions, but are easier to obtain, only a simple identification is required for registration. This kind of cards can usually be bought from other place, like kiosks, retail locations, not necessarily a bank.

The smart card can communicate with other devices in two ways: contact-based or contactless.

- Contact-based smart cards have a chip embedded into the plastic card. Only the surface of the chip is not covered as it needs to be brought into contact with a reader (e.g. bankcards, telephone cards). Figure 3.1 displays a possible design and structure of a smart card.
- Contactless smart cards contain a chip completely embedded into plastic. The card needs to be placed in close proximity (about 10 cm) to the reading device to start the communication process with the reader, done by high-frequency waves similar to RFID. Figure 3.2 displays the basic structure of a contactless smart card, represented by a chip whose surface is completely covered by plastic.


Figure 3.1 Contact-based smart card [12]

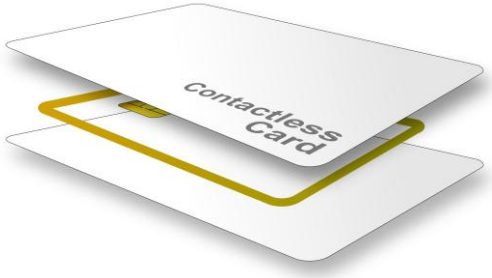



Figure 3.2 Contactless smart card [12]

### *3.1.2.2. Mobile ticketing*

In mobile ticketing applications, the mobile phone is used as an electronic version of a ticket. Because the use of mobile phones and the Internet has changed the way users buy products and services, it is expected mobile commerce to become increasingly popular, which is an important and relevant fact for the public transportation industry [7].

Mobile ticketing is basically a virtual ticket, in a textual or graphical representation that is held on mobile phones, tablets or personal digital assistants (PDAs) and can be ordered and obtained from any location.

Basically, there are three different possibilities for mobile ticketing:

- SMS based transactional payment

Users send their request using short messages (SMS) and pay their fare with the next phone bill or, if it is a prepaid SIM, the fare will be deducted from the purchased credit value. SMS allows ticketing operators to gain more widespread access to mobile users than bar codes; not only does every phone have SMS, but also mobile phone owners are very familiar with usage.

- OCR (Optical Character Recognition)

Users receive and image that functions as a code (e.g. 2D barcode) that contains all required information.

- Contactless Near Field Communication

The process is similar to OCR, but the information is instead stored in the NFC memory of the phone. That way, many different tickets can be stored on a single phone.

In the following section, identification technologies used in mobile ticketing will be discussed.

## 3.2. Identification technologies

By identification technologies we refer to the methods of automatically identifying objects, collecting data about them, and entering that data directly into computer systems (i.e. without human involvement). Technologies typically considered include bar codes, Radio Frequency Identification (RFID), biometrics, magnetic stripes, Optical Character Recognition (OCR), smart cards, and voice recognition. Only the technologies of interest will be detailed.

### *3.2.1. Optical Character Recognition (OCR)*

Optical Character Recognition is the electronic conversion of scanned images, such as 1-dimensional barcodes or QR codes (but also scanned documents, PDF files or images) into machine readable characters. Original sources, e.g. receipts, tickets or other forms of printed records can be captured by a digital camera and converted into editable and searchable data.

A barcode is the visual representation of information that can be read and understood by computers. 1-dimensional barcodes consist of vertical bars and spaces that contain information only in the horizontal direction; data that can be stored in one barcode is thus limited. They can be used to track objects and persons; this way barcode tickets allow their holders to enter sport arenas, theatres and cinemas or public transport. The barcode is read by a scanner, that can be based on laser or optical technology. Optical scanners capture the barcode and then process the data from the captured images. This technology often surpasses laser scanners on performance and reliability and is an important technology for mobile barcodes. Figure 3.3 displays an example of a 1-dimensional barcode.

Figure 3.3 1-dimensional barcode [7]

However, demand grew and more data storage capacities were required. Therefore, 2-dimensional barcodes have been developed.

QR Codes (Quick Response Codes) are one form of a 2-dimensional (2-D) matrix code that is composed of small, symmetrical elements arranged in a square or rectangle

containing information in the horizontal and vertical direction. Figure 3.4 displays a QR code.

Figure 3.4 QR code [7]

QR codes are usually attached to an item and entails information related to that respective item. They are accepted in diverse industries, such as manufacturing anf warehousing, logistics, healthcare, tourism or transportation. Especially with the increasing use of smartphone, QR codes are used for mobile marketing and location based services; they are increasingly printed on signs, billboards, posters, business cards, clothing or other items. By using the camera of a mobile phone, the QR code can be scanned and the phone then automatically accesses the Internet by reading the URL encoded in the QR code. Users are then connected to a relevant web page and receive targeted marketing. It can be used for location-based services (e.g. on timetables in subways to find out arrival times of the next transport means) or for e-payment using a mobile phone and QR code printed on tickets.

### *3.2.2. Radio Frequency Identification (RFID)*

The function of Radio Frequency Identification is basically the same as for barcodes, but with the important difference that RFID tags can, other than barcodes, process data or communicate with other RFID tags and are thereby compatible with existing contactless smart card infrastructure. The systems consist of a reader that can wirelessly read and write data in real-time to a RFID tag. These tags include an integrated circuit that usually stores a static number (ID), and an antenna to transmit the data to the reader using radio waves. Initially, RFID applications were used to precess and track the flow of goods, e.g. in the retail sector, supply chain management and warehouse management, logistics and manufacturing. But also the tourism sector can profit from RFID applications, e.g. several museums have already implmented RFID to inform their users about the exhibited pieces, hotels offer keyless entry or casin chips are tagged with RFID [7]. There is only very little security during the communication with the reader. Typically, RFID tags can be read from distances of several centimeters to several meters. For the purpose of transport ticketing, a reader will be informed about the passengers' departure and destination. The tags are usually attached to smart cards carried by the passengers. This allows passengers to be charged automatically, according to the zones or the time they have travelled (depending on the tariff structure).

However, the very high one-time investments and ongoing operating costs for the technology infrastructure (check-in/check-out devices at stations and in vehicles, smart

cards, back-end systems and communication infrastructure) have so far hindered a wide scale implementation of e-ticketing solutions.

Near Field Communication (NFC) is basically an advancement of RFID technology, which will be discussed in the following section.

## 3.3. Near Field Communication

NFC is a short-range wireless communication technology that is based on approved and mature standards in the field of RFID and smart cards. RFID, which has already been introduced in the 1970s, realizes automatic identification and data transfer via electromagnetic radio signals, typically by means of an active reader that is connected to a source of energy and a passive electronic tag that is a transponder receiving its power from the reader by magnetic induction. The RFID tag normally contains an antenna for receiving and transmitting the radio signal and an integrated circuit for processing and storing information and for modulating and demodulating the signal [8]. A usual tag is shown in Figure 3.5. A small microchip is surrounded by a copper antenna. The shown tag has a side length of around five centimeters.

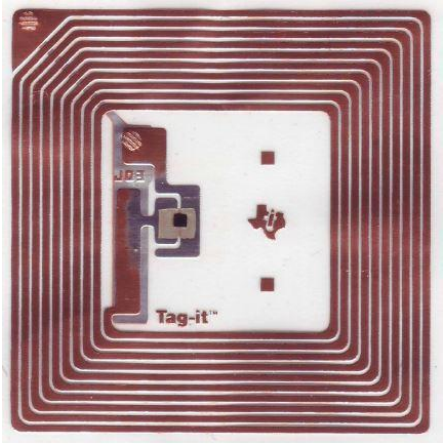



Figure 3.5 A passive RFID tag that is compatible with NFC [6]

The RFID tag can be placed almost everywhere and is normally hidden behind existing material, like the packaging of a product, thus being invisible to the user.

In 2004, NXP Semiconductors, Sony and Nokia founded the NFC Forum in order to bring existing standards and efforts of the RFID and smart card technology together and to create a novel and innovative capability for short-range communication. Up to now, the NFC Forum counts more than 100 members and supporting companies aiming to find a worldwide standardization for the NFC technology [8]. For a long time, only a small handful of NFC mobilephones were available, mainly manufactured by Nokia, until Samsung and Google attracted a large audience when releasing the NFC supporting Nexus S phone in 2010. With the current rush on smartphones mentioned above and further successful NFC field tests in the past years, it is expected that in near future most of the top class smartphones will be equipped with NFC support [6].

### *3.3.1. NFC overview*

Near Field Communication (NFC) is a wireless proximity communication technology that enables data transfers between two devices which are close to each other,

typically to a distance of less than 10 centimeters [8]. This simple means of establishing a connection is a major advantage of NFC over other wireless communication technologies, such as Bluetooth and Wi-Fi. However, compared to connection speeds of Bluetooth and Wi-Fi, NFC provides slower data transmission rate of up to 424 kilobits per seconds (kbps) [8].

In addition, the communication range of NFC is shorter than that of other communication technologies. However, this is not considered a drawback, but an inherent characteristic and a technical advantage of this technology. To be more specific, the close communication range enables intuitive transfers of data by tapping one device against a desired peer device and ignoring other devices that are outside this communication range. This not only helps to prevent signal interference between devices, but also secures users and applications, since users must be close enough to an NFC device to be able to nearly „touch" it, or in other words, in most cases, they intentionally wish to use the application.

NFC operates at 13.56 MHz and relies on ISO14443 and ISO 18092 for low level data exchange between two NFC devices [8]. Specifically, these two ISO standards specify the operating frequency, modulation, coding schemes, anti-collision routines, and communication protocols. NFC data exchange formats and NFC tag formats are defined by the NFC Forum [8].

### *3.3.2. Hardware architecture*

Hardware architecture comprises a system's physical components and their interrelationships. The main components of the NFC hardware architecture are [8]:

- The Host-Controller

Application Execution Environment (AEE), the environment where the application rests, such as the mobile phone.

- The Secure Element

Trusted Execution Environment (TEE), the secure environment where sensitive information such as debit card data is stored within the host controller.

- The NFC-Controller

Contactless Front-end (CLF), the link between the host and NFC, with an interface to the Secure Element.

- NFC-Antenna

Simply, put this is simply loops of wire, occupying as much surface area as the device allows. Here we will discuss in detail the two central components from the list above: the secure element and the NFC controller. Figure 3.6 shows the NFC elements

within a mobile device.

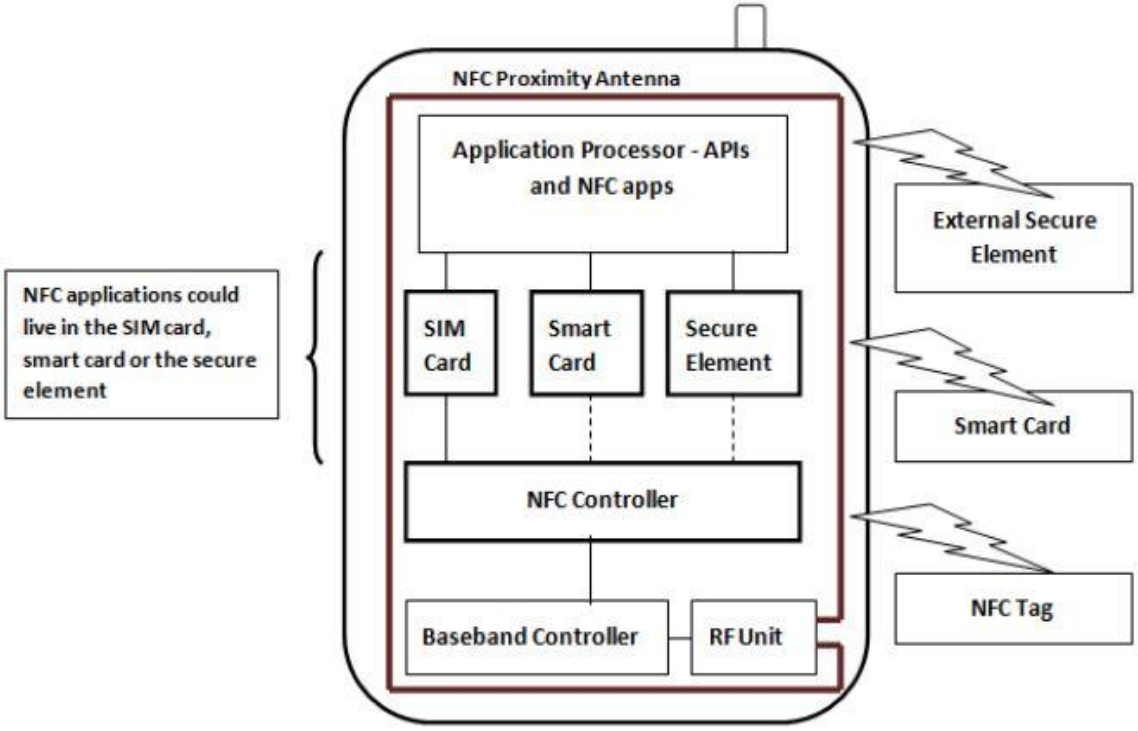



Figure 3.6 NFC related elements in mobile devices [18]

### *3.3.2.1. NFC Controller*

The NFC-Controller is the link between the air interface (radio-based communication link between the mobile station and the active base station), the host-controller and the secure element. The Host-Controller is most likely a mobile device like a mobile phone, or a smart car key. There are various interfaces between the host controller and the NFC controller such as the serial peripheral interface (SPI), and universal serial bus (USB). For the communication with the secure element there are typically smartcard interfaces, the NFC wired interface or the single wire protocol in use. The controller works as a modulator/demodulator between the analogue air interface and other digital interfaces. The NFC-controllers have integrated microcontrollers, which implement the low level services, so the exchange with the host controller is limited to the application data and some control commands [18].

### *3.3.2.2. Secure Element*

On most mobile devices, such as mobile phones, there is no way to store secure data directly. For most NFC applications, i.e. payment and authentication solutions, secure storage systems are essential. For sensitive data, the storage needs to be resistant to manipulation and it must be able to execute cryptographic functions and to execute security-relevant software. Smartcards usually implement these requirements [8]. To implement such secure elements, there are different possibilities, each with its own advantages and disadvantages [18]:

- Software without secure hardware

Software is the most flexible and independent solution, but software could not be optimally secured without the hardware as there is always the possibility that the unsecured hardware is manipulated.

- Device integrated hardware

This is the most host dependent, but most reliable solution. The secure element is either a part of the host or is built in as its own chip. The communication with the element and the NFC-Controller works like a smartcard or over the NFC Wired Interface. The biggest disadvantage of this solution is, if the user changes the device, the provider of the secure service has to remove the data from the old device and to put it on the new one.

- Changeable hardware

In most cases, this would be the best compromise between reliability, usability and costs. Because a hardware interface is needed to plug in the removable secure element, the production costs of the host device are higher. Such removable devices could be a Secure Memory Card (SMC), which combines the secure smartcard functions with a usual memory card function, or a Universal Integrated Circuit Card (UICC); for example in a mobile phone this is the Subscriber Identity Module (SIM) card. While the SMC is usually owned by the user, which allows him/her to change his/her data independently, the SIM card of a mobile phone is owned by the network provider and, thus, the network provider must cooperate with the secure service provider.

An NFC system implementing a Secure Element is often abbreviated as Secure NFC, this is misleading because only the data stored on the secure element is secured, not the whole NFC communication.

### *3.3.3. NFC tags*

In June 2006, the NFC Forum introduced standardized technology architecture, initial specifications and tag formats for NFC-compliant devices [8]. These include Data Exchange Format (NDEF), and three initial Record Type Definition (RTD) specifications for smart poster, text and Internet resource reading applications. In addition, the NFC Forum announced the initial set of four tag formats that all NFC Forum-compliant devices must support. These are based on ISO 14443 Types A and B (the international standards for contactless smartcards) and FeliCa (conformant with the ISO 18092, passive communication mode, standard). Already more than one billion tags of this kind have been deployed globally, although for non-NFC applications like mass transit and access control.

The NFC Forum chose the initial tag formats to cater for the broadest possible range of applications and device capabilities [3]:

- Tag 1 Type

Based on the ISO14443A standard. Theyare read and re-write capable. Memory availability is 96 bytes and is expandable up to 2 kbyte. The communication speed of this NFC tag is 106 kbit/s.

- Tag 2 Type

Based on ISO14443A. They are read and re-write capable. The basic memory size is 48 bytes and can be expanded to 2 kbyte. The communication speed is 106 kbit/s.

- Tag 3 Type

Based on the Sony FeliCa system. Memory availability is variable, theoretical memory limit is 1MByte per service.

- Tag 4 Type

Defined to be compatible with ISO14443A and B standards. These tags are pre-configured at manufacture and they can be read, re-writable, or read-only. They have a memory capacity up to 32 kbytes. The communication speed is in the range of 106 kbit/s and 424 kbit/s.

It is worth noting that Type 1 and 2 tags and Type 3 and 4 tags are two very different groups, with very different memory capacities. There is very little overlap in the types of applications they are likely to be used for.

### *3.3.4. Operation modes*

There are three operating modes: reader/writer, peer-to-peer, and card emulation [11]. The reader/writer mode enables one NFC mobile to exchange data with one NFC tag. The peer-to-peer mode enables two NFC enabled mobiles to exchange data with each other. In card emulation mode, a mobile phone can be used as a smart card to interact with an NFC reader. Each operating mode has a different technical infrastructure as well as advantages for the users.

#### *3.3.4.1. Reader/writer mode*

The NFC device behaves as a reader for NFC tags, such as the contactless smart cards and RFID tags. It detects a tag immediately in close proximity by using the collision avoidance mechanism. An application on an NFC device can read data from and write data to the detected tag using the read/write mode operations. Figure 3.7 displays the basic principle of the reader/writer mode. The initiator generates a 13.56 MHz magnetic field (1) and the tag is powered by the magnetic field and sends a response to the request (2).

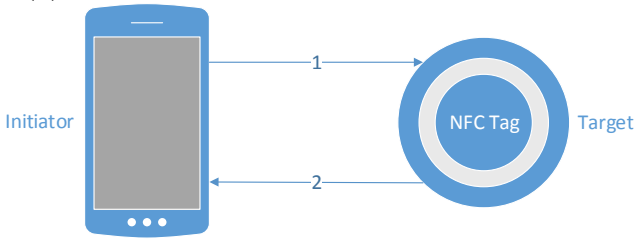



Figure 3.7 Reader/writer mode of communication

The reader/writer mode is about the communication of an NFC enabled mobile phone with an NFC tag for the purpose of either reading or writing data from or to those tags. It internally defines two different modes: reader mode and writer mode.

In reader mode, the initiator reads data from an NFC tag which already consists of the requested data. In addition to the requirement that the NFC tag already consists of the requested data, it also consists of the program which performs returning the requested data to the initiator.

In writer mode, the mobile phone acts as the initiator and writes data to the tag. If the tag already consists of any data prior to the writing process, it will be overwritten.

The algorithm may even be designed so that the initiator will update by modifying the previously existing data as well [11].

### *3.3.4.2. Peer-to-peer mode*

Peer-to-peer mode enables two NFC enabled mobile devices to exchange information such as a contact record, a text message, or any other kind of data. This mode has two standardized options; NFCIP-1 and LLCP. NFCIP-1 takes advantage of the initiator–target paradigm in which the initiator and the target devices are defined prior to starting the communication. However, the devices are identical in LLCP communication. After the initial handshake, the decision is made by the application that is running in the application layer. On account of the embedded power to mobile phones,both devices are in active mode during the communication in peer-to-peer mode. Data are sent over a bidirectional half duplex channel, meaning that when one device is transmitting, the other one has to listen and should start to transmit data after the first one finishes [11]. This mode of communication is represented in figure 3.8.

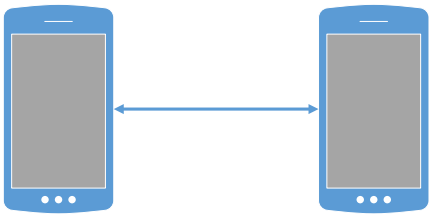


Figure 3.8 Mobile peer-to-peer mode

### *3.3.4.3. Card emulation mode*

Card emulation mode provides the opportunity for an NFC enabled mobile device to function as a contactless smart card. Mobile devices can even store multiple contactless smart card applications in the smart card. The leading examples of emulated contactless smart cards are credit card, debit card, loyalty card, transport cards, identity or access cards. Card emulation mode only removes the need for carrying the cards. People carry mobile phones with them most of the time so coupling mobile phones with the human body fits with their use. One can expect that in the near future people will carry NFC enabled mobile phones not just to gain mobility but also to perform daily functions as well. All credit cards, keys, tickets, and so on will be possibly embedded into mobile phones. Hence, there will be more opportunities to integrate daily objects into NFC enabled mobile phones in the future [11].

In this operating mode an NFC enabled mobile phone does not generate its own RF field; the NFC reader creates this field instead. Currently supported communication interfaces for the card emulation mode are ISO/IEC 14443 Type A and Type B, and FeliCa. Card emulation mode is an important mode since it enables payment and ticketing applications and is compatible with existing smart card infrastructure. Figure 3.9 shows the communication steps of this operating mode: the NFC reader (initiator) generates 13.56 MHz magnetic field (1), after which the reader (initiator) reads the information stored on the card (2).

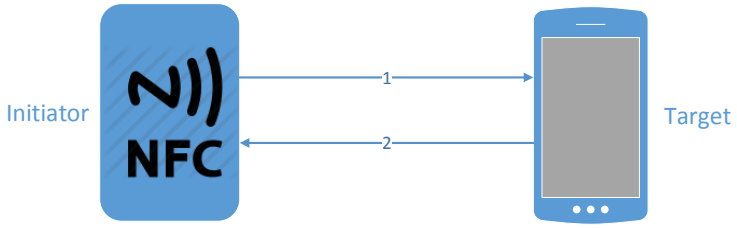



Figure 3.9 Card emulation mode

### *3.3.5. Data formats*

In all modes of operation an NFC Data Exchange Format (NDEF) message is used for the transfer of data, no matter whether the communication takes places between two NFC devices or between one device and a passive NFC tag [6].

To transfer NFC information from one device or tag to another device or tag, a standard encapsulated format is used. The NFC Data Exchange Format (NDEF) describes how information should be sent and organised in the exchange. The standard contains only information on how to organise the information transferred and does not define what is transferred or how the transmission is done. Two NFC devices close to each other will start sending NDEF messages over the NFC Forum Logical Link Control Protocol which is a part of ISO18092, but when a device is close to a tag it will start communicating NDEF messages over the specific tag protocol [19].

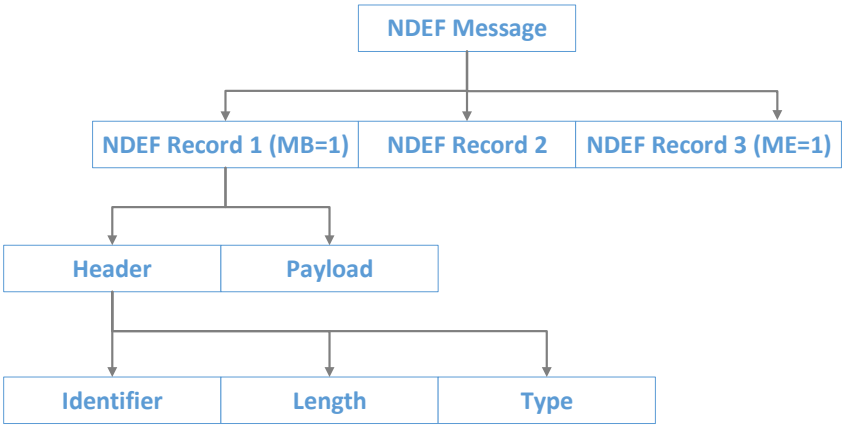



Figure 3.10 Structure of an NDEF message with three NDEF records (adapted from [8])

Specifically, an NDEF message contains one or more NDEF records. These records can be chained together to support a larger payload. Each NDEF record uses three parameters: payload length, payload type, and an optional payload identifier to describe its payload. The first NDEF record in an NDEF message has the MB (Message Begin) flag set, while the last NDEF record is marked with an ME (Message End) flag. This means that an NDEF message with only one NDEF record has both MB and ME flags set. Figure 3.10 illustrates the general structure of an NDEF message.

An NDEF record is the unit for carrying a payload within an NDEF message.

### *3.3.6. Security aspects*

Most NFC scenarios are required to deal with sensitive data like credit card numbers, bank account details,account balances, personalized tickets or other personal data [6]. For data storage and wireless data transfer security is therefore an essential issue. NFC thereby provides several mechanisms for security and immunity.

First of all, there is obviously a certain physical security due to the touch-to-connect principle. As a matter of fact, the NFC technology only provides data transfer between two devices or between a device and a tagged object when the distance between the two items falls below a certain range. Data skimming, that is capturing and intercepting transferred data by a distant attacker, is hence not possible that easily [6]. Misuse is however possible in the form of relay attacks. Such attack is basically accomplished via an attacker serving as a man-in-the-middle who is forwarding transmitted data between a reader, e.g. a reading device for NFC payments, and a target transponder, e.g. a NFC device serving as a credit card, that is actually out of the reading range. A possible countermeasure for such relay attack could for example be built upon a quite short timeout threshold that avoids transactions if the response time is too slow. The concept of Google Wallet is however also secured against such attacks as thereby a PIN has to be entered in order to activate the phone's NFC broadcast hardware and to activate the Secure Element that is storing all the sensitive data. The Wallet PIN also prevents unauthorized usage of the payment card in case the NFC phone is stolen.

For the wireless channel communication itself the NFC specifications do not ensure any secure encrypting mechanisms. On higher layers however, NFC applications can of course use industry-standard cryptographic protocols like SSL/TLS based methods to encrypt the data that is to be transferred over the air and that is stored in the Secure Element.

Other possible NFC vulnerabilities involve the manipulation of NFC tags [6]. Existing passive NFC tags, e.g. on a smart poster, could be replaced by spoofed tags such that a modified NDEF message is read by a NFC reading device. Possible attacks could for example replace the URI record with a malicious URL, e.g. in order to make the user load a phishing website that steals the users credentials. Furthermore, it is even possible to create a malformed NDEF message that causes the some applications to crash. This form of manipulation could be used by attackers to debase the relationship between a user and the pretended service provider. The Signature Record Type that was previously addressed in this paper and that signs a whole NDEF message can however serve as a countermeasure against URL spoofing and similar attacks. By manipulating the signed tag payload a signed NDEF message will lose its integrity and will be recognized as not-trusted [6].

In general, one can summarize that NFC is not more insecure than other related technologies. It offers options for encrypting data on the application layer the same way as WiFi or Bluetooth, but additionally provides safety through the requirement of very close physical proximity. Compared to traditional payment methods including magnetic strip cards that can easily be skimmed or cloned, it is quite difficult to tamper NFC hardware. Beside the physiological concern of transferring money or sensitive data without wired connection over the air, the NFC technology can thoroughly be considered as secure enough for mobile ticketing and payment - at least, if the application developers make use of the provided security mechanisms.

## 3.4. Similar systems

As already outlined, different types of e-ticketing exist. The most prominent schemes have their origin in public transport; and of those the most extensive schemes are domiciled in Asia [7]. However, not only Asian systems enjoy the status of being exemplary. The Oyster card scheme in London and the OV-chipkaart in the Netherlands as well are often referred to as being excellent, though they can also not easily be transferred to other contexts. In the following, two such e-ticketing systems will be presented in more detail, followed by a short comparison with our system.

### *3.4.1. Oyster Card*

Public transport in London is mainly organized and managed by "Transport for London" (TfL). TfL was created in 2000 as an integrated body responsible for all transport issues in London, for public transport, road based transport, cycling and walking, for managing the congestion charge and regulating the taxis [20].

The Oyster card was a success story from the beginning on: more than 3000 applicants registered for the card on the first weekend alone [7]. Smart card ticketing is available in a number of UK cities and regions with London's Oyster card being by far the most successful card scheme. The governments vision is to extend e-ticketing across the country and to potentially use mobile phones or contactless bank cards instead of smart cards [20].

For using Oyster cards, passengers need to touch their smart card on a reader at ticket gates at the start and end of their journey. It is applicable for everyone, whether living or just visiting London. Oyster smart cards can be obtained online, at ticket offices and at London Information Centres for a deposit of 5 pounds [7].

The Oyster card can store pre-purchased tickets (including weekly or monthly passes) and single fares (that are cheaper than single cash fares). An online auto top-up option that is linked to ones bank account ensures that credit never drops below a specified amount.

In 2007/2008 TfL and $O_2$ ran a six month trial with Oyster products stored on an NFC enabled phone which proved to be a success from the customers point of view [7]. More than seven million cards are used regularly in London, each week 57 million journeys are made using Oyster and more than 80% of all bus and tube payments are with Oyster [20].

### *3.4.2. U'Go Mobile Application*

In France, each city has its own ticketing system. CTS, the transport operator in Strasbourg, launched ticketing on NFC smart phones running Android in June 2013 in partnership with mobile operators Orange, Bouygues Telecom, SFR and NRJ Mobile. CTS has negotiated a transaction fee with each individual mobile network operator, which it pays for each purchase [2].

Using the U'Go mobile application, passengers can buy either monthly or daily tickets for bus and tram journeys. Payments of up to €15 could be charged to the user's mobile phone bill or paid via their bank card. For payments higher than €15, users are required to enter their bank card details to complete the purchase [7].

NFC tags are located in tram stations and attached to ticketing machines on buses in the Strasbourg region. Customers can tap their phones against the tags to validate their prepaid ticket before travelling. The transport applet on the passenger's SIM card is then updated via a mobile data connection, as soon as it's possible. The system also allows ticket inspectors to check mobile tickets for up to 24 hours after a passenger's battery goes flat or while their phone is turned off [2].

CTS isn't using photos for ID. The inspectors typically just check that the ticket has been validated. If they have doubts, they can check the name encrypted on the SIM card and ask the passenger to show some matching ID [2].

### *3.4.3. Our system*

The proposed system is a mobile application running on NFC-enabled devices. The tickets of each user are stored on the smartphone device and in the service provider's database, such that when the user replaces his smartphone he can receive access to his/her tickets, independently on the smartphone or the mobile network operator.

The tickets by themselves do not store any information about the user, the client application being responsible for granting access to the tickets based on the user's identity.

To purchase tickets, the user needs to access the mobile application and select the corresponding option (to buy more tickets). The tickets are, therefore, prepaid, and the user is constantly deciding if he/she wants to get more. No bill is issued in his name and no automatic „top-up" is performed. The user also has the possibility to purchase monthly passes on one or several bus lines, by purchasing a subscription during a freely-chosen period.

In order to validate the ticket, the user needs to tap the tag located in the common transportation vehicles with his/her NFC-enabled device. When performing this action, a ticket is subtracted from the user's remaining tickets and the details of this validation are displayed. To check that a tickets has been validated, the user only needs to show to the inspector the details of the last ticket validation.

Table 3.2 summarizes the capabilities of our system against the previously mentioned similar systems.

Table 3.1 Main features of discussed systems

| | **Oyster** | **U'Go** | **Our system** |
|---|---|---|---|
| **Scope of application** | Public transportation | Public transportation | Public transportation |
| **System technology** | Contactless smart card using MIFARE technology | NFC smartphones running Android | NFC smartphones running Windows Phone 8 |
| **Payment system** | Stored value smart card, possibility to link card to bank account for "auto top up" | Payments charged to the user's mobile phone bill or paid via their bank card | Prepaid tickets, bought through PayPal, using the mobile application |
| **Ticket validation** | Need to touch smart card on a reader at ticket gates at the start and end of a journey | Need to tap phone against the tag located in tram stations or in buses to validate prepaid ticket before travelling | Need to tap phone against the tag located in trams, trolleybuses or buses to validate prepaid ticket before travelling |
| **Services related information** | - | None | Bus lines information, routes, stations |

# Chapter 4. Analysis and Theoretical Foundation

In this section, we will present in detail the technologies used to develop this project, emphasizing the main advantages and benefits that lead to choosing that specific technology for our implementation.

## 4.1. Used technologies

### *4.1.1. Windows Phone 8 OS*

We will begin with a look at what Windows Phone 8 is, both as a new competitor in a crowded mobile space, and as a reloaded successor to Windows Phone 7. Next, we will move on to an overview of the platform architecture and of the application models against which we will be building apps. We will also look at the brand new hardware capabilities that Windows Phone 8 was designed to support and we will briefly examine the platform security model, which revolves around capability declarations.

Windows Phone 8 is the second generation of Microsoft's new mobile client operating system. Windows Phone was a complete and breaking departure from Microsoft's previous platform, Windows Mobile, but Windows Phone 8 is a major evolution and a generation change from Windows Phone 7 [22]. At the heart of this evolution is Microsoft's cross-device convergence strategy. With this release, the phone OS is getting one step closer to the Windows 8 operating system for tablets and PCs. In other words, Windows Phone 8 now has a shared core with Windows 8, which includes the well-proven NT kernel that's optimized for multicore chips, that very same kernel that can run on desktop machines and servers with up to 64 cores [1].

A key consequence of the shared core is that we now have the same .NET engine on the phone and on the desktop, as well as a shared set of native APIs for things such as media, networking, and storage. This means we can build apps that share an increasing amount of code between the phone and the PC. Another consequence of the shared core is that Windows Phone now supports native code and Direct3D [23].

Another main thing that has made Windows Phone unique in the market since its launch in 2010 is its distinctive user experience as compared to other mobile platforms [22]. Windows Phone positions itself as the most personal phone in the marketplace, and it does so by putting the user rather than the apps at the center of the experience. At the UI level, this is reflected in the tiles and hubs approach, which represents a major paradigm shift from competing platforms such as iOS and Android. Tiles are really Windows Phone signature feature. The traditional static application icons that are found in other platforms are replaced with live squares that animate in the start screen and can display constantly updated content. The other key component of the Windows Phone user experience is the Microsoft design language, which is a typography-based set of design principles that define the look and feel of the phone. One of these key principles is a focus on application content versus UI frames and graphics. This same design philosophy is also used for the Windows 8 user experience.

## *4.1.1.1. Windows Phone 8 overview*

### 4.1.1.1.1. Architecture

We will discuss the platform architecture, presented in figure 4.1.

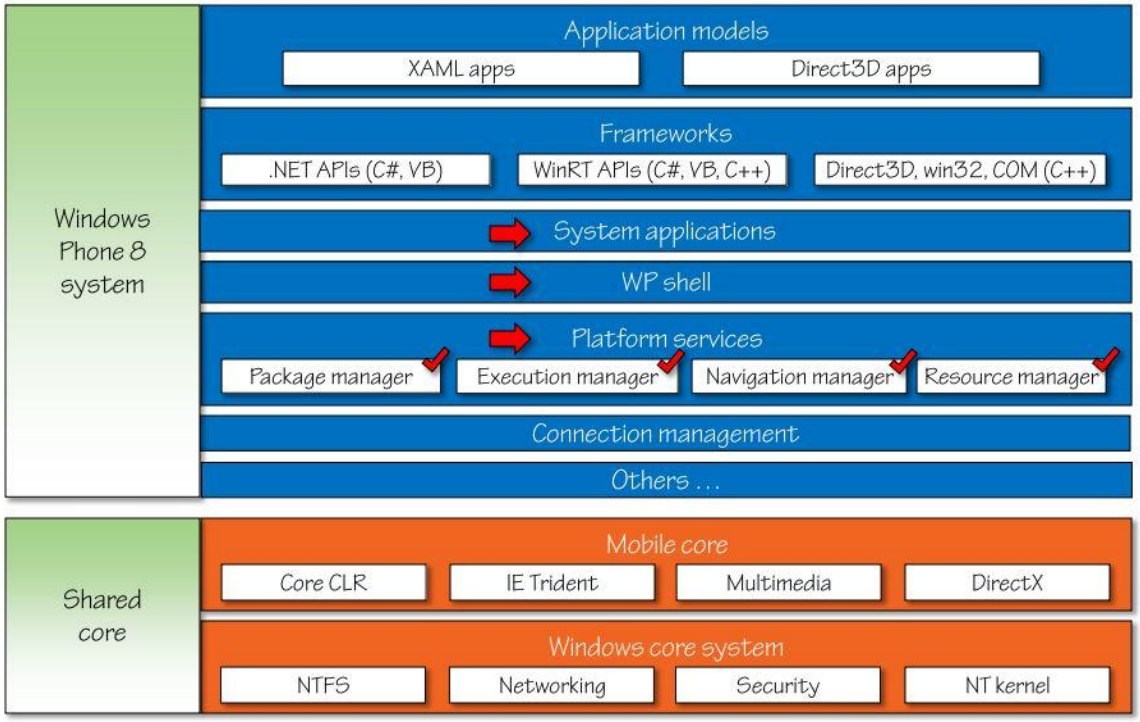



Figure 4.1 Windows Phone 8 platform architecture [23]

At the bottom of the stack is the shared core, which is composed of two main pieces [1]. The first piece is the Windows core system that contains common based operating system functionality shared across many types of Windows devices, and that includes the NT file system, the kernel, and the networking and security stacks. The second piece is a so-called mobile core, which is also shared with Windows 8, although Windows 8 bundles it together with other functionality. At the top end of the stack are the two application models that Windows Phone 8 supports, and against which we build apps. The XAML model, which has been the primary model since Windows Phone 7, and the new native code and Direct3D model brought in from Windows 8. The application models are part of the Windows Phone system that sits on top of the shared core. The phone infrastructure includes the Windows Phone shell, a set of built-in system apps such as the start application and the hub apps, and a set of core platform services that the apps rely upon. And these platform services include the package manager, which manages the deployment and licensing information for each application from installation to un-installation, and it keeps track of the installed apps, their application tiles and any other phone extensions that are associated with the installed apps; the execution manager, which manages the execution lifecycle of the apps and their associated background processes; the navigation manager, which handles the transition between running apps including both built-in and third party apps; and the resource manager, which is responsible for monitoring and maintaining the system's health and responsiveness by managing system resources such as memory and CPU.

#### 4.1.1.1.2. Security

We will briefly discuss the security model of the application platform in the Windows 8 operating system. Mobile devices usually carry significant personal information from the user such as contacts, photos, messages, geographic location information, and even financial information. All of this personal information needs to be protected from the apps that run on the device. So, the Windows Phone OS has the concept of security chamber, which provides isolation boundaries within which an application process runs [1].

Each application runs in its own isolated chamber, and each chamber initially has a minimal set of access rights that include access to its own isolated storage, but an application can never access another application's memory or data. Most modern apps need to access or use some sensitive resources. Each protected operating system resource is defined as a Windows Phone capability and after the application is published to the Windows Phone Store, these capabilities are disclosed for customers to review the capabilities, and a customer must explicitly grant his consent before the application can be downloaded and installed on the device. So, basically the user gets to decide whether or not to grant the application the requested access rights to sensitive resources such as the user's own personal information on the phone [1].

#### 4.1.1.1.3. Development fundamentals

The easiest way for an application to follow the Microsoft design language principles is to use built-in controls including layout controls such as panorama and pivot and data bound list controls, use styles, templates, and animations to customize the application’s look and feel, and the page framework for content and navigation structure [22].

Besides user experience aspects, the phone's application life cycle model must be taken into consideration. The Windows Phone operating system lets only a single application run in the foreground at any one time, so an application will automatically get deactivated if the user switches away to another application. The application may later get either reactivated or terminated by the system. It is the developer’s job to provide the user with a seamless experience across the lifecycle by saving and restoring the application's state at the right time and as quickly as possible. Even when the application is not running, it may also run some code in the background under specific and constraint conditions [22].

Beyond the fundamentals, that is building a store-compliant UI, handling the application lifecycle, and working with local and even remote data, there is the phone's real value proposition, which is its extensibility or integration points, giving the opportunity of working with live tiles and notifications, launches and choosers, application-to-application communication, and more.

### *4.1.1.2. User experience*

#### 4.1.1.2.1. Design principles

We will briefly enumerate the key design principles that the whole platform experience is based on, and which also apply to our own user experiences.

The first principle is that phone designs are clean and uncluttered. Visual elements follow a grid-like structure with consistent spacing and margins, and we achieve a sense of hierarchy and balance through the use of typography [21].

Another fundamental concept is content before chrome. We should avoid any chrome in the UI, which are the boxes and frames and lines that are found in traditional UI's. Instead, the application's content itself constitutes the experience, and the user can interact directly with that content [21].

Another key tenant on the phone is motion, and motion is found in the live tiles in the start screen and also in the transitions that connect one screen to the next, and that provides for its sense of continuity and flow, and it helps people understand the interaction with the UI [23].

Another principle is a focus on truly digital designs. And what this means is that the visuals that we use should break away from the tradition of using virtual representations of real world objects as is the case on many competing platforms and embrace the digital world (Android and iOS) [23].

#### 4.1.1.2.2. Layout and controls

The main controls that are available to on the platform can be broken down in six categories: Layout controls, Text controls, Button and Selection controls, List controls, Progress indicators, and Media and Web controls [1]. This classification is shown in figure 4.3.

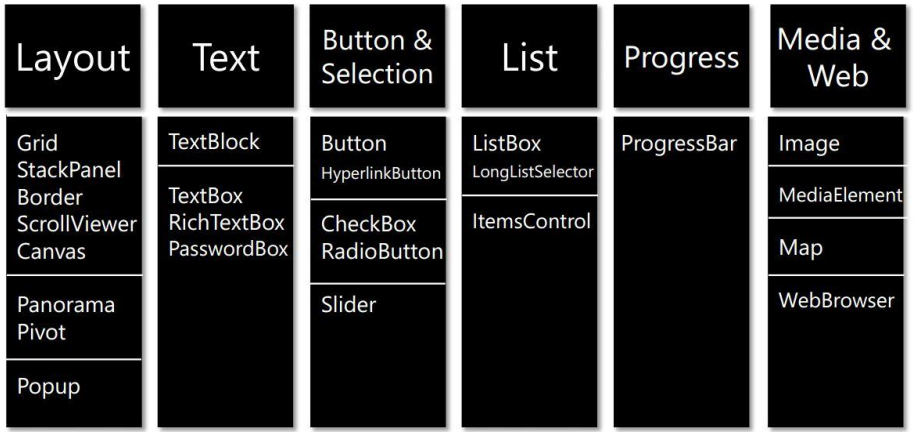



Figure 4.2 Windows Phone 8 controls classification [23]

We will take a closer look at the panorama and pivot controls. The panorama control allows to build experiences where the content spans horizontally beyond the size of the phone screen. We add content panes that each fit on the actual screen, and the user can pan from one pane to the next as if the panes were laid out on a long continuous horizontal canvas [21].

The pivot control also has multiple content panes, but pivot's focus, rather than being on providing a continuous panorama scrolling experience is on presenting multiple views of related data such as filtered lists or different views of the same page.

#### 4.1.1.2.3. Styles, resources and themes

In the UI we typically want to have a consistent look and feel for the visual elements across the application. Styles help us achieve this. A style is simply a set of property value pairs that we can apply as a group to an instance of a given UI element

type. Windows Phone defines a set of system styles that can be applied to specific UI elements, but we can actually define our own style. By using styles we can apply a consistent look and feel for your UI elements across the application [23].

Resources are common objects that pages and UI elements can use such as styles, but also things such as brushes, colors, font sizes, margins, and so on. Resources are stored in resource dictionaries, which are key dictionaries of objects created either in XAML or in code. A resource dictionary can be stored at the Element level, Page level or Application level.

#### 4.1.1.2.4. Templates

Using properties and styles we can define a common appearance for the UI elements in the application. Some types of UI elements, however, can be customized beyond just changing color, fonts, border or margin properties. These elements derive from the control class.

Controls use a template to define their visual appearance, that is the tree of visual elements that the control is composed of, and we can access this visual tree to deeply customize the look and feel of the control [22].

One can create a brand new template for a control and customize its visual look. However, the best way to approach this is to customize the existing default template so that you don't lose the control's functionality [21].

#### 4.1.1.2.5. Data binding

Data binding lets us connect a UI element called binding target to a data object called binding source. We bind a property of the target to a property of the source. The source object can be any .NET object including another UI element on the same page or even the same UI element with one property binding to another [1]. Figure 4.4 displays the data binding options.

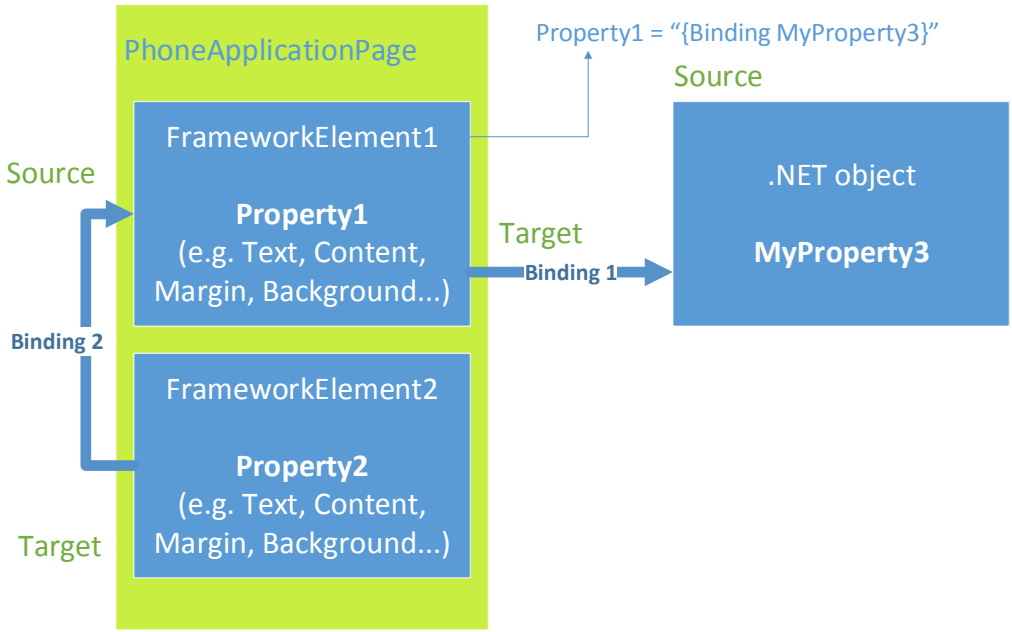



Figure 4.3 Data binding

A binding connection is created by setting the target objects DataContext property to the source object. And in XAML, we specify the source property by using the binding XAML markup extension on the target property.

Property1 property of FrameworkElement2 is binded to the MyProperty property in a .NET object. In this case, the target of the binding is Property1, while the source is MyProperty3. This is an example of binding an UI element to a data source.

Property2 property of FrameworkElement2 is in turn binded to the Property1 property of the FrameworkElement1. The target of the binding is Property2 and the source is Property1. This is an example of binding an UI element to another UI element.

Assuming that Property1 property is the Text property of a TextBox, it gets updated with the value of the source property, which is the MyProperty3 property of the .NET object. We would like the reverse to be true as well, that is the MyProperty3 property to be updated with the new value entered in the Text property of the TextBox. We can enable this by setting the Mode property of the Binding object to TwoWay instead of the default OneWay, and as a result, when we update the value of the text in the TextBox, the MyProperty3 value will be updated accordingly.

If the value of MyProperty3 changes, in order for the Text property of the TextBox to be updated automatically, the binding source object needs to notify the UI that a change has occurred, and it does this through a special interface called INotifyPropertyChanged, which the XAML system knows about. And this interface is defined in the system component model namespace, and it actually exposes a single member, which is an event called PropertyChanged. Now, any property that changes needs to raise that event for the UI to get notified and for any bound elements, any bound objects, to automatically update themselves.

When there is a type mismatch between the source property and the target property, we need to bridge this gap. This is accomplished using a simple class called a ValueConverter, that implements the IValueConverter interface, which exposes two methods, convert and convert back. And what convert does is take the source property value and it returns a type that can be consumed by the target property.

#### 4.1.1.2.6. Data templates

Data templates are used to specify the way content, including content from a bound object, is displayed in a control, and you can use a data template in content control such as Button and Pivot and Panorama, and in list controls through the ItemTemplate property, which is of type DataTemplate. Through that property, you specify how each item in a bound collection is to be displayed [1].

#### 4.1.1.2.7. Visual states

Visual states are a powerful technique for creating and managing multiple views of the UI, and transitioning from one view to another based on some event [22]. Animations are created by animating properties of UI elements such as margins, colors, opacity, and so on. In its simplest form, animating a property means specifying a start and an end value for the property and a duration. And so when the application starts, the framework smoothly transitions between the two values using linear interpolation. Animations are contained in a Storyboard, which exposes methods to start, stop, and pause the contained animations.

A visual state defines the appearance of a UI element such as a control or a page in a given state [22]. For example, a button has a normal state, a press state, a disable state. A state contains a Storyboard with animations that will start playing when the UI element transitions to that state. Visual states are grouped in VisualStateGroups, and we use the VisualStateManager class to transition to a specific state.

#### 4.1.1.2.8. Live tiles

Live tiles flashing in the start screen are Windows Phone's signature feature, one that visually sets it apart from competing platforms [23]. But tiles are also a powerful channel for an application to communicate information to the user on a continuous basis whether or not the application is running. Based on the application's scenarios, we can send update notifications to the application's tiles, including the main tile, to surface up-to-date information.

### *4.1.1.3. Navigation and lifecycle*

#### 4.1.1.3.1. Navigation

Windows Phone navigation revolves around the PhoneApplicationFrame class. In a phone application the pages are hosted in a PhoneApplicationFrame, which is a navigation control associated with each phone application. The control lets us navigate between the application's pages and work with the navigation stack. PhoneApplicationPage has a navigation service property that exposes the navigation service used by the frame hosting the page [23]. NavigationService actually shares many of the frames properties and methods, and we use these properties to perform navigation tasks in the application.

When we navigate from one page to another, we can pass parameters between the pages. This is done by appending some parameters to the URI of the target page in the Navigate method using a similar technique to web parameters (if you're familiar with them).

The navigation back stack is a collection of journal entries that represent the pages the user navigates to in the application. It uses a stack approach, so it adds a new journal entry on top of the pile when the user navigates forward to a page, and it removes the last entry from the top of the pile when the user navigates backwards to a previous page or application, either because the phone's back button was pressed or because the GoBack API of the navigation service was invoked [1].

#### 4.1.1.3.2. Application lifecycle

The application lifecycle is an essential piece of the platform architecture. The lifecycle governs when and how the application gets launched, runs, and stops running. The guiding principle here is that there's only one application that can run in the foreground on the phone at any given time to ensure the active application has enough resources to remain highly responsive all the time [23].

When an application is launched, the launching event is raised at the application level, which is important as we use lifecycle events to manage application state [21].

Once the application is in the running state, if the user navigates backwards past their first page on the stack, the application gets terminated back into the closed state, and the closing event is first raised, and we can use this event to save some data before the

application closes. If while the application is running the user hits the hardware start button or navigates forward to another application, the current application gets deactivated and goes into a dormant state. Dormant means the application's execution is paused, but the application is fully preserved in memory. If the user subsequently returns to a dormant application by navigating back to the application using the back button, the application resumes immediately as if it never paused. When the application goes into dormant mode, the deactivated event is raised.

As the user keeps navigating forward and launching new apps, more apps are successively put into a dormant state, which means more phone memory is used up. Again, to ensure the active application always has sufficient resources to run seamlessly, the resource manager will reclaim some of that memory by tombstoning some of the dormant apps.

Tombstoned apps are terminated, but the application's page and application state and its navigation state is saved. If the application is later relaunched, that is if the user navigates back to the tombstoned application, that state can be restored to memory. We want to manage application state to give the user the impression that on relaunch the application has continued running in the background, even though it was actually tombstoned. We can save state by handling the deactivated event for the case the application later gets tombstoned. And we can restore state using the activated event that's raised if the tombstoned application is relaunched. Figure 4.5 shows the stages an application goes through during its life, and the events that are triggered when the application goes from one state to another.

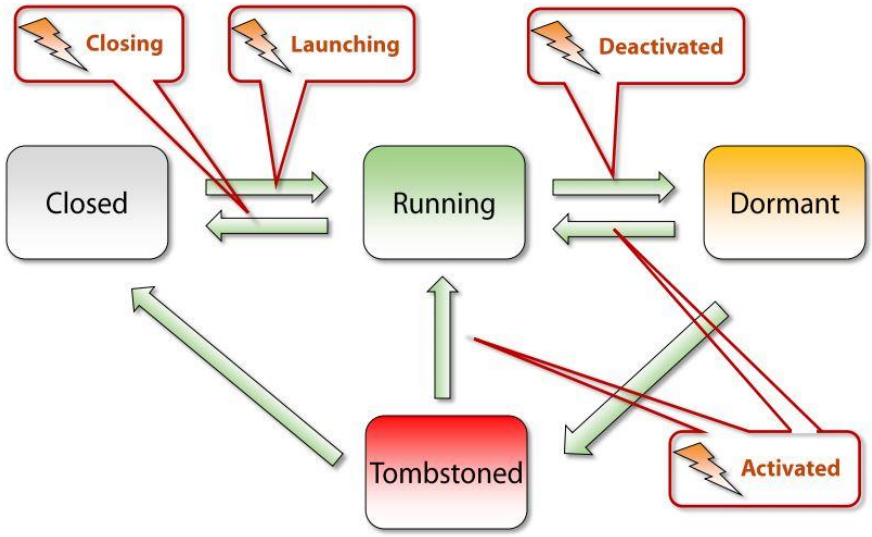



Figure 4.4 Windows Phone 8 application lifecycle [21]

### *4.1.1.4. Why Windows Phone 8?*

Windows Phone's predecessor, Windows Mobile operating system, was not seen as easy to use and intuitive. In order to change this, Microsoft radically changed its operating system in 2010 when launching Windows Phone, and put an important focus on ease of use with the introduction of its innovative Modern UI. Android, on the other hand, has long been seen as a "geeky" OS, with complicated features, an unintuitive and inconsistent user experience.

The first thing you notice on Windows Phone 8 is its pure interface. All the additional icons, buttons and chunks of text are gone and only the essential is left. When you compare it to Android, you realize the latter has a bit too much information on the homescreen and that some of it could be hidden. Windows Phone 8's homescreen looks neat and well organized, unlike Android's inconsistent and diverse widgets.

Microsoft said it from the start, carriers won't be able to alter the Windows Phone 8 interface. When we look at Android, we're often disappointed to see carrier branding throughout the interface, not to mention the removal of several features and other unpleasant tweaks.

In terms of emulators, Windows Phone has a big advantage here. The Android emulator's speed hasn't improved over the years and is still tremendously slow. Windows Phone emulator works like a charm and starts at the speed of light compared to the Android emulator. Also, Eclipse's Android plugin and toolset is very buggy and lacks many features while Visual Studio is a much more stable and mature IDE.

Windows Phone offers a great drag and drop tool and the UI creation is a straight-forward procedure. Things are a little bit more complicated on Android. There is a basic drag and drop tool, but it doesn't do a lot of things, and designing the UI for so many different types of Android screen sizes and shapes is more complicated than having to support a limited and documented set of screens.

Another important issue is the design guideline. Microsoft came with the Metro UI. The design guidelines for Android are constantly changing so you have to redesign your application often. Metro UI has come to stay..

Google made the decision to use Java as the programming language of Android to attract many Java developers. Microsoft did exactly the same, only they used C#. Cloning the good elements and coming to fix Java's weaknesses, offering extra functionality, C# is a more modern programming language and this is a point for Windows Phone.

Windows Phone 8 packs in Microsoft Wallet, making use of any NFC technology packed into handsets. This is all set to tie in to varying applications installed on the phone, allowing payment via services such as Paypal. But what sets Windows Phone apart from Android and iOS is that it makes use of secure NFC elements stored on SIM cards, which will allow for more flexibility and security when it comes to the preferred standards of card companies and mobile carriers.

The fact that Windows Phone 8 switched to Windows NT kernel means that the OS will heavily share code with its desktop counterpart (networking and multimedia code is pretty much the same across platforms), and it will take advantage of many under the hood performance enhancements made in Windows 8 for mobile devices. Apps can be easily ported across platforms, and desktop-grade features are all made possible because of this. Put simply, Windows Phone will become more of a brother to Windows 8 than a cousin.

### *4.1.2. WCF services*

Today, more than ever before, companies demand widespread technology freedom, and interoperability throughout their connected systems. We can accomplish this through what are called services. A service is simply a unit of functionality that you can expose to the outside world via messaging. The way we accomplish interoperability is by implementing that messaging using standard protocols, and message formats. For

example, one of the most common transport choices is HTTP, because of its widespread ubiquity throughout the web today. Pretty much every platform has built-in support for this popular protocol. And one of the most common message format choices is XML, either a custom XML dialect of your choosing, or a standard XML dialect that has well-defined semantic meaning, like RSS, or SOAP. The bottom line is, as long as your services communicate using standard protocols and message formats, you will be able to achieve your interoperability requirements. And thanks to this focus on messaging, you will also achieve a more loosely coupled architecture.

Today, there are two primary philosophies that one can adopt when designing service-oriented solutions. The first design philosophy centers on the ideas of SOAP, and the various WS-* specifications. Another design philosophy centers on the idea of REST, or Representational State Transfer. Each of these philosophies comes with its own pros and cons, so it's important to understand the tradeoffs between them. Today, the SOAP approach is probably most commonly used within the context of the enterprise, where you have a more controlled environment to take advantage of the various WS-* protocols. REST, on the other hand, is probably more commonly used in public-facing web service scenarios, where you might have the requirement for a high degree of scalability.

#### *4.1.2.1. SOAP-based services*

SOAP and WS-* based services build on a great deal of work to implement a completely new protocol stack for services. One of the primary design goals for this new protocol stack was that of transport neutrality, meaning that any additional features, or capabilities that we want to implement for our services, should be possible to implement in a transport neutral way. And we'll accomplish that in this protocol stack by using an XML-based messaging layer. This is where SOAP comes into the picture. SOAP is a particular XML vocabulary for packaging up messages that we need to transmit to our services.

#### *4.1.2.2. RESTful services*

RESTful services are fundamentally different than SOAP-based services, because they don't attempt to achieve transport neutrality. In fact, RESTful services typically embrace HTTP as the only transport used throughout the system. With REST, you model services as resources, and you give them unique identifiers in the form of URIs. And then you interact with those resources through a standard uniform interface, or service contract. In this case, it would be the methods defined by the HTTP protocol; specifically GET, POST, PUT, DELETE, and HEAD. By standardizing on a uniform interface, we can build infrastructure around the semantic meaning of each operation, and make performance, and scalability improvements when possible. This turns out to be extremely advantageous when building highly scalable web applications and services. We also need to be able to represent resources using a concrete representation, a message format, such as XML, RSS, JSON, etc. So when we work with our resources, when we request them, or update them, or create them, we're going to be passing a representation of that resource at a particular point in time. Now, when you build services this way, the assumption is that HTTP can provide you with all the necessary features that you need around security,

and scalability, and it has proven to be a very successful design pattern used throughout the web in a wide variety of web applications, and highly scalable web services.

#### *4.1.2.3. WCF (Windows Communication Foundation)*

WCF is the a unified communications framework that takes all the best ideas from the existing communication frameworks on Windows (ASMX – ASP.NET Web Services, WSE - Web Services Enhancement Toolkit), and it attempts to unify them into one logical model. This simplifies the communications landscape on Windows, and makes WCF the default choice for connecting your applications from this point on. When you're faced with a particular communication scenario, you simply use WCF to write the communication logic, and then you decide which of its many features you wish to employ. In one case, you may decide to use a distributed object style of communication, where you use TCP-based communications, and raw binary messages for performance reasons. In another case, you may decide to use more of a SOAP style service, where you use the SOAP-based message format on the wire, and use MSMQ to transmit those messages in a secure, reliable, and asynchronous manner. And then in another scenario, you may choose to use a RESTful style, where you actually embrace HTTP as the transport mechanism, and you retrieve resources that may be represented using the RSS format.

The point is, WCF supports all these different communication styles. We have just one way to write the code, but many ways to connect the dots between the services. This, in a nutshell, is the value of WCF. In WCF, we're going to write traditional .NET classes and interfaces. The classes will represent the messages that we're going to send as part of a service operation. And the interface definition defines the set of operations that we want to expose on a particular service contract. And so we'll use traditional .NET code to layout the basic structure of these things, and then we'll annotate them with some special WCF attributes. The combination of these two things basically defines the communication contracts for our WCF services. We can then go implement these contracts on a .NET class.

To summarize, with WCF you write services that expose endpoints to the outside world. Your service implementation defines the actual business logic. That's the way you write the code. And the endpoints define the various communication options you want to support by the service. That's how you connect the dots. An endpoint is a piece of information that tells WCF how to build the underlying communication channels that will be used at runtime, to both send and receive messages. Each endpoint definition consists of three things; an address, a binding, and a contract. The address defines a network address for both sending, and receiving messages. In the case of a WCF service, the WCF runtime will produce a transport channel that's prepared to listen for messages at that address. And in the case of a WCF client, it'll produce a transport channel that's prepared to transmit messages to that address. The binding specifies how to send the messages. For example, it specifies what transport protocol to use, what message format, and which, if any of the WS-* protocols you want to use through that particular endpoint. The binding actually conveys quite a bit of information, and it's used quite heavily by the runtime to build all of the communication channels that will be needed at runtime to support that particular style of communication. And finally, we have the contract, which tells us what the messages must contain. This basically provides additional detail around the

structuring contents of the various messages that will be used by the operations exposed through this particular endpoint. So services expose endpoints, while clients consume them, providing for a very symmetric model on both sides of the wire when you're using WCF to build your clients, as well as your services.

#### *4.1.2.4. Consuming services with WCF*

In order to consume a service, you're going to need to know several things. The client will need to know where to send the message. It'll need to know how to send the message, such as what transport, and WS-* protocols to use. And it will need to know what the messages should contain. So it's going to need to basically know all the details that are made available in an endpoint; the address, the binding, and the contract. With WCF you consume services through what are called channels, and you build channels based on endpoints. First, the client has to somehow retrieve the endpoint definitions from the service metadata. So here we have a service sitting out there on the network, and it has exposed several endpoints, and let's say that the client wants to talk to it through that middle endpoint. Well, somehow the client's going to have to walk up to this service and request the metadata, typically in the form of a WSDL definition. That metadata will come down to the client, and the client will run that metadata file through what's called a metadata import tool. That will produce a client-side version of those endpoint definitions. When using WCF on both sides of the wire, it'll essentially look just like the endpoint definitions that we used on the server side. Once the client has those local endpoint definitions, it can then construct a channel based on the endpoint of interest, and once it has constructed that channel, it can simply make method calls through that channel to transmit the appropriate messages to the service. So, as you can see, endpoints provide for some very nice symmetry across clients and services.

#### *4.1.2.5. SOAP versus REST*

REST is fundamentally different from SOAP. SOAP defines a transport-neutral model focused on defining custom service contracts with custom operations, and you can invoke those operations over a variety of different transports using different messaging codings. REST, on the other hand, defines more of a transport-specific model.

REST doesn't actually tie you to HTTP, but in reality, HTTP is the only protocol that is used in practice today for building RESTful architecture, so typically the two go hand-in-hand. When you say you're doing REST, that typically implies that you're also doing HTTP. With the RESTful model, instead of being focused on custom service contracts, and custom operations, is focused on defining resources, and identifying those resources through a unique identifier, or a URI. Then, you build services around a uniform interface. So, instead of defining a custom service contract, we're going to define a uniform service contract that all of our services will use. If you use HTTP to implement your RESTful services, that uniform interface is defined by the various HTTP methods, so GET, POST, PUT, DELETE, HEAD. And so those are the operations that we'll be able to invoke on our resources. And when we get a resource, we'll then be retrieving a representation of that resource over the protocol. So, what's actually returned to us over the wire is a representation, and we can use a wide variety of data formats to represent that resource. We could use HTML for example, or XML, or even things like RSS, or JSON. So there are still a variety of ways that you can represent resources when you

move them back and forth between clients and services, but in the end, we'll always be using a uniform interface to interact with those resources that are exposed by our service.

To summarize, SOAP emphasizes verbs, or actions, while REST emphasizes nouns, or resources. When you define a SOAP service, your focus is on defining the service contract, the set of custom operations that you're going to expose through that service. So, you're essentially defining actions, or verbs, that you're going to be able to use through a variety of different transports. When you use SOAP with HTTP, you're always going through the HTTP POST method. So the action is not really defined by the transport, it's defined by the message that you're sending through that transport.

With REST, on the other hand, your focus is on defining resources, and identifying those resources with unique identifiers, or URIs. Then, you'll interact with those resources through a uniform interface, typically defined by HTTP. So you'll use the various HTTP verbs, like GET, POST, PUT, DELETE, and HEAD, to interact with the resources. So, for example, I could GET a User to retrieve a representation of it, or I could POST a User to create a new one, or I could PUT a User to update it in the system, or I could DELETE a User to actually remove it from the system. So, I can use any of those standard operations with any of these resources. I could do the exact same thing with the Location resource. So that's the beauty of REST. We have a standard uniform interface that can apply equally to all of our resources that we expose through that service, and they provide the standard CRUD operations that you would typically expose through a SOAP service interface as well. And when you do that, you'll define standard representations for those resources that you're going to be sending back and forth across the wire. There are many different ways that you can represent a single resource. That, in a nutshell, is the primary difference between REST and SOAP.

### *4.1.2.6. Why WCF?*

Each architecture has its own benefits, and there's no clear winner for all use cases. SOAP and WS-* is primarily seen today within the enterprise. So, if you're building services that are going to be used within the enterprise, where you have more control over the different pieces within the system, and you have more complex service requirements, maybe you have high performance requirements, and you need to use a wide variety of transports within the service-oriented solution, you'll probably need to use SOAP and WS-* to achieve all those goals. But in scenarios where you need a high degree of interoperability, and a high degree of scalability, REST will tend to be a better fit. REST is all about scalability. So, anytime you're building services that will be web facing to the public, and you're going to need a high degree of interoperability, REST will probably do a better job of helping you achieve those goals. Now, one of the big tradeoffs between SOAP, and WS-*, and REST, is the tool support. With SOAP and WS-*, there's great tool support provided by all the big SOAP vendors, like Microsoft, IBM, BEA, Sun, etc. With REST, on the other hand, there's very limited tool support beyond the basic HTTP and XML stacks that you can find pretty much on any platform out there.

Why should I choose WCF? The most fundamental reason is that it's bound to increase your developer productivity. Thanks to WCF's unified programming model, developers only have to worry about a single programming model for writing all their communication logic. And that fact alone is bound to increase your developer productivity results over time. Another reason is that if you move to WCF, you will

immediately increase your interoperability potential. Like we discussed, WCF is capable of doing basic web services, and all the way up to advanced SOAP and WS-* communications, giving you a wide range of communication options on the wire. WCF also provides increased flexibility. Not only in terms of the communication, but also in terms of being able to plug in your own code. If you're not happy with some of the built-in functionality that comes with WCF out of the box, you can always inject your own behaviors, and communication protocols that address your precise needs. And, of course, thanks to its interoperability potential, and the fact that it can integrate on the wire with some of today's existing Microsoft frameworks, it's very easy to plug those new WCF services into your existing connected systems. And the last, perhaps most practical reason for moving towards WCF, is that all of Microsoft's future communications work is focused here. So, if you want to benefit from all of Microsoft's future investments on the communications front, you'll really need to move over to WCF at some point in time.

### *4.1.3. PayPal*

PayPal is an online banking service. As one of the oldest and most recognized names in the internet banking business, PayPal enjoys widespread acceptance as a payment medium for all kinds of online transactions. In addition to basic online payment services it offers additional payment services via mobile phones, browser add-ons, and more.

The fundamentals of PayPal are pretty easy to grasp. Your PayPal account is much like any savings or checking account, except PayPal was designed specifically for online transactions. Because it is so easy to use, PayPal is the favorite of millions of amateur sellers and buyers around the world.

According to [24], today over 140 million internet users prefer to use PayPal to send money to each other via email. PayPal has become such a convenient and trusted way to transfer money online, 90% of eBay's purchases go through PayPal. However, one of the keys to PayPal's success has been its ability to expand beyond the eBay market. You can use it send money to a friend, donate to a charity and buy items from online merchants.

PayPal doesn't fundamentally change the way merchants interact with banks and credit card companies. It just acts as a middleman. Credit and debit card transactions travel on different networks. When a merchant accepts a charge from a card, the merchant pays an interchange, which is a small fee. The interchange is made up of a variety of small fees paid to all the different companies that have a part in the transaction: the merchant's bank, the credit card association and the company that issued the card. If someone pays by check, a different network is used, one that costs the merchant less but moves more slowly.

What part does PayPal play in all this? Both buyer and seller deal with PayPal, having already provided their bank account or credit card information. PayPal, in turn, handles all the transactions with various banks and credit card companies, and pays the interchange. They make this back on the fees they charge for receiving money, as well as the interest they collect on money left in PayPal accounts.

PayPal promotes its presence as an extra layer, as a security feature, because everyone's information, including credit card numbers, bank account numbers and

address, stays with PayPal. With other online transactions, that information is transmitted from the buyer to the merchant to the credit card processor.

#### *4.1.3.1. Buyers*

For buyers who want to make payments online, the PayPal platform is a financial gateway that allows simple, private, and secure payments on a large number of websites. Unlike paying directly with credit cards online, paying with PayPal does not require sharing sensitive information such as credit card numbers on merchants' websites and simplifies payment information entry to only an email address and password.

#### *4.1.3.2. Sellers*

For vendors who want to sell products online and collect payments electronically, the PayPal platform is a payment processing solution that allows them to accept secure online payments from customers in over 190 markets and in 19 currencies [25]. Unlike online solutions provided by traditional merchant accounts, PayPal's product can be set up quickly and easily and has a low cost transaction fee structure.

#### *4.1.3.3. PayPal Windows 8 Checkout SDK*

The PayPal Windows 8 Checkout SDK gives the ability to integrate PayPal payment functionality into the apps created for Windows 8 Store and Windows 8 Phone.

PayPal is the most trusted payment system for online transactions, and offering PayPal as a payment method lets customers be secure with the transactions they complete within the application. In addition to offering secure payments, PayPal also gives the ability to sell digital goods from within an application, which is a unique offering among Microsoft platform payment methods. Accepting PayPal can open new revenue streams because one can sell addition program content and software updates from within the Windows application.

Windows 8 Checkout SDK uses Mobile Express Checkout for its underlying technology, and it offers the same functionality with the same pricing model. PayPal's Mobile Express Checkout (MEC) gives the ability to place a Check Out with PayPal button on a mobile website/application, which enables customers to check out via their PayPal accounts. MEC is based on Express Checkout. It gives mobile users a streamlined checkout process—they don't have to enter their banking or shipping information into their mobile devices because this information is contained, and shielded, in their PayPal accounts. In addition, MEC is flexible in that it can give customers without PayPal accounts the option to check out using their credit or debit cards.

#### *4.1.3.4. Why PayPal?*

In order to motivate our choice, we will compare PayPal with another online payment system, Fortumo.

Fortumo allows any merchant to set up payment processing for web and mobile services, games or apps. Users with a mobile phone are then able to make one-click payments using Fortumo without the need for a credit card - payments are charged to their mobile operator bill instead. Fortumo payments are cross-platform and work in PC applications, web services and HTML5, Android, Windows Phone and Windows 8 apps.

Fortumo supports payments in 81 countries through 300 mobile operators, including a number of exclusive direct carrier billing partnerships. Fortumo offers self-service setup with no monthly fees or minimum volume commitments, allowing any developer regardless of size and location to sign up for an account and get started with mobile payments.

Fortumo In-App Payments for Windows 8 and Windows Phone is designed to be extremely simple and straightforward. The payment flow uses Premium SMS messages and HTTP requests to process payments. Payment is considered to be successful after your application has received payment confirmation from Fortumo server, and virtual goods can be granted to the end-user.

From this, we can conclude Fortumo is a viable and promising option for our solution. The disadvantage of Fortumo, that ultimately gave PayPal the upper hand, is the fact that all transactions are billed to the mobile operator. For our current needs, this is not an option, because many people have a prepaid sim card, and overlooking that side of the market would be a bad choice.

### *4.1.4. Entity Framework*

Entity Framework is an Object Relational Mapper(ORM). ORMs are aimed to increase developer productivity by relieving of the tedium and redundant task of persisting the data that you use in your applications. Taking advantage of its default behaviors, Entity Framework can generate the necessary database commands for reading or writing data and execute them for you in the database. If you're querying, you can express your queries against your own domain objects using LINQ to entities. Entity Framework will execute the relevant query in the database and then materializes results into instances of your domain objects for you to work within your application. So you focus on your domain and Entity Framework takes care of the database work.

There are other ORMs in the marketplace such as NHibernate and LLBLGen Pro. Most ORMs typically map domain types directly to the database schema. Entity Framework has a more granular mapping layer so you can customize mappings, for example, by mapping the single entity to multiple database tables or even multiple entities to a single table. Microsoft recommends that you use Entity Framework over ADO.NET or LINQ to SQL for all new development.

#### *4.1.4.1.EF conceptual model*

Entity Framework lets you focus on your business domain instead of database development. With Entity Framework, the focal point is referred as a conceptual model. It's a model of the objects in your application, not a model of the database you use to persist your application data. Your conceptual model may happen to align with your database schema or it may be quite different. You can use a Visual Designer to define your conceptual model which can then generate the classes you'll ultimately use in your application. Or you can just define your classes and use a feature of Entity Framework called Code First, and then Entity Framework will comprehend the conceptual model. Either way, Entity Framework is able to work out how to move from your conceptual model to your database. So you can query against your conceptual model objects and work directly with them.

While Microsoft provides Entity Framework support in the SQL Server provider, there are many third party providers that allow to you to use Entity Framework with a variety of databases, from commercial databases like Oracle, to Open Source databases like MySQL and Firebird.

Entity Framework is able to keep track of changes made to objects that it's aware of, including adding new objects or deleting some. If you're designing disconnected apps where Entity Framework won't be around at the time the objects are actually being edited, there are patterns you can follow to let Entity Framework know what changes it should be aware of when you pass the objects back, for Entity Framework to save back to the database. Entity Framework has a variety of ways that it supports toward procedures and a complex API that let's you have granular control over everything from its modeling to it's runtime behavior.

### *4.1.4.2. EF capabilities*

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. The high-level capabilities of Entity Framework:

- Works with a variety of database servers (including Microsoft SQL Server, Oracle, and DB2)
- Provides integrated Visual Studio tools to visually create entity models and to auto-generate models from an existing database. New databases can be deployed from a model, which can also be hand-edited for full control
- Provides a Code First experience to create entity models using code. Code First can map to an existing database or generate a database from the model.
- Integrates well into all the .NET application programming models including ASP.NET, Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), and WCF Data Services (formerly ADO.NET Data Services)

The Entity Framework is built on the existing ADO.NET provider model, with existing providers being updated additively to support the new Entity Framework functionality.

### *4.1.4.3. Entity Framework versus other ORMs*

#### 4.1.4.3.1. Entity Framework vs. traditional ADO.Net

The highlights are that you can write code against the Entity Framework and the system will automatically produce objects for you as well as track changes on those objects and simplify the process of updating the database. The EF can therefore replace a large chunk of code you would otherwise have to write and maintain yourself.

Further, because the mapping between your objects and your database is specified declaratively instead of in code, if you need to change your database schema, you can minimize the impact on the code you have to modify in your application, so the system provides a level of abstraction which helps isolate the application from the database.

Finally, the queries and other operations you write into your code are specified in a syntax that is not specific to any particular database vendor. ADO.NET, prior to the EF,

provided a common syntax for creating connections, executing queries and processing results, but there was no common language for the queries themselves; ADO.NET just passed a string from your program down to the provider without manipulating that string at all, and if you wanted to move an application from Oracle to SQL Server, you would have to change a number of the queries. With the EF, the queries are written in LINQ or Entity SQL and then translated at runtime by the providers to the particular back-end query syntax for that database

#### 4.1.4.3.2. Entity Framework vs. nHibernate

The big difference between the EF and nHibernate is around the Entity Data Model (EDM) and the long-term vision for the data platform we are building around it. The EF was specifically structured to separate the process of mapping queries/shaping results from building objects and tracking changes. This makes it easier to create a conceptual model which is how you want to think about your data and then reuse that conceptual model for a number of other services besides just building objects. So the differentiator is not that the EF supports more flexible mapping than nHibernate, it's that the EF is not just an ORM, it's the first step in a much larger vision of an entity-aware data platform.

### *4.1.4.4. Why EF?*

Using the Entity Framework 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
- 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 (LINQ to Entities) provides IntelliSense and compile-time syntax validation for writing queries against a conceptual model.

## *4.1.5. AutoMapper*

Many times we need to map objects between different application layers, we need to translate data from one object type to another. Common examples include DTOs (Data Transfer Objects), View Models, or even just some request or response object from a service or Web API call. The solution to these problems is to create some mapping mechanism with minimal configuration that we can use in a fairly generic way in our code, but thankfully, we don't have to actually do that: third-party libraries exist that we can lean upon, specifically, AutoMapper.

AutoMapper is a mapper between two objects. It maps two different entities by transforming an input object of one type to an output object of another type. Figure 4.1. displays the working principle of AutoMapper.

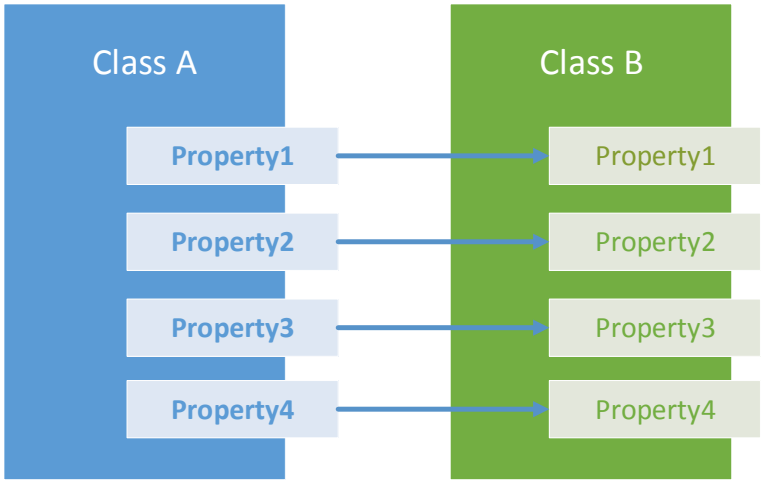



Figure 4.5 Mapping between entitities A and B

It is very tough job to map two different entities. AutoMapper resolves that tough job by mapping it with proper convention and removes all the dirty work. Not only that, you can also customize the mapping with proper parameters. Also, AutoMapper shines in the ability to map lists and nested properties by convention,o r constructing an object based on another object's properties.

### *4.1.5.1. Automapping*

Auto-mapping is done at runtime, so when the entity definitions change, there are no upstream code changes. AutoMap uses reflection, but the performance hit is relatively small and the map can be cached if it's going to be used repeatedly.

Any properties not explicitly specified are auto-mapped. Mapping degrades gracefully, so any properties which can't be mapped (either because the names can't be matched, or the source cannot be read from, or the target cannot be written to) are not populated. (Optionally you can force exceptions to be thrown on mismatches).

AutoMap uses a naming strategy to match properties. By default this uses a simple matching algorithm, ignoring case and stripping non-alphanumeric characters. You can override the default to use exact name matching, aggressive name matching or to supply your own strategy (implementing IMatchingStrategy).

### *4.1.5.2. Static mapping*

For complex maps, or for scenarios where you don't want the reflection performance hit at all, you can define a static map. The interface is the same as AutoMap, except by default all properties have to be specified – there is no auto-mapping of unspecified targets, so additionally the naming and caching strategies are ignored. Static object maps are derived from ClassMap, with the specifications made in the constructor

You can specify mappings in an action, or specify source and target with funcs as you prefer. Execute the map in the same way by calling Create or Populate to map from the source instance to a target. You can mix static and auto-mapping by setting AutoMapUnspecifiedTargets, meaning that the auto-map will be used for any target properties which have not been explicitly specified, which also allows your static map to leverage the naming and caching strategies of AutoMap.

#### *4.1.5.3. Nested maps*

AutoMap doesn’t traverse object graphs, it will only populate properties in the first-level object (except where you have specified a mapping for a child object). To populate full graphs you can use nested auto-maps or static maps.

#### *4.1.5.4. Performance*

As always, the generic solution has a performance implication, although the mapping has had a couple of rounds of optimisation done to minimise the overhead. Up to 1,000 objects, the performance hit in using the AutoMapper is negligible, but above 1,000 objects the cost is more pronounced.

#### *4.1.5.5. Why AutoMapper?*

You can write extension methods to move data between properties (perform the mapping manually), but why write countless mapping lines just for shipping data across types? AutoMapper will automatically move data from your business entities into the corresponding properties on your DTO, saving you many, many lines of repetitious assignment statements. It also means that if you add a new property to your DTO, AutoMapper will automatically pick up the corresponding property from the entity; no further changes to the code are required.

AutoMapper provides all the convention you need to get the simple cases done for free, and simple patterns for managing the more complex scenarios where you are reducing shapes or changing types. Very little setup is needed. The CreateMap call sets up a mapping between a source class and a destination class.

By default, AutoMapper creates a new instance of the object that the data is being moved into. The object that's returned from AutoMapper is a different object from the object retrieved from the DbContext. That's bad in the context of Entity Framework, because EF won't track the changes made to that new object, meaning that when SaveChanges is called, nothing will be saved back to the database. Entities can be updated with AutoMapper by passing both the DTO and the entity object to AutoMapper's Map method. Rather than creating a new entity object to return from the Map method, AutoMapper now updates the entity object passed in the second parameter and returns it to the existing variable. As a result, EF will track the changes made to the entity object and when SaveChanges is called, the updates will be sent to the database.

### *4.1.6. NDEF Library for Proximity APIs/NFC*

NFC tags and the content sent in device-to-device communication when tapping two phones is based on certain standards by the NFC Forum (called NDEF – NFC Data Exchange Format). When it comes to storing data on NFC tags that can have as little writable storage as around 40 bytes, very efficient and complex data storage schemes are necessary. The downside is that most operating systems do integrate the NFC data transmission at the base level, but offer developers very little support for the NDEF standards on top. Obviously, creating an own implementation of a message that stores a simple URL on a tag for example, is not an easy job.

#### *4.1.6.1. The NFC Library*

The open source NFC/NDEF Library contains a large set of classes that take care of formatting data according to NDEF standards, so that these can be directly written to NFC tags or sent to other devices.

In your application, you choose the corresponding record type (e.g., for URLs, emails or geo tags) and provide the necessary data. The library creates an NDEF message out of the data, which you can directly send to the NFC stack in your operating system as a byte array, which takes care of writing it to a tag or publishing it to another device.

Additionally, the library can parse NDEF byte arrays that you read from tags or receive from other devices and create a list (NDEF Message) of data classes (NDEF records) that you can easily analyze and use in your application.

For Windows Phone 8, the NFC stack is represented through the Proximity APIs - they encapsulate NFC hardware communication and basic NDEF formatting for a very limited subset of the NDEF standards. This missing part is added by this NDEF library.

#### *4.1.6.2. Availability*

The NFC / NDEF library is written in C# and can therefore be used on any operating system that supports C# development. The library is available as a ready-made portable class library, which can be used on the Windows 8 platform, as well as on Windows Phone 8. Both platforms provide support for interacting with the NFC hardware through the Proximity APIs.

Additional platform-specific functionality is added through the the separate extension library. It integrates with the platform APIs for WinRT / WP8.0 and allows real-life tasks like creating a business card record based on a Contact from the Windows 8 address book.

#### *4.1.6.3. NDEF library features*

- Parse NDEF message and records from raw byte arrays
- Supports fully standardized basic record types:
  - Smart Poster
  - URI
  - Text
- Smart URI class: automatically represents itself as the smallest possible NDEF type (URI or Smart Poster), depending on supplied data
- LaunchApp tags - launch a Windows (Phone) application just by tapping a tag

## 4.2. Use cases specification

A use case is a kind of story of how a system and its actors collaborate to achieve a specific goal. It is a step-by-step description of a particular way of using a system. The structure of a use case is narrative in nature. The story tells how the system and its actors work together to achieve something of significance to the actors involved.

Each use case expresses a goal of the actors involved and describes a task that the system, with the assistance of the appropriate actors will perform. Different user types are represented as actors. An actor is anything that exchanges information with the system.

An actor can be a user, external hardware, or another system. Each actor must have goals with respect to the use case it is interacting with in the problem domain.

We will continue by identifying the actors that will be interacting with our system, the use cases present in the application and we will provide a detailed overview of the main identified use cases.

First, we identify the actors present in our system by answering the questions:

- Who will operate on the system?
- Who will supply, use or remove information from the system?

This gives us the following actors:

- The traveler
- The transport operator

Then we ask the following questions to identify use cases:

- What functions will the actor want from the system?
- Does the system store information? What actors will create, read, update or delete this information?

### *4.2.1. The traveler*

The traveler is the primary actor of the system. The functions he/she wants to perform using the system are:

- Register into the system
- Authenticate into the system
- Validate a ticket
- See past validations
- See subscriptions (past and recent)
- See the status of his/her tickets
- Buy tickets
- Buy monthly pass (subscribe)
- Consult bus lines
- Log out

The most relevant use cases for our application will be presented in more detail.

#### *4.2.1.1. Validate ticket*

*Brief description*

The purpose of this use case is to present the flow of events that an actor must follow in order to manage the validation of a ticket in a custom scenario.

*Primary actor*

The primary actor is the traveler which performs a ticket validation.

*Stakeholders and interests*

The stakeholders are:

- The actor itself, interested in having an intuitive way of validating tickets and a user friendly application with a short response time.
- The transport operator, who wants to maximize profits and reduce travel fraud.

*Flow of events*

Basic flow

Use case start

This use case starts when the actor gets into a bus or other transportation vehicle and needs to obtain the right to use its services.

Steps

- The actor unlocks the screen of his mobile device and taps the mobile device to the NFC tag installed on the bus.
- The operating system asks the actor to open the ticketing application on his device.
- The actor opens the application.
- The system checks the number of tickets the actor has.
- The system will decrement the number of tickets he has left.
- The system performs the validation and synchronizes it with the server and the details of the validation are displayed to the actor.

Use case end

This use case ends when the validation is saved to the database and displayed to the user.

Alternative flows

*Abort validation*

This flow can occur when:

1. The actor does not open the ticketing application when notified.

The system remains unchanged.

*Validation failed*

This flow can occur when:

1. The actor does not have any remaining tickets to use.

The system displays a message notifying the actor of this error and asks him to buy more tickets.

The system remains unchanged.

*Preconditions*

The actor has the ticketing application installed on his/her device.

The actor's device has an active Internet connection.

The actor is authenticated and authorized for this use case.

*Postconditions*

The data on the actor's device and the data on the server should be consistent if the validation succeeds.

The data on the server and on the user's device should remain unchanged if the validation fails.

The data on the server and on the user's device should remain unchanged if the validation is aborted.

### *4.2.1.2. Buy ticket(s)*

*Brief description*

The purpose of this use case is to present the flow of events that an actor must follow in order to buy tickets in a custom scenario.

*Primary actor*

The primary actor is the traveler which uses the tickets for traveling.

*Stakeholders and interests*

The stakeholders are:

- The actor itself, interested in buying tickets in an easy way, anytime and anywhere he/she is, without the need to go to a facility to purchase them.
- The transport operators, who want to simplify their work, while minimizing costs and maximizing profit.
- The payment industry, interested in integrating its services in the transportation system, to increase its profit.

*Flow of events*

Basic flow

Use case start

This use case starts when the actor wants to buy tickets.

Steps

- The actor accesses the page within the application from where he/she can buy tickets, selects the number of tickets he/she wants to buy and proceeds by pressing "Pay now".
- The system will redirect the user to a PayPal login page, where information related to his/her purchase are displayed, along with the amount he/she needs to pay.
- The actor logs in into his PayPal account, using his PayPal credentials, and confirms the payment by pressing "Continue".
- The system redirects the user back to the page from where the actor can buy tickets, and displays a message confirming the success of the actor's purchase.

Use case end

This use case ends when the new number of tickets the user has is saved to the database and the actor's ticket status is updated to reveal these changes.

Alternative flows

*Payment canceled*

This flow can occur when:

1. The actor does not login into his/her PayPal account and returns to the previous page.
2. The actor does not confirm his payment and returns to the previous page.

The system displays a message to the actor notifying the actor that the payment was canceled.

The system remains unchanged.

*Payment failed*

This flow can occur when:

1. The actor does not have enough money.
2. An error occurs while processing the payment.

The system displays a message notifying the actor of this error.

The system remains unchanged.

*Preconditions*

The actor's device has an active Internet connection.

The actor is authenticated and authorized for this use case.

The actor has a PayPal account and money in his account.

*Postconditions*

The data on the actor's device and the data on the server should be consistent if the purchase succeeds.

The data on the server and on the user's device should remain unchanged if the payment is canceled.

The data on the server and on the user's device should remain unchanged if the payment fails.

*Extension points*

Pay using debit/credit card

- When redirected to the PayPal login page, the actor selects to pay with a credit/debit card instead of logging in into his/her PayPal account.

### *4.2.1.3. Subscribe (buy monthly pass)*

*Brief description*

The purpose of this use case is to present the flow of events that an actor must follow in order to make a month's subscription (buy a monthly pass) in a custom scenario.

*Primary actor*

The primary actor is the traveler which uses the monthly pass for traveling.

*Stakeholders and interests*

The stakeholders are:

- The actor itself, interested in subscribing for a monthly pass in an easy way, anytime and anywhere he/she is, without the need to go to a facility to purchase them.
- The transport operators, who want to simplify their work, while minimizing costs and maximizing profit.
- The payment industry, interested in integrating its services in the transportation system, to increase its profit.

*Flow of events*

Basic flow

Use case start

This use case starts when the actor wants to buy a monthly pass.

Steps

- The actor accesses the page within the application from where he/she can subscribe, selects the starting date of the monthly pass, selects one, two, three or all available bus lines and then proceeds by pressing "Pay now".
- The system will redirect the user to a PayPal login page, where information related to his/her purchase are displayed, along with the amount he/she needs to pay.
- The actor logs in into his PayPal account, using his PayPal credentials, and confirms the payment by pressing "Continue".
- The system redirects the user back to the page from where the actor can subscribe, and displays a message confirming the success of the actor's purchase.

Use case end

This use case ends when the new user's new subscription is saved to the database and the list of the actor's subscriptions is updated to display these changes.

Alternative flows

*Incorrect input*

This flow can occur when:

1. The actor selects a date from the past
2. The actor doesn't select any bus line, or he/she selects a number of bus lines different from one, two, three or all bus lines.

The system displays a message to the actor notifying him/her of the incorrect input.

The actor continues the use case from the point where he/she inputs the data according to his/her needs.

*Payment canceled*

This flow can occur when:

1. The actor does not login into his/her PayPal account and returns to the previous page.
2. The actor does not confirm his payment and returns to the previous page.

The system displays a message to the actor notifying the actor that the payment was canceled.

The system remains unchanged.

*Payment failed*

This flow can occur when:

1. The actor does not have enough money.
2. An error occurs while processing the payment.

The system displays a message notifying the actor of this error.

The system remains unchanged.

*Preconditions*

The actor's device has an active Internet connection.

The actor is authenticated and authorized for this use case.

The actor has a PayPal account and money in his account.

*Postconditions*

The data on the actor's device and the data on the server should be consistent if the purchase succeeds.

The data on the server and on the user's device should remain unchanged if the payment is canceled.

The data on the server and on the user's device should remain unchanged if the payment fails.

*Extension points*

Pay using debit/credit card

- When redirected to the PayPal login page, the actor selects to pay with a credit/debit card instead of logging in into his/her PayPal account.

### *4.2.2. The transport operator*

There is only one use case associated with the transport operator, and that is configuring the tag.

### *4.2.2.1. Tag configuring*

*Brief description*

The purpose of this use case is to present the flow of events that an actor must follow in order to configure a tag in a custom scenario.

*Primary actor*

The primary actor is the transport operator which configures the tag with the corresponding data of the bus where the tag will be placed.

*Stakeholders and interests*

The stakeholders are:

- The transport operator, who wants to perform his /her job in an easy and quick manner.
- The transport authorities, which are interested in an environment friendly transportation system.
- The suppliers, who want their NFC tags on the market.

*Flow of events*

Basic flow

Use case start

This use case starts when the actor wants to configure a tag.

Steps

- The actor accesses the tag writing application, selects the data he/she has to write on the tag and then proceeds by tapping the device to the tag and pressing the "write tag" button.
- The system writes the data to the tag.

Use case end

This use case ends when the new data is written to the tag.

Alternative flows

*Writing fails*

This flow can occur when:

1. The tag is not formatted.

The system remains unchanged.

*Preconditions*

The actor's device has an active Internet connection.

*Postconditions*

The tag is written with the new data if the tag writing succeeds.

# Chapter 5. Detailed Design and Implementation

In this chapter, we are going to present the detailed implementation of the project. We will first describe the system as a whole and present its architecture, and then we will break it into its main components and discuss each component's functionality, how it accomplishes its goals and what are some strengths and weaknesses.

## 5.1. Initial system approach

Before diving into the system's architecture, we need to understand how the system's components will interact. In figure 5.1 we present the involved entities in the processes of validating a ticket, buying tickets, etc.

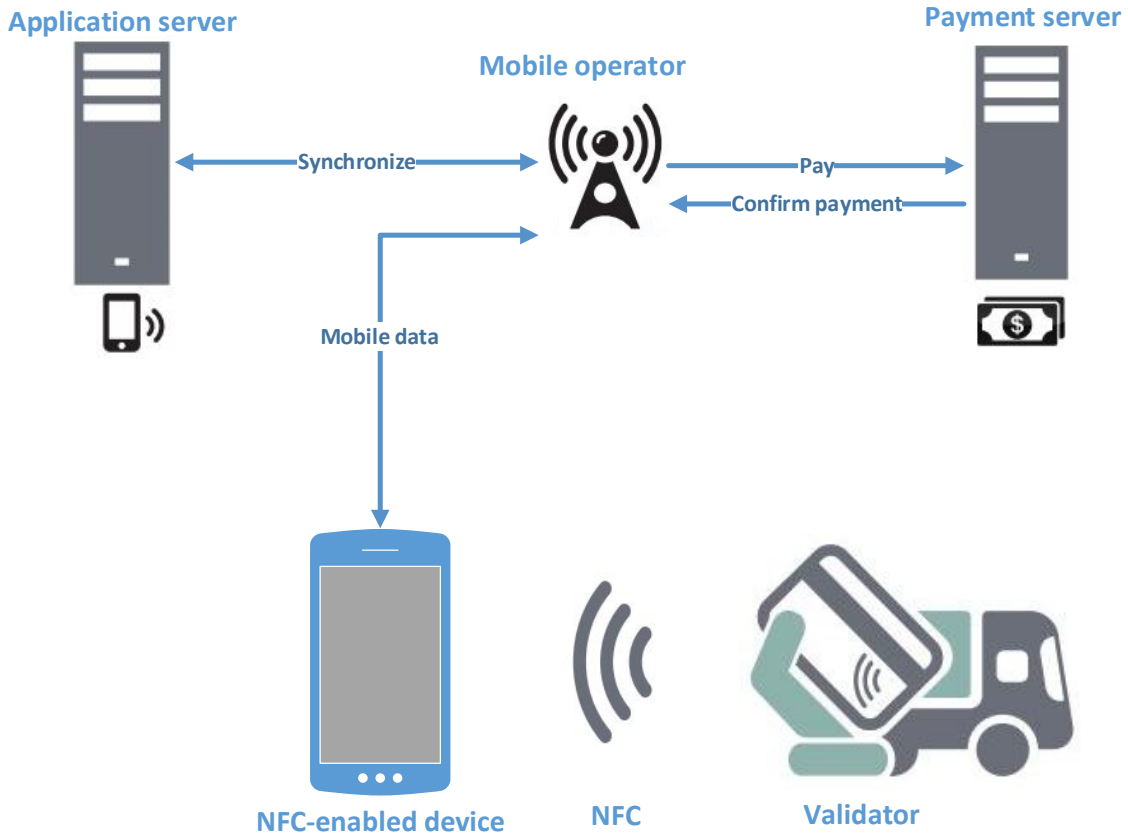



Figure 5.1 System overview

The ticketing application is installed on the NFC-enabled device and it communicates with the application server and the payment server through an active Internet connection (mobile data). The device also communicates with the validator (tag) from the bus/tram/trolleybus, through Near Field Communication (discussed in chapters 3 and 4). When buying tickets or a monthly pass, or when validating a ticket using the application installed on the NFC-enabled mobile device, the result of each of these operations is stored in the database, on the application server.

Furthermore, the tag writing application, also installed on the NFC-enabled device, needs to communicate with the application server to retrieve the data that will be

written to the tag, and this commnunication is performed in the same manner, through an active Internet connection. The tag writing application will write the tag using NFC.

As a conclusion, we can establish that our system requires NFC-enabled devices with an active Internet connection, NFC configurable tags and an application server to persist our data. The payment service will be provided by a third party.

## 5.2. System architecture

For the needs of our application, a classic client-server architecture will suffice, without adding any overhead. The major components of the system implementing this architecture are shown in figure 5.2.

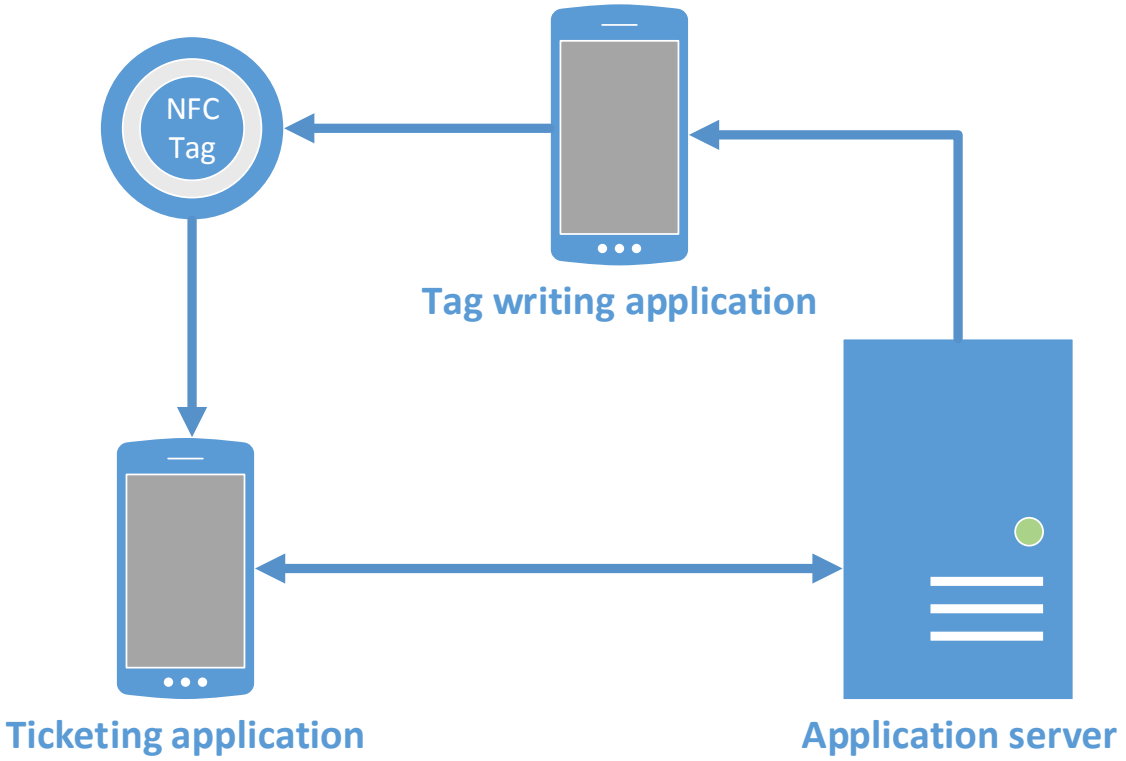



Figure 5.2 System architecture

From this figure, we can observe this is a modular architecture which represents a strong point for reusability and maintainability. The major downfall of this architecture is its single point of failure: the application server. If the application server is offline, then the whole system is impaired, but yet this is a classic problem in the client-server architecture and can be resolved by adding a backup server. The backup server will kick in when the main server is down.

An issue that is generated by the solution to the first problem is that we need to keep the data from the two servers synchronized. This is a subject that we will not discuss, since it is not in the scope of our project.

The next major challenge we need to consider is the performance of the system. The application server represents the bottleneck of the system, because asumming multiple concurrent users access the server simultaneously, this will be reflected in the performance of the system.

The solution to the single point of failure problem can help us also to improve the performance, because by having multiple server we can use a load balancer to share the requests to the servers, thus minimizing the stress on each server.

We will shortly describe how each individual component communicates with the others, and after that, we will see the implementation details of each of these components.

We begin by showing how the tag writing application communicates with the application server and the tag.

The purpose of the tag writing application is to configure the NFC tag with the needed data from the server. The communication with the server is done via an Internet connection (mobile data or WiFi), and the communication with the tag is done via NFC (see chapter 3).

The purpose of the ticketing application is to provide the the traveler with tickets and monthly passes in an electronic format, along with the ability to use these tickets in an intuitive way. The device on which the ticketing application is installed needs to be tapped against the tag in order to validate a ticket. When this tap occurs, the data from the tag, containing information about the bus the user is currently traveling on, is transferred to the NFC-enabled device via NFC, which in turn sends this information to the application server for further processing. The communication of the ticketing application to the application server is also performed via Internet.

To conclude, the ticketing application uses the tag for read operations only, while the tag writing application uses the tag for write operations only. The tag writing application only retrieves data from the application server, while the ticketing application performs all CRUD operation on the database located on the application server.

### *5.2.1. Application server*

The application server exposes a web service implemented on the .NET Framework. The web service is implemented with Windows Communication Foundation (WCF), discussed in chapter 4. In order for the web service to be online and functional it needs to be hosted, which in our project is done via IIS Express.

The architecture of the application server is a 3-tier architecture, as shown in figure 5.3.

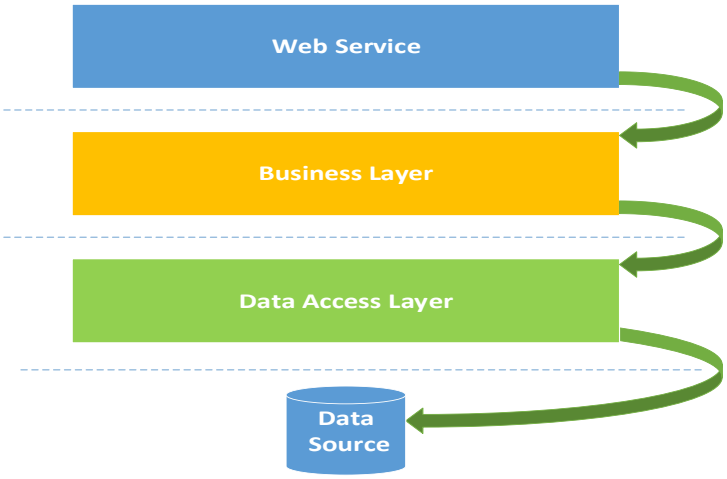



Figure 5.3 Application server architecture

For a better understanding of the application server, we will discuss all three tiers separately, but we will start with the detailed description of the Data Source.

### *5.2.1.1. Data Source*

Our system is database-centric, hence the modeling of the database is crucial to our application. We need to model the database such that we can encapsulate change and provide maintainability. Furthermore, it is important to add new functionalities to our system with ease, thus we need to take this into account when designing the database. The resulting database is depicted in figure 5.4.

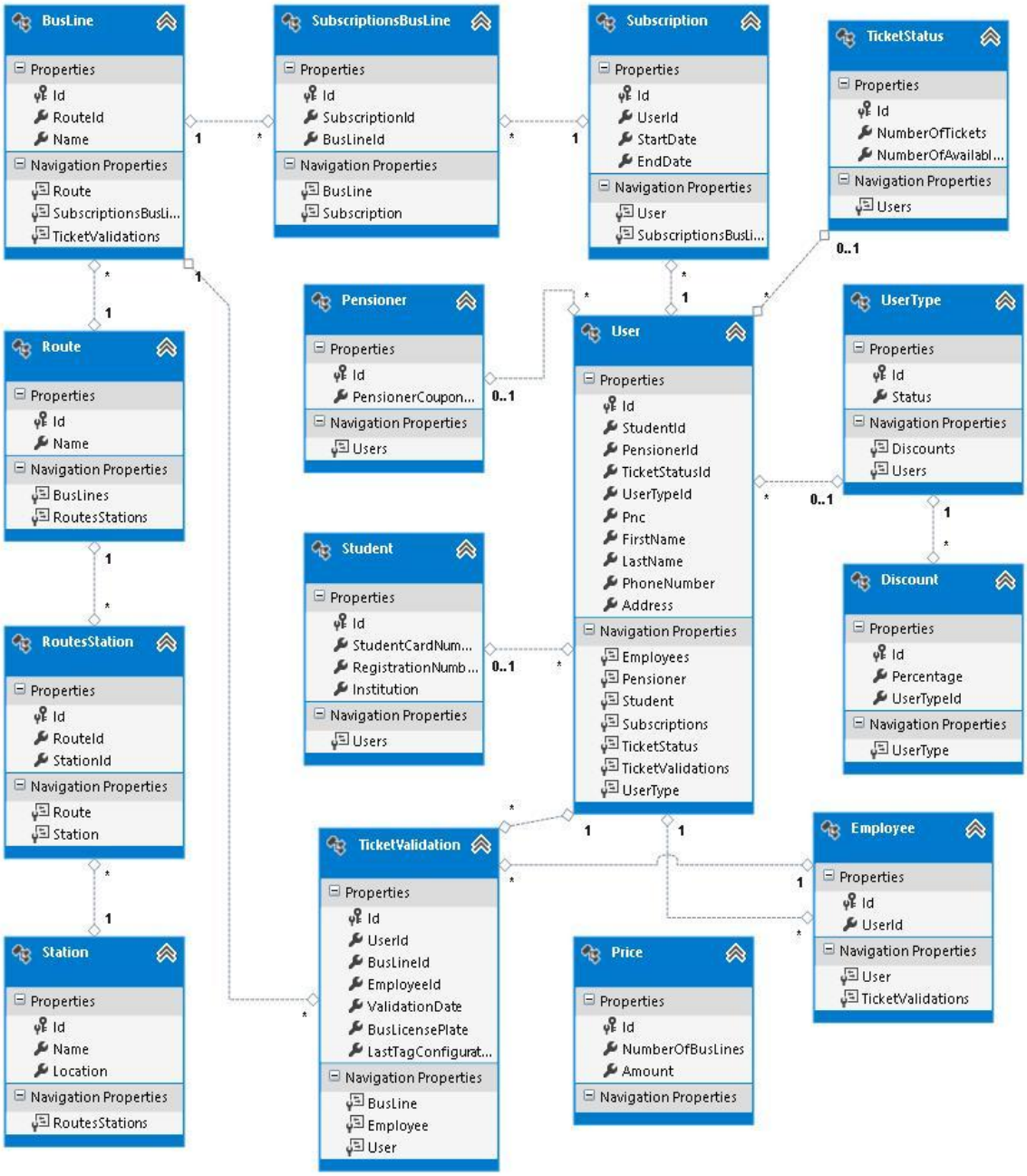



Figure 5.4 Database model

Our application should be able to allow authentication and registration, requirement which lead to creating a User table to persist that information.

Because we want our application to allow differential pricing, which means to have different prices for different types of users, we created a new table UserType and also added two new tables, Student and Pensioner for the first two identified user types, to hold the data which differentiates the users.

In order to have a many to many relationship between Subscription and BusLine (a subscription can have multiple bus lines and the same bus line can appear on multiple subscriptions), we added an intermediary table, SubscriptionsBusLine. We also added an intermediary table to implement the many-to-many relationship between Route and Station.

We keep record of the users that are also employees, such that we can retrieve the employees' information when needed, i.e. when a tag is configured.

The TicketStatus table allows the user to view certain statistics about the purchased tickets and the available tickets.

The TicketValidation, Price and Discount tables are self-explanatory.

#### 5.2.1.2. Data Access Layer

The purpose of this layer is to ensure and implement the connection to the database. When bulding this layer, we had in mind the following: avoid duplicated code, decrease potential for programming errors and have a strong typing of the business data.

To achieve this, we used a repository to separate the logic that retrieves the data and maps it to the entity model from the business logic that acts on the model. The business logic should be agnostic to the type of data that comprises the data source layer.

The repository mediates between the data source layer and the business layers of the application. It queries the data source for the data, and persists changes in the business entity to the data source. A repository separates the business logic from the interactions with the underlying data source. The separation between the data and business tiers has the following benefits:

- It centralizes the data logic.
- It provides a flexible architecture that can be adapted as the overall design of the application evolves.
- Improves the code's maintainability and readability by separating business logic from data or service access logic.
- Uses business entities that are strongly typed so that we can identify problems at compile time instead of at run time.

The Repository Pattern has been implemented using Entity Framework, presented earlier in chapter 4. Entity Framework allows operations on generic types, hence the implementation of a generic Repository Pattern is natural and brings along reuse, extensability, maintainability and avoids duplicate code.

From the generic repository, several repositories were derived, which perform specific functionalities on the database. Figure 5.5 shows the class diagram of the Data Access Layer.

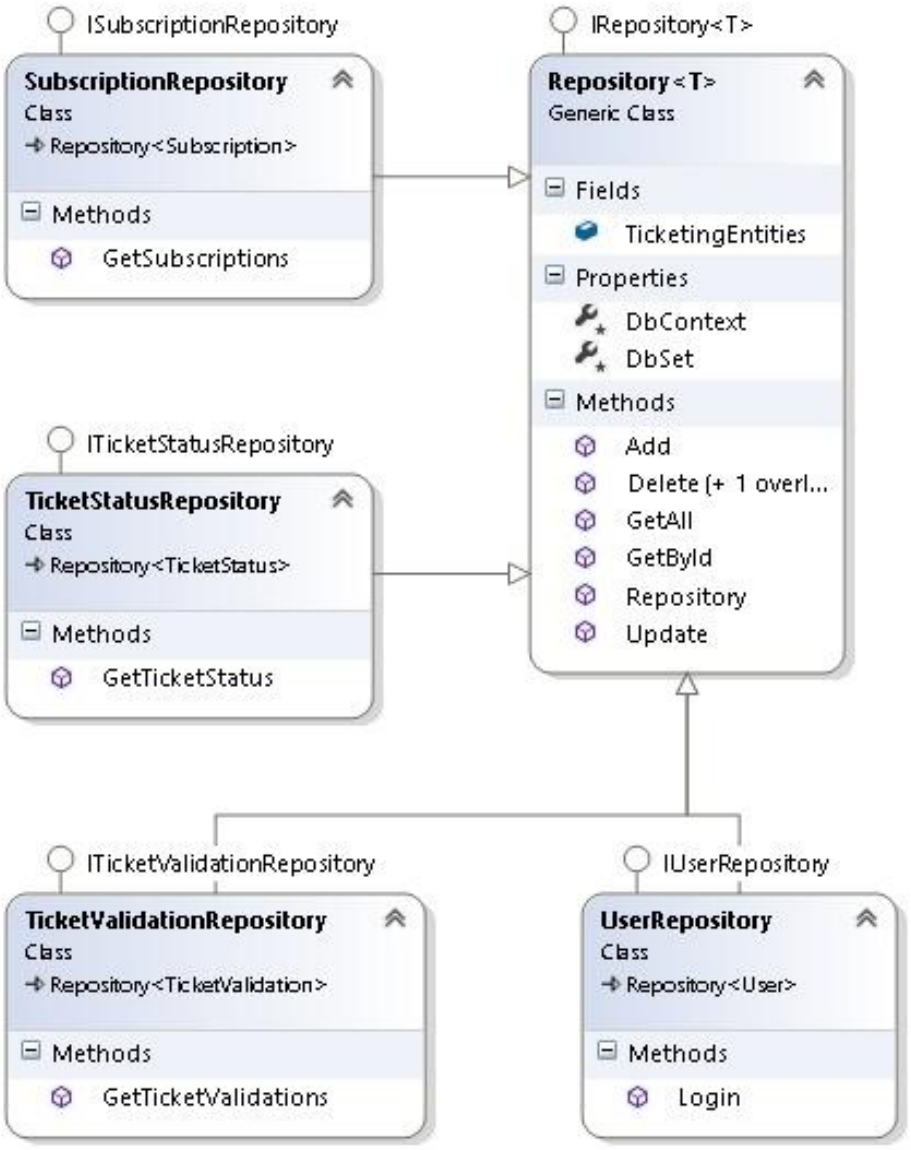



Figure 5.5 Data Layer class diagram

### *5.2.1.3. Business Layer*

The purpose of this layer is to encapsulate the logic of the application. This layer should be totally decoupled from any other layer such that it can be reused in other applications. This results in a cost efficient implementation. We accomplished this by totally encapsulating the Business Layer. The approach to do this was to create a set of Data Transfer Objects, which contain the necessary information needed by the upper layer.

By using AutoMapper (presented in chapter 4), we created a component in this layer that handles the mapping between the Data Transfer Objects and the entities objects returned by Entity Framework. This component should not impact memory performance, nor CPU performance (it has no functional value), hence it was implemented as a static component.

The Business Layer needs to access the Data Access Layer in order to retrieve information from the Data Source. By looking at the current implementation of the Data Access Layer, we can infer that a generic DataDervice is in order. This approach allows us to reduce duplicate code in this layer, and makes this component flexible, maintainable and reusable.

Resembling the Repository implemented in the Data Access Layer, the DataService is built in a similar manner. The differences are that the Business Layer has as data source the Data Access Layer and that it works with Data Transfer Objects instead of entity objects.

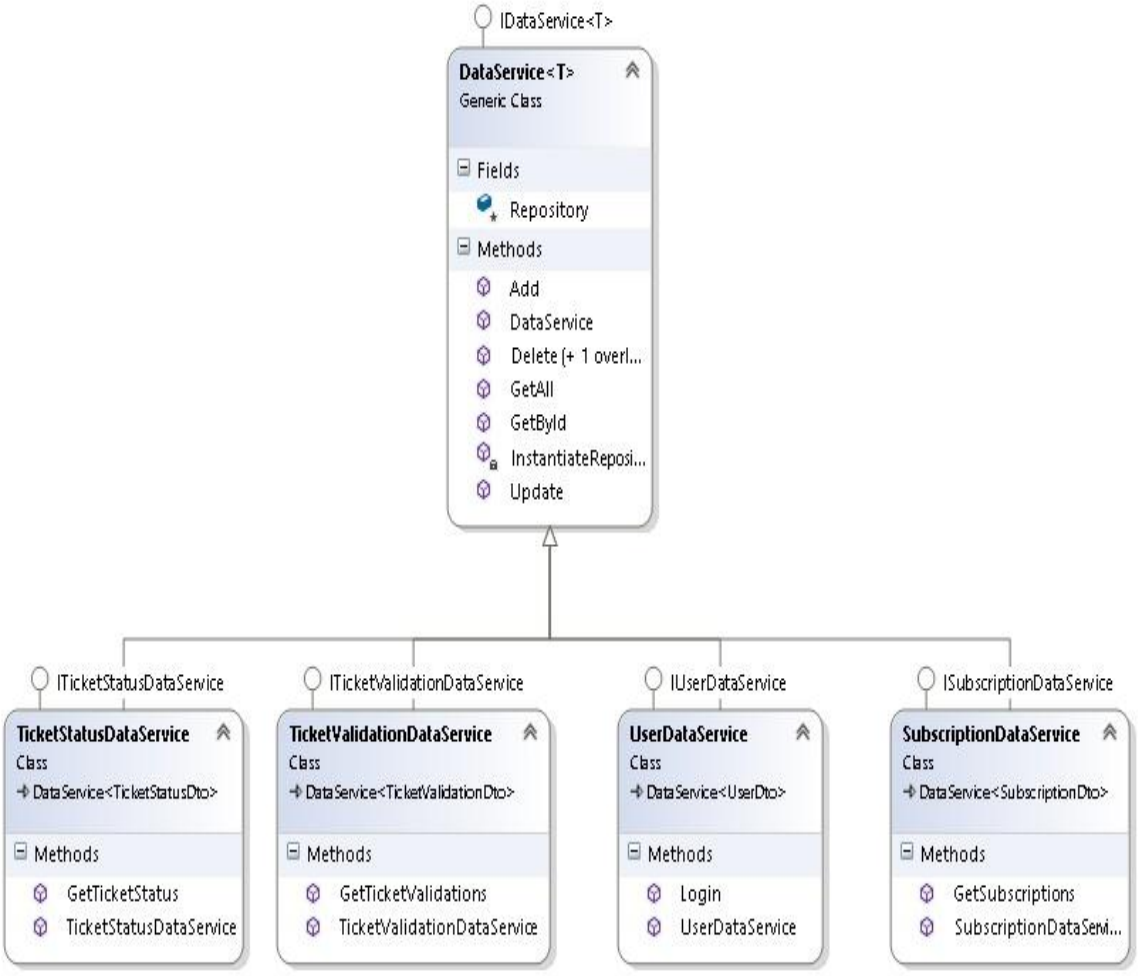



Figure 5.6 Business Layer class diagram

An aspect worth mentioning here, is the use of the dynamic type provided by the .NET Framework The dynamic type enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time. Type dynamic behaves like type object in most circumstances. However, operations that contain expressions of type dynamic are not resolved or type checked by the compiler. Variables of type dynamic are compiled into variables of type object. Therefore, type dynamic exists only at compile time, not at run time.

This capability was needed for the following reason: because the DataService is a generic class and all of its methods revolve around the generic parameter T, at compile

time we do not know the type of T. This implies we do not know how to instantiate our repository, which is done by providing an actual type. To solve this, we declare the Repository as a dynamic type. If we were to declare it as type object, VisualStudio's IntelliSense would alert us that the object does not have the specified methods. Instead, if we declare the Repository as dynamic we avoid this checking. The Repository is instantiated in the constructor of the DataService, when the type of T can be found.

```
public DataService()
{
            InstantiateRepository();
}

private void InstantiateRepository()
{
   MemberInfo info = typeof(T);
   var customAttribute = info.GetCustomAttributes(typeof(CustomAttribute),
                            true).First() as CustomAttribute;
   Type entityType = Type.GetType(string.Format("{0}, {1}",
                                  customAttribute.EntityType,
                                  StringResources.DataAccessLayerAssemblyName));
   Type classType = typeof(Repository<>);
   Type[] typeArguments = new[] { entityType };
   Type constructedType = classType.MakeGenericType(typeArguments);
   Repository = Activator.CreateInstance(constructedType);
}
```

The way we managed to instantiate the Repository (shown in the above snippet code) with the appropriate type is the following:

1. We decorated each Data Transfer Object with a CustomAttribute previously implemented, which basically specifies the type of the respective class as a string.
2. We retrieved the MemberInfo object of the type of T (remember, T is known when the method is executed), which discovers the attributes of a particular member and provides access to member metadata.
3. We retrieved our CustomAttribute from the MemberInfo.
4. We found the type of the T entity by using the string representation of the type and the name of the assembly where this type is declared.
5. We retrieve the type of the Repository<> class.
6. We create an array of types, to which we add the type of the entity found in step 4.
7. We call the MakeGenericType method of the Type class on the type retrieved in step 5, which substitutes the elements of the array of types for the type parameters of the current generic type definition and returns a Type object representing the resulting constructed type.
8. We finally create the Repository by calling the CreateInstance method of the Activator class, which creates an instance of the specified type using the constructor that best matches the specified parameters.

An important difference between the Data Access Layer and the Business Layer is that the Business Layer needs to be able to accommodate any new business rule. To do

so, we need to just simply derive from the generic Data Service. This can be observed in figure 5.6.

Another aspect we needed to consider was the use of enum types. In our application, we are currently using two enums: one for the UserType and the other one for the Price. To ensure maintanability and to respect the open-closed principle, we needed to find a way that when new types of users were added to the database, we did not have the need to return to the code and add another user type in the UserType enum. Fortunately, the .NET Framework offers a great way of doing this through T4 Templates.

T4 stands for Text Template Transformation Toolkit. T4 produces dynamically generated text, using text-based template files. A T4 template is a mixture of text blocks and control logic that can be written in C# or Visual Basic. T4 transforms this templates into executable code and then executes that code to produce the final output. The produced text can be of any type: standard text, XML, HTML, C# etc. The most common use of T4 templates is document and code generation.

There are two types of t4 templates:

- Design time t4 templates
- Run time t4 templates

Design time templates are executed by Visual Studio at design time and are used to generate code. They have the advantage of producing text that can be consumed by the project, such as classes, enums etc.

Run time templates are executed during the application's execution. Run time templates are preprocessed templates, meaning that they are turned into classes and compiled into the assembly. Because of this, run time templates can be instantiated and executed via code, which means that parameters can be passed to them and used by control logic. Run time templates can be executed outside of VisualStudio and do not require VisualStudio to be installed on the client.

There are a lot of benefits for code generation. Code takes time to write and maintain. Properly designed templates can significantly reduce development time by writing code dynamically on demand and can update code automatically when their external dependencies change. Code that you generate is code that you do not have to write.Maintenance of generated code only has to be done in a single location: the template that generated it. Bug fixes can instantly be propagated by executing the template and regenerating the code. Templates are reusable and it uses the language we're already familiar with.

For the needs of our application, we decided to use design time templates for generating the code of our enums, due to the fact that the enums are used along the application, so they are needed at compile time.

The way we wrote the UserType enum using a design time t4 template is shown in the following code snippet:

```
using System;
using System.Runtime.Serialization;

namespace BusinessLayer.Enums
{
    [DataContract]
    public enum UserTypeEnum
    {
```

```
<#
    using (SqlConnection conn = new SqlConnection(connectionString)){
        string command = "select Id, Status from UserType";
        SqlCommand comm = new SqlCommand(command, conn);

        conn.Open();

        SqlDataReader reader = comm.ExecuteReader();
        while(reader.Read())
        {
            int ordinal= reader.GetOrdinal("Id");
            int Id = reader.GetInt32(ordinal);
            ordinal= reader.GetOrdinal("Status");
            string Status= reader.GetString(ordinal);
#>
        [EnumMember]
        <#=Status #> =  <#=Id #>,
<#}
    }
#>    }
      }
```

The code resembles JavaServerPages in the sense that JSP uses <% ... %> to encapsulate Java statements and <%= … %> to evaluate a Java expression and display its result. The syntax is similar, due to the fact that in t4 templates we use <#=…#> to enclose a C# expression and display its result, and we enclose C# statements between <# and #>. The text outside any angular quote brackets (< >) is simply printed in the result (it is not executed).

### *5.2.1.4. Web Service*

The Web Service's role is to expose the functionality from the Business Layer to the other components of the system. When implementing the functionality of the Web Service, we need to be careful to not include any business logic, because we might end up compromising the maintainability of the entire system, i.e. it will be hard to replace the Web Service with other components, such as a Web API. The Web Service was implemented using Windows Communication Foundation (WCF – see chapter 4). This implies the declaration of the service contract and the data contract, which will be used in generating the Web Service proxies.

The layer represented by the Web Service communicates directly with the business layer and establishes a composition relationship with a series of wrappers that encapsulate the Data Service and holds business logic. These wrappers are the place where the business rules meet the information retrieved from the Data Service, and can be easily manipulated. In figure 5.7 we emphasize the relationship between the Web Service and the Business Layer. We can see that the web service accesses only the Business Layer classes and does not perform any logical operations, just method calls.

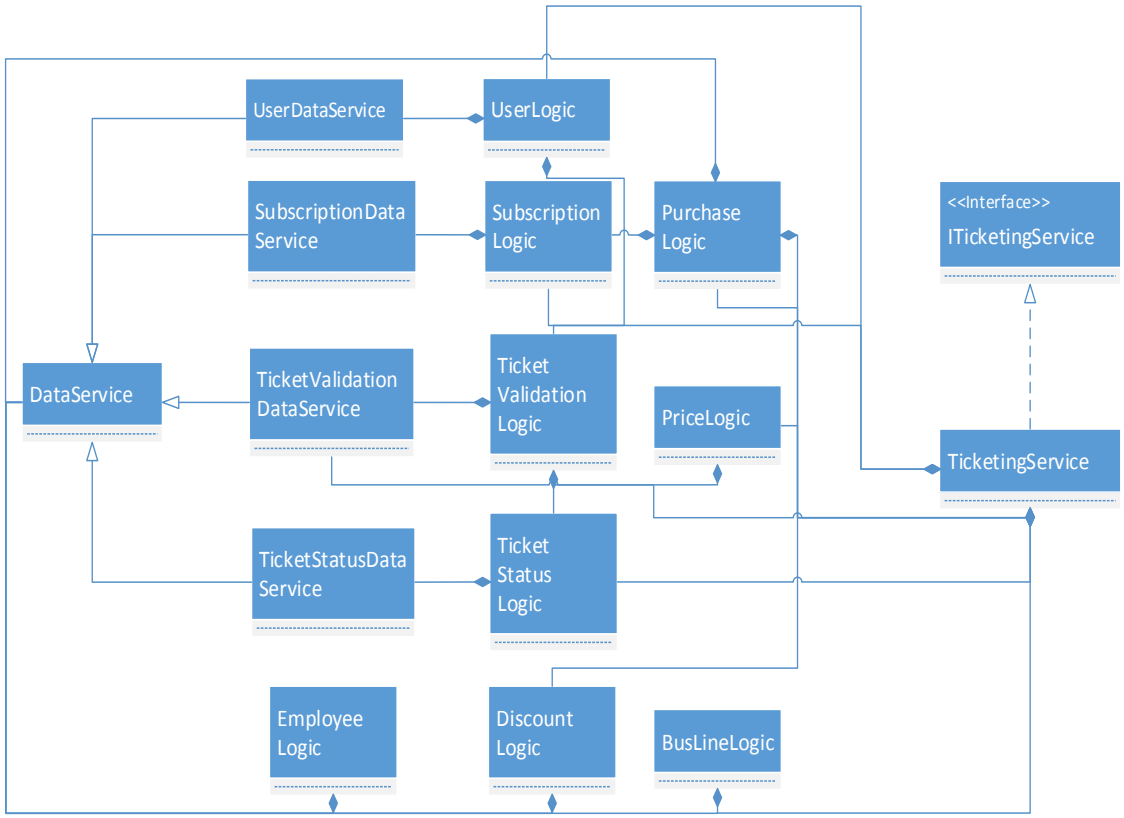



Figure 5.7 Web Service's relationship with the Business Layer

## *5.2.2. Ticketing application*

The purpose of this component is to offer the traveler all the functionality he/she requires, meaning the ability to buy tickets and monthly passes, to validate tickets, to consult the application for bus related information (bus lines) and to display an overview of his/her validations and subscriptions.

The architecture of this component is based on the Model-View-ViewModel architectural pattern. This is a best practice recommended by Microsoft for building Windows Phone 8 applications (but also Windows Store and WPF applications).

The Model-View-ViewModel (MVVM) pattern helps to cleanly separate the business and presentation logic of the application from its user interface (UI). Maintaining a clean separation between application logic and UI helps to address numerous development and design issues and can make the application much easier to test, maintain, and evolve. It also improves greatly code reuse opportunities and allows developers and UI designers to collaborate more easily when developing their respective parts of the application. The diagram in figure 5.8 shows the basic architecture of an application built using the MVVM design pattern.

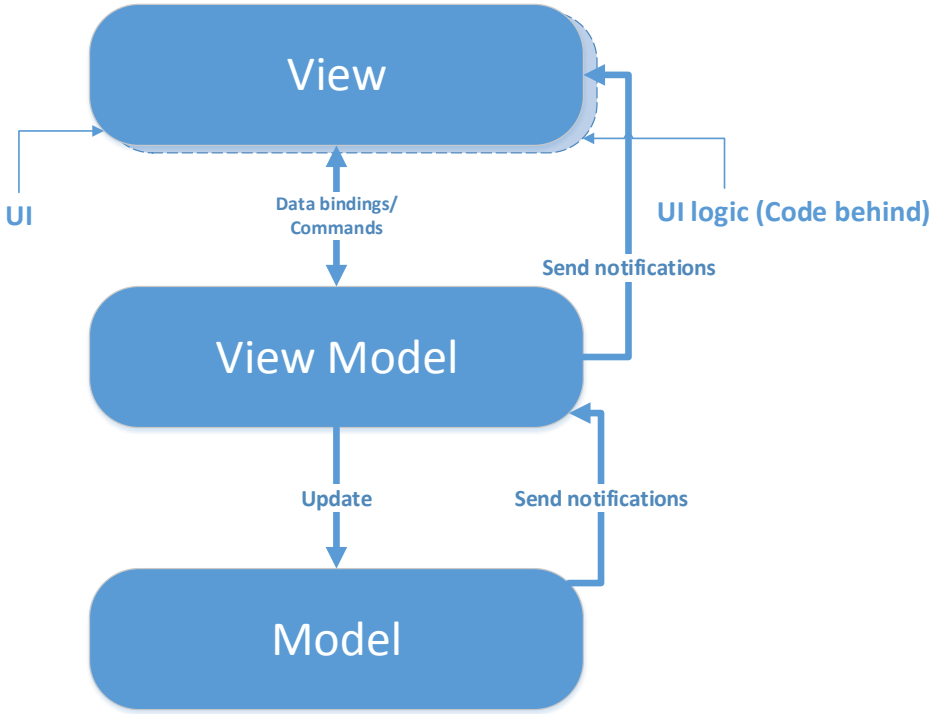



Figure 5.8 Basic architecture built using MVVM

Using the MVVM pattern, the UI of the application and the underlying presentation and business logic is separated into three separate classes:

- the view, which encapsulates the UI and UI logic;
- the view model, which encapsulates presentation logic and state;
- the model, which encapsulates the application's business logic and data.

In a nutshell, this means that MVVM is a way to separate the data from the UI. In the MVVM design pattern, developers can code application logic, and designers can create the UI. XAML is designed to make it easy to build apps using MVVM, and they complement each other in a way that makes separation of UI and application logic a natural thing to do.

**The View Class**

The view's responsibility is to define the structure and appearance of what the user sees on the screen. Ideally, the code-behind of a view contains only a constructor that calls the InitializeComponent method. In some cases, the code-behind may contain UI logic code that implements visual behavior that is difficult or inefficient to express in Extensible Application Markup Language (XAML), such as complex animations, or when the code needs to directly manipulate visual elements that are part of the view.

In MVVM, the view's data context is set to the view model. The view model implements properties and commands to which the view can bind and notifies the view of any changes in state through change notification events. There is typically a one-to-one relationship between a view and its view model.

Typically, views are Control-derived or UserControl-derived classes. However, in some cases, the view may be represented by a data template, which specifies the UI

elements to be used to visually represent an object when it is displayed. Using data templates, a visual designer can easily define how a view model will be rendered or can modify its default visual representation without changing the underlying object itself or the behavior of the control that is used to display it (see chapter 4).

Data templates can be thought of as views that do not have any code-behind. They are designed to bind to a specific view model type whenever one is required to be displayed in the UI. At run time, the view, as defined by the data template, will be automatically instantiated and its data context set to the corresponding view model.

To summarize, the view has the following key characteristics:

- The view is a visual element, such as a window, page, user control, or data template. The view defines the controls contained in the view and their visual layout and styling.
- The view references the view model through its DataContext property. The controls in the view are data bound to the properties and commands exposed by the view model.
- The view may customize the data binding behavior between the view and the view model. For example, the view may use value converters to format the data to be displayed in the UI, or it may use validation rules to provide additional input data validation to the user.
- The view defines and handles UI visual behavior, such as animations or transitions that may be triggered from a state change in the view model or via the user's interaction with the UI.
- The view's code-behind may define UI logic to implement visual behavior that is difficult to express in XAML or that requires direct references to the specific UI controls defined in the view.

### The View Model Class

The view model in the MVVM pattern encapsulates the presentation logic and data for the view. It has no direct reference to the view or any knowledge about the view's specific implementation or type. The view model implements properties and commands to which the view can data bind and notifies the view of any state changes through change notification events. The properties and commands that the view model provides define the functionality to be offered by the UI, but the view determines how that functionality is to be rendered.

The view model is responsible for coordinating the view's interaction with any model classes that are required. Typically, there is a one-to many-relationship between the view model and the model classes. The view model may choose to expose model classes directly to the view so that controls in the view can data bind directly to them.

The view model may convert or manipulate model data so that it can be easily consumed by the view. The view model may define additional properties to specifically support the view; these properties would not normally be part of (or cannot be added to) the model. For example, the view model may combine the value of two fields to make it easier for the view to present, or it may calculate the number of characters remaining for input for fields with a maximum length. The view model may also implement data validation logic to ensure data consistency.

The view model may also define logical states the view can use to provide visual changes in the UI. The view may define layout or styling changes that reflect the state of the view model. For example, the view model may define a state that indicates that data is being submitted asynchronously to a web service. The view can display an animation during this state to provide visual feedback to the user.

Typically, the view model will define commands or actions that can be represented in the UI and that the user can invoke. The view may choose to represent that command with a button so that the user can click the button to submit the data. Typically, when the command becomes unavailable, its associated UI representation becomes disabled. Commands provide a way to encapsulate user actions and to cleanly separate them from their visual representation in the UI.

To summarize, the view model has the following key characteristics:

- The view model is a non-visual class that encapsulates the presentation logic required to support a use case or user task in the application. The view model is testable independently of the view and the model.
- The view model typically does not directly reference the view. It implements properties and commands to which the view can data bind. It notifies the view of any state changes via change notification events (via the INotifyPropertyChanged and INotifyCollectionChanged interfaces).
- The view model coordinates the view's interaction with the model. It may convert or manipulate data so that it can be easily consumed by the view and may implement additional properties that may not be present on the model. It may also implement data validation via the IDataErrorInfo or INotifyDataErrorInfo interfaces.
- The view model may define logical states that the view can represent visually to the user.

**The Model Class**

The model in the MVVM pattern encapsulates business logic and data. Business logic is defined as any application logic that is concerned with the retrieval and management of application data and for making sure that any business rules that ensure data consistency and validity are imposed. To maximize re-use opportunities, models should not contain any use case–specific or user task–specific behavior or application logic.

Typically, the model represents the client-side domain model for the application. It can define data structures based on the application's data model and any supporting business and validation logic. The model may also include the code to support data access and caching, though typically a separate data repository or service is employed for this. Often, the model and data access layer are generated as part of a data access or service strategy, such as the ADO.NET Entity Framework, WCF Data Services.

Typically, the model implements the facilities that make it easy to bind to the view. This usually means it supports property and collection changed notification through the INotifyPropertyChanged and INotifyCollectionChanged interfaces. Models classes that represent collections of objects typically derive from the ObservableCollection<T> class, which provides an implementation of the INotifyCollectionChanged interface.

The model may also support data validation and error reporting through the IDataErrorInfo (or INotifyDataErrorInfo) interfaces. The IDataErrorInfo and INotifyDataErrorInfo interfaces allow WPF data binding to be notified when values change so that the UI can be updated. They also enable support for data validation and error reporting in the UI layer.

The model has the following key characteristics:

- Model classes are non-visual classes that encapsulate the application's data and business logic. They are responsible for managing the application's data and for ensuring its consistency and validity by encapsulating the required business rules and data validation logic.
- The model classes do not directly reference the view or view model classes and have no dependency on how they are implemented.
- The model classes typically provide property and collection change notification events through the INotifyPropertyChanged and INotifyCollectionChanged interfaces. This allows them to be easily data bound in the view. Model classes that represent collections of objects typically derive from the ObservableCollection<T> class.
- The model classes typically provide data validation and error reporting through either the IDataErrorInfo or INotifyDataErrorInfo interfaces.
- The model classes are typically used in conjunction with a service or repository that encapsulates data access and caching.

Having a clear understanding of the MVVM pattern, the way the ticketing application was implemented becomes more clear and easy to understand.

In what follows, we will detail the way MVVM was used in our application, by presenting the ViewModels and their connection to the Views and Models, as well as the motivation behind this implementation.

When starting the application, the first page we come in contact with is the LoginPage. This page has a password box where the user needs to enter his/her Personal Numerical Code (PNC) in order to authenticate into the application.

If the user comes in contact with the application for the first time, he/she has the option to register, by pressing the “register” button from the application bar, which will redirect the user to the RegistrationPage.

The class diagram in figure 5.9 shows the interaction between the LoginViewModel, the views associated with it (LoginPage and RegistrationPage) and the Models it uses.

The LoginViewModel inherits the BindableBase abstract class, which is an implementation of the INotifyPropertyChanged interface. This is needed in order for the ViewModel to notify the View when changes occur.

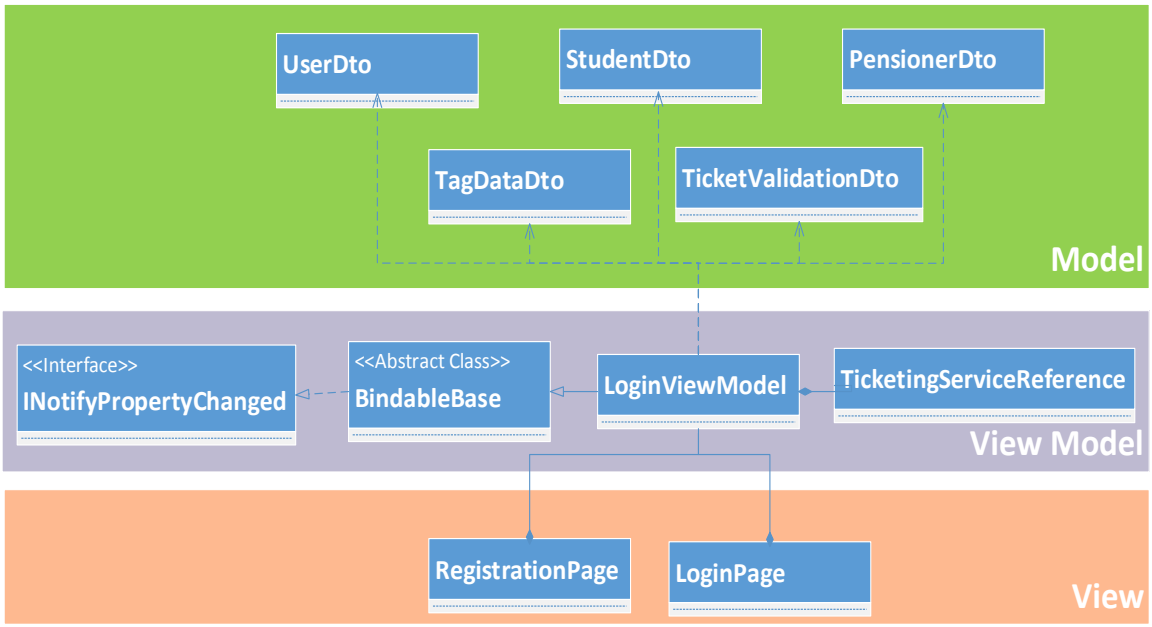



Figure 5.9 LoginViewModel interactions

What is worth mentioning in the LoginViewModel (and also the other View Models), is the fact that it uses commands and bindings to have a clear separation of the UI from the business logic. This is done by data binding and the use of the ICommand interface. The ICommand interface was implemented by the Command class, and the command is binded in the XAML markup language to specify the method a certain UI action should trigger. The LoginViewModel gets the information it displays to the View by making asynchronous calls to the Web Service, through the TicketingServiceReference.

The other View Model in our solution, is the MainViewModel. This View Model is common for the rest of the Views, due to the fact they all require the same or related information. In figure 5.10 are presented the interactions between the MainViewModel, the views associated with it, the involved Models and third party libraries.

The MainViewModel inherits the BindableBase abstract class, which implements the INotifyPropertyChanged. The MainViewModel has a TicketingServiceReference through which it makes calls to the Web Service. Each of the Views (MainPage, SubscriptionDetailsPage, ValidationDetailsPage, AllBusLinesPage, BuyTicketsPage and BuySubscriptionPage) contains a MainViewModel, which is actually the same instance, due to the fact that we created a static ViewModelLocator class which returns the same View Model when the getter is called.

The same principle of data binding and commands is applied throughout these pages. What is worth mentioning is the BuyNow class from the PayPal Checkout SDK (see chapter 4).

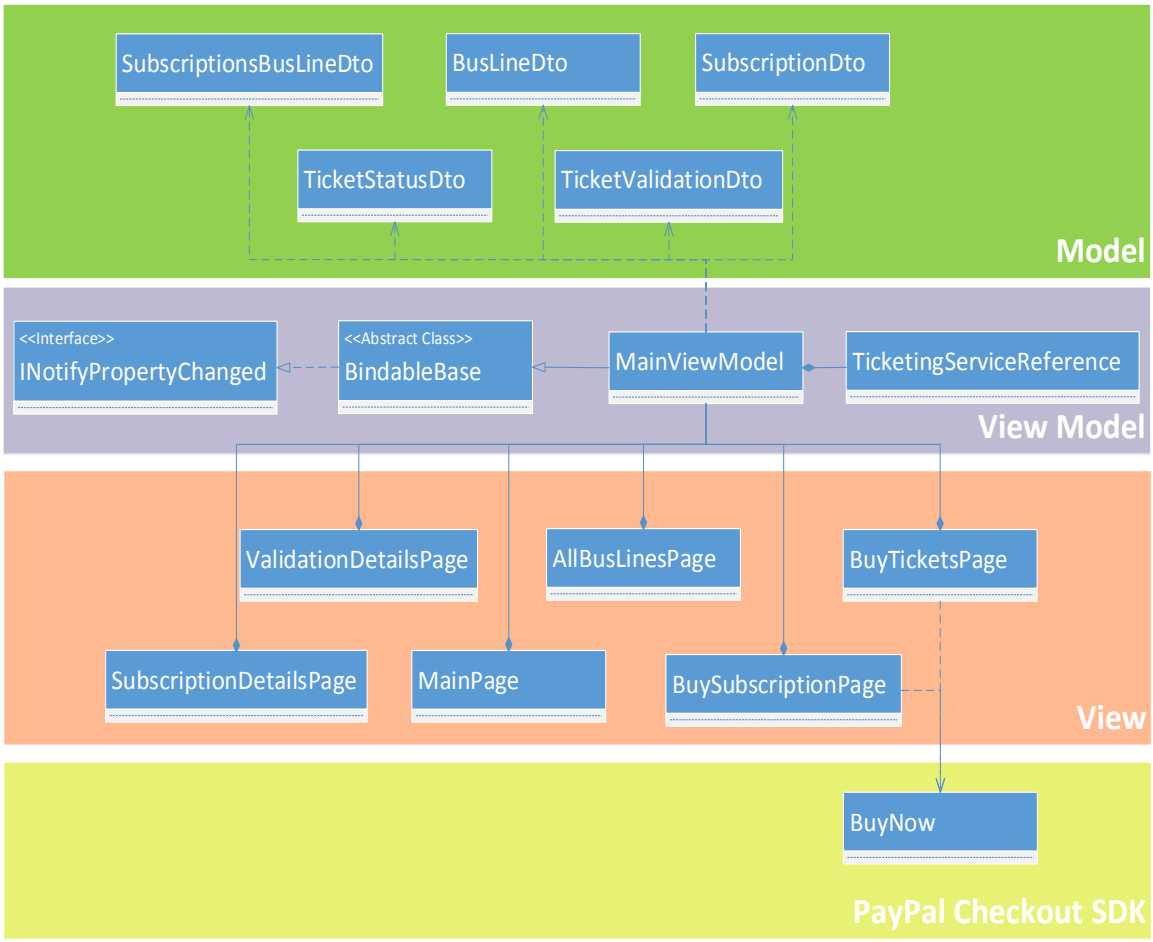



Figure 5.10 MainViewModel interactions

The PayPal Checkout SDK exposes the BuyNow class, which contains some properties like the MerchantId, UseSandbox (if this is set to true, the test environment is used), PaymentMethod etc., and some event handlers to handle the following events:

- AuthEventArgs (triggered when the payment is being authenticated)
- CancelEventArgs (triggered when the payment is canceled)
- CompleteEventArgs (triggered when the payment is complete)
- ErrorEventArgs (triggered when an error occurs while processing the payment)
- StartEventArgs (triggered when the payment is initiated)

The most important method that concerns us is the *IAsyncOperation<bool> Execute()*. This method is executed separately on a different thread asynchronously with respect to the main thread. This is a best practice when dealing with third party services and we do not have a guaranteed response time, or simply the operation is too expensive to execute it synchronously. Without doing this, we would leave the UI unresponsive until the method returns. The return result of the IAsyncOperation is handled via a generic event handler, that returns the needed information. When the method finishes execution, one of the previous events are triggered, based on the execution state of the method.

### Implementing the UI

We will now present some of the points of interest in implementing the UI, based on the design principles recommended by Microsoft (see chapter 4).

We will start by presenting the **MainPage**, which is implemented as a panorama control. As discussed in chapter 4, in the panorama control the content spans horizontally beyond the size of the phone screen. Content panes that each fit on the actual screen are added, and the user can pan from one pane to the next as if the panes were laid out on a long continuous horizontal canvas. The MainPage has three PanoramaItems (content pane): „validations", „subscriptions" and „ticket status".

The first PanoramaItem, the „validations" content pane, contains the list of the users validations, orderered from the most recent one at the top, to the oldest ones at the bottom. They appear as a scrollable vertical list, where each individual item is represented as a live tile (see chapter 4). To achieve this, we used the HubTile control as follows: the container of the validations is a LongListSelecter, which uses as ItemTemplate a DataTemplate defined in a resource file (Templates and DataTemplates are discussed in chapter 4). That DataTemplate, which is applied to each item that will populate the list, contains a CustomHubTile control and it is implemented in the following way:

```
<DataTemplate x:Key="ValidationTemplate">
        <customControls:CustomHubTile
            x:Name="Tile" HorizontalAlignment="Left"
            Message="{Binding ValidationDate, Converter={StaticResource
            DateTimeToStringConverter}}"
            Title="{Binding BusLine.Name}" FontSize="10" />
</DataTemplate>
```

Another important design principle, which is used throughout the implementation of all the pages, is the use of converters (see subchapter 4.1.1, User experience, Data binding). As it can be noticed in the ValidationTemplate above, the DateTimeToStringConverter is used, because there is a type mismatch between the source property and the target property. The source property (ValidationDate) is a DateTime, while the target property (Message) is of type string. This gap was bridged by using a class that implements the IValueConverter in the following way:

```
public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
{
       var date = value as DateTime?;
       if (parameter == null)
       return string.Format("{0} {1}, {2}", new CultureInfo("en-
                    US").DateTimeFormat.GetMonthName(date.Value.Month),
                                       date.Value.Day, date.Value.Year);
            return null;
 }
```

The ConvertBack method was not implemented, because we only need to convert from the source type to the target type.

The need that made us implement a CustomHubTile control (derived from the HubTile), is the randomization of the background picture of each tile. In the original control, each of the tiles needed to be binded to a specific image background. Due to the fact that we did not want to define a separate DataTemplate for each different background picture we wanted to apply, we overrode the ArrangeOverride method of the HubTile class to be able to set a random background picture for each tile in the following way:

```
protected override Size ArrangeOverride(Size finalSize)
{
     if (Source == null)
     {
           Source = new BitmapImage(
                    new Uri(ImagePathForValidationTemplate, UriKind.Relative));
     }
     return base.ArrangeOverride(finalSize);
}
```

The second PanoramaItem displays the user's subscriptions, in a LongListSelector. What is worth mentioning about the implementation of this content pane, is the fact that the items use the same template. The DataTemplate is implemented in such a way that based on the subscription's EndDate, it displays a different visual representation of the subscription, by using converters to convert the EndDate to a visibility property.

The last PanoramaItem, the "ticket status" one, displays the number of tickets the user bought along the time, the number of tickets he/she has remaining, plus a quick overview of the last validation he/she performed (on which date and bus line). What makes this PanoramaItem notable from the implementation point of view, is the use of user controls. The last validation displayed is implemented as a user control. A user control is a customized control that can be reused multiple times in a project. This control inherits from the UserControl class and therefore does not use templates, but has an appeareance designed from scratch.

```
<UserControl
 x:Class="TicketingClient.Views.UserControls.LastValidationUserControl"
       <!--namespaces and imports-->            >
    <UserControl.Resources>
       !--user control resources (converters, styles, templates, etc.)-->
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot">
       <!—define UI appearance-->
    </Grid>
</UserControl>
```

The **BuyTicketsPage** stands out trough its LoopingSelector control. The LoopingSelector control enables users to implement various looping scenarios in an easy way. Basically it is an infinite list selector which contains all the functionality for a looping control. To use this component we need to provide an appropriate data source. We can create a suitable data source by implementing the ILoopingSelectorDataSource interface.

To be able to select the numbers of tickets, some conditions needed to be met: the number of tickets needed to be a non-negative, different from zero number. As the

maximum limit we set 100 and the minimum is obviously one. The two methods in the implementation of the ILoopingSelectorDataSource interface were implemented in the following way:

```
public object GetNext(object relativeTo)
{
      int nextValue = ((int) relativeTo) + 1;
      return nextValue <= Maximum ? nextValue : Minimum;
}

public object GetPrevious(object relativeTo)
{
       int previousValue = ((int) relativeTo) - 1;
       return previousValue >= Minimum ? previousValue : Maximum;
}
```

The **BuySubscriptionPage** is implemented using two DatePicker controls and one LongListMultiSelector control. The tricky part about the date pickers is that only the first DatePicker is enabled. When the user selects the starting date of his/her monthly pass, the second DatePicker will automatically update its value with the current value of the first DatePicker plus one month.

The LongListMultiSelector control allows you to group items inside a list and allowing for multiple item selection. It implements a jump-list style of UI as seen in the address book of the phone. This type of UI is usually used to display long lists of data. The LongListMultiSelector control implements full UI and data virtualization. It is recommended to use the LongListSelector or LongListMultiSelector instead of the ListBox control, whenever we want to display lists of data, even if the data does not need to be grouped.

A notable characteristic of the BuySubscriptionPage is the way the ApplicationBar buttons are interchanged, depending on the performed action. This feature was implemented in code behind, where all buttons are dynamically created, added and removed from the ApplicationBar, according to some conditions.

All the **other pages** are implemented in a similar manner, taking into account the UX experience Windows Phone 8 applications offer by to using built-in controls including layout controls such as panorama and data bound list controls, using styles, templates, and animations to customize the application's look and feel, and the page framework for content and navigation structure.

We applied to all the pages a navigation transition to have a more fluid application, and we designed the application such that it will use the same colors the current theme and accent color the phone applies across all applications and OS.

Also, most of the pages need to display messages to the user and get his/her explicit answer. Such a feature was not offered by the SDK, so we implemented a CustomMessageBox. For example, when the users presses back up until the point he/she will exit the application, that last time he/she presses back, a message box will appear, notifying the user he/she is about to log out, and if he/she wants to proceed. This is done by calling the ShowMessageBox method from our custom message box:

```
var messageBox = new CustomizedMessageBox(AppResources.PleaseConfirmTitle,
                                          AppResources.LogOutMessage,
                                          () =>
```

```
                                        {
                                        NavigationService.GoBack();
                                        NavigationService.RemoveBackEntry(
                                        );
                                    },
                                    null);
messageBox.ShowMessageBox();
```

What this code snippet actually performs is it creates a CustomizedMessageBox and gives as parameters to the constructor the following:

- the title of the message (string)
- the content of the message (string)
- the action to perform if the ok button is pressed (Action)
- the action to perform if the cancel button is pressed (Action)

The action to be executed when the user presses ok (or yes) is specified as a statement lambda. A lambda expression is an anonymous function that we can use to create delegates. By using lambda expressions, we can write local functions that can be passed as arguments or returned as the value of function calls. A lambda expression with an expression on the right side of the "=>" operator is called an expression lambda. A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces "{}".

To optimize the usage of memory in a small amount, resource files were used throughout the application for storing string objects, to avoid instantiating a string with the same value multiple times.

### *5.2.3. Tag writing application*

This application's whole reasons of being is to provide a way to configure tags with the appropriate data. This application is implemented as a Windows Phone 8 application and thus it respects the Model-View-ViewModel architectural pattern and Microsoft's recommended design principles.

This application also uses a TicketingServiceReference to make calls to the WebService. The data required by this application are:

- The list of all bus lines
- The list of all employees

As discussed in chapters 3 and 4, NFC tags content is based on the NDEF message standard (NFC Data Exchange Format). In order to use these capabilities, we needed to use the Windows.Networking.Proximity namespace, referencing the Proximity API. By using this API, the application can establish a connection through a tap within wireless range. The Proximity API encapsulates NFC hardware communication and basic NDEF formatting for a very limited subset of the NDEF standards. This missing part, the support for the NDEF standards on top, is added by the NDEF library (see chapter 4).

After the data needed to configure the tag is retrieved from the view (it is inputed by the user), we save it in a TagDataDto object. This class does not have a corresponding entity in the database. In order to send this data to the NFC tag, we must transform it in a format known by the NFC tag.

For the purpose of our system, we need the ticketing application to be opened when the user taps the NFC-enabled device to an NFC tag. We achieve this by using the NdefLaunchAppRecord class provided by the NDEF library.

The NdefLaunchAppRecord is a special type of record that will open an application, based on the application ID specified in the record, when a NFC connection is established. This class contains several properties, but the property that concerns us is the „Arguments" property. This property is of type string and we will use it to send the data we need to the tag. We do this by serializing the TagDataDto object with the help of a JSON serializer. The TagDataDto class is marked with the DataContract attribute and all the members that we want to take part in the serialization are marked with the DataMember attribute. The following code snippet shows the creation and pusblishing of an NdefLaunchAppRecord:

```
var record = new NdefLaunchAppRecord {Arguments = (DataContext as
                                        MainViewModel).CreateRecordArguments()};
// add the platform identifier and the ID of the application we want to launch
record.AddPlatformAppId(AppResources.Platform, AppResources.TicketingAppId);
// Publish the record using the proximity device
PublishRecord(record);
```

In figure 5.11, the interactions between the classes of the tag writing application can be observed.

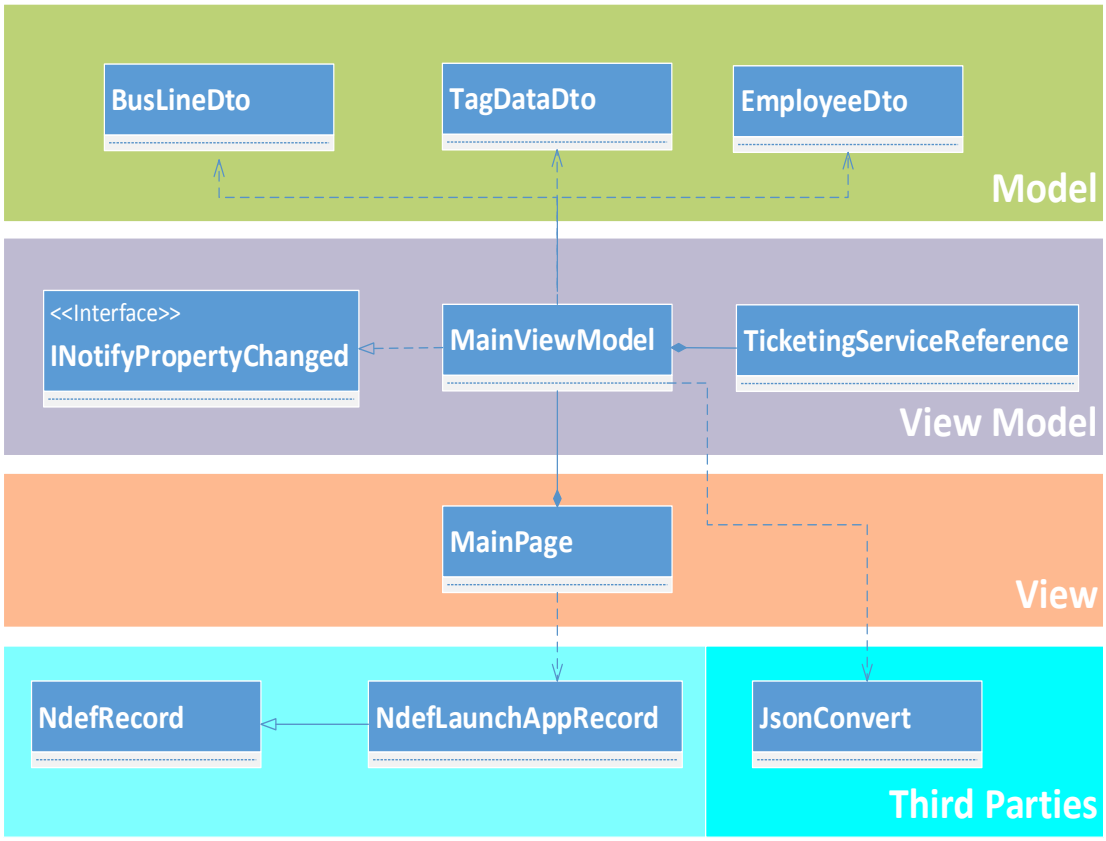



Figure 5.11 Tag writing application class diagram

# Chapter 6. Testing and Validation

The purpose of this chapter is to offer an overview of the test suite. For our project, a test suite has been built for the most important functionalities of our ticketing application. A test suite is composed of multiple test cases. A test case provides the functional knowledge for an application and contains the preconditions of the test case, i.e. in what state should the system be in such that the desired functionality is available, all the necessary steps to be performed in order for that functionality to reach its goal, an expected result for each step of the test case, and finally, the expected result of the test case, i.e. the goal of the test case.

It is important to start developing test cases and test suite early in the application life cycle because it increases maintainability, is cost efficient and we can create Smoke and Regression testing based on which we can start automating the testing process.

Smoke testing is a set of basic cheap to run tests that precede actual testing. It aims to verify that the build is deployed successfully and that all test environment aspects are running and ready for the actual test process. It saves you bringing the full extent of your testing wrath down a faulty build and just realizing that you have been testing on a bad environment or erroneously deployed build possibly too late.

During a regression test, we run through the application testing features that were known to work in the previous build. We look specifically for parts of the application that may not have been directly modified, but depend on (and could have residual bugs from) code that was modified. Those bugs (ones caused by bugs in dependent code, even though they were working before) are known as regressions (because the feature was working properly and now has a bug and therefore, regressed).

In what follows we will present the most important test cases for the application.

**Register Test Case**
**Preconditions: W**eb Service is online

| Step | Expected Result |
|---|---|
| Start application | Login page is displayed |
| Go to Register page | Register page is displayed |
| Fill the form with personal data | The watermark is replaced by the inputed information |
| Save the user | A pop-up appears confirming the successful registration of the user |
| Press ok on pop-up message | Login page is displayed |

**Login Test Case**
**Preconditions:** User is registered

| Step | Expected Result |
|---|---|
| Enter wrong credentials | The credentials written in the field are masked. |
| Press Sign In | Warning message is displayed: 'Wrong PNC or inexistent user' |
| Enter valid credentials | The credentials written in the field are masked. |
| Press Sign In | The main page is displayed |

**View Account Details Test Case**
**Preconditions:** User is logged in, main page is displayed

| Step | Expected Result |
|---|---|
| Slide right | A list of the current and past subscriptions is displayed in a chronological order. |
| Slide right | The available ticket is displayed along with historical data |
| Slide right | A list of detailed passed validations is displayed. |
| Press a tile in the Validation PanoramaItem | An overview containing detailed information about that validation is displayed |
| Press a tile in the Subscription PanoramaItem | An overview containing detailed information about that subscription is displayed |

**View Bus Lines Test Case**
**Preconditions:** User is logged in, main page is displayed

| Step | Expected Result |
|---|---|
| Expand application bar | three buttons are displayed: all bus lines, buy tickets, subscribe |
| Press button All bus lines | The available lines page is displayed showing the bus lines |

**Buy Tickets Test Case**

**Preconditions:** User is logged in, main page is displayed

| Step | Expected Result |
|---|---|
| Expand application bar | 3 buttons are shown: all bus lines, buy tickets, subscribe |
| Press button Buy tickets | The ticketing page is displayed, showing the available tickets and the price of one ticket. |
| Select the number of tickets you want to buy | The selector shows the number of tickets that the user wants to buy. Total price is updated by multiplying t number of tickets desired with the price of a ticket |
| Press button pay now | The application is redirected to the official PayPal webpage. |
| Enter PayPal credentials | The form is filled |
| Press button Login | The application is redirected to a cost overview page where detailed information about the transaction is displayed. |
| Press button Continue | The account balanced is modified accordingly. The application redirected to the Ticketing page. A pop is displayed showing the transaction id. |

**Subscribe Test Case**

**Preconditions:** User is logged in, main page is displayed

| Step | Expected Result |
|---|---|
| Expand application bar | three buttons are displayed: all bus lines, buy tickets, subscribe |
| Press subscribe button | The application is redirected to the BuySubscription page where a selectable list containing all bus lines is available and a date picker. |
| Select the start date of the subscription | The end date is computed and displayed. |
| Select the line(s) for the subscription | The total price is computed with respect to the number of lines selected. |
| Press Pay Now button | The application is redirected to the official mobile PayPal webpage. |
| Enter PayPal credentials | The form is filled |
| Press Login button | The application is redirected to a cost overview page where detailed information about the transaction is displayed. |
| Press Continue button | The account balance is modified accordingly. The application redirects to the BuyTickets page. A pop-up is displayed showing the ID of the transaction. |

# Chapter 7. User's manual

In this chapter we will present a step by step approach on how to achieve the major functionalities of our system.

## 7.1. Installation

**How to install the application**

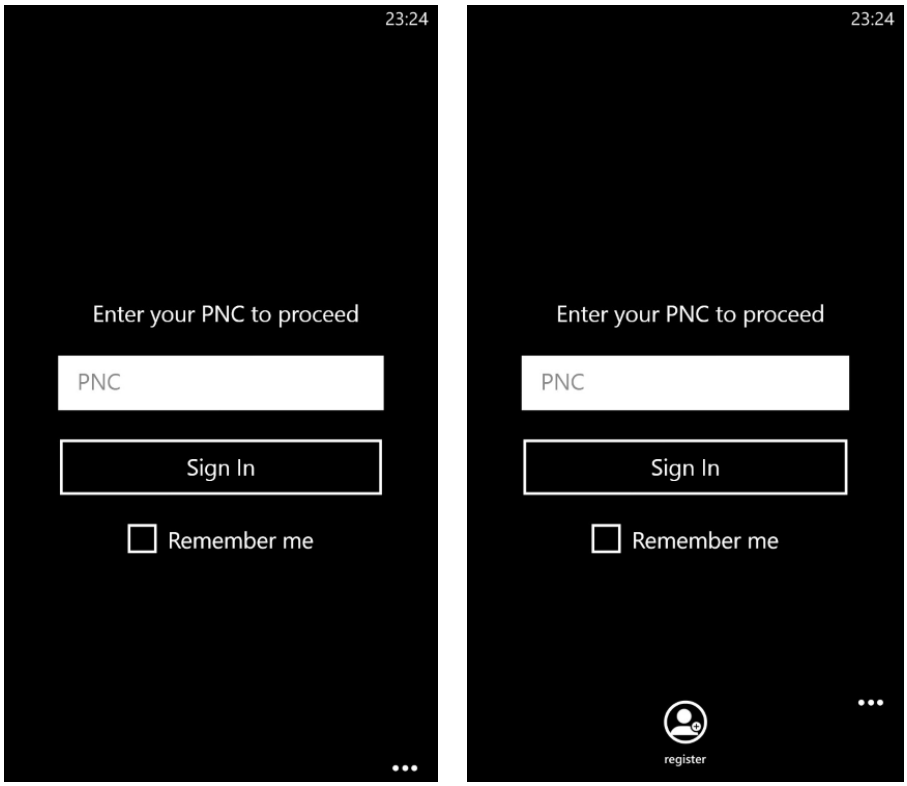
The application can be downloaded from the Store by manually searching for it. The OS can also ask you if you want to download the application from the Store if you previously tapped an NFC tag configured for our application and the application was not installed on the device.

Another way to install the application is to deploy the executable file, the .xap file to the device, by using the Application Deployment tool, a stand-alone application that was installed along with the Windows Phone SDK.

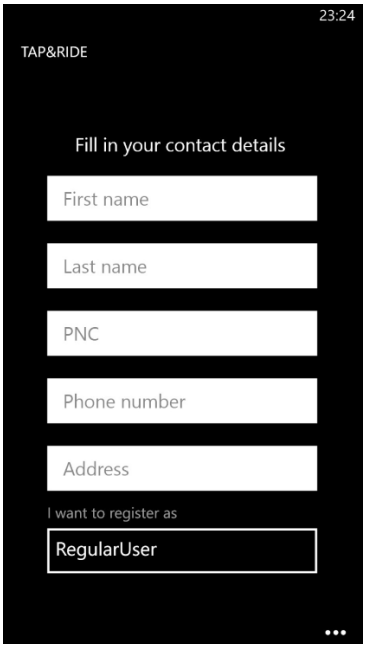
## 7.2. Usage

**How to register**

Go to first page of the application, expand the application bar and tap Register.

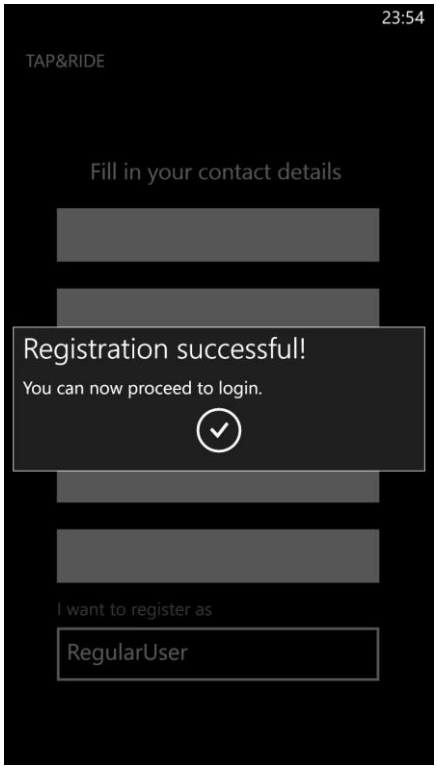


Fill the form and tap save.



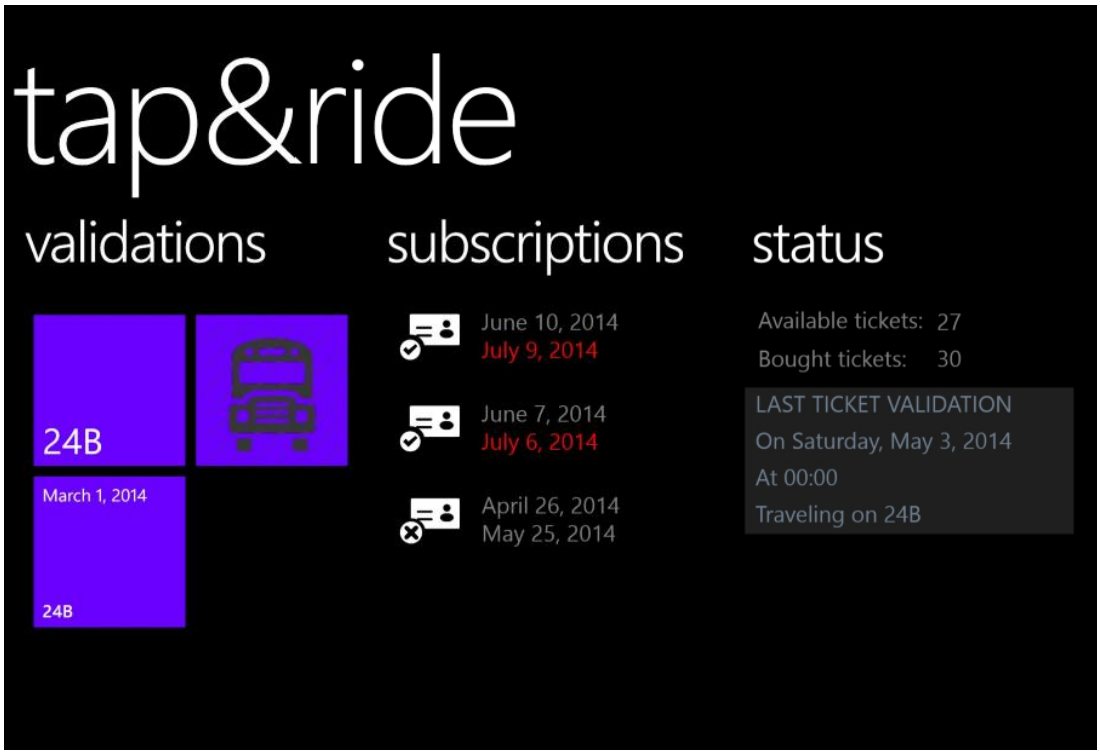


A pop-up will appear confirming the success of the operation.



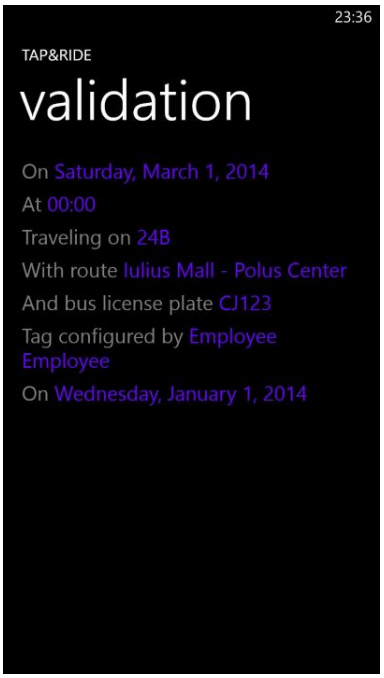
**How to view your account information**
Start the application, fill in the form and tap “Sign In”.


After a successful login, the main page is displayed, where the user can slide left and right to see different information about his/her account.



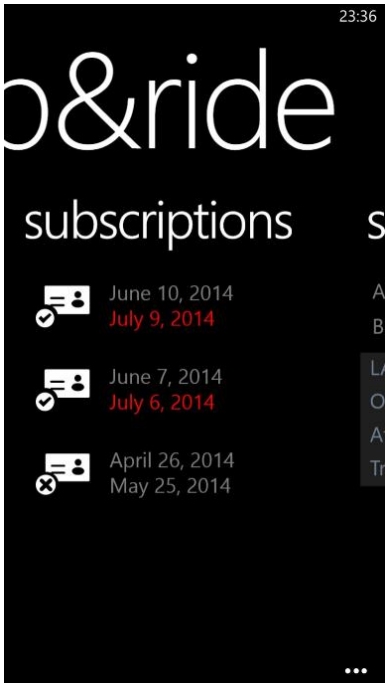
**How to view the details of a ticket validation**

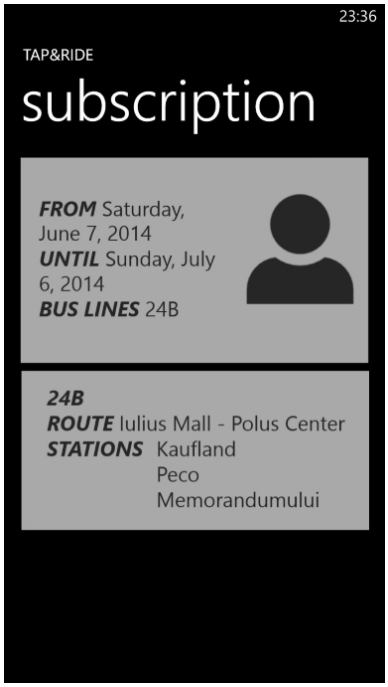
From the main page, the validations pane, tap a ticket validation (represented by a tile). The application will redirect the user to a page displaying the details of the tapped validation.


**How to view the details of a subscription**

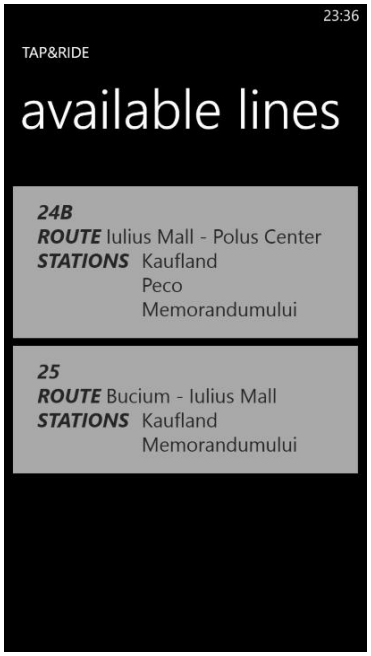
From the main page, the subscriptions pane, tap a subscription. The application will redirect the user to a page displaying the details of the tapped subscription.



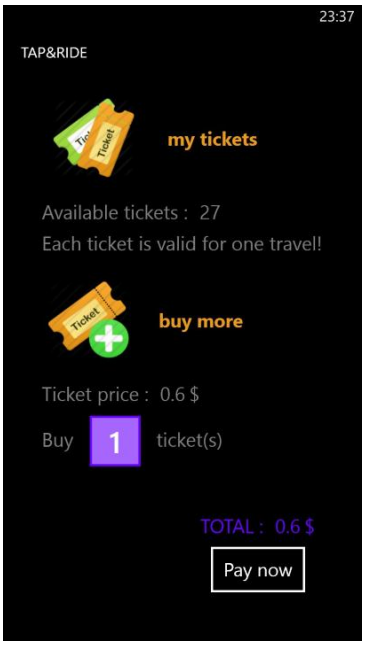
**How to view the available bus lines**

From the main page, open the application bar and press the all bus lines button. The application will redirect the user to another page where all the available bus lines are shown.

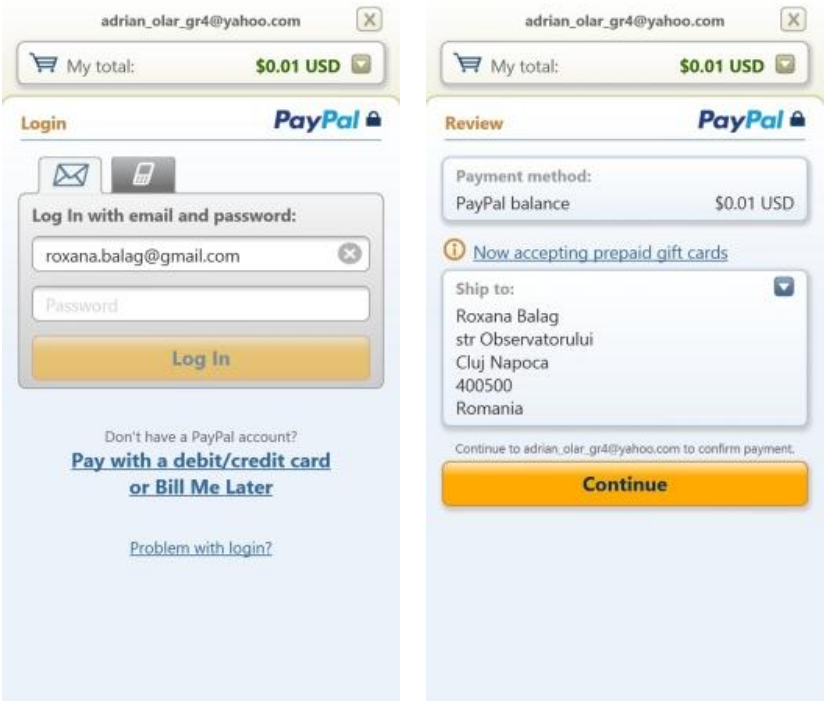




**How to buy tickets**

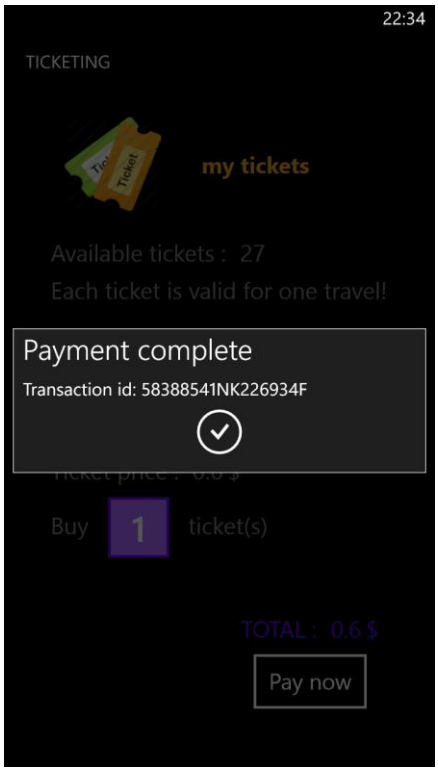
From the main page, open the application bar and press the buy tickets button. The application will redirect the user to the buy tickets page from where the user can select the number of tickets to buy along with other information.

After pressing “Pay Now” the application is redirected to a PayPal page where the user needs to enter his PayPal credentials to login into his PayPal account.




After confirming the payment by pressing “Continue”, the application will redirect the user to the buy tickets page and show a pop-up with the ID of the transaction.

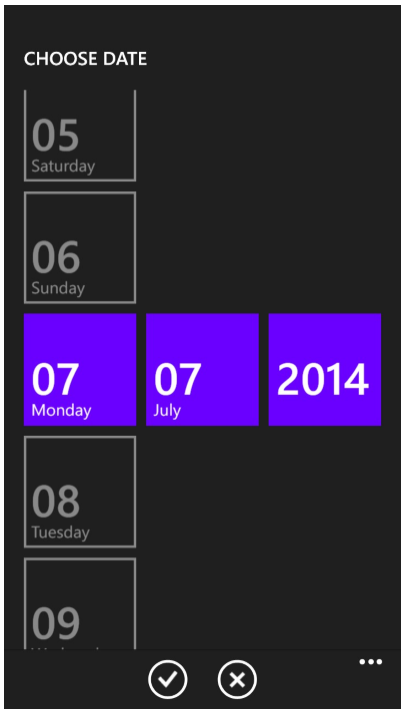


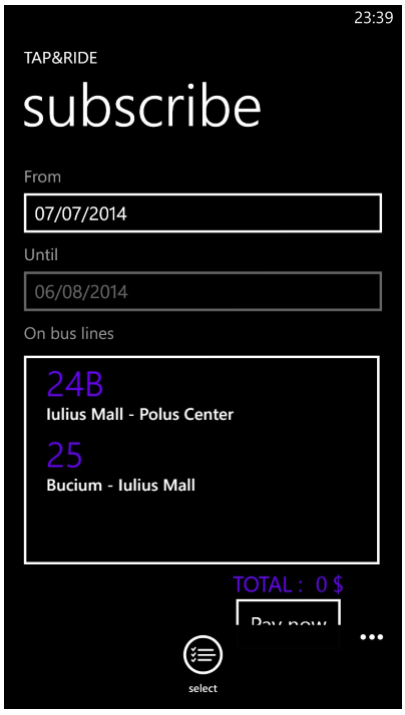
**How to buy a monthly pass (subscribe)**

From the main page, open the application bar and press the subscribe button. The application will redirect the user to the subscribe page.

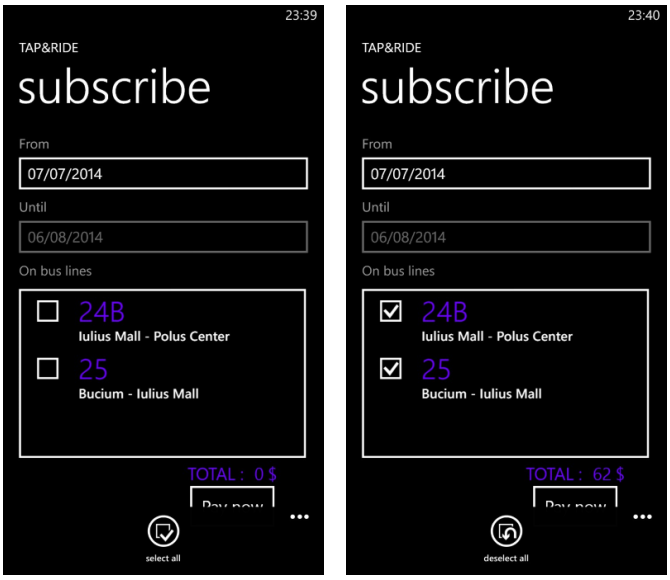


Tap the first date picker, select the start date of the monthly subscription and press save.



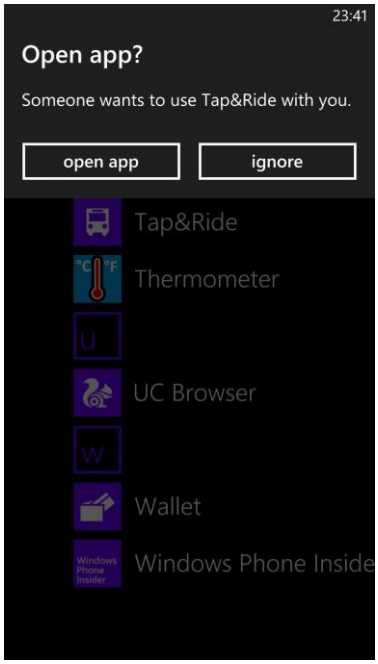
From the application bar press the select button to start selecting bus lines. The buttons in the application bar will allow you to select all and deselect all.




After pressing "Pay Now" the application is redirected to a PayPal page where the user needs to enter his PayPal credentials to login into his PayPal account. (See **How to buy tickets**).

**How to validate a ticket**

Unlock the screen of the NFC-enabled device and tap it to an NFC tag. A pop-up will appear to confirm opening the application.




Press open app. The Tap&Ride application will be opened.

# Chapter 8. Conclusions

In this chapter we will present some conclusions regarding the developed system. In the first part we will present an overview of our achievements, followed by an analysis of the obtained results and finally we will talk about future development and improvements.

## 8.1. Achievements

The implemented system is the result of a set of activities, starting with setting the project's objectives up until bug fixing, after the solution was implemented. The steps I followed while working on this project are the steps that are required in any software development process: make SMART objectives, research the problem domain and the current context, identify the functional and non-functional requirements of the desired system, identify the stakeholders and involved actors, choose the right tools and programming language for the development process, model the data, design the infrastructure, implement the solution, test the solution, bug-fix the solution, document the solution by answering the "what?", "when?", "who?" and "how?" questions and by including future improvements or points of interest in the system (where changes are foreseen).

This project was a challenge, due to the fact it had a time span of eight months, a higher complexity than any other project I've (individually) worked on and it required a lot of work. From my point of view, the hardest part was the research performed prior to starting implementation and the writing of this document, while the easiest part was writing the actual code.

What I achieved while working on this project is a deeper understanding of the Near Field Communication and the multiple benefits a ticketing system using this technology could provide to all involved parties. I learned to use Near Field Communication technology in a Windows Phone 8 application and experienced the feeling of what is now present in many countries: validation using a smartphone via NFC.

Another feature that offered a real-life experience that is encountered in so many other applications is the online payment, performed directly from the mobile phone. By using PayPal's Checkout SDK I offered a payment method without needing to worry about the transaction security, confidentiality, authorization and other aspects that are generally crucial in transactions involving money. All of those aspects were managed and handled by the PayPal SDK, all I had to do was to use it. This gave me an opportunity to realize there are already solutions for these problems and there is no need for me to reinvent the wheel, but just use the existing tested, deployed and trusted solutions to avoid the problem of reinventing a square wheel.

During the bibliography study, which I later summarized as best as I could in this document, in the Bibliographic Research chapter, I had the opportunity to see that what for us is the future, in other countries that is the present. More evolved countries have already taken advantage of the NFC technology, which is a clear indication that NFC is the way to go and its possibilities need to be explored further.

When writing the Analysis and Theoretical Foundation chapter of this document, I had the chance to cement the already known concepts of the presented technologies and

even to understand some concepts that until then I thought I understood, but it turned out I did not.

The most valuable thing gained while working on this project, was experience. I can now use WCF services, Entity Framework, AutoMapper and other of the technologies I used in this project, with more confidence. I can handle serializing and deserializing JSON formats, I can implement Windows Phone 8 applications having a more consolidated foundation of knowledge, and most importantly, I no longer have a great difficulty in expressing functionalities and interactions through diagrams and words.

## 8.2. Results

In order to observe the achieved results, we need to take a look at the requirements of the system. The requirements of the system, previously defined in the second chapter are:

- Registration

The user has the option to register, if he/she does not have an account, i.e. this requirement is met.

- Authentication

The user can authenticate into the application, such that only he/she can access his/her own data, meaning that this requirement was also fulfilled.

- Ticket validation

The user can validate his/her ticket by using the ticketing application, so this requirement was met.

- Validations tracking

The user is provided with a list of his/her validation, which fulfills the requirement.

- Subscriptions tracking

The user has access to a list of all his/her subscriptions, meaning this requirement was also met.

- Tickets status tracking

The user can see the number of tickets he/she has remaining and the number of all the tickets ever bought. This fulfills the requirement.

- Ticket purchasing

The user can purchase tickets by using the mobile ticketing application, i.e. the requirement is met.

- Bus lines consulting

The user can access a list of all the bus lines the transport company runs, fulfilling this requirement.

- Subscription purchasing

The user can purchase a monthly pass, a.k.a. subscribe or purchase a subscription, so this requirement was met.

- Logging out

The user was provided with the option to log out, i.e. this requirement is met.

- Tag writing

The tag writing application gives the transport operator a way to configure tags, meaning this requirement was met.

For the non-functional requirements of the system, we obtained the following result:

- The system is available all the time, as long as the user has an active Internet connection and as long as the application server is up and running.
- The system is accessible from any location and anytime, with the same condition that the user's device has an active Internet connection.
- The system manifests reliability because it behaves in a consistently manner.
- The authentication requirement adds security to the system, by granting access and permission of use only to authenticated users
- The system has a user-friendly interface based on Microsoft's UX recommendations, with a clean, uncluttered design, with a focus on the content. The intuitive way of validating a ticket, by tapping the mobile device to an NFC tag provides a natural and clear way of usage
- The maintanabiliy of the system was obtained by applying best practices and design patterns throughout the application, which offer a clear separation of concerns.
- Extendibility is also possible due to the architecture, design patterns and applied principles.
- The performance of the system is subjective, due to the fact that until now, the system was tested with a small amount of data and no concurrent users. Under the mentioned conditions, all the system operations did indeed take under 5 seconds.

To summarize, the results were the ones we were expecting. We obtained a system that meets all the functional and nonfunctional requirements expressed at the begininning of the project

## 8.3. Future development and improvements

This project has a lot of place for improvement and future development. Although it is impossible to mention all the improvements and future development work that can be done in this project, we will offer a discrete list, suggesting some of them:

- Encrypt the sensitive data, like the user's personal information. This will enhance the system's security. A possible encryption algorithm is AES (Advanced Encryption Standard)
- Add offline capabilities to the system. The user should be able to use the system even when an Internet connection is not available. This can be done by possibly storing data on the device and when an Internet connection becomes available, synchronize all the data with the application server.
- Add multiple payment methods. A possibility would be to use SMS-based payment, or implement an API like Fortumo, that makes payments to be charged to the user's mobile operator bill. Possibilities here are countless.
- Manage the lifecycle of the application. Implement a SuspensionManager class that handles the changing of states in the application's lifecycle.
- Add extra validation rules on the inputs provided by the users. Constrain PNC to be numerical.

- Add more business rules. Give the user the option to subscribe for passes on wider range of periods, varying from weeks to even trimesters and on a number of bus lines of his willing, not restricted to one, two, three or all bus lines.
- Offer discounts based on the purchases performed by the user. A fidelity program is a nice idea to keep people interested and attracted to the application.
- Offer the user the possibility to edit his account. The user should be able to see and edit his personal information.

## Bibliography

[1] Whitechapel, S. McKenna, Windows Phone 8 Development Internals, Microsoft Press, October 2012
[2] GSMA, White Paper: The Value of Mobile Commerce in Transport, February 2014. [Online]. Available: http://www.gsma.com/digitalcommerce/wp-content/uploads/2014/01/The-Value-in-Mobile-Commerce-in-Transport-Paper_updated-V2.pdf
[3] T. Tuikka, M. Isomursu, Touch the Future with a Smart Touch, Helsinki 2009, VTT Tiedotteita – Research Notes 2492. 280 p. [Online]. Available: http://www.vtt.fi/inf/pdf/tiedotteet/2009/T2492.pdf
[4] M. Mut-Puigserver, M. M. Payeras-Capella, J. L. Ferrer-Gomila, A. Vives-Guasch, J. Castella-Roca, A Survey Of Electronic Ticketing Applied to Transport, Computers & Security, Volume 31, Issue 8, November 2012, Pages 925–939
[5] M. Mezghani, Study on electronic ticketing in public transport, May 2008, European Metropolitan Transport Authorities. [Online]. Available: http://www.emta.com/IMG/pdf/EMTA-Ticketing.pdf
[6] S. Burkard, Near Field Communication in Smartphones, 2012. [Online]. Available: https://www.snet.tu-berlin.de/fileadmin/fg220/courses/WS1112/snet-project/nfc-in-smartphones_burkard.pdf
[7] European Parliamentary Research Service, Integrated urban e-ticketing for public transport and touristic sites, January 2014. [Online]. Available: http://www.europarl.europa.eu/RegData/etudes/etudes/join/2014/513551/IPOL-JOIN_ET(2014)513551_EN.pdf
[8] NFC Forum, NFC Forum Technical Specifications, 2006. [Online]. Available: http://members.nfc-forum.org/specs/spec_list/
[9] GSMA, White Paper: Mobile NFC in Transport, September 2012. [Online]. Available: http://www.gsma.com/digitalcommerce/wp-content/uploads/2012/10/Transport_White_Paper_April13_amended.pdf
[10] S. Stroh, D. Schneiderbauer, C. Kreft, Next Generation eTicketing, 2007. [Online]. Available: http://www.booz.com/media/uploads/Next_Generation_eTicketing.pdf
[11] Desai, M. G. Shajan, A Review on the Operating Modes of Near Field Communication, International Journal of Engineering and Advanced Technology (IJEAT) ISSN: 2249 – 8958, Volume-2, Issue-2, December 2012. [Online]. Available: http://www.ijeat.org/attachments/File/v2i2/B0956112212.pdf
[12] T. Agarwal, How does the Smart Card Work?, September 23, 2013. [Online]. Available: http://www.elprocus.com/working-of-smart-card/
[13] J. Liebenau, S. Elaluf-Calderwood, P.Karrberg, G. Hosein, Near Field Communications; Privacy, Regulation & Business Model, October 2011. [Online]. Available: http://www.lse.ac.uk/management/documents/LSE-White-Paper_-_Near-Field-Communications-Privacy-Regulation-Business-Models.pdf
[14] A. Juntunen, S.Luukkainen, V. K. Tuunainen, Deploying NFC Technology for Mobile Ticketing Services – Identification of Critical Business Model Issues, 2010 Ninth International Conference on Mobile Business

[15] N. Mallat, Exploring consumer adoption of mobile payments – a qualitative study, The Journal of Strategic Information Systems, vol. 16, Dec. 2007, pp. 413-432
[16] Public Transport ITS Committee, White Paper on the Application of NFC Technology in Public Transport, First Edition, December 2013. [Online]. Available: http://www.fomento.gob.es/NR/rdonlyres/F5BBB37E-F29E-4C47-AE7B-77C3875CAC92/122698/White_Paper_NFC.pdf
[17] M. Kerschberger, Near Field Communication: A survey of safety and security measures, July 17, 2011. [Online]. Available: https://www.auto.tuwien.ac.at/bib/pdf_TR/TR0156.pdf
[18] L. Church, M. Moloney, State of the Art for Near Field Communication: security and privacy within the field, May 10th, 2012. [Online]. Available: http://www.eschergroup.com/_common/file.cfm?id=D53DE4BDF5248B41DE0BCFD8EC50A2AD
[19] T. Van Anh Pham, Security of NFC applications, June 2013. [Online]. Available: http://nordsecmob.aalto.fi/en/publications/theses2013/thesis_pham/

[20] B. Dobson, NFC in Transport for London - Setting a context for the role of NFC technology in TfL's ticketing strategy, June 2008. [Online]. Available: http://67.222.41.204/wp-content/uploads/2013/12/Brian-Dobson-Transport-for-London.pdf
[21] M. Pagany, Windows Phone 8 Development Succintly, Syncfusion Inc., 2014.
[22] Microsoft, Designing for Windows Phone. [Online]. Available: http://www.avlade.com/WindowsPhone/Windows%20Phone%20Curriculum_sm.pdf
[23] A. Wigley, R. Tiffany, Building Apps for Windows Phone 8 Jump Start. [Online]. Available: http://www.karabinaacademy.com/media/1492/s1-introducing-wp8-development.pdf
[24] J. Gil, PayPal 101: What exactly is PayPal?, May 2014. [Online]. Available: http://netforbeginners.about.com/od/ebay101/ss/paypal101.htm
[25] C. Anderson, V. Dadok, A. Galsky, S. Lin, C. Mendez, B. Zhu, PayPal: A Marketing Strategy for the Future, August 2008. [Online]. Available: http://anderson.lbl.gov/publications/PayPal.pdf

# Appendix

## Table of figures



## Glossary

M-ticketing – mobile ticketing
E-ticketing – electronic ticketing
SIM – Subscriber Identity Module
OCR – Optical Character Recognition
RFID – Radio Frequency Identification
NFC – Near Field Communication
RTD – Record Type Definition
NDEF – NFC Data Exchange Format
UI – User Interface
UX – User Experience

XAML - Extensible Application Markup Language
SOAP - Simple Object Access Protocol
REST - Representational State Transfer
WCF – Windows Communication Foundation
XML - Extensible Markup Language
RSS - Rich Site Summary
HTML - Hypertext Markup Language
IIS - Internet Information Services
SDK – Software Development Kit
ORM – Object-Relational Mapping
EF – Entity Framework
LINQ - Language-Integrated Query
MVVM – Model View ViewModel
JSON - JavaScript Object Notation
DTO – Data Transfer Object
CRUD – Create Read Update Delete
T4 - Text Template Transformation Toolkit
WPF – Windows Presentation Foundation