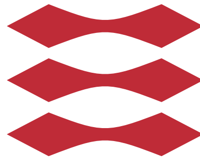
Mobile Application Development of the Erhvervsstyrelsen Frekvensregistret Service – Challenges and Implementation

Martin Olsen



Kongens Lyngby 2012
IMM-MSc-2012-119

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-MSc-2012-119

Abstract

This thesis covers the process of creating a mobile app version of Erhvervsstyrelsen's established website, the Frequency Registry. The app should be available on multiple software platforms, with Android and iOS as a minimum. Because of this minimum requirement, the various methods for cross-platform development is investigated. Each development platform is evaluated for its pros and cons, followed by a comparison of all of the platforms. Based on the requirements for the app, the development platform that is most suited for the task, is chosen. The choice falls on Appcelerator's Titanium Mobile, which is a platform especially designed for creating cross-platform apps for Android and iOS. The unique feature about Titanium Mobile is that the developer can write all of the code in JavaScript while still have access to native elements through the Titanium Mobile API. This way it is possible to write native apps for both Android and iOS from the same codebase.

Since smartphones are still much more common to have, than a tablet, this thesis is focussing primarily on adapting the app to a smartphone.

This thesis covers the design phase of the app, both in regards to graphical user interface as well as how this app will communicate with the existing environment.

Followed by this is the actual implementation in Titanium Mobile, and the final result is evaluated.

As part of the conclusion, my personal experiences with Titanium Mobile are covered in detail.

Key Contributions

This thesis covers some of the most popular ways to make cross-platform apps for mobile devices. The pros and cons of all of the development platforms are evaluated which serves to give an overview of the avaialble technologies that exist today.

This thesis also takes a closer look at Appcelerators Titanium Mobile platform, by showing the details of the design and implementation process of a mobile app, using this platform. Titanium Mobile is evaluated in more detail, based on the experiences from creating the Frequency Registry app.

At last this thesis serves to provide a concrete example of how Erhvervsstyrelsen could expand their current service of the Frequency Registry to the world of mobile computing.

Structure of the Thesis

Chapter 1

This chapter serves as an introduction to the project. In this chapter the existing website of the Frequency Registry service is being presented. Hereafter the reader is introduced to the world of mobile computing, which is the background for the creation of the app version of the Frequency Registry.

Chapter 2

This chapter introduces the various software platforms that exist for the smartphones and tablets as of today. It also describes the development platforms that exist for creating apps for these software platforms. Finally it compares all of the development platforms according to which one will be most the appropriate one to use for the implementation of the Frequency Registry app.

Chapter 3

Chapter 3 deals with the overall design of the system, without going into implementation details, which are covered in chapter 4. Chapter 3 covers both the conceptual design of the graphical user interface of the app, as well the task of incorporating the new app in the existing environment, in order for it to communicate with the existing database. The various options for establishing this communication are presented and a choice is made among them.

Chapter 4

Chapter 4 covers the details of the implementation as well as an evaluation of the performance of the final app.

Chapter 5

Chapter 5 covers the experiences I had while working with both the Titanium Mobile SDK and the IDE Titanium Studio.

Chapter 6

Chapter 6 gives a conclusion of the project as well as suggestions to future work.

Preface

This thesis was prepared partially at the department of Informatics and Mathematical Modelling, IMM, at the Technical University of Denmark and at the company NNIT A/S, in fulfilment of the requirements for acquiring a M.Sc. in Computer Science and Engineering.

The thesis deals with development of a mobile application (app) for the Danish Business Authority Erhvervsstyrelsen. This app is being developed with the help of a fairly new cross-platform environment called Titanium Mobile, which is created by the company Appcelerator.

Lyngby, 18-September-2012

Martin Olsen

Acknowledgements

Throughout this project, notable persons have contributed with their helpful support and encouragement, making this project a success. I would like to thank:

Supervisor Associate Professor Nicola Dragoni, for offering to supervise this project, and to give me the help and guidance i needed throughout the project.

Advanced Developer at NNIT A/S, Allan Fejerskov Ravn, for being the initiator of the project, and for helping me get started and giving me advice along the way.

Advanced Developer at NNIT A/S, Rasmus Lohals, for giving me a speed course in app development and the use of RESTful services, and for allowing me to use his personal webhotel for hosting of the Frequency Registry RESTful service.

Senior Solution Architect at NNIT A/S, Jon Bille, for allowing me to borrow hardware to test my app on, and helping me deploy the app to my iPhone.

Contents

Abstract i

Preface iv

Acknowledgements v

1 Introduction 1
- 1.1 Background . . . 2
  - 1.1.1 Existing Website . . . 2
  - 1.1.2 Usage statistics on the existing website . . . 5
  - 1.1.3 Mobile Computing . . . 6
  - 1.1.4 Smartphone market in Denmark . . . 7
- 1.2 Problem . . . 9
- 1.3 Requirements for the Frequency Registry app . . . 9
- 1.4 Project delimitation . . . 9

2 Platforms and development tools 10
- 2.1 Software platforms . . . 11
- 2.2 Development platforms . . . 12
  - 2.2.1 Native development . . . 12
  - 2.2.2 Mobile Web Applications using HTML5 . . . 13
  - 2.2.3 PhoneGap . . . 16
  - 2.2.4 Appcelerator Titanium Mobile . . . 18
  - 2.2.5 Mono . . . 20
- 2.3 Comparisons . . . 22
  - 2.3.1 Choice of development platform . . . 26

3 Design of The System 27
3.1 Structure of a Titanium app 28
3.2 Design of user interface 30
3.3 Communication with the existing system 40
3.3.1 Using the existing website 40
3.3.2 Connecting directly to the database 41
3.3.3 Using a service 42
3.4 Choosing the right kind of service 42
3.4.1 Asp.net services 43
3.4.2 WCF service 44
3.4.3 A RESTful service 44
3.4.4 Choice of service 47
3.5 System overview with the service 49

4 Implementation 50
4.1 Final Design of The Frequency Registry App 51
4.2 Program structure 58
4.3 The Frequency Registry RESTful service 62
4.4 Performance 63

5 Experiences with Titanium Mobile 66
5.1 Build once deploy everywhere 66
5.2 Active community 71
5.3 Titanium Studio 71

6 Conclusion 72
6.1 Future Work 73

A Source Code 75

Bibliography 76

List of Figures

1.1 Frequency Registry start up page 3

1.2 Frequency Registry results page 4

1.3 Number of page loads on existing website in the period from 1/12-2010 to 1/12-2011 5

1.4 Operating systems running on smartphones in Denmark [15] . . . 7

1.5 Operating systems running on smartphones world wide [11] . . . 8

2.1 Todays most popluar platforms on smartphones 11

2.2 Comparison of the frameworks 22

3.1 Basic structure of a Titanium Mobile app 28

3.2 Titanium Mobile app with 2 tabs 29

3.3 Startup window of the App 31

3.4 Enter value of a search criterion 33

3.5 A list of the search results 35

3.6 Details on a specific case 37
3.7 Details on the sending position of a case 39
3.8 Existing Frequency Registry system 40
3.9 Interacting with the database using a service 42
3.10 System overview with the new REST service 49

4.1 Start up window 52
4.2 List of case types 54
4.3 Results of search 55
4.4 Case details 56
4.5 The three categories of information on a case 57

5.1 Standard application on both platforms (Android to the left), (iPhone to the right) 67
5.2 Differences on Android (left) and iOS (right) -– Logo 68
5.3 Differences on Android (left) and iOS (right) – Custom made row 69
5.4 Differences on Android (left) and iOS (right) – Table with rows showing only text 70

Chapter 1

Introduction

This chapter serves as an introduction to the project, and covers the reasons behind the making of the Frequency Registry app, as well as introducing the reader to the whole world of mobile computing.

Section 1.1 introduces the reader to the concept of the Frequency Registry, which is a public service provided by Erhversstyrelsen. It takes a look at the existing website, which currently offers this service, and how high the activity level is on this website. Then it introduces the reader to the concept of mobile computing which is the motivation for creating a mobile app version of the Frequency Registry service. It raps up by providing a brief look at the smartphone market of today.

Section 1.2 describes the problem that this project is dealing with, which in short is determining the most appropriate way to create a mobile app version of the Frequency Registry.

Section 1.3 is a list of the requirements for the new Frequency Registry app.

Section 1.4 is a description of the delimitations of the project.

1.1 Background

In today's world it is getting more and more common to have a smartphone. These high technology cell phones are not only used by private persons, but are also widely used in the business world. The potential of these smartphones extends way beyond sending emails, synchronizing calendars and playing games. Part of the reason for this, is the huge amount of applications that are available to these smartphones. These applications are commonly known as "apps" and they are gathered in "app stores" where they can be downloaded directly to the smartphones, usually for the cost of a few euros. Not only is this a great opportunity to make a business, but with all the hardware that is being put into these smartphones (GPS, camera, file storage, etc.) it enables new ways for companies to offer their services to their customers.

This is this case for the Danish Business Authority called "Erhvervsstyrelsen", which is managed by the Danish government. One of their responsibilities is to manage and issue licences to persons, associations and companies, that wish to use the frequency space in Denmark. As part of their business they have a website where all the issued licences can be looked up. This service is called "Registry of Frequencies" and it is public available at the website http://frekvensregister.itst.dk.

Erhvervsstyrelsen are currently considering expanding this service to the smartphone market. Because Erhvervsstyrelsen is an official organisation run by the government, it is important to them that such an application would be available to as many people as possible. Knowing how big of the market shares are held by devices running the software platforms Android and iOS, they require that this new app would support at least these two platforms.

1.1.1 Existing Website

As mentioned, Erhvervsstyrelsen has created a website in order to make the Frequency Registry available to the general public. The general functionality of the website is to perform searches in the registry, which holds information about who has acquired licences to use the frequence space of Denmark. In business terms, a licence is stored in a case, and it is actually cases that are being searched for in the Frequency Registry. Each case can grant licences to several sending positions, on several frequencies.

Upon entering the website, the user is met with the page shown on figure 1.1.

Figure 1.1: Frequency Registry start up page

Here the user is presented with a lot of different fields which can be used to filter the search. The fields are divided into the following 3 subgroups: Technical data, sending position and user data.

The technical data hold information about the sending device including:

- Which frequencies are being used.
- What category does the sending device belong to.
- When does the licence expire.
- Are the frequencies allowed to be used across the entire country, in a region or in a custom defined area.

The sending position part holds information about the address of the location of the device. It is also possible to draw an area on a map, and the search will then return all the cases where the sending device is located within the drawn area.

The user data part includes the user's id in the system as well as the user's name and address.

Once the search criteria have been input as desired, the user can perform a search, which will bring up the results page, seen on Figure 1.2.

Figure 1.2: Frequency Registry results page

On this page the cases found by the search is listed in rows, with some of the most significant information shown for each case:

- The ID of the case.
- The sending and receiving frequency (if any).
- The name of the licence holder.
- The case type.
- The city where the antenna is placed.

On every row in the table, there is a button next to the case ID field, which can be used to expand and collapse a list of further information. Expanding this list shows all the available information the case has on this specific licence. Since a case can include several licences, the same case id can figure several times on this list, describing different licences. Like on the search page, the information is divided into three sections, here denoted as “Sending position”, “Technical specifications” and “User data”.

1.1.2 Usage statistics on the existing website

The potential customers of the Frequency Registry app are the users that are already using the Frequency Registry website. So in order to get an idea the magnitude of the potential customers of the app, it makes sense to look at the activity on the existing website. The IT department of Erhvervsstyrelsen has provided the statistics of the usage of this website, which is shown on figure 1.3.

Figure 1.3: Number of page loads on existing website in the period from 1/12-2010 to 1/12-2011

These statistics show that from between December 2010 and December 2011, there has been around 30 page loads each day on average. The total amount of page loads is counted up to be 10485. When divided 365 days, it gives the number of 28.73 page loads per day. Erhvervsstyrelsen also informed that on average will a user spend 2 minutes using the Frequency Registry. This indicates that it is a service that is used briefly, which is also usually what you expect from a mobile app.

1.1.3 Mobile Computing

For many years, computers were stationary machines that rarely got moved from one location to another. Part of the reason for this is that computers used to be quite large devices, especially if they were to meet the performance requirements of an average private user. As technology progressed, the hardware manufacturers were able to produce smaller and smaller components while still increase the performance of these components. Computers were suddenly not a big unhandy stationary machine anymore, but were becoming available as laptops, handheld devices, tablets and even as telephones.

Alongside with the evolution of the computer, there was another important development going on in the field of computer science, namely wireless communication. One of the most important wireless technologies is the wireless LAN protocol 802.11, which is an IEEE standard, often referred to as “IEEE 802.11”. The first official version was announced in 1997 by IEEE [16]. Since then, many improvements have been made on the technology and it is still the de facto standard for wireless communication between computers today.

Another important wireless technology is the telecommunication. This technology has long been used for cell phone voice calls, but as the 3rd generation mobile telecommunication (abbreviated 3G) were developed and widely used in the industry, the telecommunication network got more suited for transferring data beside voice calls. As of 2012, the proclaimed speed of the 3G network, provided by the biggest telephone company in Denmark, is at 384kbit/s [18]. When compared to the early days of the Internet, where private users used 56kbit modems to browse the Internet, the speed of the 3G network should be more than enough for just browsing the Internet and the occasional emailing.

The progressing in the wireless technologies together with the decreasing size of the ordinary computer, lead to the increasing demand for computers to be more mobile. Not only do people want to bring their computers with them, when they are on the move, they also want to be able to use it while doing so. Soon all sorts of mobile devices started to flood the market.

One of the newer types of mobile devices, are the smartphone. This device is a combination of a cell phone and a computer. One of the characteristics about a smartphone, is that it can run these apps, as introduced earlier. Apps are widely used in both the private and industrial sector. Today there exists more than 25 billion apps combined, when counting the 2 biggest app stores (Apple’s app store and Google Play) [4] [12]. The use of apps on smartphones has opened a whole new way for companies to get in touch with their customers. At first the app stores mainly consisted of games and entertainment applications, but

as time went by, all kinds of organizations could see opportunities of including apps in their business plans, including banks and institutions of the government.

1.1.4 Smartphone market in Denmark

Since Erhvervsstyrelsen only operate in Denmark, the primary market of interest is the Danish smartphone market. Here are the latest statistics of the Danish smartphone market, produced by the company "Wilke Mobile Life".

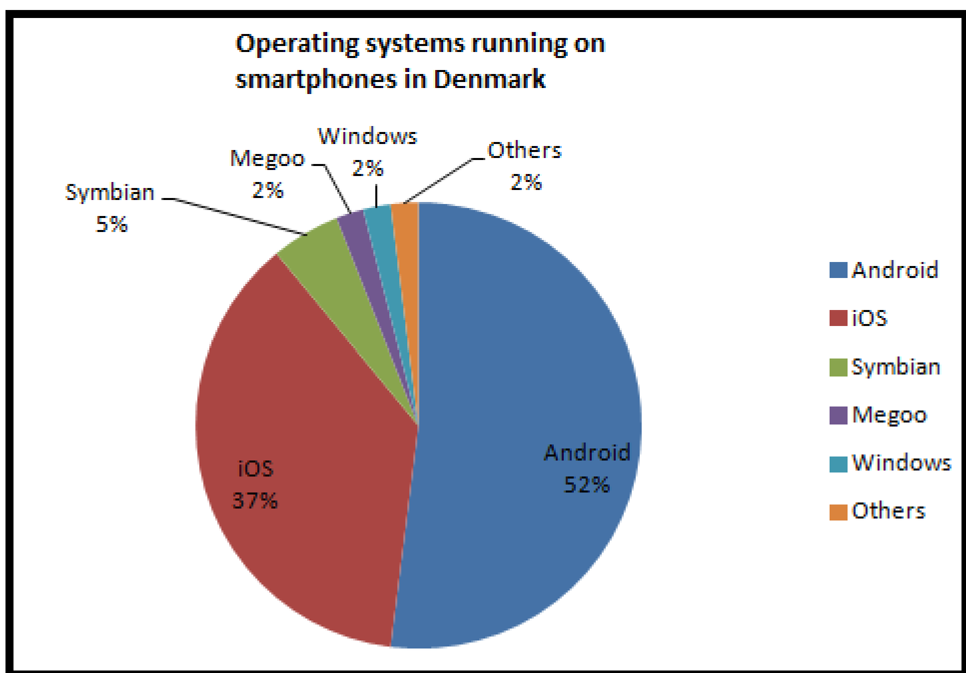


Figure 1.4: Operating systems running on smartphones in Denmark [15]

As the market analysis shown, iOS and Android are quite dominant on the Danish market, sitting on 89% of the market combined. Even though it is the Danish market that matters to Erhvervsstyrelsen, it makes sense to take a look on the market on a world wide scale. Changes in the market on the world wide scale are likely to affect the Danish market as well, and are therefore still relevant to know about.

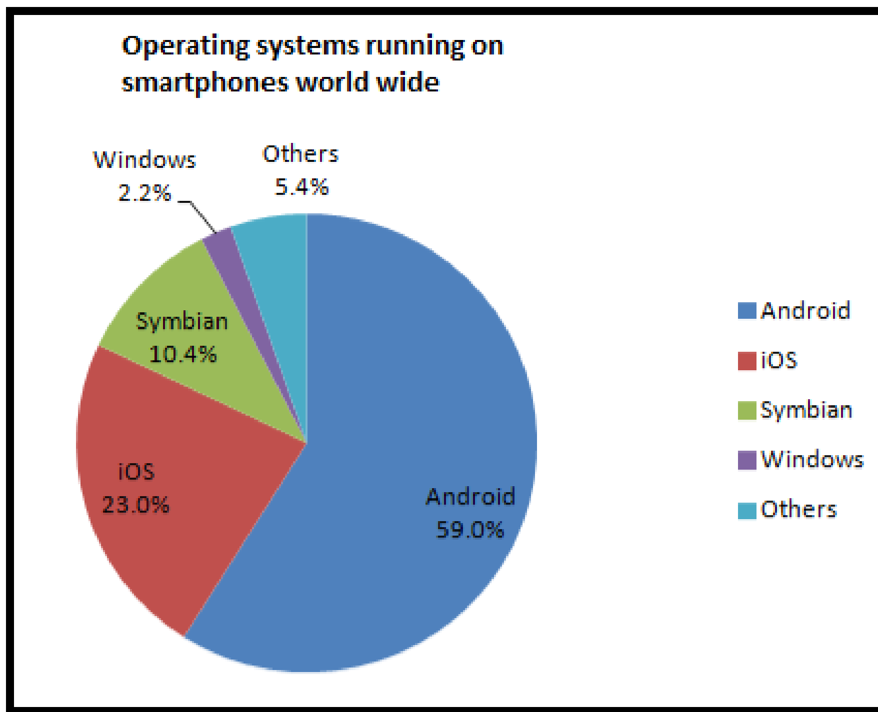


Figure 1.5: Operating systems running on smartphones world wide [11]

As the statistics on the world wide smartphone markets shows, iOS and Android are sitting quite a big part of the market. A total of 82% of the market combined. This is somewhat similar to the Danish market, which leads no indication of a change in the Danish market.

1.2 Problem

Erhvervsstyrelsen would like to make their existing service, the Frequency Registry, available as a mobile application. They want their service to be available to as many of their users as possible, and therefore they do not wish to limit their service to be available on 1 platform only. This thesis looks at the challenge of making this service available on multiple platforms in the most cost efficient way, without compromising the access to the hardware functionality available to the high-end smartphones.

1.3 Requirements for the Frequency Registry app

- Must be able to perform the basic functionalities of the existing Frequencies Registry website, which is the searching for licences.
- Support for multiple platforms, with iOS and Android as a minimum.
- Be easily available to customers.
- Have a low development cost.
- Be able to access hardware functionality for future extensions of the app.

1.4 Project delimitation

The solution for the Frequency Registry app will neither be deployed to any of Erhversstyrelsen's existing environments, nor will it have any forms of contact with the real databases of the existing Frequency Registry. All communication to and from the app will be to an independent webserver, which hosts a Frequency Registry service, created as part of this thesis.

The reason for this is to avoid making any disturbances to the real environments, and to focus on the challenges regarding creating the app and not on the communication with the outside system.

The app version of the Frequency Registry will not include all of the functionality that the website includes.

Chapter 2

Platforms and development tools

This chapter introduces the various software platforms that exist for the smartphones and tablets as of today. It also describes the development platforms that exist for creating apps for these software platforms. Finally it compares all of the development platforms according to which one will be most the appropriate one to use for the implementation of the Frequency Registry app.

Section 2.1 looks into the various software platforms that exist for smartphones today, and determines which ones should be supported by the Frequency Registry app.

Section 2.2 takes a look at each of the relevant development platforms in detail, covering the pros and cons for each of them.

Section 2.3 starts off by making a comparison of all the development platforms, and ends up choosing the one that is most appropriate to use in this project.

2.1 Software platforms

There are many considerations to take into account, when deciding to create a mobile application. Choosing the software platform(s) which the application should support, being one of the most important ones. The choice of supported platforms can have a major impact on how successful the application will become. If Android is chosen as a platform, and all the potential users of the application is using iPhones, then the chance of the app being successful is obviously very low, regardless of how good it is.

The choice of platform may not be an easy task, considering that there exist quite a few platforms as of today. Figure 2.1 shows the most popular and well-known platforms that exists today.

Platform	Language(s)	IDE	OS X
iPhone	Objective-C, C	Xcode	OS X
Android	Java, C, C++	Eclipse/Netbeans	Windows, Linux, OS X
Blackberry	Java	Eclipse	Windows
Symbian	C, C++, QT, Java	Any text editor	Windows, Linux, OS X
Windows Phone	C#, VB.NET	Visual Studio	Windows
webOS	HTML, Javascript, CSS	Any text editor	Widows, Linux, OS X

Figure 2.1: Todays most popluar platforms on smartphones

As figure 2.1 shows, all of them differ in some way from each other. The languages listed next to each platform, is the programming language which is used to make native applications to that specific platform. The programming language for making native applications for each platform, differ quite a lot. This makes it difficult to reuse code, if a company decides that they want to make a Blackberry version of their iPhone application.

When the choice of supported software platform(s) has been decided, the decision of how to implement the application, must be made. The variety of options is quite large in this area as well. First off, it must be decided if the application should be done in the native language or if it should be done in some other way. The two extremes that exist today is native development on one side and a pure HTML5 application on the other side, which basically is a website customised to be viewed on a smartphone. In between these two extremes there are a lot of different options that should be taken into considerations by the person or company who is responsible for the app. When facing the decision of choosing the right technology, there is no silver bullet. It depends on the purpose of the app. Therefore an individual evaluation of the app's purpose should always be made before making the decision. For example, if there is no need for native

functionality in the app, then it might be a good idea to do a pure HTML5 app and having the benefit of an app that is platform independent.

As mentioned earlier, Erhvervsstyrelsen do not want to limit their service to one platform only. And based on the market situation in Denmark, the app must be able to run on both the iPhone and smartphones running Android. This narrows the options down a little bit, but there are still a lot of ways this app could be implemented. The following sections describe some of the most well-known ways to achieve this goal.

2.2 Development platforms

2.2.1 Native development

This is the way apps originally were meant to be developed. Each platform has one or more programming languages which are used to create its native apps, and the operating system of the platform expects to run apps build with its native programming language. Apps that is wrapped in native code is also able to run on the system.

Pros.

Utilize every feature of the device

When creating a native app, the developer has full control of device, thereby making it possible to fully utilize the potential of the device [7]. This also includes the UI, which often is an issue for apps that is not created with the native framework of the device. Users of smartphones often talk about the "native look and feel" of an app. This is given when creating the app using native development technology.

New features will be available immediately

If being a first mover on utilizing new features in apps is a high priority for a company creating apps, then there is no way faster way to get it done, than using native development.

Fast performance

Performance is often a critical property of any kind of software product, but is

especially important when there is user interaction involved. As smartphones are equipped with touchscreens, it is fairly important that the device reacts fast and properly when the user touches the screen. Slow or incorrect reactions to a user interaction can really hurt the overall user experience.

Cons.

No cross-platform possibilities

As mentioned in the beginning of this chapter, there exists several software platforms for smartphones today, and they all use a different set of programming languages for their native apps. This means that any native app build for one of these platforms, cannot be used on a different platform. The consequence of this is that, in cases where a company wants to distribute their apps on more than one platform, they would have to develop the entire app from scratch on each platform they would like to support. This is undesirable for a couple of reasons:

- This cost a lot of money, as the implementation time is proportional to the number of platforms supported.
- Supporting a single application on multiple platforms, where each platform has its own codebase, is extremely hard to maintain.

Development for each software platform has its own learning curve

The programming languages uses for each software platform are quite different, so if a company wants their apps to support multiple platforms, the developers would either have to climb multiple learning curves, or they would need separate developers for each platform. In the community of mobile app development, the learning curve of programming languages such as Objective-C for iOS apps and Java for Android apps is considered to be longer, than e.g. the learning curve of HTML5 based apps.

2.2.2 Mobile Web Applications using HTML5

As native apps, written in the native language, represent one extreme of type of mobile apps that exists today, the other extreme is the mobile web applications. A mobile web application is basically a webpage that is wrapped in a shell of native code. The shell consist of a native program, which only performs the task of opening a WebView, which is a visual component used to display a

web browser inside a native app. Using the WebView on a device will open the default web browser on the device. This way, the users will be under the impression that they are running a native app and not just opening a web page. However that is essentially what happens behind the scenes.

These types of applications are built using HTML5, CSS and JavaScript. When it comes to being platform independent, it does not get any better than with this type of app. All it requires in order to run on a device is a browser that supports HTML5. Since HTML5 is combined by a lot of different features, it is possible for browsers to only support parts of the API. Therefore it is possible that a HTML5 application cannot be run correctly even though the browser claims to support HTML5. As of February 2012, Firefox Mobile 9 is the browser that offers the best support of HTML5 in terms of numbers of features supported [14].

Pros.

One codebase

One of the most important benefits of the fact that mobile web applications are essentially just a web page, is that the developers only need to develop one web page in order to reach out to all of the major smartphone platforms, as a web page is platform independent by nature. In the case where a company wishes to create an app that runs on multiple platforms, creating a mobile web application greatly helps to reduce the time spent on the development of the app, and thereby reducing the overall cost of creating the app. This also helps the company getting their product ready in a much shorter time frame. This aspect can be very important, if the company has the intention of securing market shares in a competitive market. Being able to produce a product in a short time frame, can give a company a competitive advantage, as being the first-mover is often a key to success.

Fast deployment process

As mobile web applications are normally not able to enter the native app stores, there exists special app stores where mobile web applications are allowed to enter. These app stores have a fast deployment process since there is no “review and approval” process whenever a new deployment needs to take place. Compared to the “review and approval” process that Apple has on their App Store, this is a thing to take into consideration if the app is required to be supported on Apples products. The process on the Android Market is not as strict as the Apple App Store, but some work is still required in order to make the app ready to enter the Android Market.

Cons.

Access to native app stores is not guaranteed

Despite the fast deployment that mobile web applications have when eluding the native app stores, it is still a big drawback that they are not guaranteed to enter the native app stores, should they wish to. There are examples of HTML5 apps being rejected from the entering the Apple App Store, but also examples where they are allowed to enter. It depends on an individual evaluation of the app, and therefore it cannot be guaranteed that the app will enter the app store. Apple has a much more strict approving policy than Google has on their Google play store, but still there are no guarantees.

There are several reasons why it is a drawback not to enter the official app stores. First off, the number of users using these app stores is very high. Both Android apps from Google Play, and apps from Apples App Store has surpassed 10 billion downloads going into 2012 [4] [12]. So that would be missing a huge opportunity to reach customers who only uses native app stores to search for new apps.

Secondly seen from a business point of view, the native stores has a much better business model, than the app stores for mobile web applications. In native stores it is possible, and very normal, to charge a fee each time a customer downloads the app. The stores for mobile web applications are essentially a web page that provides a collection of links to the various mobile web applications that is added to the store [7].

The native app stores have the advantage that customers consider it natural to pay for an application downloaded from this store, where as they are much less likely to pay for a link to a mobile web application. So the in order to make profit of a mobile web application, the company who owns it, will mostly have to rely on advertisement in the application, like the ones known from normal websites.

Slower than native

As mentioned, mobile web applications is created with the use of HTML5, CSS and JavaScript. It is in the JavaScript part that all the logic is placed and where the background work is being processed. Therefore it is in this area, that performance issue can arise. When a company decides to release a product that supports different platforms, it is desired to give the customers an equal experience, no matter which of the supported platforms they are using. Ideally the performance should be uniform on all platforms. This can be hard to achieve since different browsers are using different JavaScript engines. In fact there exist

a lot of different JavaScript engines today. And due to this fact, some browsers simply run JavaScript code faster than others [1]. This makes it very hard for companies to give performance guarantees about their application.

Not all features are accessible

There are a lot of features that can only be used through native API calls [7]. One of the areas where a mobile web application meets it's limits is when CPU-intensive processes are needed, as hardware acceleration and multithreading is not running as good as on a native application [7]. Another area is the lack of native look and feeling that the user expects from a smartphone app. As some native UI features are not accessible with this type of app, the developers must try to mirror the behaviour to the best of their abilities, or accept that the app is not looking and feeling native. This has been an issue for many users and developers of mobile web application ever since the release of HTML5 [1]. New features keep getting supported by HTML5 based apps, but naturally there will always be a certain timeframe, before brand new native features are getting supported by HTML5 based apps. The native apps will always be one step ahead.

2.2.3 PhoneGap

HTML5 has definitely opened a lot of doors for cross-platform development of mobile applications. One of the big hurdles has been that these mobile web applications, without any native wrapping, can have problems getting accepted into the native app stores, especially in Apples App Store. So in order to overcome this, the open source project "PhoneGap" was established. PhoneGap provides the native wrapping that is needed for apps to even be considered accepted into the app stores, but also it provides a big API which gives the developer access to many native functionalities. Many of which normal HTML5 mobile applications do not have access to. These are the main reasons that has paved the way for the project known as PhoneGap.

Pros.

Many native device features are supported

The goal of this project is to close the "gap" that exists between mobile web applications (HTML5 applications) and native applications. In order to do this, the developers of PhoneGap has created their own API, which can be called using JavaScript on a HTML5 page. The PhoneGap framework contains native code pieces to interact with the underlying operating system and pass information

back to the JavaScript running in the WebView container [1]. This way the developer is able to access the native features on the device, such as Geolocation, Accelerometer Camera, Contacts, Database, File system and more [9]. In fact, most of the features for the popular platforms are supported.

Access to native app stores

With the use of the native WebView feature, what PhoneGap essentially does is to wrap a native shell around a mobile web application. This greatly improves the likelihood that the app is getting accepted into the native app stores, and thereby enjoying all the benefits that the native app stores offers.

Many platforms are supported

By the time this thesis was created, PhoneGap supports a total of seven different platforms. The supported platforms are iOS, Android, Windows Phone, Symbian, Samsung Bada, webOS and BlackBerry. This means that if a company needs to make an app that must be able to run on all of these platforms, with the use of PhoneGap they would only have to develop the application once, instead of a potential of 7 times.

Looks more like a native app than a web page

PhoneGap makes it possible to make a web page and display it in the browser of the device, without having it look like a normal web page. This is done by hiding the browser frame around the content of the web page. Thereby the user will not notice that it is in fact a web page he or she is using. Having the app installed on the user's device, instead of just being a bookmark in the browser of the device, enhances the user's perception that he or she is using a native app. An app made from PhoneGap will generally not meet this expectation.

Cons.

No native UI support gives a generic look on all platforms

As of may 2012, no native UI elements are being supported by the PhoneGap API. This means that a PhoneGap application will look the same way, no matter which platform it is running on. This is quite a downside to this type of app. The vast majority of app users have a well defined standard for what a certain type of app should look like and behave, upon user interaction with the device. A common user of Android apps will expect any apps found in Google Play to behave like a native Android app.

New features take time to get supported on all platforms

Because PhoneGap covers so many platforms, it takes a while to get new features implemented for all the supported platforms. This is the one of the tradeoffs the people behind the PhoneGap project has chosen to make. So if access to brand new features is really important to the company responsible for the app, PhoneGap might not be the best choice.

PhoneGap is considered a hack

Many people in the industry of mobile applications consider this "bridge" that the PhoneGap project has made between mobile web applications and native apps, to be a hack [1]. The nature of a "hack" is typically that it is a temporary way of overcoming a problem in an inelegant but effective way. Even though hacks are usually temporary solutions, there are currently no indications of PhoneGap's future being in any danger. But the industry's perception of PhoneGap could perhaps hold off some companies from using it professionally.

2.2.4 Appcelerator Titanium Mobile

The company Appcelerator has developed a platform which is used to create applications for tablets, smartphones and desktops. The platform is called Appcelerator Titanium. For this platform, they have created a framework which specifically focuses on development of apps for the iOS and the Android platform. This framework is called Titanium Mobile, and was launched in 2008, supporting only the Android platform, since Apple did not allow third party development tools to use the iOS API. However in September 2010, Apple relaxed their restrictions in this area, making it possible to create iOS native apps with Titanium [2]. This meant a huge deal for Appcelerator as they now had developed a cross-platform framework that made it possible to create apps for iOS and Android with the same codebase.

Essentially the way Titanium Mobile works is that it uses JavaScript as a main programming language, and through the Titanium API, the developer can access native methods and properties [17]. Appcelerator has also developed their own IDE called Titanium Studio, which must be used to create Titanium apps.

Pros.

One codebase for iOS and Android apps

Appcelerator has decided to focus their attention on the two most popular platforms of today's smartphones, namely the iOS and Android platforms. With Titanium Mobile, developers have the possibility of creating a single app, that

can be deployed to both iOS and Android devices. This means less development time, easier maintenance and a smaller cost price.

Executing native

An application written in Titanium Mobile is compiled into "native code", in the sense that every method and property used in the JavaScript code is being executed in the same way as if it was called from a native program written in either Objective-C for iOS or Java for Android [17]. This means that the performance of the app could potentially be just as good as apps written entirely in native languages. This will however depend on the performance of the JavaScript engine that Titanium is using, and how the developer chooses to build the app. Appcelerator recommends doing as much as possible with native API calls, and keeping the amount of "heavy work" in the JavaScript part to a minimum.

Native look and feel

The Titanium Mobile API also includes the UI parts for both iOS and Android. Therefore it is possible to have a Titanium Mobile app that looks and feels just like if it was build in Objective-C or Java. This is a quality that many users of smartphone apps consider to be quite important. However, not all native UI elements are included in the Titanium Mobile API as of June 2012, but a very large part of the native elements are there, and new features continue to be added to the Titanium Mobile API as time goes by. Still there exists some UI customizations that is not currently possible using Titanium Mobile [7].

Massive API

The API of Titanium Mobile is quite massive. As of release 1.8.2, the API consists of more than 5500 native methods, and more than 3000 native properties.

Widely used

Titanium is widely used today. According to Appcelerator, big brands such as NBC, eBay, MTV, Budweiser and Jaguar, have all chosen Titanium as their platform to build applications for tablets, desktops and mobile devices [2]. Also the community of developers using Titanium consist of more than 300,000 people worldwide.

Faster to build applications

Appcelerator claims that building a mobile app with the use of the Titanium Mobile platform will be 70% faster than building the same app using Objective-C or Java [3].

Free of charge

The Titanium Mobile SDK as well as the Titanium Studio is available to everyone for free.

Cons.

Not all UI elements are supported

As mentioned, it is not all the native UI elements that are supported in the Titanium Mobile API. Some custom elements are only useable when creating the app in Objective-C or Java.

Currently only supports iOS and Android platform

Appcelerator has focussed their efforts on the iOS and Android platforms. As of may 2012, Appcelerator is still working on supporting the BlackBerry platform as well, but the development is still in closed beta. Titanium Mobile might be able to support more platforms in the future, but currently only BlackBerry support is in scope.

2.2.5 Mono

Mono is an open source project headed by the company Xamarin, which specializes in making the .NET framework compatible with other platforms, than the windows platform. They have created implementations of Mono which can be used to create apps for iOS and Android based smartphones. The implementation for use with iOS is called "MonoTouch" and for use with Android systems the implementation is called "Mono for Android". It allows native development of iOS and Android apps with the use of the C# language.

Pros.

One language for the leading platforms

The nature of Mono is to make it possible to build software on non-windows platforms. With the help of MonoTouch and Mono for Android, developers do not need to worry about learning a new language when switching between iOS development and Android development.

Non-UI code can be reused across platforms

While the UI elements of MonoTouch and Mono for Android are unique for each platform, the business logic used for an Android app can be reused in an iOS app, as well as a Windows Phone app. This adds value to for companies who want their app to support these platforms. However, they would still need to develop the UI on each of the platforms separately. As of may 2012, it is only these 3 platforms that the Mono framework is supporting.

Native look and feel

Much like Titanium Mobile, the Mono frameworks also makes use of the native UI elements of iOS and Android. This makes it possible to have the native look and feel that is so important to many customers. MonoTouch even integrates to Apple's own IDE called Xcode, allowing the developer to use the design tools that comes with Xcode. Likewise for Android, Mono for Android allows the use of the layout tool DroidDraw, to create the UI for Android apps [20].

Cons.

Requires a commercial licence to deploy to devices

Even though the Mono project is an open source project, the two implementations for smartphone development is released under a commercial licence. This means that a licence is needed in order to deploy projects to a device or to a native app store. MonoTouch and Mono for Android do not share the same licence. Each licence cost in the range from 400$ to 2500$ per year depending on the size of the company buying it and the level of support that is included in the package.

UI elements cannot be reused across platforms

As mentioned earlier in this section, the implementations of Mono for both Android and iOS, does not include a cross-platform implementation of the UI elements. Each platform has their own API for accessing their UI elements. This means that if a company uses Mono to develop an app for both iOS and Android, they must create the UI for both platforms separately. On the bright side of things, both implementations use the same programming language that is C#.

2.3 Comparisons

When building a mobile application there are many choices to make and one of the most important ones is to choose which technology that is actually going to be used for building the app. The first thing that comes to mind is to use the tools that originally were intended for this purpose, which are the native tools. But that is not always the best choice in every situation. Every time a mobile app needs to be built, the company responsible for the app should have a look at the requirements for the project at hand, and determine which technology is the most appropriate choice to go with. As mentioned earlier, there is no silver bullet for this type of project.

On figure 2.2 is an overview on how the different frameworks compares with each other.

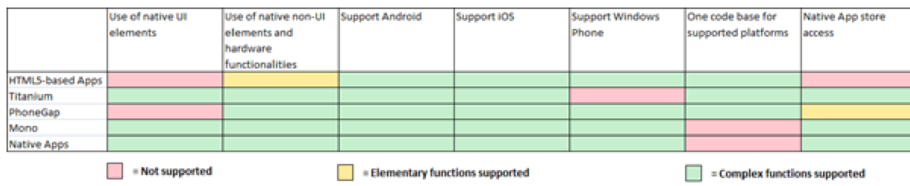


Figure 2.2: Comparison of the frameworks

To make a choice between the different platforms, one must look at the requirements for the Frequency Registry app.

Requirement 1: Basic functionalities of the existing website must be supported.

- All the platforms are able to fulfil this requirement.

Requirement 2: Support for multiple platforms, with iPhone and Android as a minimum.

Primary choices:

- Native applications trivially support all platforms.
- HTML5 supports the platforms which has a browser that supports HTML5. (Almost every device today lives up to this)

- PhoneGap support great variety of platforms, including iPhone and Android.

Secondary choices:

- Mono support iPhone and Android, and Windows Phone. Other platforms are not supported.
- Titanium support iPhone and Android.

Requirement 3: Be easily available to customers.

Primary choices:

- Native applications have access to native app stores which are the easiest way to reach customers.
- Currently PhoneGap can access native app stores.
- Apps built with Mono can access native app stores.
- Titanium apps can access native app stores.

Secondary choices:

- HTML5 apps are currently struggling with getting access Apple's app store, and cannot be sure to fulfil this requirement.

Requirement 4: Low development cost.

Primary choices:

- HTML5 apps only need to be built once in order to be available to all its supported platforms. It is also very similar to creating a normal webpage, which is a well known technique. This enables web developers to make a HTML5 app very quickly.

Secondary choices:

- PhoneGap is a "build once deploy everywhere" framework. However since the framework is new it must be expected that there will be some learning curve for an average developer.
- Titanium is also a "build once deploy everywhere" framework. Like PhoneGap, Titanium is a quite new framework, and will have some learning curve.

Tertiary choices:

- Mono has the disadvantage that the UI for iPhone and Android cannot be reused between the two frameworks. Therefore the cost of creating an app for these two platforms will take significantly more time (potentially twice the time) compared to a "build once deploy everywhere" framework.

Quaternary choices:

- Native apps must be built completely from scratch for every supported platform. Seen from this perspective, it makes native apps a very undesirable choice if multiple platforms must be supported, and if development time and cost is an issue (which it normally is).

Requirement 5: Able to access hardware functionality for future extensions of the app

Primary choices:

- Native apps will naturally be the best choice when it comes preparing for future functionality.
- The Titanium Mobile framework has done a very good job supporting the various hardware functionalities of both Android and iOS devices. With this framework the app are well prepared for any future extensions that might be needed. The only thing that can be an issue is the time it takes when a brand new feature is being released till it gets supported by Titanium Mobile.

Secondary choices:

- PhoneGap does not support all hardware functions on all its platforms [9]. However on iOS and Android systems, it currently supports quite a lot of the hardware functions.

Tertiary choices:

- The Mono frameworks have a list of limitations to both iOS and Android systems [22] [21].

Quaternary choices:

- HTML5 apps are by far the type of apps where the support for native hardware functionality is lowest. As mentioned, this "gap" between the HTML5 website and native features, is the reason that project PhoneGap was initiated.

2.3.1 Choice of development platform

One way to make a decision between the development platforms would be to give each platform a score to reflect the rank of choice they received on each requirement. One example could be to give 4 points for each primary choice, 3 points for each secondary choice and so forth. This would lead to the following scores:

- Titanium Mobile 14 points
- PhoneGap 14 points
- Native 13 points
- HTML5 12 points
- Mono 11 points

Based on the scores given to the development platforms on each requirement, it gives a tie on the first place between Titanium Mobile and PhoneGap. One of the main differences between Titanium Mobile and PhoneGap is that Titanium is going to look and feel more native than PhoneGap, since Titanium Mobile apps are compiled into native code, and Titanium Mobile also makes use of the native UI elements, which PhoneGap does not use at all.

PhoneGap does however support many more platforms than Titanium Mobile does. However this won't make much of a difference, considering that Android and iOS is sitting on 89% of the smartphone market in Denmark.

Titanium Mobile also has the advantage that it is being used by big brands, and has proven that it can be used on a big scale.

Considering the cons and pros, I choose Appcelerator's Titanium Mobile as the framework to create Erhvervsstyrelsen's new mobile app.

CHAPTER 3

Design of The System

This chapter deals with the overall design of the system, and covers both the conceptual design of the graphical user interface of the app, as well the task of incorporating the app in the existing environment of the Frequency Registry system.

Section 3.1 explains the standard structure of a Titanium Mobile app in detail. This is done since the structure is a central part of the conceptual design of the graphical user interface, and many of the terms introduced here is used throughout the rest of this chapter.

Section 3.2 presents the conceptual design of the graphical user interface.

Section 3.3 covers establishment of the communication between the app and the existing system. The various models for doing this are presented and it is found optimal to use some kind of service for the app to communicate with.

Section 3.4 is focussing on choosing the right kind of service for this project. The most popular types of services of today are presented, and the most appropriate one is chosen.

Section 3.5 raps this chapter up by presenting a concept design of what the final production environment would look like, incorporating the new Frequency

Registry app.

3.1 Structure of a Titanium app

Every application build with Appcelator's Titanium Mobile SDK is based on it's core structure. On the highest level, the apps consist of windows and views. Windows can contain views. One window can contain as many views as needed. And a view can contain other views inside it. These concepts are shown on the following on figure 3.1.

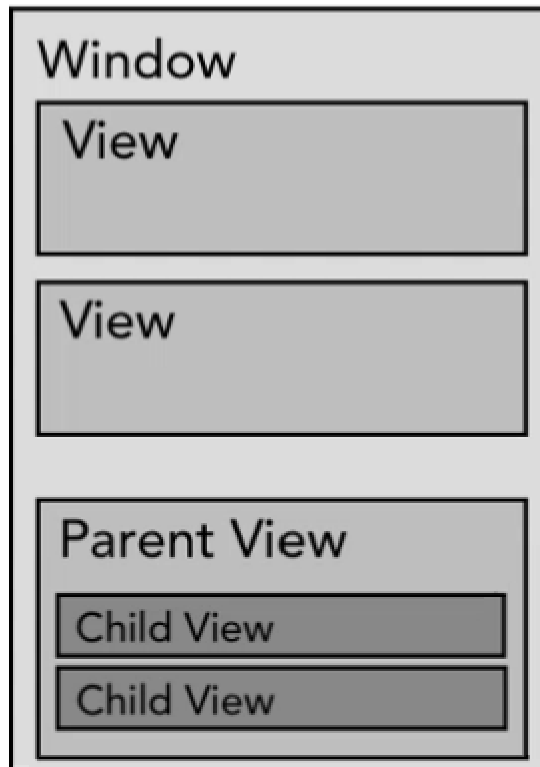


Figure 3.1: Basic structure of a Titanium Mobile app

As shown in figure 3.1, the window is at the highest level of a Titanium Mobile app. Without any windows, the app would show nothing. All Titanium Mobile apps therefore have at least one window. An app can have as many windows as the programmer wishes. The figure also shows that windows can contain views.

Each window in an app can contain as many views as needed. By default the views are private to the window which they are added to. A view can only be added to one window at a time. If a view is added to window1, and then later added to window2, it will no longer be visible on window1. This is because an object in Titanium Mobile can only belong to one specific object. A child object can only have 1 parent object, no matter if the object is a view, a window or some other construct.

As mentioned a view can also contain other views. The top layer view is called the "parent view", and each parent view can contain several views inside its dimension boundaries. These views are referred to as its "child views". On figure 3.1 there is shown a parent view, which contains two child views. Each of these child views can take on the role of a parent view to another set of child views, and so on.

One of the most commonly used layout components in applications for mobile devices, are the so called "tabs". Tabs are often located at the bottom of the screen, and works a lot like tabs in modern day Internet browsers. Using tabs are a good way to help the user navigate between the different windows in the app, without the user loosing the overview of the app.

Figure 3.2: Titanium Mobile app with 2 tabs

On figure 3.2 is shown the concepts of a tabbed application. The tabs are placed

in the button, and one of them is always active. To the left, it is “Tab 1” that is active. On the right, “Tab 2” is the active one. The tabs are both added to an object called “TabGroup”. The TabGroup manages the tabs, and can manipulate with them in various ways. It is considered good practice, only to have 1 TabGroup in a Titanium Mobile app. It is possible to have more than one TabGroup in the same app, but that complicates the navigation of the entire app, as well as the structure.

Windows can be added to a tab. Just like a view only can belong to one window at a time, a window can only belong to one tab at a time. Each tab can contain many windows. Tabs are very handy to use when navigating through windows, as they provide navigation buttons on the windows title bar. Whenever a new window is opened in a specific tab, the title bar will automatically show a button which will take the user back to the previous window. If several windows are opened in the same tab, each window will have a back-button leading back to the previous window, forming a chain of windows. This makes navigation easy, and the programmer of the app is free from creating navigation buttons manually and deal with all the hassle of making sure they point to the correct window.

3.2 Design of user interface

The design of the user interface of any application has always been a very important aspect of a project. A bad user interface can ruin a perfectly well functioning application. As mentioned earlier in the introduction to mobile computing, people are expecting to be able to use computers as they are on the move. This naturally leads to a few additional requirements that arise, solely from the fact that the application is running on a handheld mobile device.

- The app should be simple, as people often don’t have much time to spend on using the app on the move.
- It should be very intuitive.

Based on these requirements, the following figures show a conceptual design of how the app could look like.

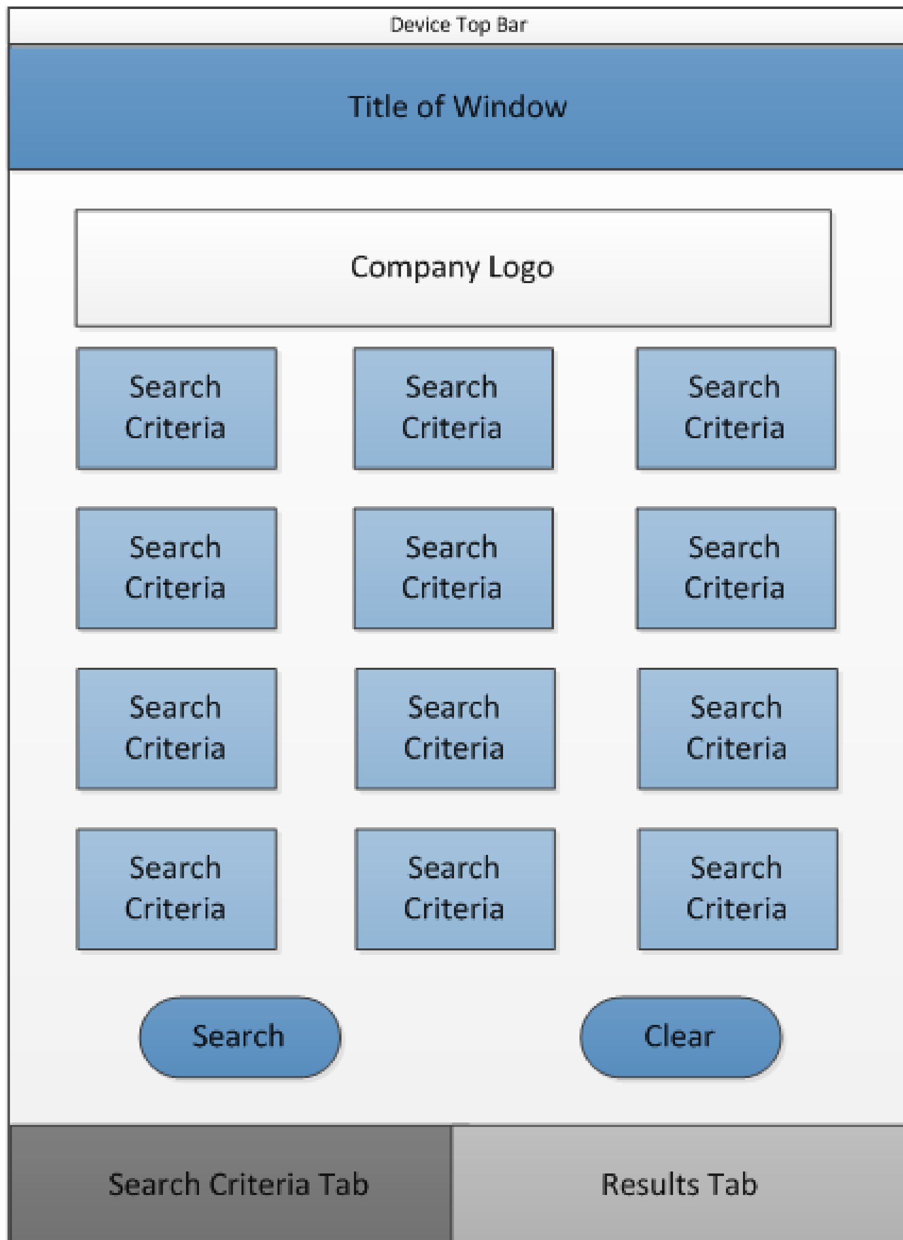


Figure 3.3: Startup window of the App

Figure 3.3 shows the first window that the user will be presented with, when starting the app from his or her mobile device. At the top of the screen, a

device dependent bar is shown. This bar is created by the operating system of the device, here denoted as "Device Top Bar". It displays information about the connection, the current time, and the status of the battery. It is possible to hide it, and let the frame of the app utilize the entire screen, however an argument for keeping it visible is that it would be nice to be able to easily check the connection of the network, while using the app, in case the app is not responding or behaving in an unexpected manner. If the device top bar was hidden, the user would have to exit the app to check the connection or the time and battery, for that matter. Therefore it is more convenient to keep it visible.

Below the device top bar is the title bar. This is meant to show the title of the window currently being displayed. This will be used throughout the entire application. On the start up page it will display the name of the app, which is "Frequency Registry". As the user navigates through the windows of the app, the title bar will change correspondingly.

Below the title bar is the logo of Erhvervsstyrelsen. Showing the logo of the company owning the app, helps to give it a more official look.

Next are all the search criteria that can be set in order to perform a search in the Frequency Registry. Each criterion exists of a label and a textbox for input. If the users do not wish to fill out all of the criteria, he or she can just leave out some of them, and the search will only take the filled ones into account. Upon clicking on any of the textboxes, the app will bring up an appropriate keyboard to let the user type in the criteria. On figure 3.4 is an example of this, showing a numeric keyboard.

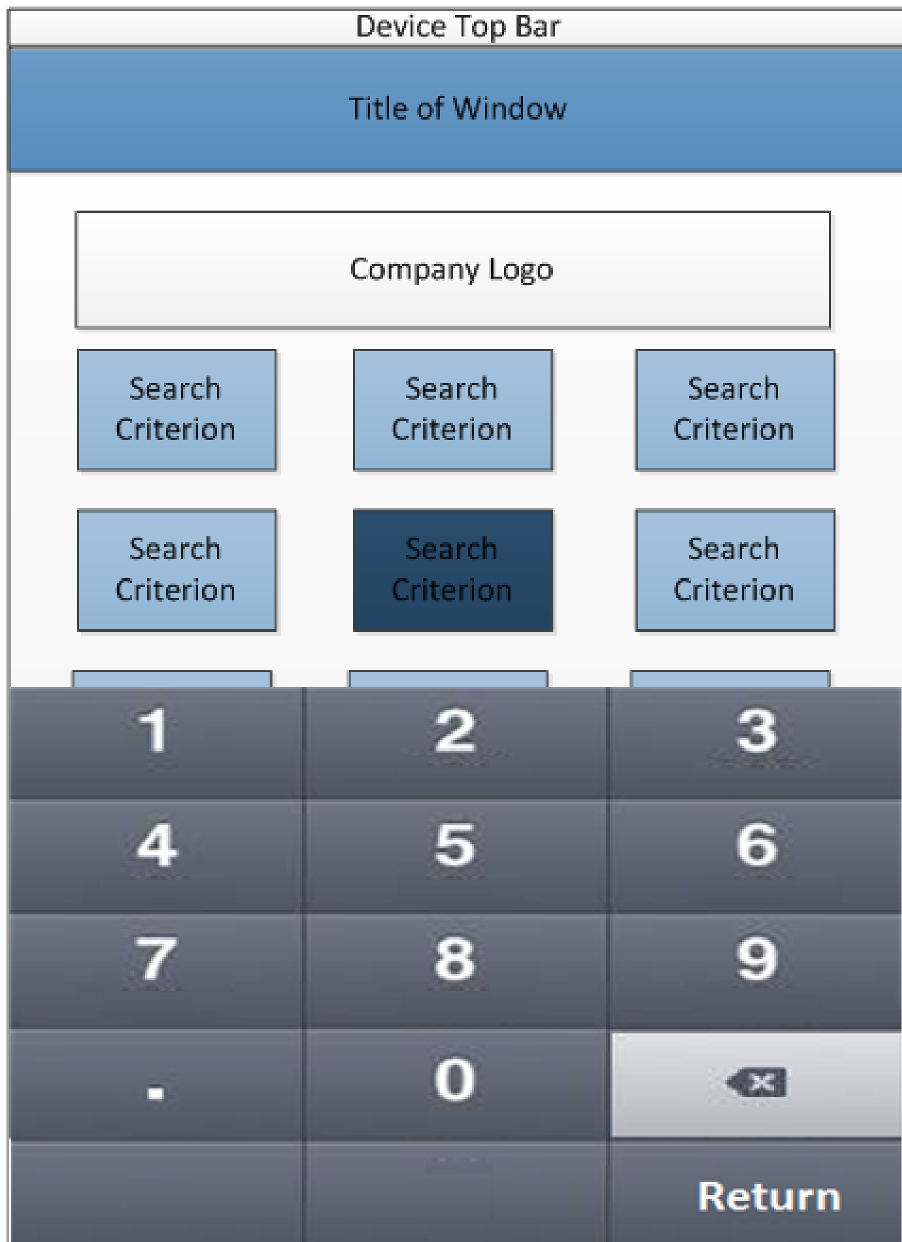


Figure 3.4: Enter value of a search criterion

A numeric keyboard is shown and will allow the user to enter the value for the criterion. Once the user clicks outside the keyboard, or clicks on the return key, the value is stored in the textfield, and the keyboard is hidden again. This is a good way to make sure the user does not enter an invalid value into the textbox.

The start up window also contains 2 buttons. The “search” button is used to perform the actual search, once the preferred criteria have been filled out. The “clear” button is used to clear all the criteria at once. A click on the search button will change the screen and open the results window, which is covered later in this chapter.

At the very bottom there is a bar that shows the tabs of the app. As mentioned, these are very handy for navigation, and are therefore often used in apps. This app will have two tabs, one for the search window, and one for the results window. The search tab will only contain the start up window, but the results tab will include several windows. The user can, at any given time, go to the search tab, change the criteria and perform a new search, and the app should bring the user to the results tab, showing the newest search result, as seen on figure 3.5.

Figure 3.5: A list of the search results

As seen on figure 3.5 the results tab is now active and is showing the results window. This window shows a list of all the cases that was found by the performed search. If there are more results than fits the window, a scroll-function and an index would be needed to allow the user to browse through the many results. In a real live system, the results could be in the thousands, therefore it might be relevant to put a maximum limit on how many elements the list should contain.

If the user decides to take a look at the details of one of the cases, the user just needs to tap it and this brings up the details on that specific case. This is illustrated on figure 3.6.

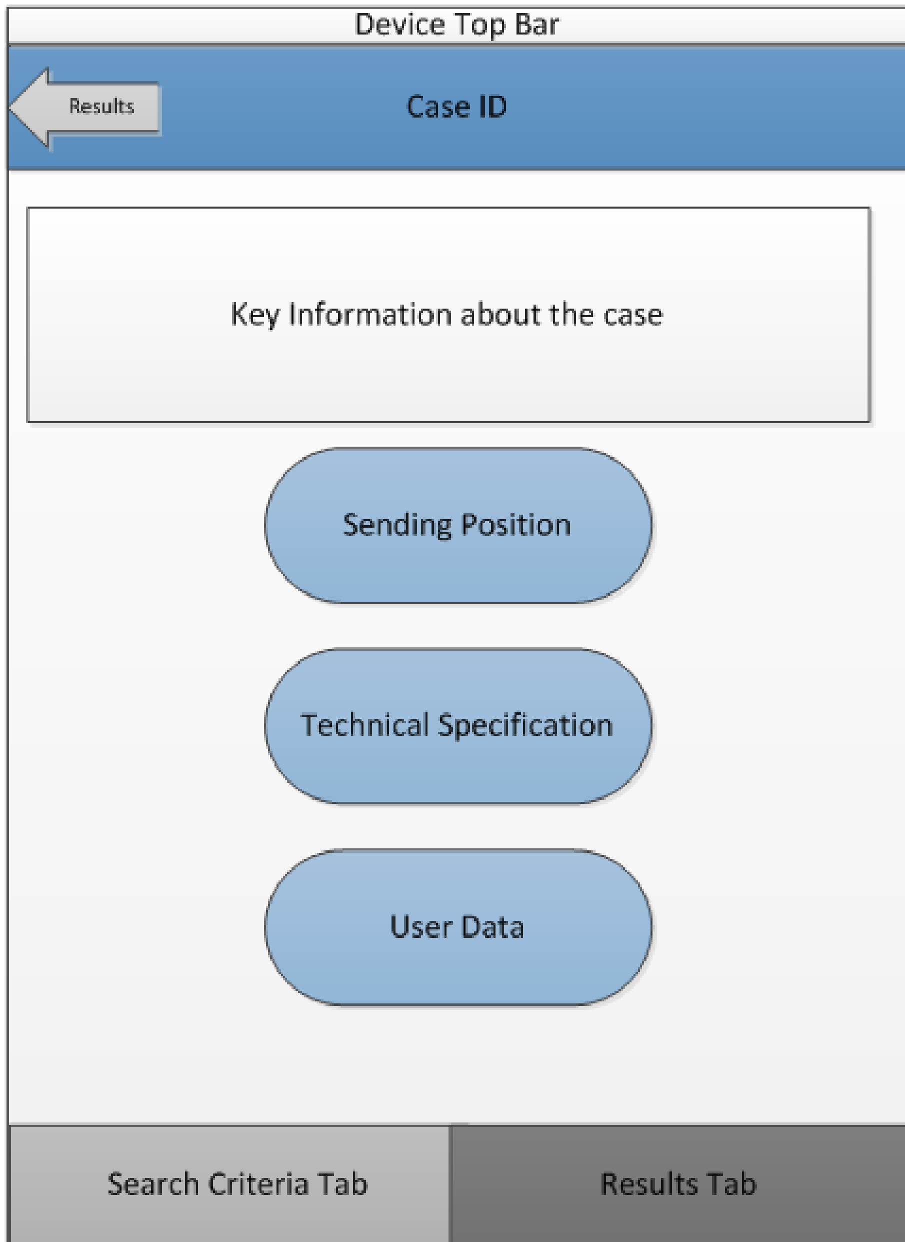


Figure 3.6: Details on a specific case

Figure 3.6 shows the window that is shown when the user taps a specific case in the results window. This window shows the details of the case tapped by the user. The title bar is showing the case ID of the case, and does also include a back-button to let the user navigate back to the results window. The text of the back-button is always showing the name of the destination window, to which the back-button will take the user, if clicked. Here it says “Results”.

Below the title bar some of the most important information about the current case is shown. Each case has a lot of details, and it might not be appropriate to list them all in one big view. Therefore the most commonly used information is placed in this view. Below the key information view, three buttons is placed. These buttons can be tapped to show even more information on the current case. As mentioned earlier, the information of a case is grouped into three categories (sending position, technical specification and user data). As seen, each button represents each of the three categories. Upon clicking on each of the buttons, the user will get full details on each of these categories. As an example, the user might want to look into the details of the sending position, and therefore taps the “sending position”-button. The result is seen on figure 3.7.

Figure 3.7: Details on the sending position of a case

Figure 3.7 shows the screen that is shown once the user taps the “sending position”-button. All the data which is related to the sending position is listed here. The user can return to the overview of the case, by tapping the back-button in the title bar, which now has the ID of the current case, as it’s text value. The window for displaying details on the “technical specification” and “user data” is using the same layout as the “sending position” window does, and is not shown here to avoid redundancy.

As mentioned earlier, if the user decides to make a new search based on new criteria, then he or she can always tap the Search Criteria tab, and the app will show the start screen again, allowing the user to make a new search.

3.3 Communication with the existing system

The frequency registry as it is today, basically consist of a database and a website as seen on figure 3.8.

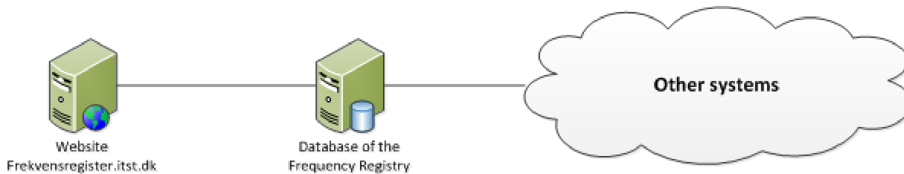


Figure 3.8: Existing Frequency Registry system

The database server communicates with other systems in order to keep the database updated, but the details of this communication are not of importance and are therefore left out of the scope of this project. However it is in the scope to consider how the app will be communicating with the database in a live production environment. This can be achieved in a number of ways, which is being presented and evaluated in the following sections.

3.3.1 Using the existing website

As figure 3.8 shows, the website is communicating with the database directly. This means that a potential app version of the Frequency Registry could potentially use the website to communicate with the database instead of implementing its own form of communication. However, using the website to communicate with the database seems like a bad idea, for a number of reasons:

- The performance of the app will suffer, since the website would have to listen for request coming from the app, while still servicing the normal users of the website. The resources available to the website would have to be split in some way, to make sure the requests coming from the app would get handled. And the performance of the website would also decrease for the very same reason.

- This introduces an extra dependency to the system. The availability of the app would rely on the webserver hosting the website. If this server breaks down in some way, or just down for scheduled maintenance, the app will not be available. This is very undesirable, since there would be scenarios where the app and the database are working fine, but are unable to communicate because something has happened with the website. This is very undesirable.

- Changes would have to be made to the existing website. This would give the undesired side effect, that whenever changes were made to the communication between the app and the database, it would require a change in the website. This would lead to a new deployment of the website, thus making the website unavailable during this time.

- Implementing a connection between the app and the database would require a considerable amount of work, since the website is not built to handle request from an app. Errors in the redefined website would affect both the app and the existing website.

The only good reason for choosing this solution is that the communication with the database is already implemented, and could probably be reused. However this is still a very poor solution to the problem, due to the amount and significance of the bad reasons just listed. Another solution must be found.

3.3.2 Connecting directly to the database

The existing database of the Frequency Registry is a Microsoft SQL server database. Currently the Titanium Mobile API does not offer any support for connecting to this type of database. There are supports for SQLite databases however, but this type of database has very limited functionality and does not support stored procedures, which are heavily used in the real Frequency Registry database. Microsoft do have a framework, called Microsoft Sync Framework, to do synchronization between these two types of databases, but this would not solve the problem with the missing support for the stored procedures. And should this work, it would still not achieve the objective of connecting directly

with the real database, which was the whole idea in the first place. And the task of keeping these two databases in sync all the time, will likely impact the performance of both the app and the existing website, since the real database would be doing a lot of synchronization.

3.3.3 Using a service

A very common way to solve this type of issue is to use some kind of service. Usually this type of service will only have the task of handling incoming request from a specific application, and is therefore often very simple. The reason for this is to avoid unnecessary decrease in performance of the overall system. Therefore the complexity and the amount of work of the service should be kept to an absolute minimum. In this setup, the app will use the service for all kinds of interactions with the database, as shown on figure 3.9.

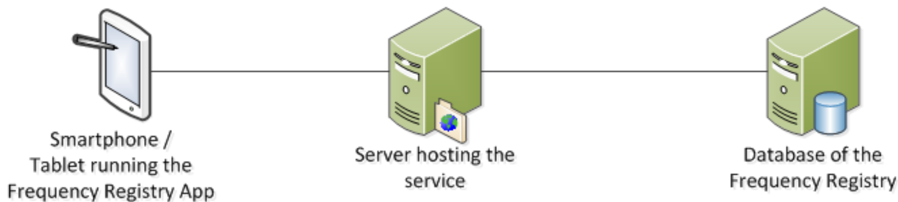


Figure 3.9: Interacting with the database using a service

As seen on figure 3.9, the Frequency Registry App communicates directly with the service at all times. As mentioned earlier, this approach is very widely used, and has no real downside, and thereby the obvious choice for this project. However the service could be made in a number of ways. To optimize the performance of the app, the right type of service must be selected.

3.4 Choosing the right kind of service

Since the Frequency Registry app should be able to communicate with the service over the Internet, the type of the service should be some kind of web service. There are several ways to create a web service today, and the one which is most suitable for this project must be chosen. Two of the mostly used forms of web services today are the asp.net services (sometimes referred to as ASMX services or SOAP services), and WCF services. Some people like to refer to services that apply the RESTful principles as RESTful services, although it can be created

with a framework like WCF, and therefore it is just a specific type of WCF service.

3.4.1 Asp.net services

Asp.net services are considered by many to be the first real type of web service. The web service is described by a language known as the Web Services Description Language, abbreviated WSDL. Most importantly it describes the operations which the web service is exposing, and the endpoints at which the service can be contacted through. These endpoints are used by applications that wish to use call the operations of the web service. These applications are often referred to as “clients” and are said to be consumers of the web service.

In order to communicate with clients, this type of web services makes use of the web protocol SOAP (Simple Object Access Protocol), and thus many people refer to this type of service as a SOAP service. The SOAP protocol makes use of the Extensible Markup Language known as XML, to create the messages used for communication. Each message consists of an optional SOAP-header and a SOAP-body, and is wrapped inside a SOAP-envelope. The simplest form of a SOAP message is as follows [13]:

<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP:Body>

<m:HelloWorld xmlns:m="http://www.soapware.org/"/>

</SOAP:Body>
</SOAP:Envelope>

The details of this message is not important for the purpose of the project, however the size of the message is worth noting. It takes 5 lines and a lot of characters to send a simple “HelloWorld” message, using SOAP. The optional header is even left out in order to get the message to be as small as possible. This is important in the world of mobile computing, where the network connection often is through the telephone network and the bandwidth tends to vary a lot. This is especially an issue in areas that is not covered by the 3G network. Therefore the messages between the web service and the Frequency Registry app, should be as small as possible, and should primarily consist of the actual payload and only a minimum of wrapping. The wrapping of this SOAP message is quite large compared to the actual payload. However the ratio between the wrapping and the payload will be evened out as the payload increases in size.

The SOAP web services also makes use of the UDDI framework (Universal Description, Discovery and Integration), which makes it possible for clients to discover SOAP web services over the Internet. The UDDI framework is supported by all the major software houses and has great potential [19]. However in a project like the Frequency Registry app project, UDDI will not be needed, since the service is created for the sole purpose of servicing the Frequency Registry app. There is no need to discover the web service, since it is part of the project.

3.4.2 WCF service

Windows Communication Foundation or simply WCF, is a framework created to build service-oriented applications. It was released together with Windows Vista, back in 2007 [8]. The WCF framework supports the creation of web services. Web services created with WCF has a lot of features, apart from just sending messages back and forth to its consumers. Some of them are multiple message patterns, tight security, reliable message transfer, message encoding and the use of transactions. The WCF web service is seen as a replacement for the old asp.net web service, as it can do the same things, plus a whole lot more. A WCF service is basically an improvement of the old asp.net service in every way. In addition, the WCF framework also supports creation of RESTful services.

3.4.3 A RESTful service

For many years, SOAP has been the preferred way to exchange messages between web services and their consumers. But in the last few years, a new method has been gaining popularity. This method is using the RESTful principles, and web services applying these principles are said to be RESTful services. The RESTful principles are as follows:

Explicit HTTP method use.

RESTful web services should use HTTP methods explicitly, according to the definition of the HTTP version 1.1 in RFC 2616. This means that the four basic methods, create, read, update and delete (known as CRUD), should be used as follows [10]:

- Creating a resource on the server should be done using the POST command.

- Retrieving a resource should be done using the GET command.
- In order to update the state of a resource, the PUT command should be used.
- To remove a resource, the DELETE command is the one to use.

Stateless communication

RESTful web services uses client-server architecture, in the sense that the web service has the role of the server, and the consumer of the web service has the role of the client. The client itself is responsible for storing any state information of the system it is running. This means that the server does not need to hold any state information, and can treat all incoming requests independently. This simplifies the server and makes it more scalable, as a server can easily be replaced, and more servers can easily be added to perform load-balancing.

Expose directory structure-like URIs.

In order to access a RESTful web service, the client will have to know about the identity (URI) of the service. This identity includes the location (URL) of the web service, at which it can be contacted by the client. The URL of a RESTful web service is designed to be easy to understand and self-explanatory of its function. Because of this, the URL of a RESTful web service is designed much like the URL of an ordinary website. That means that the URL is build like a path, with a root folder and subfolders [22]. A typical path to a REST web service could look like this:

http://www.myservice.com/pictures/vacations/amsterdam/

This URL is quite straightforward to interpret, which is exactly what a URL of a RESTful web service should be. Intuitively this URL points to a location where, there are pictures stored from a vacation in Amsterdam. Whether or not this URL points to a normal website or a RESTful web service, is actually not clear by the URL alone. In order to determine that it is a RESTful web service that is located at the given URL, the client must look at the response it gets. The type of response is the 4th principle of the RESTful principles and is addressed in the following section.

Transfer messages in XML, JSON, or both.

RESTful services do not make use of SOAP to create messages. One of the motivating factors for creating RESTful services was to have simpler and smaller

messages, than those of the traditional asp.net web services. Since SOAP messages come with quite a lot of wrapping with its envelope and header, another type of messaging format was required. The idea of using XML to create messages was still considered a fair solution. XML is a nicely structured language, and therefore is a good way to represent data. A simple XML message will look like this:

<SampleItem xmlns="http://schemas.datacontract.org/2004/07/TestService" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">

<Item>HelloWorld</Item>

</SampleItem>

The message contains a root tag containing the necessary namespaces, and an element with the actual data. Here the data is a string saying “HelloWorld”. However, the whole size of the message is not that much smaller than the SOAP message showed earlier. Therefore RESTful services also support another type of message format, which is of much smaller size. This format is known as JSON which is short for JavaScript Object Notation. This format was created as an alternative to XML back in 2006 and is derived from the JavaScript language [5]. The advantages of JSON are that it is simple, human readable and easy to process by program, while at the same time being formal and concise [6]. The same simple “HelloWorld” message looks like the following when created with JSON:

{"Item":”HelloWorld”}

As shown, JSON has the ability to create very short messages. Essentially, all it uses are a set of curly brackets to encapsulate the message. A JSON message can hold several values of different types. This is helpful in order to represent an object consisting of a set of different typed variables:

{"myInt":23, “myString”:“someText”, “myDouble”:3.17}

As seen, the variables are of different types and separated by a comma, and put into the curly brackets. This JSON message shows a single object consisting of an integer, a string and a double.

JSON also allows the use of arrays. To indicate that the message includes an array, JSON makes use of the angle brackets. An array in JSON looks like the following:

[{"myInt":23, “myString”:“someText1”, “myDouble”:3.17},

{"myInt":3, “myString”:“someText2”, “myDouble”:15.25},
{"myInt":34, “myString”:“someText3”, “myDouble”:4.75}]

In this JSON message, there are 3 objects in an array, which are separated by commas. The array is a 1 dimensional array. It is possible both to include several arrays into a single JSON message as well as having multidimensional arrays. However, this requires a great deal of nested brackets, which can be hard to work with, if the message gets very big. This is one of the drawbacks of the JSON format. Heavy nesting is hard to read for humans, and hard to keep track of when processing it in a program. On this point XML is somewhat superior to JSON, as the structure of XML is better suited for large and complex messages.

A RESTful service can send messages in both formats, depending on what format the client wishes to receive. The client can request response messages to be in a specific format, by setting up the “content-type” attribute in the HTTP header of the request. For an XML response, the HTTP header should include:

content-type: application/xml

And for the response to be in JSON format, the HTTP header should include:

content-type: application/json

There are clearly pros and cons with both formats. To sum it up, XML has a good structure and imposes schema checking and works with entities. This greatly helps to avoid errors. JSON is simple and well fitted especially for smaller task, and is easy to work with in programs because of its array-like structured format. Some like to describe the two formats as “XML focuses on documents, while JSON focuses on data” [6]. And XML are certainly a good choice when the task is to returning HTML fragments that needs to be inserted into a webpage. If JSON is used for this task, it would require the use of JavaScript to convert the data into an HTML fragment, before it could be used on the page [1]. So the choice of data format must be evaluated separately for each project, taking all the requirements for the service into account.

3.4.4 Choice of service

The original Frequency Registry website is not overly complex. Therefore the app version will likewise not be very complex. Because of this, it is natural to expect that it will then have a good performance. Therefore should the choice of service not jeopardise the performance of the app. One thing that could be

a potential drawback on performance is the size of the messages. Therefore the messages should be as small as possible, since the connection between the app and the service might not have a large bandwidth.

Because of this issue, it would not make sense to use SOAP, since it simply adds more characters per message, than both XML and JSON do. The smallest messages are clearly JSON messages. XML do offer some features that JSON does not, but the structure and schema checking of XML, is not strictly needed in this project, and therefore the smaller message sizes is more important. XML do have its pros when dealing with HTML webpages, but since this project does not operate with a HTML webpage, but instead needs pure data transferred between the service and the app, JSON is the better choice. XML offers some features that are just not needed for the service needed to fulfil the job of the Frequency Registry service.

Since WCF services is an upgrade of the old asp.net services, it makes sense to use WCF to build the service. However, all the features that standard WCF web services offer, is not really needed for this project. There are not requirements saying that it should use reliable message transfer or implement any special security measures. One could argue that since the Frequency Registry is public available, security is not really an issue. On the other hand it could be convenient for Erhvervsstyrelsen to control who consumes the web services. Because there are no strict requirements stating that these features should be applied, the simplest choice would be to make the service a RESTful service. RESTful services support the use of JSON for data representation, and is stateless and thereby making the servers hosting the service, more scalable. Since WCF supports creation of RESTful services, the choice falls on this framework.

A WCF RESTful service, using JSON for data representation seems like the right choice for the Frequency Registry app. Even though the service is made with the WCF framework, it will not make use of the features WCF offers. This is to keep the service as simple as possible. JSON is also a good choice for the data representation, since Titanium Mobile is using JavaScript as the main programming language, and JSON is derived from JavaScript.

3.5 System overview with the service

In order to get the Frequency Registry app up and running in the real environment, the RESTful service would need to be part of the final system. As discussed earlier, the app should communicate only with the RESTful service, and never directly to either the existing website or the database. This results into the following system overview as seen in figure 3.10.

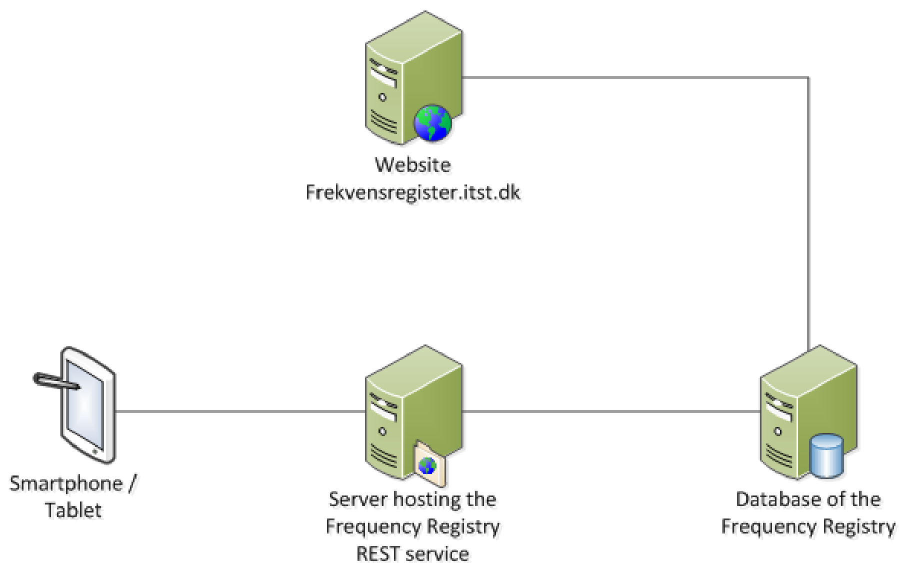


Figure 3.10: System overview with the new REST service

As figure 3.10 shows, the database can now communicate both with the existing website and the REST service. The end user is using the system via his or her smartphone or tablet. The app runs on the user's device and communicates only with the REST service. The existing website is not influence by this new addition to the system.

Chapter 4

Implementation

In this chapter the details of the implementation is being covered.

Section 4.1 covers the design of the graphical user interface, aswell as the most important logic that lies behind it.

Section 4.2 explains the structure of the program.

Section 4.3 covers the implementation of the RESTful service

Section 4.4 takes a look at the performance of the app.

4.1 Final Design of The Frequency Registry App

This section shows the final design of the graphical user interface of the Frequency Registry app. The code provided with this project, built in Titanium Mobile, will deploy this version of the app to the device. The app can be deployed on both iOS and Android devices. For the demonstration of the apps appearance, the iPhone simulator has been chosen.

The start up window

When the user runs the app from his or her device, the app shows a quick splash screen with logo of Appcelerator, followed by the start up window of the app. This is shown in figure 4.1.

Figure 4.1: Start up window

This is where the user types in the criteria for the search. Here are only shown 3 search criteria, but if the app were to be put into a production environment, this window would show the remainder of the search criteria, similar to those on the existing website.

In the Case ID field, the user can type in the ID of the case he or she is looking for. Case ID's in the Frequency Registry are made out of 1 letter followed by 6 numbers, e.g. H100489. Therefore the user needs a keyboard with both numbers and letters to type in the case ID.

In the next 2 fields, labelled "Frequency Range (MHz), the user can type in a minimum and maximum frequency for the search. These values can only be numbers, and therefore the user is presented with a numeric keypad when tapping on each of these two textfields.

This way, the program prevents the user from typing in a non-numeric value for these fields. This is one of the benefits with using custom keyboards for each field.

In the last field, the user can choose a case type to search for. There exist a limited number of case types in the Frequency Registry, and therefore they are put into a fixed list that shows when the user taps the case type field, as shown in figure 4.2.

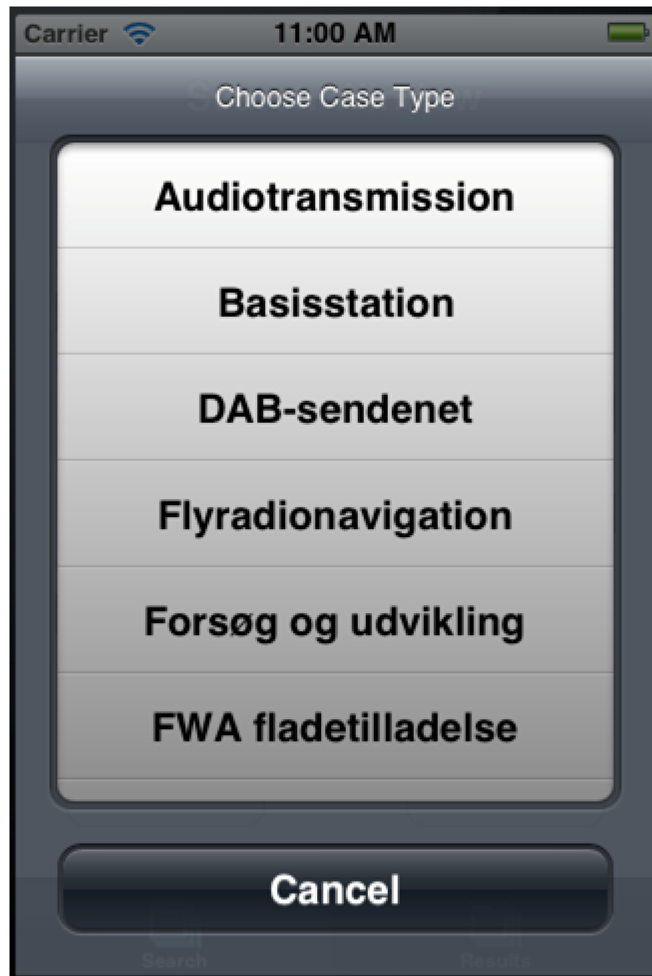


Figure 4.2: List of case types

When all the desired criteria have been filled, the user can click search to perform a search and the app will contact the RESTful service and wait for a response. If no response is received within the defined timeout period, an error will show, and the user can try again. If a response is received, the app switches to the second tab and shows the results of the search. This is shown in figure 4.3.

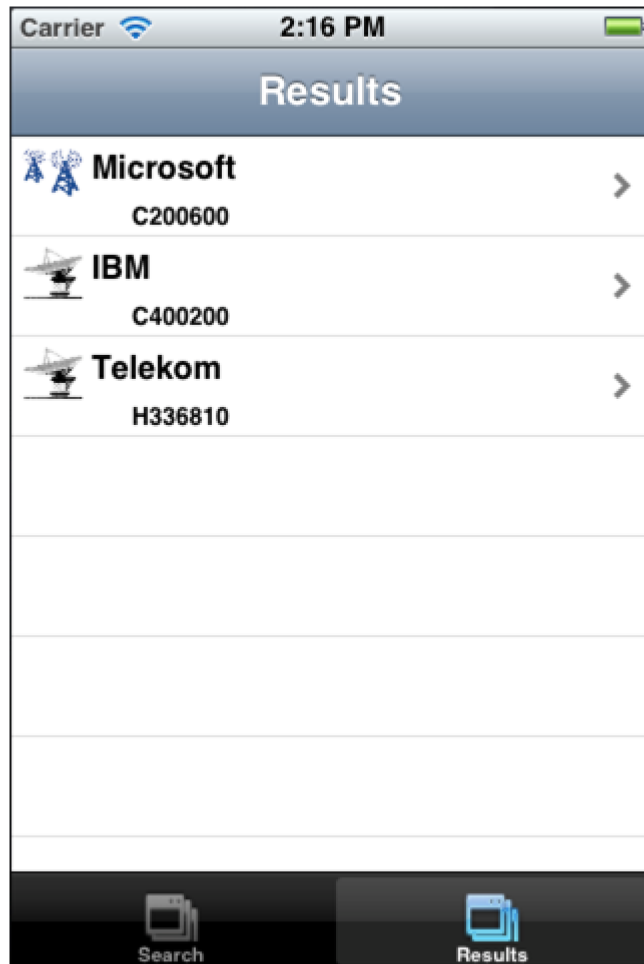


Figure 4.3: Results of search

The results are listed in a table. Each row shows the name of the registered with the case, the id of the case, and small image to indicate what type of licence this case is providing. In the first row, the image shows two antennas sending signals to each other. This is to indicate that this case gives licence to a "point to point" transmission.

In the two consecutive rows is shown a satellite dish located on the ground. This is to indicate that these two cases provide licence to use a ground station to send signals with. With the help of these images, the user can quickly find the exact case that he or she is looking for, as long as they know the type of the

case.

When the user has found the case he or she wants information about, the user clicks on the corresponding row and the app changes window to the one shown in figure 4.4.

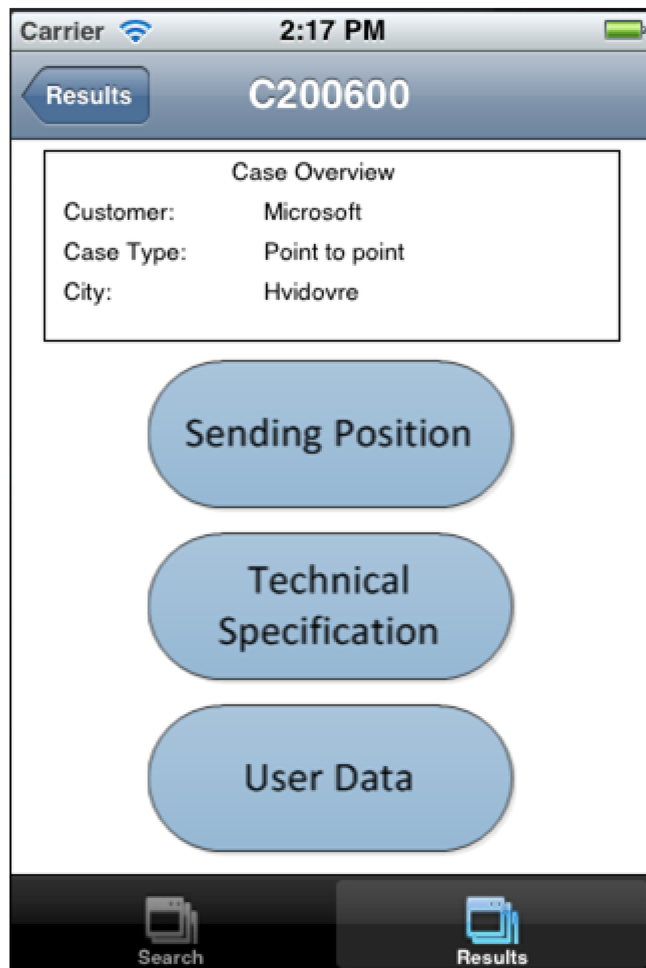


Figure 4.4: Case details

The case details window shows a couple of essential information about the case just below the title bar. This is the case overview and shows the name of the customer, the type of the current case and in what city the sending device is located. If the user requires more information, the three buttons can be used

to open new windows. The information is spread into these three categories in order to avoid having one big list of information that the user would have to scroll through to find the information that he or she was looking for. This way, the user will have a better overview of the case. The three categories are all presented in the same way, as seen on figure 4.5. The information is listed in a simple table, with a label and a value field. When the user wants to return to the case overview, the back-button in the title bar can be used. The text value of the back-button holds the case ID of the opened case. On the Android platform, the physical back button should be used to achieve this.

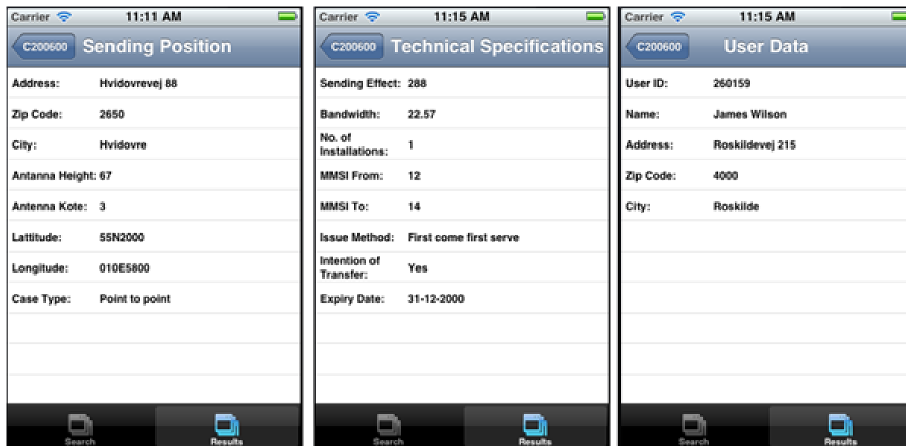


Figure 4.5: The three categories of information on a case

4.2 Program structure

As mentioned earlier, programs built with Titanium Mobile follow the same overall structure. Every program starts executing the code in the “app.js” file. This is similar to the “main” method in a classic console application built in Java or C#.

Every object in Titanium Mobile is created in a similar manner. Here is an example of how a typical object is created:

```
var MyLabel = Titanium.UI.createLabel({

    text: "This is a label",

    font:fontSize:12,fontWeight:'bold',

    width:50,

    textAlign:'left',

    bottom:0,

    left:60,

    height:20

});
```

Every variable in Titanium is created with the type “var”. This is the nature of the JavaScript language. Here a label view is created. The properties of the label are set immediately when the label is created, by the use of the object’s constructor. Here the basic properties are set, such as the width and height of the label, where it should be placed in its parent view, and what text value it should display.

Since a lot of UI elements on smartphones are meant to be tapped or touched in some way by the user, every view can have an eventlistener attached to it. There are many different types of events, depending on the type of view its being added to. Some of the most common events in Titanium Mobile are “click”, “tap”, “touchMove”, “touchEnd”, “singletap”, “twoFingerTap” and many more. An eventlistener is added to a view in the following maner:

```
MyLabel.addEventListener('click', function(e){

        Ti.API.info('MyLabel was clicked')

});
```

The type of event is put given as the first argument, and the second argument is a callback function that is being executed once the event has been fired. Here all it does is to write "MyLabel was clicked" into the console of Titanium Studio.

As with many other programming languages, it can sometimes give a better overview of the code, if the code is put into several files, instead of just one big file. The Frequency Registry project consists of 3 JavaScript files. The app.js file, case.js file, and the details.js file.

App.js file

The app.js file of Frequency Registry app is where the start up window is created, and along with all of the different views on this window. When all the views have been created, they are added to their parent view, whether that is the start up window or another view, by calling the "add" function on the parent view, and providing the child view as the argument. An example of this is shown here:

```
MyWindow.add(MyLabel);
```

Here is the label "MyLabel" being added to the window "MyWindow". Apart from creating all the views, and adding them to their correct parent views, the app.js file also holds the responsibility of contacting the RESTful service, when the user presses the "search" button. This is done by creating an instance of the "HTTPClient" object. In the eventlistener of the search button, the callback function calls the "open" method of the HTTPclient object. This method takes 2 arguments. First one is the HTTP method that is being used, here it is the GET method (since the app just wants to retrieve a resource from the service), and the second one is the URL at which the RESTful service is located. In the code is looks like this:

```
xhr.open("GET", URL);
```

The variable "xhr" is the instance of the HTTPClient. The URL variable is a string value of the actual URL to the service. Before the HTTP request can be send, it needs to have a content type set in its header. This tells the service whether the response should be delivered in XML format or JSON format. For this project only the JSON format is used. The header is altered by the following statement:

```
xhr.setRequestHeader("Content-type", "application/json");
```

If it is needed to get the response in XML format, the argument "application/json", is simply substituted with the string "application/xml".

When the header is set with the desired content type, the request is being sent by calling the send function:

```
xhr.send();
```

When the request is sent, the app does not do anything until an event is fired. That could either be an event coming from a tap on the screen by the user, or the HTTPClient that receives a response. Instead of having to add an event listener to the HTTPClient object, the developer has to provide the "onload" property with a function that is called whenever a response is received from the receiving party (here the receiving party is the RESTful service). Essential it acts the same way as an eventhandler with a callback function. The "onload" function in the app.js file, parses the response it gets into a standard array variable in JavaScript. This is done using the JSON API provided in Titanium Mobile:

```
JSONobj = JSON.parse(this.responseText);
```

The "this" keyword refers here to the HTTPClient object. After this statement has been executed, the content of the response message can easily be accessed like any other array. The content is used to fill out rows to display the cases that were found during the search. The name and case id of each case is placed into an instance of the "TableViewRow" object. This is the standard row type in Titanium Mobile. The rows are used to create the table on the results window. The results window is added to the second tab of the application, called the "results tab". The table in the results window gets added an eventlistener that fires an event any time the user taps any of the rows in the table. The callback function of this event opens a new window in results tab, called the "Case window". This window is titled with the id of the case and shows an overview of the opened case. The code for opening this window is shown here:

```
tableView.addEventListener("click",function(e){
    var newWindow = Titanium.UI.createWindow({
        backgroundColor:"#FFFFFF",
        dataToPass:JSONobj[e.index],
        url:"case.js"
        });
    tab2.open(newWindow,{animated:true});
});
```

As the code shows, a new window is created in the callback function for the event. The logic for this window is placed in the case.js file, and therefore the url property is set to point to this file. When creating a new window, it is possible to add a custom property. This is done here with the “dataToPass” property. Here the data in the JSON object for the specific case that the user clicked on, is stored as a custom property. Only the data of the specific case is needed, and therefore only this part of the entire JSON message is passed to the new window. When the window is being accessed in the case.js file, this property can be accessed aswell. Thereby the data of the case has been passed to the case.js file. Once the new window has been created it is opened in the second tab, here denoted as “tab2”.

Case.js file

In the beginning of this file, the custom property of the newly created window is stored in a local variable in order for it to be referenced in the case.js file. It is also possible to access the current showing window and the current active tab. The case.js file is responsible for creating the window which shows the case overview when a specific case is selected. This window consists of a view that shows key information about the case and three buttons that each represents details on the case, grouped in three categories, which are “sending position”, “technical specifications” and “user data”. Each of these three buttons has its own eventlistener that listens for the “click” event. Once the event is fired, the callback function creates a new window and opens it in the current tab, which is “tab2”. The logic of the new window is held in another JavaScript file called “details.js”. As a custom property, the detail category is passed to the “details.js” file, just as the JSON message was passed from the app.js file, to the case.js file. This is done to make the details.js file simpler.

Details.js file

This file is responsible for showing the details on each of the three categories, mentioned earlier. Common for the three detail windows are that the information is shown in a normal table, with the use of the tableView object. Each table is then populated with a number of rows, depending on what type of detail has been chosen by the user on the case overview. This information is stored in the window object in the custom property "detailType". Most of the code of this file is the same for the three types of detail, so by having this property a lot of code is reused, making the file simpler and easier to maintain. The only thing that is different from each detail type is the title of the window, the row labels and the data in the JSON object used. Everything else is reused among all three detail types.

4.3 The Frequency Registry RESTful service

When the user performs a search in the Frequency Registry, he tap the search button on the search window inside the app. If any of the search criteria is filled out, their values are sent together with the search request, as the requests arguments. The request is a HTTP GET method request for the operation "PerformSearch". The operation currently takes 4 arguments, which corresponds to the 4 search criteria fields that has been implemented in the final version of the app. If any of these fields are not filled out, the request will be filled with their "empty" values, which is simply an empty string for the string types and "-1" for the double types. The format of the URL used to contact the RESTful service is as follows:

HOSTPATH\PerformSearch?arg1=value&arg2=value&arg3=value&arg3=value

As shown, the request has four arguments, one for each of the search criteria. If the production version of the app, should include more search criteria, the request would have to take more arguments, according to the number of search criteria.

When the request is received in the RESTful service, it performs the actual search, which is done in the PerformSearch-method. The RESTful service has stored a list of sample cases for this purpose, since it is not connected to the live database. The search works in the following way:

- It loops through all the case objects stored in the list called CaseList.

- For every case it compares the four arguments with the value of the corresponding property in the case object.
- If either the argument and the property matches in value or if the argument is empty, the search will treat this as a hit. This is done to allow the user to leave out some search criteria and still get a valid search result. This means that if all of the arguments have a value specified, then the method will only return those cases where all of the arguments match with the corresponding case properties. If all of the arguments are empty, then the method will return all of the cases it has stored in the CaseList.
- If a hit occurs, the case is stored in a new list called ResultList.
- Once all of the cases in the list has been looped through, the method returns the ResultList.

4.4 Performance

There are many ways to measure performance of an app. One subject worth investigating is the usage of the app. How many people are actually using the app? And if not, is it because they cannot find it or because they don't like it for some reason? This gives lead to the following metrics to measure:

- Total number of downloads.
- Number of unique users accessing the app over a period of time.
- Active user rate. The rate between the total number of downloads and the number of active users.
- Number of new users of a period of time or a ratio of new user per day/week/month.

Another interesting aspect to look into is how much the users are actually using the app. This could be measured by the following metrics:

- How frequently are the users opening the app?
- The average amount of time using the app, each time the app is opened.

All of these metrics would be relevant to test on the Frequency Registry app, however since the app has not been put into a production environment and available to the public, this has not been possible to test.

One aspect that can be tested during development is how native the app feels to the user. The user would ofcouse have to be used to using apps in order to tell if this app have captured the native look and feel, that has been mentioned earlier. As an experienced app user of iPhone apps myself, I would say that it does indeed have the native look and feel to it. It reacts immediately upon touching the screen and the animations and scrolling works just like any other iPhone app. I also tested the Android version on a Samsung Galaxy Tablet, and compared the experience with the usage of other Android apps. The results were the same. For me, there is no way to tell that the Frequency Registry app is not made in the native programming language on either of the platforms.

Since the app has not been put into the app stores I asked some friends to test out the app on my own iPhone. They agreed with me that the app felt completely native.

One thing I did notice during the testing was that when the device was not connected to a Wi-Fi network, the HTTP request to the RESTful service sometimes suffered a timeout. This timeout occurs after 5 seconds if no response has been received. The problem is clearly related to the network connection, since I've tried to make the same HTTP request with a web debugger (Fiddler), and the server hosting the RESTful service replies immediately. This was done both before and right after a timeout in the app appeared. One way to deal with this could be to increase the timeout limit of the app.

I have also made some measurements on the apps performance. These measurements are performed on an iPhone 4S 16 GB version. The iPhone is running version 5.1.1 of iOS.

- Start up time: 1.5 seconds.
- Search performed while connected to Wi-Fi: 1.5 seconds.
- Search performed while connected to 3G with a signal strength value of 2 out of possible 5: 2.75 seconds.
- Search performed while connected to 3G with a signal strength value of 3 out of possible 5: 3.75 seconds.
- Time used to load a new window or tab: Happens instantaneously
- Number of app crashes observed during entire development phase: 0

These numbers show that the app itself, as well as the RESTful service, work pretty well. The only thing that can be an issue for this app is either a bad network connection, or if the webserver running the RESTful service handles the request very slowly. As I did not have the opportunity to stress test the RESTful service, by making multiple request at the same time, it is not clear how the REST service performs when a lot of clients makes requests at the same time. The Frequency Registry website has approximately 30 visits per day on average, so it is entirely possible that multiple request would happen quite often if the app gets in production. But any service should be able to handle multiple request of this magnitude, the webserver used in a production environment is most likely more powerful than the one used in this project.

Overall the performance of the app has proven to be quite well.

Chapter 5

Experiences with Titanium Mobile

Working with Titanium Mobile has been a great and educational experience. Some of the things I learned came as a surprise to me and is something that I did not stumble upon when reading about the framework. One of the most important features about Titanium Mobile is that it allows the developer to only have to develop the app once, and then it will be running perfect on both Android and iOS devices. At least that is the impression one gets from reading about the product on the creators Appcelerator's website. I found this statement to be a little exaggerated, as I will explain in this chapter.

5.1 Build once deploy everywhere

This theory of "Build once, deploy everywhere" is a very important aspect of Titanium Mobile, as this reason alone can be enough for a company to choose Titanium Mobile over some of its rivals. So does it live up to the expectations? Well for starters, there are some slight differences in the UI between the two platforms (Android and iOS). Creating the standard Titanium Mobile project, which shows 2 tabs, with each window having a single label, illustrates some of the differences.

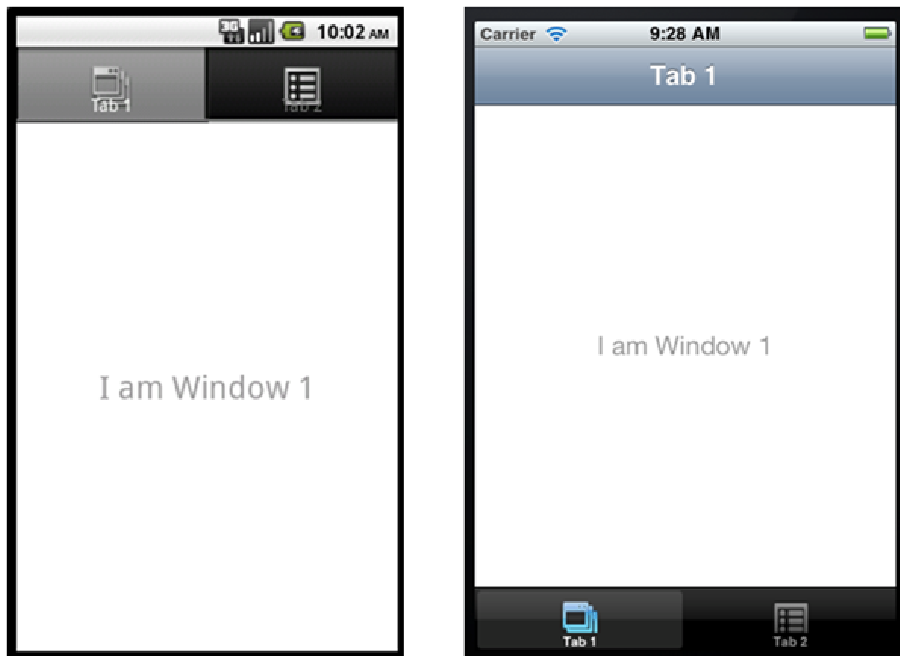


Figure 5.1: Standard application on both platforms (Android to the left), (iPhone to the right)

On figure 5.1 is shown how the standard 2-tabbed project looks on both the android platform (to the left), and the iOS platform (to the right). The first difference to notice is that the tabs are not located at the same place. Android has the standard tabs located in the top and iOS has its tabs located at the bottom of the screen. Also, Android does not have a title bar for its windows. This is just a small portion of the differences that exist between the two platforms, and all it took to visualise this, was to deploy the simplest app available, and compare the results. These differences can be evened out by some manual adjustments of the code, but part of the reason why people choose Titanium Mobile as their development platform is that the get the impression that they are free of such manual adjustments across the platforms.

During development of the Frequency Registry app, more differences started to show. In the early development I tried putting the logo of Erhversstyrelsen on the start up window. The opacity of the picture had a level so that the logo would look like a "watermark" in the background. This picture would come to look quite different on the two platforms.

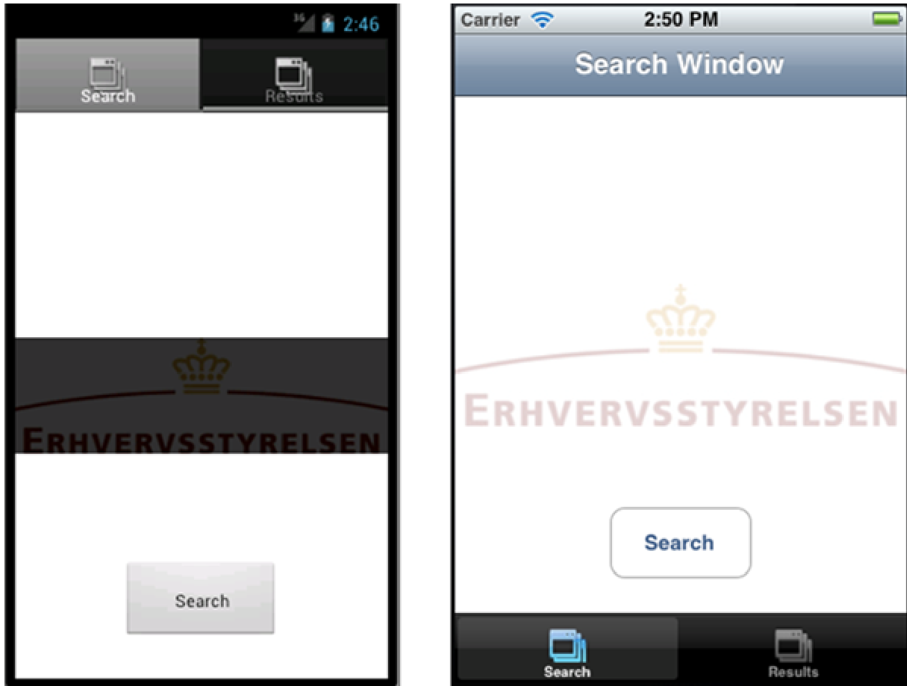


Figure 5.2: Differences on Android (left) and iOS (right) -– Logo

As figure 5.2 shows, the logo is not being displayed correctly on the Android platform. It seems that the opacity of the picture is being displayed as a dark grey color. On iOS on the other hand, the picture is being displayed as expected.

Titanium Mobile gives the developer the ability to use custom defined rows to put into a table. These custom rows can consist of several views of any type. I tried creating a custom row with an image, 2 labels, and an expand/collapse link that let the user know that the row can be expanded if it is tapped. On figure 5.3 is shown a window containing a table with 2 of these custom rows in it. As seen on the figure, the text of both the labels has been cut off in the bottom on the Android platform. This phenomenon has only been seen on the Android platform during this project. Also the standard color scheme of the 2 platforms is quite different, which also is made quite clear on figure 5.3.

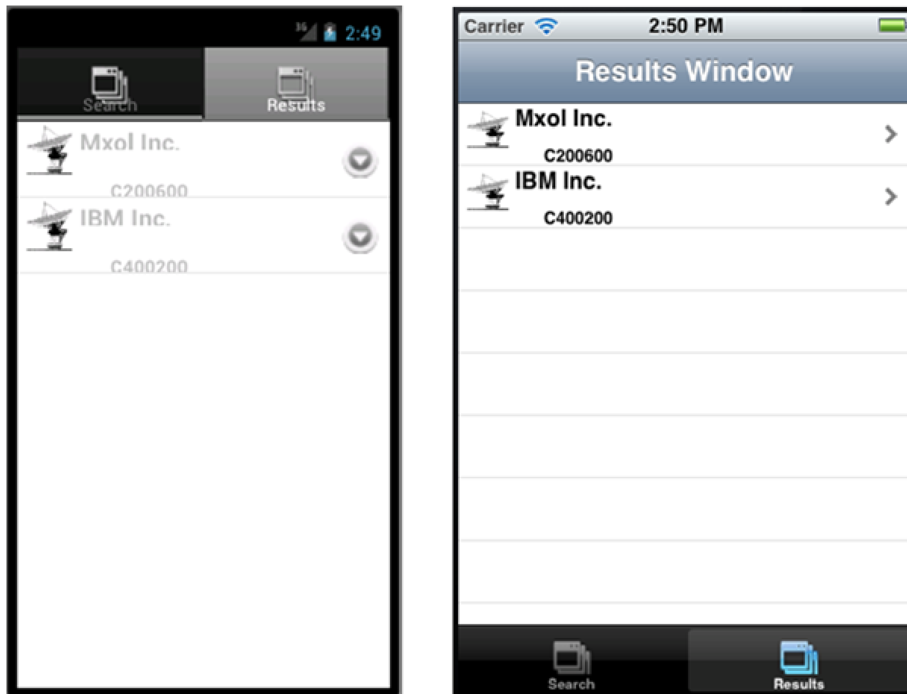


Figure 5.3: Differences on Android (left) and iOS (right) – Custom made row

The cutting off part of the text in the labels that were seen on figure 5.3 is caused since the size of the letters is too big to fit into the row size. This is unexpected since the size is set to a fixed value in the code, and it works fine on the iOS platform.

One obvious challenge that exists when creating an app for mobile devices is the different sizes of the actual device. iOS devices only consist of iPhone (and iPod Touch which is the same size) and iPad, but the Android system is running on many different devices, all with different size screens. This makes it hard to make a one single solution that fits all screen sizes. This may not be the case with the label text being cut off on the Android version, but nonetheless I have observed other behaviours that I believe are related to the screen size. One issue was found during the development of the window that presents the details on the sending position of a case. The issue can be seen on figure 5.4. The data shown is just test data.

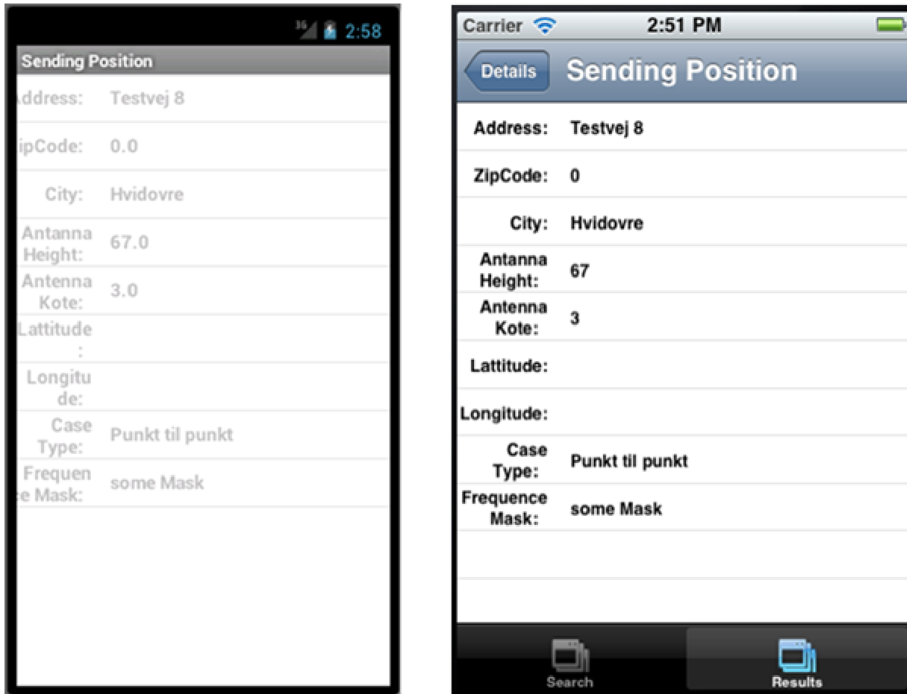


Figure 5.4: Differences on Android (left) and iOS (right) – Table with rows showing only text

As seen on figure 5.4, the Android system has problems showing the whole text in the rows. The first row is supposed to display the text "Address:" followed by a string with the value of the address of the sending position. This works fine on the iOS system, but on Android, the first letter has been cut off from the screen. This is also the case in the second row, where it is supposed to display "ZipCode". Strangely enough, this is not a problem on all of the rows. And apparently it is not because the text is just one word, with no whitespaces, since at row 7 it is capable of splitting the word "Longitude" into two lines.

Figure 5.4 also shows the fact that there is a problem with the tabs on Android, when a new window is opened in the same tab. This happens when the user clicks to expand a row shown on figure 5.3, a new window is opened in the same tab, but the tab bar is missing. However it is still possible to move backwards in the sequence of windows opened in the same tab, by using the physical "back-button" that all Android devices posses. So this means that without further modifications to the program as it is shown in figure 5.4, in order for the user to navigate back to the start screen and perform a new search, the user would have to press the physical back-button two times, before the tabs would again

be shown, allowing the user to go to the start screen. This would have to be changed manually in the code, if the app should ever get in production.

5.2 Active community

As mentioned earlier the community of Appcelerator's Titanium software (both for desktop applications and mobile applications) is quite large, with 300,000 registered people worldwide. Many times during this project I stumbled across a problem, that I could not find the solution to by searching the Internet and API docs of Appcelerator. The "Question and answers"-section of Appcelerator's developer forum is a great place to find help from the community. I have asked a dozen of questions at this site, and every time I have received fast and constructive replies. Within the first half hour of posting a question, the post would get about 50 views, and usually a constructive reply. This has really impressed me.

5.3 Titanium Studio

This is the IDE that is used to develop all the types of Titanium applications. It is built on top of the popular open source IDE, Eclipse. Eclipse is a solid IDE with a good editor and good debugging possibilities, and Titanium Studio is enjoying the benefits of using Eclipse as a base. Titanium Studio also makes use of intellisense, which help provide the developer with the Titanium Mobile API as the developer types in part of the statement.

Titanium Studio also comes with emulators for both the Android and iOS platforms. These emulators are an invaluable help when developing the apps, as they can simulate the app running on the desired device. They support a great variety of devices of different size, especially for the Android platform. However the Android emulator was considerably slower to start then the iPhone emulator, which is why must of the testing was done using the iPhone emulator. It takes about a minute and a half to open the Android emulator on the laptop I was using during this project while it only took 5-10 seconds to open the iPhone emulator. Therefore the iPhone emulator was the primary testing tool.

Overall my experiences the Titanium Studio were quite good.

Chapter 6

Conclusion

In this thesis I have focussed on creating a mobile version of the Frequency Registry, which is a public service provided by Erhvervsstyrelsen. Since the app should be available on multiple platforms, various cross-platform development technologies have been investigated. This thesis compares four of the most popular cross-platform development technologies of today, and concludes that the Titanium Mobile platform, created by Appcelerator, is the most appropriate platform to use in this project. There are two main reasons behind this choice. The first one being that Titanium Mobile is using the same codebase to deploy the app both as an Android version and an iOS version. The second reason is that app built with Titanium Mobile is compiled into native code. Thereby it looks and feels just like if the app were created with the native development language for both platforms separately.

Working with Titanium Mobile shows that there are some issues to be aware of. Android and iOS does not share the same default layout scheme. Often there is a need for manual adjustments, if the two versions are to look alike. Also there are some differences in the way colors, shadows and transparent pictures are shown. Still Titanium Mobile is a great platform and is already being used professionally by big companies and respected brands. The apps build with this platform looks and feel native, because they are native. Every element that these apps consist of is native elements, and this really shows when testing the app. However there is still room for improvements. The IDE Titanium Studio

would improve a lot if it was possible for the developer to use designer tools like Xcode and DroidDraw to create the graphical user interface, like the Mono framework does.

During the tests of the app, it was observed that the 5 second timeout limit that was set on the HTTP request going from the app to the RESTful service was sometimes not enough. This became a problem, as soon as the device was not connected to a Wi-Fi network and had low/medium signal strength to the 3G mobile network. This indicates that the response time relies heavily on the network connection. The timeout period can be adjusted manually in the code, to cope for this issue. While the timeout limit was set to 10 seconds, none of these issues has been observed. The long response time is clearly a network related issue, and has not anything to do with the development platform of which the app is created.

A long response time does not necessarily lead to a bad user experiences, as long as the user is informed of the waiting time in a meaningful way. Since the response time should be kept to a minimum, the choice of using a RESTful service, and sending messages in JSON format, proves to be a good one, since this gives the simplest and fastest service and the shortest of messages. All of which is having an impact on the response time.

As part of the testing of the app, the performance was evaluated, and the app performed was just as good as any other native app. The only thing worth noting is the response time increase caused by the network connection.

6.1 Future Work

Titanium Mobile is definitely capable of being the platform for creating a production version of the Frequency Registry app, for Android and iOS. Whether this will be reality or not, is not in the scope of the thesis to predict. However, if it does become reality, then developers should be aware what this thesis has found to be one of the most critical areas of the app, namely the response time when it needs to contact the RESTful service. One way to improve this response time could be to store a local copy of the production database together with the RESTful service. This way, the response time does not suffer from the communication time spent between the service and the database. The local copy of the database would need to be updated from time to time. This could easily be done during the time where the service is just being idle, waiting for incoming requests.

There are many ways to make this project into a reality. Whether it will be or not, remains to be seen and only time will tell. Titanium Mobile is certainly capable of making it happen.

Appendix A

Source Code

The source code used to create this project is provided on a CD. This CD provides both the source code for the Titanium App, and the source code for the RESTful service. The structure of the CD is explained here:

In the root directory there are 2 folders. The first folder is named "Titanium Mobile Source Code". This folder has the following 2 sub folders:

- "Source Code" which includes only the JavaScript files of the Titanium Mobile app.
- "Workspace" which includes the entire workspace of the Titanium Mobile project.

The second folder located in the root directory of the CD, is named "RESTful Service Source Code". This folder similarly has 2 sub folders:

- "Source Code" which includes only the C# files files of the RESTful service.
- "Workspace" which includes the entire workspace of the RESTful service project.

Bibliography

[1] Brian Leroux Andre Charland. Mobile application development: Web vs. native, 2011. [Online; accessed 10-September-2012].

[2] Appcelerator. In the clear: Apple opens up ios to all developers., 2010. [Online; accessed 10-September-2012].

[3] Appcelerator. The titanium sdk docs, 2012. [Online; accessed 10-September-2012].

[4] Christina Bonnington. Google's 10 billion android app downloads: By the numbers., 2011. [Online; accessed 10-September-2012].

[5] Douglas Crockford. Rfc 4627: The application/json media type for javascript object notation (json), 2006. [Online; accessed 10-September-2012].

[6] DigitalBazaar. Web services: Json vs. xml, 2010. [Online; accessed 10-September-2012].

[7] Niclas Jonasson Erik Jansson. Speedflirt for android, a study of porting a mobile application and its effects on user experience. Master's thesis, Umeå University, Department of Computer Science, may 2011.

[8] Microsoft MSDN Forum. When wcf release will be available?, 2006. [Online; accessed 10-September-2012].

[9] PhoneGap Project Group. Supported device features using phonegap, 2012. [Online; accessed 10-September-2012].

[10] IBM. Restful web services: The basics, 2008. [Online; accessed 10-September-2012].

[11] IDC. Idc worldwide mobile phone tracker, 2012. [Online; accessed 10-September-2012].

[12] Apple Press Info. Apple's app store downloads top 15 billion, 2011. [Online; accessed 10-September-2012].

[13] Intertwingly. A gentle introduction to soap, 2002. [Online; accessed 10-September-2012].

[14] Ben Lang. Which mobile browser has the best html5 support?, 2012. [Online; accessed 10-September-2012].

[15] Wilke Mobile Life. Danish smartphone market, 2011. [Online; accessed 10-September-2012].

[16] PCmag.com. Wireless lan time line, 2003. [Online; accessed 10-September-2012].

[17] VP of Developer Relations @ appcelerator.com Scott Schwarzhoff. Is appcelerator titanium native?, 2011. [Online; accessed 10-September-2012].

[18] TDC. Speed on the internet when using 3g on a cell phone, 2012. [Online; accessed 10-September-2012].

[19] W3schools. Wsdl and uddi, 2012. [Online; accessed 10-September-2012].

[20] Xamarin. Build cross-platform mobile apps using c# and .net, 2012. [Online; accessed 10-September-2012].

[21] Xamarin. Limitations to mono android, 2012. [Online; accessed 10-September-2012].

[22] Xamarin. Limitations to monotouch, 2012. [Online; accessed 10-September-2012].