CHALMERS

Extending a Web-Based Work Platform to the iOS Platform

Master of Science Thesis

SEBASTIAN REHNBY

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, February 2012

Extending a Web-Based Work Platform to the iOS Platform

SEBASTIAN REHNBY

Examiner: MIROSLAW STARON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden February 2012

Abstract

This thesis report presents the work carried out at Podio, a company developing an online work and collaboration platform. The purpose is to identify and research a number of issues with Podio's existing mobile application for the Apple iPhone. Based on these findings, common problems and challenges in mobile application development are presented and reasoning around possible solutions is given. The problem domain is approached from a technical, usability, and time-to-market perspective. This thesis aims to identify the considerations needed when adapting an existing web-based product to a mobile environment and the inherent constraints of such an environment. It also takes into consideration the limitations of mobile devices themselves in terms of physical dimensions and hardware resources.

Two main priorities are identified as the most important challenges of developing a high quality mobile application. The first is the adaptation of the desktop user experience to that of a mobile application. The importance of realizing the strengths and weaknesses of each of them is considered essential to producing a high quality mobile experience. The second area is that of request management and local data caching. Because of the ever-changing environment of mobile devices with varying connection speed and availability, it is important to handle local data storage with a great deal of afterthought. In order to build a robust client application to the Podio API, one also needs to consider the ways in which the API changes and possibly breaks compatibility with existing client applications.

Because of the many considerations when dealing with server requests and data caching, and to ease the development of new features, a standalone static library called PodioKit is developed. PodioKit provides a native Objective-C interface to access the Podio API in a convenient and extendable way. PodioKit utilizes the dynamic properties of the Objective-C programming language to automate the domain object mapping process from a simple mapping definition. The primary purpose of PodioKit is to decouple request management from data persistence and the application itself, and to reduce code duplication. Although within the Podio application PodioKit is used with the Cocoa Core Data framework for local storage, it does not require nor does it make any assumptions regarding the underlying storage technology used.

The report concludes that multiple tradeoffs are necessary when adapting an existing web-based service to a mobile context. They include determining the mobile feature set of the application given the physical and environmental limitation of mobile devices, as well as finding a balance between maintainability, testability and performance. It also identifies the task of managing network connectivity in combination with remote application data to be the most challenging area in mobile application development, one where the right approach depends heavily on the primary application use cases.

Preface

I would like to thank Podio ApS for hosting this thesis project at their office in Copenhagen during the summer and fall of 2011. I would also like to thank my supervisors; Phillip Chambers at Podio and Jörgen Hansson at Chalmers for their help and support throughout this project.

Glossary

API Application Programming Interface.

BLOB Binary Large Object.

DOM Document Object Model.

IDE Integrated Development Environment.

MVC Model-View-Controller.

ORM Object-Relational Mapping.

PaaS Platform as a Service.

REST Representational State Transfer.

SaaS Software as a Service.

SDK Software Development Kit.

UI User interface.

Contents

1 Introduction

Web-hosted software solutions, often referred to as Software as a Service (SaaS), have rapidly increased in popularity in the last couple of years. Led by industry giants such as Google and Facebook, a lot of the work previously performed by using desktop applications have moved to the Internet, often referred to as the *Cloud*. As people access their data and content from a wider variety of devices, there are several advantages to moving our repository of information from the desktop to the Internet. It makes data easily accessible from anywhere to anyone with an web browser and an Internet connection. Smaller companies can also avoid the cost overhead of implementing and managing a self-hosted solution.

In parallel, mobile computing has also taken off in the last couple of years. Many would credit this to the launch of the Apple iPhone, the first mainstream mobile device that emphasized Internet access and communication, and the subsequent launch of Google's Android platform. According to a press release by International Data Corporation (IDC) [1], smartphone market share had surpassed that of so called feature phones in Europe by September of 2011. Because of this recent boom in mobile computing, it has become increasingly important to SaaS companies to have a mobile strategy, i.e., a way for their customers to access their service from their mobile devices. The solutions range from mobile optimized version of web sites accessed through the mobile browser to native client applications.

Podio [2] is an online work and collaboration platform. It is web-based and centrally hosted. The goal of Podio is to enable and encourage collaboration in teams and projects by allowing the customer to customize their workflow and by providing a social context to working with business-related artifacts. Because of the flexibility and extendibility of the platform through the use of *apps* and an open Application Programming Interface (API), it can also be consider a Platform as a Service (PaaS) company.

Like other SaaS companies, Podio is dedicated to having a mobile strategy and a way for customers to interact with their Podio data using their everyday mobile devices. They do so by offering native client applications for the two largest mobile platforms, Google's Android and Apple's iOS. For the client applications to be able to interact with the platform, Podio offers an open and full-featured API. The API contains all the functionality of the platform and the Podio web site is in fact just another client to the API.

This thesis identifies the problems and challenges associated with developing a mobile client application as an extension of a SaaS product using a Representational State Transfer (REST) web API [3]. It covers the differences, limitations and advantages of the mobile context, devices and platforms and suggests strategies for handling these appropriately. It also presents some of the technical challenges of developing a client application based on remote data while guaranteeing the user experience. This thesis project uses the existing iPhone application as the basis and iOS as the target platform for the proposed

solutions.

The main research questions this thesis projects aims to address are:

- What are the technical aspects and potential problems that should be considered when developing a mobile client application, where data is provided by a remote third-party web service and accessed over a sometimes unreliable cellular network connection?
- What are the possible solutions to these problems and what are the advantages and drawbacks of each of them?
- How does one, as a developer, achieve a balance between the end user experience, development effort and maintainability of the application?
- How does a mobile application respond to changes in the web service it depends upon?
- What techniques can be used to improve the perceived performance and usability of a mobile application?

Brief Overview of Results

The results of this thesis project are twofold. The first part addresses the problems associated with mobile application development in general and the adaptation of an existing web-based product into a mobile application in particular. These results show that to provide a high quality mobile application one should realize the strengths and limitations of mobile devices and the environments they are designed to operate in. One should not simply try to create a one-to-one clone of the web-based version of a product, but instead leverage the context awareness and mobility inherently available to mobile applications. This part also suggests a way to determine the feature set of a mobile application based on the frequency and complexity of the action it is used to accomplish.

The second part of the result is of a more technical nature. It addresses the technical challenges with managing data and adapting to network availability in the rapidly changing environment of mobile applications. Using the existing Podio iPhone application as the base for solving these technical challenges, a new iOS client library called PodioKit is developed. PodioKit provides a pluggable and easy to use, native abstraction layer for the Podio API. PodioKit implements the suggested solutions presented in this thesis to solve existing issues with data and network request management.

The thesis work itself begins with an evaluation of issues experienced with the existing Podio iPhone application. Additionally, many core features from the Podio web site were missing in the existing iPhone application. As the first part of this thesis is concerned with how to adapt existing web-based functionality to a mobile application, implementing these core features based on the thesis research in this area provides valuable insight and a good starting point

for improving the underlying data and request management strategy. Finally, PodioKit is introduced into the main code base, providing an easy to use and far less error prone solution to work with the Podio API.

Outline of Thesis Report

The remaining parts of the report are structured as follows:

Chapter 2 provides an in-depth look at Podio as a product and its public API. A brief overview of the state of mobile application development is given, including popular platforms, key stakeholders and platform providers. The mobile platform used in this thesis project, Apple's iOS, is also introduced.

Chapter 3 outlines the problems often associated with mobile application development and how they differ from those of regular desktop application development. It states a number of known problematic areas and presents the method of work for this thesis project.

Chapter 4 presents the research carried out as part of this thesis project. It presents proposed solutions to each of the problems described in chapter 3 along with their respective advantages and disadvantages.

Chapter 5 presents the results of this thesis project. It describes how the findings in chapter 4 can be used to improve the Podio iPhone application.

Chapter 6 summarizes this thesis and provides a conclusion and a discussion around the premise and the results of this thesis.

2 Background

This thesis project covers a number of platforms, technologies and methodologies. The following section provides an introduction to some of these concepts to give context to the work performed throughout this project.

2.1 Podio

Podio is an online work and collaboration platform. Its goal is to encourage and ease collaboration in professional teams. Podio provides a flexible platform for building custom workflows using simple containers and constructs to meet many different project or team needs.

Organizations, Spaces and Apps

In Podio, an *organization* is the top-level entity. Within an organization the user can create *spaces*, typically one per team, department or project. Spaces can be either invite-only or open for any member of the organization to join. Spaces have their own activity stream, tasks and members. Figure 1 shows the space view presented when selecting a space in the gray navigation bar to the left. The activity stream, shown in the center of figure 1, is an aggregate view of all recent activity within the space. A member can post status messages to the stream and any action within the space generates a stream event.

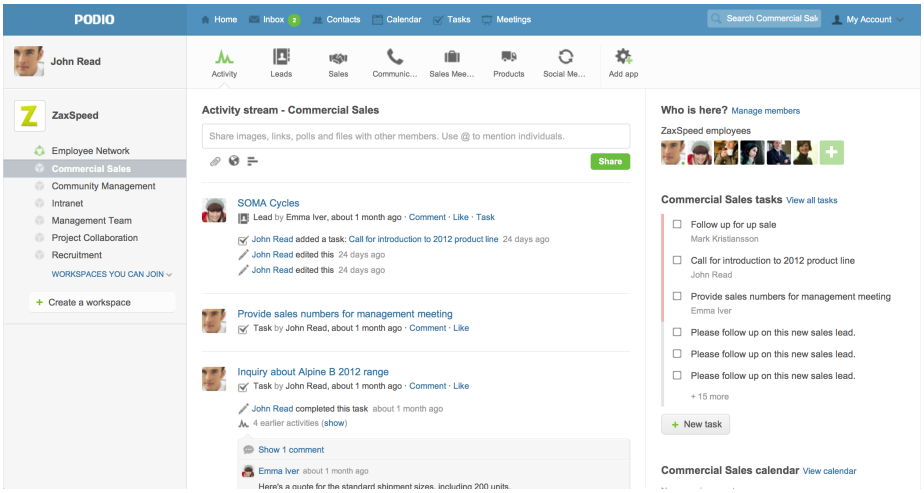

Figure 1: A Podio space

Spaces also have *apps*, the main tool for building custom workflows. Apps are shown as a horizontal list of icons above the space activity stream. Throughout this report, the term *app* is used to refer to the Podio platform apps, while *application* is used for the mobile client application. An app is simply put a container of posts, or *items*, similar to a table of rows in a database. What makes them powerful is the ability for the user to create custom apps from a set of basic *fields*, similar to columns in a database. There are multiple types of fields, for example text, number, category, progress and contact fields. There are two ways to add apps to a space. The first one is to use a simple drag-and-drop interface called the *App Builder*, where users can easily create their own apps using the basic field types. Once an app has been constructed to contain a number of fields, users can start creating *items* within the app. Each item contains a number of values for each of the fields in the app. As users continue adding items to the app, it becomes a useful and expressive container for the data. The second way to get apps is to use the *App Store*. The App Store contains pre-built apps created by Podio users. Any user can build a new app and share it on the App Store. There are also *App Packs* that are collections of apps set up to work together to solve a particular problem that might require

multiple apps, such as general project management.

Anatomy of a Podio App As mentioned apps are fundamentally collections of items, and items in turn are posts with values for the various app field types. Apps and their field types can be modified at any time. However, changes made to apps with existing items only affects newly created or edited items. Some field types hold a single value, e.g. the text field, while some can contain multiple values, e.g. the image field. All items can be interacted with by liking, commenting or creating tasks on them. Additionally, apps provide a number of optional features like the ability to add a star rating, approve/disapprove or RSVP to items. Items with a date field can optionally be added to the Podio calendar. This makes it possible to create event-based apps, e.g. for meetings.

It is also possible to create cross-references between apps within a space using a reference field type. This allows the users to hierarchically structure workflows into subcomponents and aggregate the values of referenced apps. For example, for a time tracking scenario one might create a *Projects* app and a *Time Sheets* app. The *Time Sheets* app would have a reference to the *Projects* app and a duration field. When a new time sheet item is created, a project item reference and a duration time is included. The project item can then use a calculation field to sum up the duration of all time sheets referencing the particular project. By creating these relationships between simple building blocks users can easily build rather complex workflows.

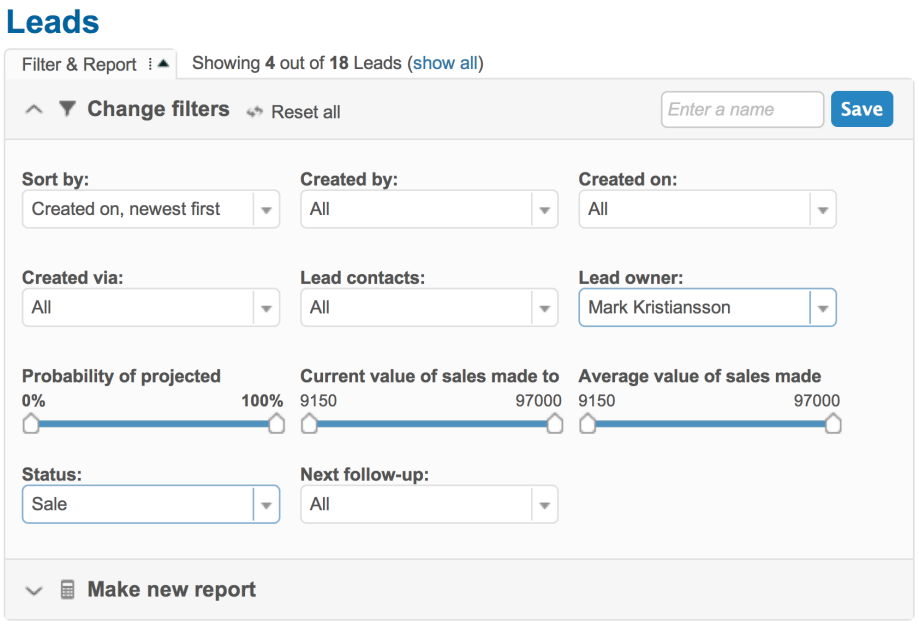

Figure 2: Filters panel

Another useful app feature is the ability to create filters. Filters are used to limit the number of items shown when browsing an app, based on the fields it contains. For example, in the case of the filters panel for the *Leads* app shown in figure 2, the view can be filtered to only display leads where the *Lead owner* is a certain person and the *status* is set to *Sale*. Filters can therefore be used as a simple reporting tool to track the performance of lead owners.

API and Client Access

Podio provides an external API [4] for third-party client applications. The API acts as the backend in the decoupled architecture of the Podio platform and is the only component with immediate access to the database. The API is based on REST and uses JSON [5] for data representation and the OAuth2 [6] protocol for authentication.

The Podio web site is a Ruby on Rails [7] application that uses the Podio Ruby client library [8] to access the API, i.e. there is no direct database access. Podio provides client libraries for many other programming languages, such as Python, Java, PHP etc. Because of the design of the Podio architecture, there is no difference in the data accessible to a third party application and the official Podio web site, except for a few potentially dangerous actions such as deleting users. Any Podio user can register for an API key to gain access to the API.

2.2 State of Mobile Application Development

The popularity of mobile computing and mobile application development has skyrocketed in the last couple of years with the launch of the iPhone Software Development Kit (SDK) in combination with the Apple App Store in 2008 and the Android Market later that year.

The two major mobile platforms for the consumer market are Apple's iOS and Google's Android [9]. iOS is a mobile adaptation of the Mac OS X operating system used on Apple's personal computers. The iPhone SDK, released in 2008 and later rebranded as iOS SDK after the launch of the iPad in 2010, contains a collection of Objective-C-based frameworks for developing native iOS applications. Objective-C is an object-oriented programming language that is a strict superset of the C programming language. As with C, there is no garbage collection offered and the responsibility of memory management is given to the developer. iOS applications are distributed to users over-the-air (OTA) through the Apple App Store application available on all iOS devices. If the developer chooses to charge for an application, Apple keeps 30 percent of the profits of the sale. Applications must adhere to the Apple App Store Guidelines [10] and pass through a manual review conducted by Apple for each new version released.

Android is a Linux-based operating system developed by the Open Handset Alliance [11] led by Google. The Android SDK is Java-based, and applications

are compiled into Java byte code and executed on a customized Java virtual machine. Since Java offers garbage collection, there is no need for manual memory management. The Android Market is, similar to the Apple App Store, used to distribute apps to users. However, Android Market requires no review process and applications can be submitted at anytime and become available to users immediately.

While the two platforms are similar in many ways, not least because of the similarity in user interface and feature set, there is much debate over whether the curated strategy of the App Store is better or worse than the openness of Android Market. Because of its curated catalog, the quality of the applications on the App Store tends to be higher in general, while certain functionality available in an Android app might not be allowed in an iOS app. In addition, the review process of the App Store makes the process of submitting critical application bug fixes to users slower.

Another difference is the availability of devices. Android is a licensed operating system, meaning any device manufacturer can use it on their devices. In contrast, iOS only runs only on Apple's own devices such as the iPhone and the iPad. Because of this, the variety of devices offered by third party manufacturers like Samsung, HTC and Motorola are far greater. The competition in this market also drives down prices with the use of cheaper hardware and components, making more Android devices available at a lower price point than iOS devices.

There are other mobile application platforms, such as Blackberry OS from RIM, Windows Phone by Microsoft, webOS by HP that offer developers similar ways develop native applications. However, at the point of this writing none of them have gained enough popularity to motivate developing a native Podio application.

2.3 iOS

iOS was derived from Mac OS X and optimized to run on the iPhone's mobile architecture. It is therefore just like OS X a POSIX-based operating system. It is not, unlike Android, a licensed operating system and it runs exclusively on Apple's own hardware. Conceptually, the iOS platform consists of four abstraction layers [12]:

- **Cocoa Touch** provides high-level frameworks for easy application development. Frameworks interacting with the lower layers are also accessible to developers in case Cocoa Touch does not expose the desired functionality.
- **Media** provides the functionality related to graphics, audio and video handling. This functionality is divided into frameworks such as Core Graphics, Core Animation, Core Audio among others.

- **Core Services** includes a number of system frameworks and features. Examples of such frameworks are Core Data, Core Location, CFNetwork and Foundation.
- **Core OS** provides the low-level system features to support the functionality of the other layers. This layer handles things like threading, networking, standard I/O, memory management etc.

iOS Applications are developed primarily using the Objective-C programming language but C, C++ and Objective-C++ can also be used due to the fact that Objective-C is simply an extension to the C programming language. Applications are developed using Apple's Xcode Integrated Development Environment (IDE), and are debugged either using the iPhone Simulator application bundled with Xcode or using a tethered physical iOS device.

3 Problem Description

The initial Podio iPhone client application existed prior the start of this thesis project. The application contained a very limited subset of the Podio platform features and most importantly was missing Podio core features such as organizations, spaces and apps. The existing application had also been experiencing a number of issues related to concurrent requests and the data model implementation.

This thesis aims to identify and evaluate the problems experienced with the existing Podio iPhone client application. In addition, the features providing the primary appeal of the Podio platform, i.e. organizations, spaces and apps should be implemented in the iPhone application. The context in which mobile applications are used is vastly different from the one of the Podio web site and imposes a completely different set of constraints in terms of connectivity, user experience, performance and power consumption. These constraints are often general and are commonly encountered when developing a mobile application as an extension of an existing product or platform. Therefore the lessons learned in this project can potentially prove to be valuable in other similar cases. The purpose of this thesis project is to look at several of the aspects and problems related to mobile application development, in particular for applications that act as clients to a remote API, and suggest and argue for their respective solutions.

3.1 Research Areas

Rodrigues et al. [13] identify a number of issues and challenges that need consideration when moving from the robust web integration of a desktop computer to a less reliable mobile environment. These include data transfer optimization, performance, user interface design etc. Additionally, other potentially problematic areas include, but are not limited to:

- Data Persistence and Request Management
- User Interface
- Changeability
- Reusability

Data Persistence and Request Management

Perhaps the hardest problem related to mobile application development is that of data persistence. Without a fast or even reliable connection to the primary application database, there is a potential risk of bad performance, reliability and perceived quality. There is also the problem of the user interacting with stale or even deleted content because the local data cache has not been updated with the latest server side changes. As argued by Christensen [14], the client application needs to handle these inconsistencies between the server and the client data in a manner that keeps the application responsive to user interaction.

User Interface

The Podio web interface contains a lot of features and is the primary interface for interacting with Podio. Not every feature makes sense or is feasible in the mobile context. Careful considerations and tradeoffs need to be made when defining the mobile Podio experience. Existing features need to be adapted and rethought to fit into the physical and environmental constraints of a mobile device.

Touch-based interface methods like the multi-touch technology used by the iPhone allows the use of advanced gesture recognition to trigger various actions. The feasibility, advantages and potential disadvantages of binding actions to gestures versus visual User interface (UI) elements like buttons should be evaluated.

Both iOS and similar popular mobile platforms such as the Android platform provide views capable of rendering web content within native applications. The possibility to create reusable interface components within the native application using web technologies such as HTML5 and JavaScript should be evaluated.

Changeability

Podio is a rapidly changing product and as such, the API changes just as rapidly. There is a need to identify the structure of and the ways in which the API and its data change. The mobile client application needs to be robust enough to deal with such changes. Also, when API changes do require an application update

to be released, these client side changes should be as small as possible and easy to make.

Reusability

Ideally, the software produced during this project should be design in such a way that it can be reused in other Objective-C-based client applications. It should also be designed in a way that minimizes code duplication within the existing code base for common use cases.

3.2 Method

The research method of this project is structured as follows:

First, a pre-study is carried out in order to assess the state of the existing Podio iPhone client application and identify the cause of its existing problems. Based on the result of this assessment, available solutions are investigated and presented along with their respective advantages and disadvantages. The study also aims to highlight scenarios, other than that of the Podio application, where one solution might be more appropriate than another one.

Based on the results of the pre-study, a prototype implementation is developed to demonstrate how the existing iPhone client application can be improved. A detailed description of the implementation is also provided to explain how the prototype application addresses the problems identified in the pre-study.

The process of implementing the prototype application is logically separated into a number of phases, where each phase includes implementing a particular part of the intended functionality. These phases include:

Improved data persistence and request management The research in this area and its resulting implementation is to be performed in parallel with the other phases and should be introduced into the main code base once completed. The benefit of this approach is that the knowledge of the existing application code will improve throughout the project and provide a good understanding of the problem to be solved. Performing a detailed evaluation up-front is not expected to generate reliable enough results in order to understand the problem domain and the features and particularities of the Podio API.

Items An item is a single entry in an app containing a number of value fields. Value fields can be of various types and should therefore be presented differently in the user interface. Items are commonly referenced from other places in the iPhone application, currently by providing a simple hyperlink button to the item web URL that opens up the item view in a non-authenticated web browser view. This requires the user to log in again to view the item in a browser window using the regular web interface. This phase includes evaluating possible

implementation approaches to provide an in-app view of an item that allows user interaction and provides a responsive user experience.

Organizations, spaces and apps This phase includes implementing functionality to allow the user to browse organizations, spaces and apps within the iPhone application. An organization contains a number of spaces that in turn contains apps and other components such as the activity stream and tasks. An app contains items with value fields defined by the app configuration. Users use a simple drag-and-drop interface in the Podio web interface to create apps.

Create and edit app items This phase includes providing the ability to create or edit existing items directly from within the iPhone application. Each individual value field type requires a custom edit mode. This iteration depends on the previous iterations to view apps in order to have a context to which new items can be added.

4 Research and Proposed Solutions

The research part of this thesis project focuses on two main topics. The first one is the process of adapting an existing web-based service to the mobile context, both in feature set, interaction and interface design.

The second part is concerned with data persistence and request management. The conditions under which mobile devices are used affect the behavior of mobile applications. The environment of a mobile device varies more than that of a desktop computer, and as such strategies need to be developed to minimize the negative effects on the user experience due to these variations.

4.1 The Mobile Context

When redesigning an existing desktop product or service for mobile use, one needs to carefully consider the differences between the two platforms and environments. Mobile devices impose hardware restrictions such are screen size, input methods and limited system resources such as CPU and memory. The mobile form factor in addition to new input mechanisms such as touch-based screens also introduce new UI paradigms and interaction patterns [13]. There are also external environmental factors to consider such as network connectivity and the availability of on-board device sensors. Dehlinger and Dixon [15] suggest mobile applications should have defined self-adaptive requirements, specifying how the application is to deal with these external environmental changes.

A Podio app on the Podio web site contains a large number of UI elements and functionality. Wasserman [16] argues not to treat the mobile application of an existing service as an afterthought, but as its own product with unique requirements. It is therefore important to not start out with the premise that the mobile

version of the service should offer the full feature set of its web-based counterpart. Instead, one should realize the context and conditions under which one should be used over the other and then try to leverage their respective strengths. As noted by Rodrigues et al. [13], people are more likely to use their mobile devices and applications when their main computer is inaccessible. Therefore, implementing the most common use cases should be of highest priority for mobile applications. Being selective in specifying the mobile feature set is therefore essential to feature planning and the user experience.

As described by Christensen [14], "smart" mobile devices come with the strength of being able to take advantage of their context awareness features. Many smartphones of today boast advanced features like GPS capabilities, digital compass and integrated camera. Most mobile smartphone platforms provide access to these components to application developers through their respective SDKs. Mobile applications can therefore enhance the user experience by providing the service with additional contextual data from the user's device, not available to a desktop user.

When determining the feature set of the mobile version of a web-based service, one should consider the frequency of the actions and the work associated with accomplishing a given use case. Wasserman [16] argues that because of the limited screen space of mobile devices, user interface design is more important than ever. He also argues that the primary use case for mobile applications is to accomplish simple and quick tasks, as opposed to using more complex features available on the full-featured web site. In Podio, lightweight actions such as liking, commenting or sending messages are frequent and quick and should be made easily accessible in the mobile application. Other more heavy actions, such as creating an app, are less common and are more time-consuming, and might therefore be more suited to remain a desktop-only feature. The action of creating an item is somewhere in-between but still common enough to be considered an important use case in the mobile application, while editing an existing item is an even more common and lightweight action. Figure 3 shows a way to visualize the categorization of a number of actions on Podio, where the frequency with which a regular user performs an action, as well as the time the action takes to perform has been plotted. The top left quadrant of the graph includes the actions considered most essential to include in the mobile application.

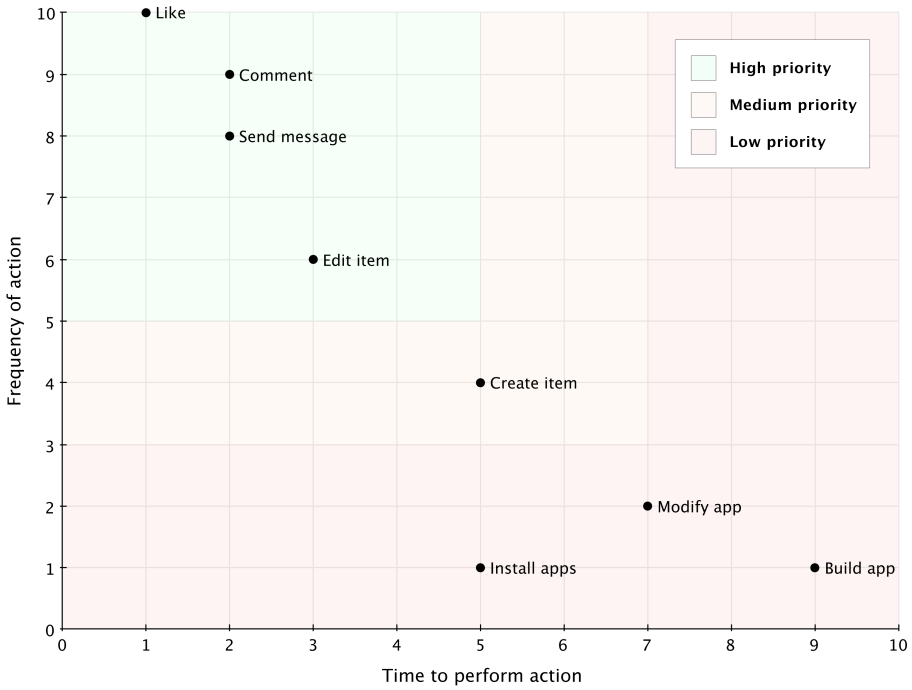

Figure 3: Plot of the respective frequency and time of various actions

There are also features that are simply not feasible in the mobile context, such as exporting or importing app data to a spreadsheet file.

User Interface Considerations

The presentation of a Podio app on the iPhone requires consideration of the conventions, restrictions and capabilities of the iOS platform and the iPhone device as stated by iOS Human Interface Guidelines [17]. As described by Benbunan-Fich and Benbunan [18], the ability for a user to perform a given task is heavily dependent on the familiarity of the environment. Therefore, common iOS UI paradigms should be used for the UI to be intuitive to the average iOS user.

It is also important to maintain the Podio look-and-feel and to add to the perception that the mobile application is a part of the Podio platform. The navigational patterns and information structure needs to as much as possible be aligned with that of the web site while still providing a platform-like experience.

Interface Partitioning and Separation of UI Concerns A desktop screen provides far more screen real estate than a mobile device screen to present app

content, which is potentially very large. On the iPhone, the small screen requires the content to be separated into multiple individual views while allowing for easy navigation between the two. An example of this shown in figure 4 is the *Comments* section of an app item. On the web site, the comments are shown below the value fields and scrolling up and down between the two is rather easy. This layout is not a good fit for the iPhone since the item content area width is narrower, therefore making the vertical scrolling distance to the comments section much larger for items containing many field or large amounts of content. The iPhone scrollable UI elements do not offer direct manipulation of the scrollbar position in the way a normal web browser does, so the user would need to manually scroll to the top or bottom.

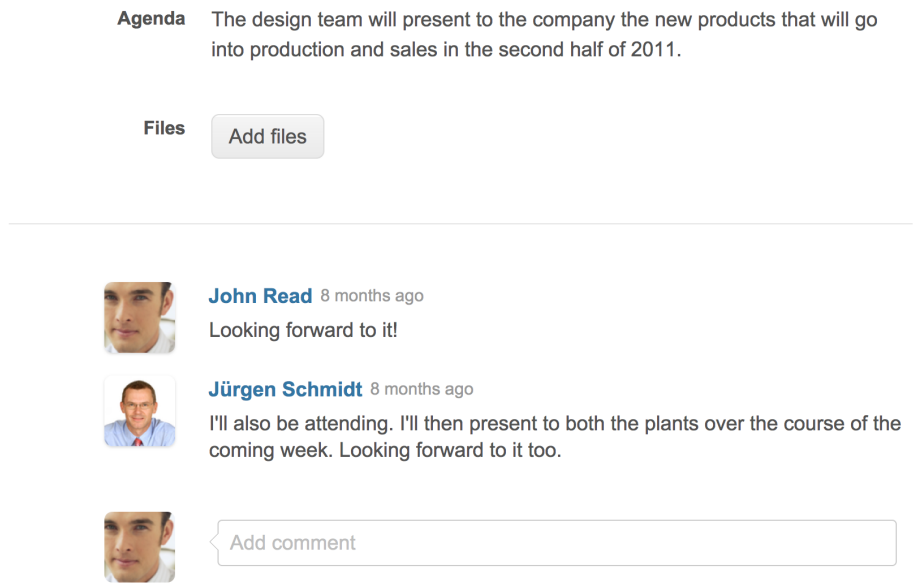

Figure 4: Item comments on the Podio web site are shown below the value fields

Another argument for content separation other than scrolling is minimizing view content. Each unit of functionality within a single view requires additional UI components for interaction, decreasing the available space for displaying content. For example, displaying the comments below the value fields requires an additional floating text entry bar to post a new comment. The entry bar would be placed at the bottom of the view and would take away additional screen space from the item content.

Based on the above considerations, related item content such as comments should ideally be displayed as separate views, easily reachable from the app item content view by a single button tap. This way the user can easily toggle between the app item and comments view without losing its scrolling position in either one.

Another example of seemingly independent item-related content is the associated files. These could also potentially be displayed in a separate view. In this case one needs to consider how they are displayed on the web site, which is similar to how an app item field is displayed with a header and sub-items. Considering that the interaction with app item files requires no additional UI components such as the entry bar in the comments case, it is reasonable to display these files below the app item fields similar to the web site interface.

Gesture-Based Interaction When using gesture-based interaction, it is important to consider the discoverability and intuitiveness of a given gesture. Lei and Wong [19] refer to this as the low affordance of gestures, and argue that gestures should be carefully designed to be simple, intuitive and not easily confused. To achieve this, they suggest limiting the number of available gestures or use context-specific gestures with supporting visual hints. Another way to improve the usability of gestures is to use well-known platform-specific gestures common across application. An example of such a common gesture in iOS is swipe-to-delete in lists. Although platform vendors in general provide user interface guidelines for application developers and designers, the are also cases where the community invents new UI paradigms that become so popular they become a kind of de facto standard for the platform. One example of this is the pull-to-refresh gesture invented by Brichter for his iPhone Twitter client application *Tweetie* [20].

The discoverability, and therefore the accessibility, of a gesture should reflect the importance of the action. A rarely used feature could be triggered by a more complex gesture, while common ones such as deleting an entry in a list, should be easier to access. In terms of providing contextual actions, gestures are similar to the classic desktop context menu that appears when a user right-clicks on a mouse. However, gestures are less scalable since the number of comfortable gestures is limited and of various complexity while a desktop context menu provides multiple equally accessible actions.

Presentation Techniques

To implement the item view, the most complex view to be implemented in this project, a number of techniques can be used. The item view contains a number of fields of various type, as well as files and metadata such as the item title and creator. It should also be possible to edit existing and create new items. The requirements of the item view are:

- Integrated and accessible from within the existing iPhone application
- Native or near native look, feel and performance
- Already viewed items should be accessible offline
- Intuitive to users of both Podio and the iOS platform

In addition, the following characteristics are desirable but not required:

- Cross-platform
- Offline support
- Adding apps to iPhone home screen
- Adding remote updatable field types

There are several approaches supported by the iOS platform to be considered:

- **Native UI framework** Cocoa Touch includes a native UI framework named UIKit. The existing iPhone application is built using this framework. Views are created and composed programmatically or using Apple's Interface Builder tool that is a part of the Xcode IDE.
- **Embedded web content using UIWebView** UIKit provides a native view classs, UIWebView, used to render web content within a native application. This includes HTML, Javascript and CSS support, and it uses the open source rendering engine WebKit [21] to render its content.
- **Third party cross-platform framework** With the recent rise of the mobile application development industry and the number of available mobile platforms, several new tools and frameworks have emerged which enables cross platform application development based on web technologies such as HTML and JavaScript. A single code base can then be used to output native applications for multiple platforms. Examples of such platforms are Appcelerator Titanium [22] and PhoneGap [23].

The following sections investigate the above technologies and determine the most suitable solution for implementing the app item view.

Native UI Framework Implementing the item view natively means using the UIKit UI framework provided by Apple as a part of the iOS SDK. UIKit, written in Objective-C, has been optimized for the iPhone and iPhone platforms and performs very well on its hardware. The UIKit controls and UI components are available and common, although customizable, across all native iOS applications. It also includes navigational components that implement the common navigation and interaction patterns of the platform. Therefore, using UIKit would provide the user with interface familiarity from the rest of the platform when using the Podio application. The iOS platform uses hardware-accelerated animated transitions extensively to provide context to user actions, improving the user's understanding of the view hierarchy and interface layout [24].

The interface is created programmatically or using Interface Builder, a subcomponent of Xcode. Views can be immediately accessed and manipulated

from the application code using Objective-C. Because of Objective-C being a compiled language and the hardware acceleration of the user interface, UIKit interfaces provide extremely smooth animations and near instant user interaction feedback, traits that iOS users have come to expect from iOS applications in general.

Additional authentication is not a problem when using UIKit, as the views are just presenters of the application data. Also, since the existing iPhone application was initially developed using UIKit no additional integration is needed.

Using the native view framework is also an advantage when dealing with locally cached data. When using Core Data, implementing the view layer using the same SDK makes integration easy and Cocoa Touch provides a number of ways to easily associate and present local data. One such example is the *NSFetchedResultsController* class, which observes changes in the model layer and notifies the controller object of when and how to update the interface.

The interest in Objective-C has skyrocketed since the launch of the iPhone SDK in 2008, moving from position 38 in October 2006 to being the 6th most popular programming language exactly 5 years later according to Tiobe Software's *Programming Community Index* [25]. However, because of it's tight coupling with Apple-related platforms it is still considered a niche language in enterprise software development. Hence, a decision to pursue a development project using the iOS SDK requires a commitment to learning Objective-C or hiring someone already familiar with the language. Therefore, a disadvantage of using the native SDK is the added learning curve, a fact that is heavily emphasized by competing third party platforms providing cross platform solutions based on well-established web technologies such as HTML and JavaScript. However, if knowledge of Objective-C is not an issue then productivity and development speed can be considered as good if not better than other web-based frameworks, not least because of the available development tools available and the quality of the iOS SDK.

The greatest disadvantage of using the native SDK is platform dependency. Podio wants to offer solutions on all major mobile platforms used in business and as such, minimizing effort and development time by using cross-platform solutions has a clear advantage.

Embedded Web Content Using UIWebView UIKit provides a custom, called *UIWebView*, a view class capable of rendering web content. It is based on the WebKit [21] rendering engine just like the default iOS browser application Safari. It provides rich support for technologies such as HTML, CSS and JavaScript.

When using a UIWebView, there are two options; loading the remote web page with the item content from a server, or bundle the web page as a resource within the application package and inject the desired data into it. Also, web views are inherently slower to render their content than a native view based on its reliance

on a generic web rendering engine and the Document Object Model (DOM). For interactive views, this can negatively affect the user experience since users have come to expect a certain level of responsiveness from the platform.

The first option of fetching the item content from the server has the advantage of being reusable across platforms. Therefore, it minimizes the effort of creating custom views for each individual mobile platform using their respective native SDK. For example, an item view web page could be reused in a native Android application that provides a view component similar to UIWebView. However, the design and behavior might anyway need to be adapted depending on the client platform to conform to the client specific interface paradigms. However, attempting to maintain client-specific customizations of a shared component decreases the greatest benefit, namely cross-platform reusability. In contrast, ignoring client differences by having the interface behaving identically on all platforms could have a jarring effect on the user experience when a user action does not yield the expected result for that platform. The second option of bundling the web content as a local resource in the application package has fewer advantages since it cannot be controlled remotely. Other than that the same reasoning applies to both options.

Another issue is integration with the rest of the application. In a highly functional view such as the item view, a user interacts with the view to preview images, edit text etc. It is not possible, without complex workarounds, to trigger native application code from for example JavaScript. In the case of loading the content remotely, it would also impose a potential security risk. Therefore the JavaScript code running in the web view is restricted to the APIs exposed to it. Hardware resources such as the camera or address book are therefore inaccessible. However, JavaScript can be injected and evaluated into the web view from native code, which makes it possible to populate a skeleton HTML resource from application data.

Tools available are also a factor. Objective-C and use of the native SDK is the default assumption on the iOS platform and its supporting tools. Xcode and its companion applications, such as Interface Builder and Instruments, provide powerful development and debugging tools. When developing complex logic and javascript for a web view, the tools are more limited.

Overall, using embedded web-based views in a native application is a reasonable solution when presenting static or near-static content with heavy formatting and various resource types such as images and styling. For functional views that require deep integration with the rest of the application, the advantages are seemingly overshadowed by the disadvantages. The issues with using web-based technologies for building highly functional mobile applications can in part be explained by the "Impedance Mismatch" problem presented by Mikkone and Taivalsaar [26], where they argue that the original purpose of the web as a document-sharing platform has a number of inherent problems when used as an application platform.

Third Party Frameworks The success of the Apple App Store has caused a wave of third party application frameworks to appear, such as Appcelerator Titanium [22] and PhoneGap [23]. The advantages they offer are that apps can be developed using web technologies such as HTML, JavaScript and CSS instead of Objective-C, a more low-level and less common programming language. However, PhoneGap and Titanium accomplish this goal in different ways. PhoneGap is primarily used to wrap a web application in a web-based view, and provides an extensive JavaScript API to access native platform features and hardware, such as the built-in camera. Titanium on the other hand uses the JavaScript source files to generate native applications for various platforms using their native UI controls.

Both of these solutions promote the advantage of not having to learn Objective-C and instead use web technologies that are considered more common and advantageous in their cross-platform support. In addition, they emphasize the time-to-market advantage of being able to use the same code to build apps for multiple platforms.

The greatest disadvantage of committing to one of these frameworks at an early stage is the risk of adding an extra layer and hence, an additional dependency, on top of Apple's iOS SDK. New features added to the iOS SDKs has to gain official support in the third party frameworks before becoming accessible to the developer. This poses a potential competitive threat since other applications might not be restricted by such a dependency. Also, given the young age of these frameworks, their future can be viewed as somewhat uncertain. This risk was manifested in October 2011, when Adobe Systems Inc acquired PhoneGap.

There is a clear place and advantage to these third party application frameworks, but for a long-term application projects one should consider the risk of the added dependency. For short-term projects or for companies with limited development resources but high demand for multi-platform presence they offer a good alternative. For example, PhoneGap could be ideal when developing promotion applications for events such as festivals, sporting events and the like.

For this project, there is an existing Podio iPhone application built using the native iOS SDK. Pursuing the use of a third party framework would not be feasible since all existing functionality would need to be rebuilt using the new framework. Additionally, the application needs to be maintained for a long time in parallel to the other Podio client interfaces and the benefits of these third party frameworks does not motivate the risk of an added dependency.

4.2 Request Management and Data Persistence

To keep application data up-to-date it needs to be regularly fetched from the Podio API to update the local data cache. As argued by Gopal [27], caching data on the client device itself is important to be able to interact with already downloaded data when network connectivity is unreliable or lost. Maintaining a data cache in a persistent model layer also makes it accessible to multiple view

controllers of the application.

Because the main thread in an iPhone application is responsible for reacting to user input and rendering the user interface, the network requests to the Podio API needs to be delegated to background threads to keep the user interface responsive. The following sections describe how to balance request and data management and keeping the user data up-to-date while keeping the application responsive to user interaction.

4.2.1 Stale Data and Validation of API Actions

Each Podio API request is associated with a user account by including an access token string in the "Authorization" HTTP header of every request that requires authentication. The access token is used by the API to identify the user who wishes to perform the operation. The API uses a permission system to determine if the authenticated user is authorized to perform the requested action on a particular resource object. In addition, when requesting a resource object from the API, the response includes a list of rights dictating the actions the user is allowed to perform on the given object. This allows the client application to adapt the user interface based on these rights, e.g. display an "Edit" button. However, if the client application were to, by accident, perform an unauthorized action, the API would return an error.

The API-side validation can be of use when working with stale data. Before any further discussion, one needs to define what is meant by stale data. As a client application, the remote Podio database is considered to contain the canonical state of all data. Prior to any action on a remote resource object, the object itself needs to be fetched by the client through an API request. A subsequent modifying request can then be performed on that object. This means that the time between the two requests always is greater than zero, which inevitable allows for the possibility that the data changed in between by another user or system event.

The conclusion is that the client application can provide a better user experience by being optimistic in its execution of actions, i.e. to act based on the state of the cached state of an object and by gracefully handling any API errors. There also needs to be a reasonable default behavior for when the data should be updated, to minimize the time between the fetch request and the modifying request. The conclusion is that a reasonable rule of thumb is to use the cached state of an object when requested to immediately populate the interface and at the same time initiate a refresh request for that particular object and update the interface once completed, as shown by figure 5.

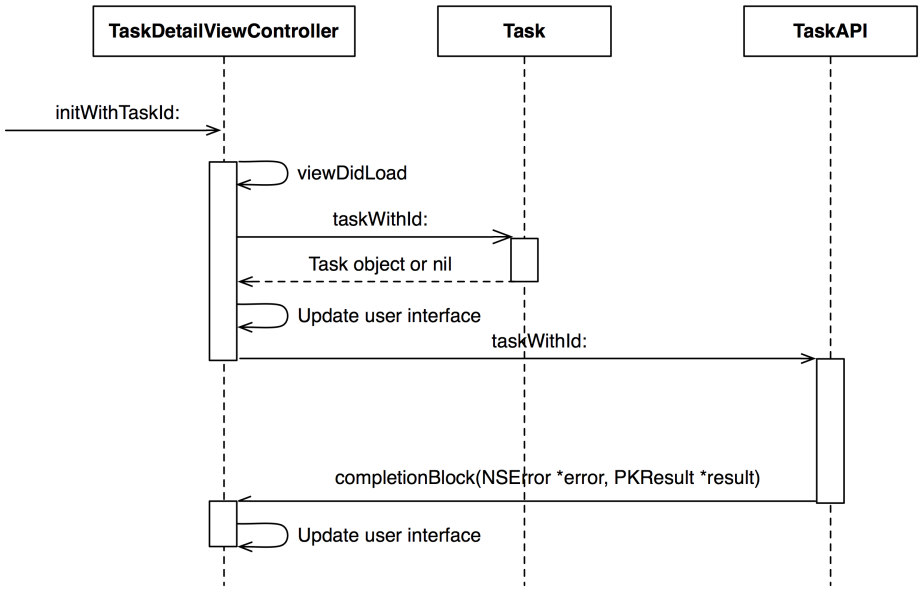

Figure 5: Sequence diagram for loading a detailed view of a task

The underlying data also follows different change patterns. For example, admin users are allowed to change the configuration and user permissions of an app, but these changes are less common than for example changing or creating items. By identifying these usage patterns, one can make informed decisions on when data needs to be refreshed from the API, and the consequences of using stale data.

4.2.2 Request Management

Request management is concerned with managing network request to the Podio API. Requesting data from a remote server takes a considerable amount of time and should be handled on a background thread. Request management includes initiating, monitoring requests and handling the returned data.

API Requests and Threading iOS applications are multithreaded. The main thread is responsible for executing the main run loop that listens for user and system events. It performs the corresponding action and updates the user interface accordingly. It is critical to the user experience to not block the main thread with long-running synchronous operations such as network requests or file system operations, as it would cause the interface to become unresponsive. Therefore, the main thread should only be used for short tasks and interface rendering. All other operations should be delegated to background threads.

The Foundation framework provides abstractions for asynchronous task execution through convenience classes such as *NSThread*, *NSOperation*, and *NSOperationQueue* to make the creation and management of background threads easier. The *NSThread* class represents a single executing thread while *NSOperation* is a more abstract concept meant to encapsulate a unit of work but it is not necessarily executed on separate thread. Instead, a *NSOperationQueue* instance manages a number of *NSOperation* objects and their execution based on their own configuration and that of the queue.

A *NSOperationQueue* provides the threads used to execute its operations. It is therefore convenient in an iPhone application to setup different queues for different long-running tasks. An operation queue can reuse the same thread for executing multiple operations as long as they are configured to run sequentially [28]. One such queue is used for all dispatched API requests in the application.

In the Podio iPhone application, *NSOperation* can provide a flexible and configurable way to execute API requests. A popular Objective-C library used for such HTTP-based requests is ASIHTTPRequest [29]. The ASIHTTPRequest library provides a custom subclass of *NSOperationQueue* called *ASINetworkQueue* as well as multiple convenient *NSOperation* subclasses that implements the HTTP protocol.

Threading and Core Data The Core Data framework defines several restrictions on how it can be used in a multi-threaded application. For further explanation of Core Data key classes and concepts read the introduction chapter about Core Data as a persistence layer [30].

It important to pay attention to the restrictions stated by the framework documentation when using Core Data in a multi-threaded environment. Perhaps the most important one is the fact that a single *NSManagedObjectContext* instance, used for accessing stored objects, can only be used by one thread at a time. This means there needs to be at least one object context exclusive to the main thread to update the interface, and one for every asynchronous background thread that interacts with the Core Data store.

The Request Lifecycle The typical lifecycle of an request made to the Podio API can be described as follows:

1. An action is triggered by an event such as user input.
2. The action is detected and handled on the main thread. An *NSOperation* object is instantiated and added to the network queue.
3. The main thread continues execution of the run loop while the network queue dispatches the operation. The network queue executes its operations concurrently or in sequence depending on its configuration.
4. The request completes.

5. The response data is passed to the main thread.

When the request completes in step 4, the application needs to take care of the returned data. The Podio API returns a response body containing data in the form of a serialized JSON [5] string. The JSON string needs to be parsed into a native collection data structure, such as an *NSArray* or *NSDictionary*. While there are many freely available open source libraries for parsing JSON, JSONKit [31] is seemingly the most popular [32] at the time of this writing. JSONKit is a full document parser, meaning it outputs nested *NSArray* or *NSDictionary* objects that holds the complete structure of the parsed JSON string.

In some cases with the Podio API, the response data does not represent an entire resource object. This is usually the case when creating, updating or deleting objects. In these cases, the response instead contains only the identifier of the related object. In this case the data can be passed to the main thread as-is, without further processing.

In the previous version of the iPhone application, passing data to the main thread was handled by using the notification mechanism [33] provided by the Foundation framework. It is based on the Observer [34] pattern and provides the ability for any object to register as an observer of a particular notification, identified by an *NSString* object, posted by any other object. Each notification is distributed through an *NSNotificationCenter* instance and is delivered to all observers registered with that notification center. One can choose to observe all notifications of a given type, or only the ones posted by a particular object.

One problem that arises when working with asynchronous background operations is the need to execute some custom code on the main thread once the request completes. There might also be a need to handle the success and failure cases independently. Hence, there is a need to identify which request finished. The previous request management implementation used the aforementioned notification mechanism to message all observing view controllers once a request completed. It provided some context to this notification by including an integer as a request type to identify which API operation was called. This showed to be an error prone approach since multiple view controllers could be active and registered to notifications of the same request types. For example, when an image was uploaded to create a new status message, the profile picture of the user would change since both view controllers used that same request type to upload a picture. This indicates that the request completion code needs to be executed with more granular control than using a request type, ideally on a per-request basis. The previous approach also resulted in large amount of boilerplate code since each new view controller needed to register and unregister to request completion notifications.

The second scenario and the common case when performing a HTTP GET request to a given resource URL, is that the response data is a serialized representation of a single or collection of remote resource object. Listing 1 shows the response for getting a single comment object.

Listing 1: Response string for fetching a comment

```
{
    "comment_id": 123456,
    "value": "This is the comment body",
}
```

The above data contains the serialized data fields of the remote resource object. To be able to effectively use this object within the application and persist it to the client database for offline access, it needs to be transformed into a native domain object and stored locally. The proposed strategy for doing so is described in the next section.

4.2.3 Data Persistence

After successfully receiving the data of a remote resource object it needs to be transformed into a native domain object for easy use within the native application. The iOS system frameworks are in large parts built using the Model-View-Controller (MVC) pattern [35]. Therefore, Apple encourages developers to build model-driven applications [36] where multiple view controllers can share and react to changes in the model layer, as shown by figure 6.

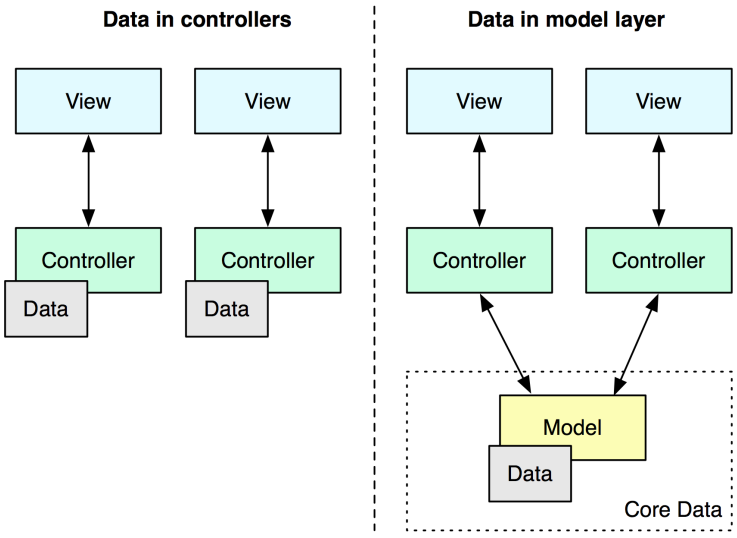

Figure 6: Keeping data in controllers vs. a separate model layer

There are several benefits to keeping data in a separate model layer:

- **Shared data** - Avoids duplication of data and redundant API requests if multiple controllers are interested in the same data.

- **Minimizing memory usage** - Keeping application data in the model layer, for example in a database, removes the need to keep everything in memory and allows for lazy loading [37] of model objects. Controllers are always kept in memory once allocated and will therefore keep all its associated data in memory as well.
- **Persisting data across application launches** - Controller objects are continuously allocated and deallocated, sometimes without the control of the application developer.
- **Offline access** - If Internet connectivity is lost, the user is still able to browse already downloaded data.

To be able to store the response data locally a persistence strategy is needed. iOS offers multiple solutions for this as described in the next section. There also needs to be a mechanism for transforming the collection objects generated by the parsing step into native persist-able domain objects, a step that from here-on is referred to as *data mapping*.

iOS Persistence Solutions The iOS platform offers a number of solutions for persisting data:

- File system
- SQLite
- Core Data

Data can be stored as files in the file system. Each application on iOS is assigned a sandboxed file system directory in which it can store arbitrary files. However, this approach is not suitable to store complex objects structures since files and directories are not ideal for querying or filtering data, but rather have to be traversed and inspected.

SQLite is a library that implements a file-based full-fledged SQL database. SQLite support was included in the initial version of the iPhone SDK, while Core Data support was not added until iPhone SDK 3.0.

Core Data is an framework for object-graph management and persistence. The underlying storage mechanism is configurable and Core Data provides four default storage types; SQLite, XML, binary or in-memory, as well as the ability to create custom ones. Given the size of the Podio domain model and the desire to maximize performance as well as minimizing memory usage, SQLite is seemingly the best option according to Apple's *Store Types and Behaviors* chart [38].

Core Data provides change tracking for objects in the graph including insertions, updates and deletions. It provides property validation, undo management and

other useful features. All changes to the database are made through an *NSManagedObjectContext* instance. A single object context should be exclusive to a single thread to avoid concurrency issues, and changes in contexts are communicated across threads using the Foundation framework's notification mechanism [39]. Because of Core Data's use of notifications, any part of an application can observe changes made to an object context. For example, a view controller can be notified of a deleted object and remove that object from the view. There is a performance overhead associated with using Core Data [40], but for a complex domain model the benefits typically outweighs this drawback.

In Cocoa Touch, built in large parts around on the MVC pattern [34], Core Data provides a sophisticated model layer. Because of its use of notifications, it is easy for the view layer to observe and update itself in response to changes in the model. When using Core Data as a persistent cache to store remote data it is possible to simply have the main thread use local data to populate the view, and to trigger asynchronous refresh operations. Upon completion, the remote data is imported into the database on a separate thread. The view layer is notified of these changes on the main thread and can update itself accordingly. In the event of lost network connectivity the user should still be able to browse any previously stored data. This approach is encouraged by Apple to improve the user experience [41].

The above indicates that Core Data is the most suitable persistence solution for most non-trivial iOS applications. The next section describes the process of populating the database with native domain objects created from remote response data.

Data Mapping The process of data mapping is to lookup, instantiate and populate native model objects from the response data returned by the API. Optionally these objects can also be persisted locally. Section 4.2.3 suggests that Core Data would be the preferred persistence strategy for the Podio application. Core Data enables seamless persisting of native Objective-C objects. The data mapping process can be a time-consuming task for large data and should therefore be delegated to a background thread. The application code interacts with the Core Data persistent store through an *NSManagedObjectContext* instance. As previously mentioned, object contexts need to be exclusive to a single thread to avoid concurrency issues.

Core Data was used in the previous version of the iPhone application, where the mapping process was initiated by creating a background import operation. This background operation defined a custom mapping procedure for each request type unique to the resource URL of the request. This resulted in problems when parsing, mapping and persisting data because of code duplication. It also resulted in a close coupling between the domain objects, mapping logic and network code, which in turn reduces testability and maintainability.

The goals of the new data mapping process should be:

- **Decouple mapping from the domain object type** This makes it possible to use the same mapping code across applications with different domain objects. It also improves testability.
- **Customizable mapping definitions** Mapping definitions need to be easily defined. The definition grammar and features should be developed based on the needs of the iPhone application and the structure of the API response data.
- **Standalone and generic mapping process** The actual mapping process should be generic and decoupled from the underlying data store. Having a persistence layer should be optional. This is important if the mapping implementation should be reused in future projects possibly not reliant on Core Data for persistence.
- **Testability** The mapping process and definitions should be easily testable. This requires it to be decoupled from any network code while still being usable in such a context.

As described in section 4.2.2, remote resource objects are returned from the API in serialized form, where the name/value pairs of each JSON object represents the remote object's data fields. These data fields need to be mapped to native domain object instance variables. Objective-C 2.0 includes a language features called *properties*. Properties provide a simple way to automatically define accessor methods for instance variables, similar to getters and setters in Java. Listing 2 shows the use of properties as accessor methods using the dot notation.

Listing 2: Objective-C properties using dot notation

```
1  @interface Person : NSObject {
2    NSString *name;
3  }
4
5  @property (copy) NSString *name;
6
7  @end
8
9  @implementation Person
10
11 @synthesize name;
12
13 - (void)greet {
14   Person *person = [[Person alloc] init];
15   person.name = @"Sebastian";
16   NSLog(@"Hello %@", person.name); // => "Hello Sebastian"
17 }
18
19 @end
```

In addition, Foundation provides an informal protocol named NSKeyValueCoding (KVC) [42]. KVC can be used to call an accessor method indirectly by name

instead of explicit invocation. Listing 3 shows how the above `greet` method would look using KVC.

Listing 3: Objective-C properties using KVC

```
- (void)greet {
  Person *person = [[Person alloc] init];
  [person setValue:@"Sebastian" forKey:@"name"];
  NSLog(@"Hello %@", [person valueForKey:@"name"]);
}
```

Objective-C is a dynamic language with an extensive runtime library [43]. This library in combination with KVC provides a good foundation for building a generic mapping strategy using Fowler's *Metadata mapping* and *Data Mapper* patterns using *Reflective Programming* [37]. The ability to access object properties at runtime using a string representation of its name also suggests that a declarative approach can be used for constructing object-mapping definitions. While Fowler's examples focus on Object-Relational Mapping (ORM) mapping and this project focuses on mapping data from a network request, the mapping problems are fundamentally similar.

As stated by Fowler [37], using reflective programming and metadata mapping does have performance implications. However, he also suggests that relative to the execution time of database queries during mapping, the increase in execution time is unlikely to have a significant impact on overall performance. In this project the mapping process is a sub-task of requesting data from a remote API, which also includes a network request. In this context the added overhead is likely to be negligible.

As previously stated the mapping process should be independent from the underlying persistence strategy. However, object lookup, creation and deletion is usually very much dependent on the underlying implementation. Hence, the responsibility of these actions can be delegated to the persistence layer by providing a simple interface to implement. The *Repository* pattern [37] could be used to achieve this.

5 Results

The result of this thesis project consists of two main parts. The first part is the implementation of the feature improvements intended by this project, including the final navigational structure and interface design. The second one is PodioKit, a library developed to improve the request management and data persistence layer of the Podio iPhone application. These results are based on the findings and reasoning around best practices in mobile application development in section 4.

5.1 Application Design and User Experience

The resulting interface design of the iPhone application is based on the observations made in section 4, and primarily affects the app, item and space sections of the application. The goal being to consider both the user experience and the technical aspects of presenting data in a intuitive way.

5.1.1 App Item

As described in section 2, an *item* is a collection of *fields*. Fields can be of various types and need individual presentation and interaction logic. However, field types rarely change. An app item can be referenced from multiple places on Podio, including the activity stream and in tasks. Because of this it was the first piece of functionality to be implemented as it could be treated as a standalone view to be loaded from multiple places within the application.

Several technologies were considered in section 4 to create the app item view. The result was to create a native view based on UIKit and create custom controls for each of the field types. The reasoning being that since field types rarely change this is an acceptable one-time effort. There would also be enough time to release a new application version in a reasonable time in the event of an added field type. Figure 7 shows an example of how a simple item looks in the iPhone application and the web site respectively.

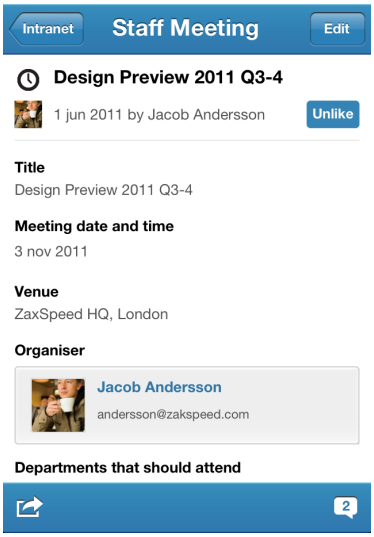

Figure 7: Item view

An app item view contains a lot of information. In order fit all of it on the smaller screen, it is important to separate primary information from secondary information and, if possible, even consider not including the latter. For example,

an item *category* field has a number of *categories* and each of them can be either selected or deselected. When only viewing an item, the more relevant of the two is the selected state. Hence, only the selected categories are shown in the iPhone, while both selected and deselected categories are shown on the web site. This provides extra context to the item data at the expense of extra screen space. However, this does not work as well on the iPhone.

Another example is the image field type. On the web site, the full images are shown inline. On the iPhone, showing full images would force the user to scroll long vertical distances. Also, fetching full size images for each item view would increase network data usage and latency, an unnecessary cost if the user never intended to view the full images. Instead, images are displayed in a small grid as shown by figure 8. It uses scaled down images from the server to give the user initial previews. The user can then tap a grid tile to view the full sized image.

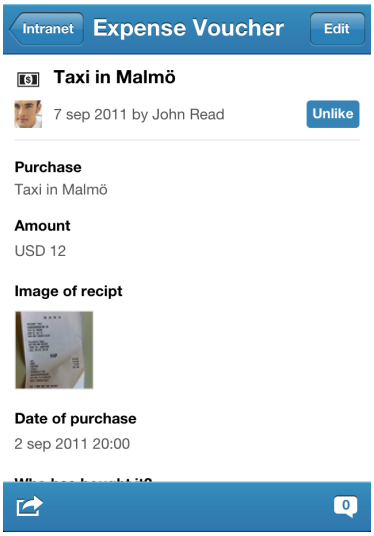

Figure 8: Taxi receipt shown in image preview grid

As an example of interface partitioning, the item comments are accessible by tapping the speech bubble in the bottom right corner of the screen in 7. This action initiates a flip animation of the entire view and reveals the comments on the "back" as shown by figure 9.

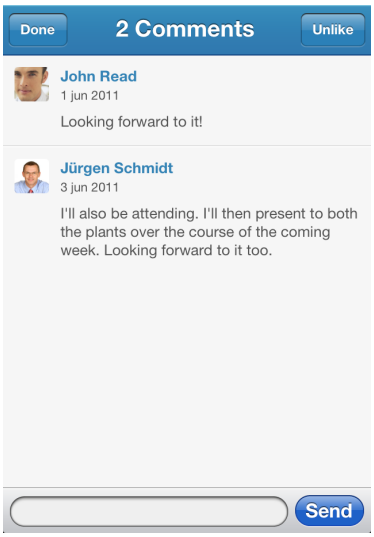

Figure 9: Item comments

In contrast, on the web site comments are shown underneath the item fields. This approach is reasonable since most users browse the web site using large landscape oriented monitors. The default orientation on the iPhone is portrait. This makes the content area narrower and as a result item fields and their content occupy more vertical space. This further increases the scrolling distance to access comments. Also, by providing comments with its own full-screen view there is more room for related view elements such as a button for adding a new comment and the comment text entry bar.

5.1.2 Spaces and Apps

A space contains a number of things that should be present on a per-space basis in the iPhone application:

- Activity stream
- Apps
- Tasks
- Contacts

A space is displayed as its own view with a bottom tab bar, as shown in figure 10.

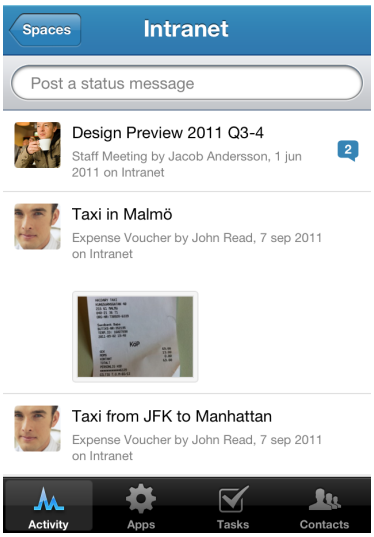

Figure 10: A space has several subsections, shown as tabs in the space view

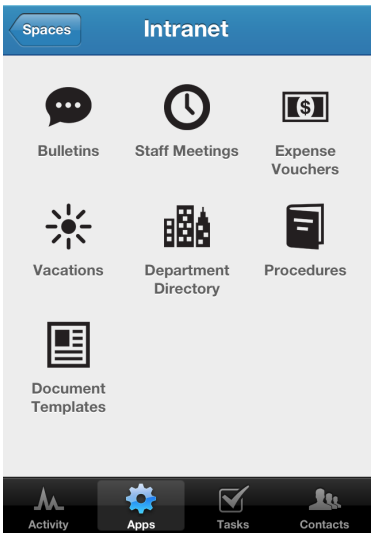

Figure 11: Space apps are shown as a grid to provide platform familiarity

Tapping the *Apps* tab displays the new interface for browsing apps within a space, shown in figure 11. The apps are laid out in a grid pattern, similar to the iPhone main home screen. Since the concept of Podio apps is similar to that of native iPhone applications, although simpler in functionality, presenting them in a similar manner provides some interface familiarity. For example, most users would recognize that the small page indicator at the bottom indicates that they can access more apps by swiping right to left. Some users might also know that an icon in this kind of grid layout might respond to a long press gesture. On the iPhone home screen, a long press puts the whole screen in edit-mode and

allows the user to reorganize or remove apps. In the Podio application's apps grid, a long press presents a contextual menu with a few options, such as "Add to Home Screen". Such an option is more of a shortcut since it is available in other places of the app, which this makes the discoverability of such a feature less of a priority. When the user launches one of the apps in the grid, she is presented with the app view, shown by figure 12.

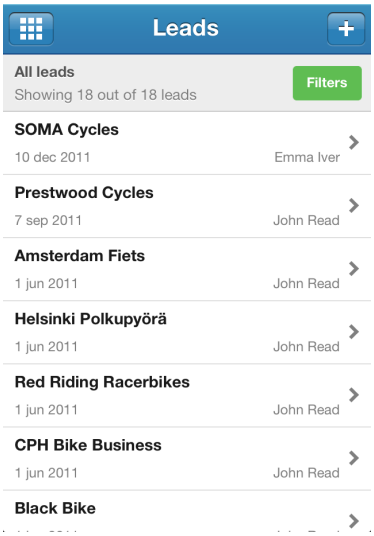

Figure 12: An app is displayed as a list of items

The app view is a simple list of items with the ability to filter previously created items by tapping the green "Filters" button. To delete an item in the list, a common platform-wide swipe gesture is used to reveal a "Delete" button. Even though not immediately discoverable, the deletion-gesture is common enough throughout the platform to use for a relatively common action such as deletion. The benefit is that no additional screen space is needed to always show a delete button.

Several features have also been intentionally left out because they do not make any or enough sense in the mobile application. Examples of such features are Excel export and import, the ability to modify apps or manage space or app settings.

5.2 PodioKit

PodioKit is a static Objective-C library built on top of the Foundation framework and a few other supporting libraries. It is intended to provide a reusable request management and data mapping mechanism for use with the Podio API. It is designed to work with iOS 4.0 or greater. The goals of PodioKit are:

- **Podio API interfaces** Provide a set of native interface classes to access the various Podio API areas.
- **Reusability and encapsulation** Provide an easy to use library that encapsulates communication with Podio API and is reusable across application projects.
- **Customizability** Allow for extending the functionality of the library and customize behavior depending on the application design and storage strategy.
- **Modularity** Allow for better testability by decoupling the various library components, such as network request management and data mapping.
- **Decouple model from mapping** Be model-agnostic and allow a client application to use domain objects of any type with existing mapping definitions.
- **Simple block-based API** Providing an opaque block-based API for making API requests and getting native domain objects in return.
- **Minimal configuration** Require minimal configuration to start using the library within a client application.
- **Test coverage** Provide good test coverage of the library code to ensure correct behavior.

The next couple of sections describe, in more detail, PodioKit and how it accomplishes the goals stated above.

5.2.1 The Asynchronous Request API

To provide the application with a way to perform an API request, PodioKit leverages the addition of blocks [44] in iOS 4.0. Blocks are a C-level implementation of *closures*, which are similar to anonymous functions but with the advantage that they capture the state of the lexical scope in which they are created. This means they keep a reference to any objects in scope at their creation, including the current object's self referencing variable `self`.

Blocks in Objective-C are treated just like regular objects and can copied to the heap by using the `copy` method defined by *NSObject*. Because of this it is possible to treat blocks as regular instance variables of an object such as an *NSOperation*. This provides a way to attach a callback block to an API request to be called upon completion. Since the API request executes in a background thread and the callback code should be executed on the main thread where it was created, the `libdispatch` library's `dispatch_async` C function is used. This way the correct callback code is executed upon completion of the API request. Identifying the callback code on an instance basis rather than using request types and notification-based callbacks eliminates many issues with the previous iPhone application. Listing 4 shows how using blocks can provide a short and simple way to initiate an API request and handle its result.

Listing 4: API request block

```
[[PKTaskAPI requestForTaskWithId:32452] startWithCompletionBlock:^(
    NSError *error, PKRequestResult *result) {
  if (error == nil) {
    // Success...
  } else {
    // Failure...
  }
}];
```

`requestForTaskWithId` returns a *PKRequest* object which defines the API interface and HTTP parameters to be used in the request. Then `startWithCompletionBlock` is called to add the request to the network queue. Listing 5 shows the actual creation of the request object.

Listing 5: API request creation

```
+ (PKRequest *)requestForTaskWithId:(NSUInteger)taskId {
  NSString *uri = [NSString stringWithFormat:@"/task/%d", taskId];
  PKRequest *request = [PKRequest requestWithURI:uri method:
      PKAPIRequestMethodGET];

  return request;
}
```

The *PKRequest* can be further configured on a per-application basis. For example, an object-mapping object can be attached to be used when mapping the response data to native domain objects. Object mapping is explained in detail in section 5.2.2. When adding the request to the network queue, it is passed to a singleton class named *PKRequestManager*. *PKRequestManager* is responsible for creating and managing asynchronous background request operations. The operations themselves are instances of *PKRequestOperation*, which is in itself a subclass of *NSOperation*.

NSOperation, and subsequently *PKRequestOperation*, also provides a way to cancel an operation by invoking its `cancel` method. Calling this method does not immediately interrupt the operation, but the operation code can check for a cancellation request at any time and terminate gracefully. This can be used in the iPhone application to avoid requiring the user to wait for a request to complete. Previous versions of the application presented a modal loading view on top of the current interface while performing an API request. It did so because it could not let user navigate to another view based on potentially stale data. In the version using PodioKit, a request can be explicitly cancelled when the user navigates away from the current view. The request operation then checks if it was cancelled before storing the local data and if so discards the response.

5.2.2 Data Persistence

The Podio iPhone application uses Core Data for persistence, and SQLite as its underlying storage. The domain objects are Core Data entity objects, i.e. subclasses of the *NSManagedObject* class. The API response data is mapped to the domain objects' property methods through a mapping process described in section 5.2.2. In brief, PodioKit uses subclassing of the parent class *PKObjectMapping* to define mapping behavior for domain objects. Object mappings can be reused and nested to create complete mapping structures for the response data.

The following sections describe the principles used when modeling the Core Data model schema based on the Podio domain model and the response data structure.

Response Data Structure The API response body contains a JSON formatted string representing the serialized data fields of remote resource objects. It is parsed by the client application into native Objective-C collection objects like *NSArray* and *NSDictionary*. A single resource object is represented by an *NSDictionary* while an *NSArray* is used to represent a collection of objects. A response object data field is referred to as an *attribute* and consists of a *key* and a *value*. An attribute is mapped to a native Objective-C object *property*, which is an accessor to an object's instance variable.

Listing 6 shows the API response body for fetching a single task object.

Listing 6: Get task response

```
{
  "task_id": 174753,
  "text": "Task text",
  "due_date": "2011-08-01",
  "description": "Task description",
  "comments": [
    {
      "comment_id": 574482,
      "value": "Comment on task",
    }
  ],
  "responsible": {
    "user_id": 123456,
    "type": "user",
    "name": "Sebastian Rehnby"
  }
  "ref_type": "space",
  "ref": {
    "space_id": 38212,
    "name": "Test space"
  },
}
```

The response data can be of varying depth with several nested object types. For example, a top-level domain object can have a relationship to a collection of objects of another entity type. A *Comment* object might have an property field named *files* containing a collection of *CommentFiles* objects. The nesting of objects in a single response is essential in API design [45] to avoid requiring the client to make additional requests for sub-objects. However, it is a tradeoff since a larger response body increases response time and data usage, and as a result potentially has a negative effect on the user experience. Therefore it is important to know in which context the data is to be used. For large collections of objects, pagination can be used to limit the number of objects included in the response.

The following attribute types for a resource object have been identified:

- **Static attribute** - The type and structure of the value are always known. This is the most common type of attribute and can be immediately mapped to a native object property.
- **Dynamic attribute** - An attribute where the value format varies depending on another static attribute, e.g. a "type" field of the same object. For example, the task object in listing 6 above has an attribute named *ref* whose content depends on the value of the static attribute *ref_type*.

Dynamic attributes cannot simply be mapped as a sub-object using their own *PKObjectMapping* sub-class since the value format of the dynamic attribute needs to be determined at runtime based on the value of another static attribute. Two suggested ways of solving this problem are:

1. **Add domain object properties for every possible dynamic sub-attribute** For example, say the *ref_type* attribute has a two possible values; "item" and "space". The "item" *ref_type* would indicate that the *ref* attribute value contains static attributes named *item_id* and *title*. For the "space" *ref_type*, the *ref* attribute value would contain the static attributes *space_id* and *name*. Using this approach, the domain object would have four properties, `itemId` `itemTitle`, `spaceId` and `spaceName`.
2. **Store dynamic attribute values as a Binary Large Object (BLOB) property** Because the dynamic attribute value is unknown and varies, the attribute cannot be expressed as an object relationship in the database schema. But by viewing the dynamic attribute value simply as complementary data to the parent object, it becomes clear that it is only relevant in the context of its parent object and will most likely not need to be accessed in isolation. This allows for constructing a custom object at runtime and serializing it into a BLOB column in the database schema. BLOB data is supported by Core Data through the *Transformable* attribute type, capable of serializing any object as long as it conforms to the *NSCoding* [46] protocol. The serializable objects can also be nested as long as they all conform to the *NSCoding* protocol.

PodioKit uses the second option to handle dynamic attributes.

Podio Domain Model Design Core Data model schemas are defined in a *.xcmodel* file in the Xcode project. The Podio domain model consists of a large number of entities and relationships. On the server side the database schema can be modeled as a relatively direct translation of the domain model. However, additional considerations need to be made for a client application based on the intended use of the cached data. One should consider how the data is presented in the user interface and the restrictions that maintaining relationships between entities put on the ability to refresh data and navigate the interface.

Data is requested over a mobile network connection usually tied to a user subscription plan, for example paying a monthly fee to the network provider. These data plans commonly have an upper limit in data usage per billing cycle. This makes data usage a shared resource between applications on the device and should be considered and respected when developing a data caching strategy. As described by Christensen [14], the two key parameters to optimize application data usage is to first limit the number of request and secondly limit the size of each requests.

The Podio API provides few ways for a client to tell the server the scope of the data to return in order to reduce the request size. Therefore it is desirable to use as much of the data returned by a single requests as possible, including returned sub-objects. In many cases the need for the data of these objects differs depending on the context in which they are to be presented. An example is an *item* object. Such an object can appear as its own top-level entity by requesting its data through the `/item/{item_id}` API operation. In this case, the entire item is shown in the interface and therefore all of its data should be persisted, including its *Field* and *Value* sub-objects.

The *item* object is also returned as a sub-object in the API operation `/item/app/{app_id}`. In this case, the API returns a list of item objects. The iPhone application calls this operation to display a listing of the items in a given *app*. All that is needed to populate the list of items is the *title* and *createdBy* attributes of each item. Parsing and storing more data than necessary would increase parsing times and database space usage for items that will likely never be viewed in full. There also needs to be a reference to the top-level app object. This relationship can be defined in the Core Data schema by adding a relationship between the two to allow a simple call to `[app items]` to get all of the items from the database.

Because of these two very different uses of the *item* object, it makes sense to separate the Core Data entity types for these operations, named *Item* and *AppItem*. The naming of the latter indicates that this entity is used for items in the context of an app. This leads to duplication of some item data stored in both entity type collections, but the drawbacks can be considered minor compared to the benefits for handling the data. This approach can been referred to as "Optimized Data Types" [36].

However, the *comment* object is an example where the above does not apply. A comment object is presented in the same way everywhere in the application and it is always showing the same data; the *author*, *text* and *creationDate*. Comments can therefore be stored as the same Core Data entity regardless of the parent entity to which it is attached. This is a significant advantage since we do not have to duplicate the domain objects for its multiple sub-objects like *link*, *file* and *question*. In this case the comment object has two static attributes, *refType* and *refId* to reference the parent, forming a sort of inverse unidirectional association. For this occasion Core Data provides a mechanism called *Fetched Properties* to declare this relationship in the model schema to allow immediate access to the comments from the parent like `[parent comments]`, similar to a regular Core Data relationship.

Model Migrations The Podio model schema is as previously mentioned defined in a Core Data model file. Core Data provides a versioning mechanism for managing changes to this model schema between releases. An older schema should never be changed but rather a new incremental model version should be created within the model file. Core Data ensures schema compatibility between a new application version and the existing persistent store by performing a hash comparison of the schema.

When the application model version is incremented, the existing store needs to be migrated to use the new schema. Core Data provides an easy mechanism called *Lightweight Migration* [47] for simple changes such as adding a new entity or attribute. A *Mapping Model* [47] is needed for more complex schema changes. It requires defining how data stored using the old schema should be transformed to match the new one.

The Podio server side domain model changes rapidly and often in significant ways. However, the initial version of the iPhone application was missing a large part of the functionality of the Podio platform such as organizations, spaces and apps. At that stage model changes could be managed fairly well using lightweight migrations. However, with more features and added model complexity in later versions the model schema changes significantly in every new release. Creating mapping models for each of these versions quickly becomes both cumbersome and error prone.

Instead, an alternative and far simpler approach is to delete the existing data store between application versions. The primary use case for storing local data is to provide a caching layer to improve the user experience and offline capabilities. The local database is entirely populated by remote data, and can therefore be re-populated from the server. By purging the database, i.e. invalidating our local cache, cached data is lost temporarily until it is re-downloaded. However the benefit of not having to manage model schema migrations is considered to outweigh the drawbacks. In addition, since iPhone application upgrades are distributed over-the-air (OTA) through the Apple App Store, it is likely that the user has a functioning network connection at the time of the upgrade and will be able to re-download any deleted data.

Data Mapping Data mapping is the process of creating and populating native domain objects from the API response data. To address the issues in the previous version of the iPhone application described in the chapter 4, a new mechanism for performing data mapping was developed as part of PodioKit. To make it possible to populate a domain object from an *NSDictionary* containing the deserialized object data, one needs to extract the data from each attribute, transform it and assign the corresponding domain object property the resulting value. A number of options were considered to achieve this:

1. **Subclassing** - This means providing a top level class in PodioKit that defines the object properties, such as *Task* or a *Space*. This is a problem since Objective-C does not support multiple inheritance. For example, to use Core Data as the persistence layer, the object is required to be a subclass of *NSManagedObject*. Therefore it is not possible to require the object to inherit from a second PodioKit object that defines the object structure.
2. **Protocols** - Provide an interface for each object type in PodioKit and require the domain objects to implement them. However, protocols do not include behavior, only method declarations.
3. **Key-Value Coding (KVC)** - Key-Value Coding [42] is an informal protocol in Foundation, which leverages the dynamic properties of Objective-C. It allows accessing object properties indirectly by using the string representations of the method names to invoke the target selector. Since Objective-C is a dynamically typed language, any object that responds to the selector identified by the property name string can be invoked. This is potentially dangerous since it triggers a runtime exception if the object does not implement the selector, but can be avoided by adding a runtime check before calling the method.

PodioKit uses the third option and relies on KVC. Building on top of the runtime features of Objective-C allows for creating powerful and generic mapping code based on the *Data Mapper* and *Meta Mapping* patterns [37] using *Reflective Programming* [37]. However, it requires caution since potential bugs are less likely to be caught by the compiler. This further increases the need for proper test coverage.

PodioKit provides a generic way to map response data to native objects by defining a reusable mapping strategy for a remote resource using a subclass of *PKObjectMapping* for each resource object type. Object mappings can also be combined, nested and reused for multiple API operations. An example of an open source library that uses a similar strategy is RestKit [48].

There are a number of key classes and protocols that are a part of the mapping process:

- *PKMappableObject* A protocol required to be implemented by every native class that is used as the domain object for a *PKObjectMapping* subclass.

This protocol is needed by *PKObjectMapper* to determine things such as object identity.

- *PKObjectMapping* This class is subclassed to define object mappings for the response data to the native domain object's value properties.
- *PKAttributeMapping* A class describing how an attribute should be mapped to a specific domain object property.
- *PKObjectMappingProvider* Every client application should provide a custom subclass of this class or use the default mapping provider class included in PodioKit to define the domain model object class for each object mapping to be used within the application.
- *PKObjectMapper* The object mapper is the core of the mapping process and is responsible for evaluating and applying all the mapped properties to a single or collection of domain objects.
- *PKObjectRepository* The object repository is an abstraction used to decouple the creation, lookup and deletion of domain objects. Its implementation differs depending on the underlying persistence layer and its interface is only concerned with object class and identity.
- *PKObjectMapperDelegate* The delegate object to receive updates from the object mapper during the mapping process. For example, in the case of Core Data the delegate is notified once the mapping completes in order to save the changes.
- *PKMappingManager* The mapping manager is responsible for providing each new request operation with an object mapper. It is needed because different persistent stores need different threading behavior. Core Data for example needs a separate *NSManagedObjectContext* for each thread, i.e. each concurrent request operation.

The object mapper performs a recursive mapping of the data based on the top-level object mapping and nested sub-mappings. It evaluates every attribute mapping and updates the resulting property value accordingly. Figure 13 shows a UML diagram of the mapping class relationships.

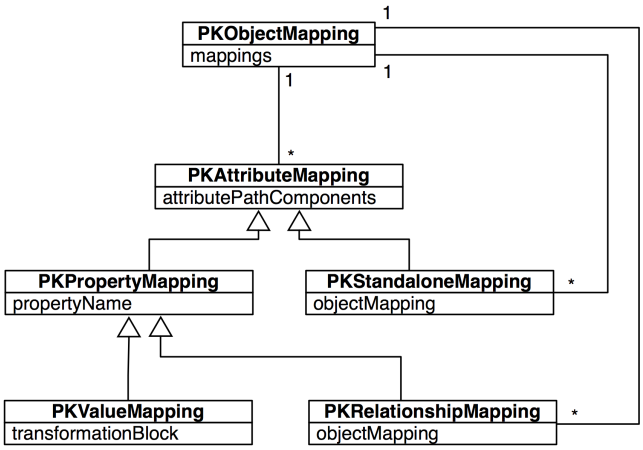

Figure 13: UML diagram of attribute mapping classes

Listing 7 shows a simplified version of a task object returned from the server in JSON format.

Listing 7: Simplified task object in JSON format

```
{
  "task_id": 12345,
  "text": "This is a task",
  "status": "active",
  "created_on": "2012-02-08 12:11:06",
  "created_by": "Sebastian Rehnby",
  "files": [
    {
      "file_id": 11122,
      "name": "Report.docx",
    },
    {
      "file_id": 11123,
      "name": "Presentation.ppt",
    }
  ],
  "comments": [
    {
      "comment_id": 654232,
      "value": "A comment on a task object",
      "created_by": "Sebastian Rehnby",
      "ref_type": "task",
      "ref_id": 12345,
    }
  ]
}
```

There are several types of attribute mappings. The following describes how these can be used to map the JSON object in listing 7 to a native domain

object.

PKValueMapping A value mapping immediately sets the object property value to that of the response data attribute. Use of this mapping requires that the parsed value is of the same type as the domain object property. This is the most common and straightforward way of mapping an attribute to an object property. Listing 8 shows how to define a value mapping.

Listing 8: Simple value mapping

```
[self hasProperty:@"taskId" forAttribute:@"task_id"];
```

Value mappings also allow for an optional block parameter to be used to evaluate or transform the attribute value, as shown in listing 9.

Listing 9: Value mapping with transformation block

```
[self hasProperty:@"createdOn"
    forAttribute:@"created_on"
          block:^id (id attrValue, NSDictionary *objDict, id parent) {
   return [NSDate dateFromDateTimeString:attrValue];
}];
```

PKRelationshipMapping A relationship mapping represents a one-to-one or one-to-many relationship, depending on if the parsed response data is an *NSArray* or an *NSDictionary*. When the object mapper applies this mapping it also configures the parent-child relationship. Listing 10 shows how to define a relationship mapping.

Listing 10: Relationship mapping

```
[self hasRelationship:@"files"
         forAttribute:@"files"
      inverseProperty:@"task"
inverseScopeProperties:[NSArray arrayWithObject:@"taskId"]
     objectMapping:[POFileMapping mapping]];
```

PKStandaloneMapping A standalone mapping can be regarded as a relationship mapping without an explicit relationship. As shown in listing 11, it is similar to a regular object mapping but with an optional context given by the parent object.

Listing 11: Standalone mapping

```
[self hasMappingForAttribute:@"comments"
                objectMapping:[POCommentMapping mapping]
         scopePredicateBlock:^NSPredicate *(id parent) {
  return [NSPredicate predicateWithFormat:@"referenceType == \"task\" &&
      referenceId == %@", [parent taskId]];
}];
```

The use of standalone mappings is relevant when there is an underlying persistent store from which data can be loaded. The standalone sub-objects of

response hierarchies are not be directly accessible through a parent object property, but the data is still persisted and is accessible by querying the persistent store.

When mapping object data retrieved from a remote server, there is a risk of unmappable object structure being included in the response. For example, there are a number of item field types that the client application knows how to handle in the user interface. To provide a robust client application, there needs to be a way for the client to inspect the data prior to mapping and determine if the field type is supported. If the type is not supported, the data for such an object should not be mapped. PKObjectMapping provides the ability to do so by overriding the `shouldPerformMappingWithData:` method shown in listing 12.

Listing 12: Method to override for filtering the object data to be mapped

```
+ (BOOL)shouldPerformMappingWithData:(NSDictionary *)data {
  NSString *type = [data objectForKey:@"status"];
  return [type isEqualToString:@"active"];
}
```

Listing 13 shows a simple example of an object mapping definition for the *Task* domain object. Notice there is no relation defined to the parent object for the standalone mapping.

Listing 13: Complete object mapping definition for the task domain object

```
@interface PKTaskMapping : PKObjectMapping
@end

@implementation PKTaskMapping

+ (BOOL)shouldPerformMappingWithData:(NSDictionary *)data {
  NSString *type = [data objectForKey:@"status"];
  return [type isEqualToString:@"active"];
}

- (void)buildMappings {
  // Properties
  [self hasProperty:@"taskId" forAttribute:@"task_id"];
  [self hasProperty:@"text" forAttribute:@"text"];
  [self hasProperty:@"createdOn"
      forAttribute:@"created_on"
            block:^id (id attrValue, NSDictionary *objDict, id parent) {
       return [NSDate dateFromDateTimeString:attrValue];
  }];

  // Relationship
  [self hasRelationship:@"files"
           forAttribute:@"files"
       inverseProperty:@"task"
       inverseScopeProperties:[NSArray arrayWithObject:@"taskId"]
         objectMapping:[PKFileMapping mapping]];

  // Standalone
```

```
29   [self hasMappingForAttribute:@"comments"
30                  objectMapping:[POCommentMapping mapping]
31              scopePredicateBlock:^NSPredicate *(id parent) {
32       return [NSPredicate predicateWithFormat:@"referenceType == \"task\"
              && referenceId == %@", [parent taskId]];
33   }];
34 }
35
36 @end
```

Once the mapping process completes the mapping manager, acting as the object mapper delegate, is notified. It saves the changes made to the object context, which triggers the *NSManagedObjectContextDidSaveNotification* notification on the background thread. The mapping manager on the main thread observes this notification and the changes are merged to the main thread's object context. Once the changes are merged, all view controllers interested in the changed objects are notified and the user interface is updated, as shown in figure 14.

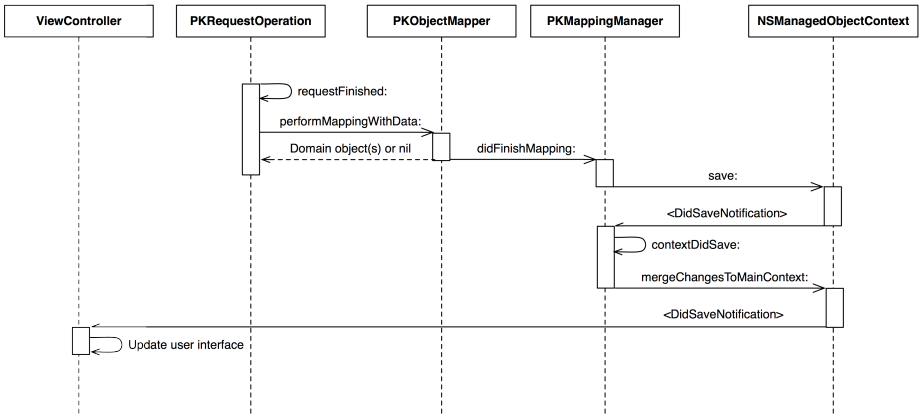

Figure 14: Sequence diagram of a finishing API request

Object ordering and pagination Object ordering needs to be considered when storing objects in Core Data. Core Data entities are treated as un-ordered collections of objects, i.e. there is no absolute ordering between them and nothing similar to the auto incrementing primary key feature found in relational databases such as MySQL. In many cases the object order is implied by the order of the objects in the API response and cannot be derived from the object attributes. For example, a list of *comment* objects could in practice be ordered by their creation date, but the order of a collection of *item* objects for an app is only given by their relative order in the API response. Since the user interface is not populated from the API response but from Core Data, that ordering needs to be maintained when storing the objects.

In PodioKit, providing an optional sequence property name in the object map-

ping helps with maintaining the response object order. At runtime, the object mapper looks up the sequence property on the target object and automatically increments its value when parsing a collection of objects from the API response.

Pagination is another problem when working with parsing potentially large collections of objects, such as the items in an app. Fetching all objects at once from the server would increase processing time on both server and client side as well as data transfer time. It would also result in a larger memory and storage footprint for the client. For such cases, the Podio API provides the ability to request a limited range of the entire collection by providing *offset* and *limit* query parameters. The offset parameter is also used by the object mapper as the sequence starting number when determining the value of the sequence property during the mapping process. This allows us to provide good UI performance and incrementally fetch the content of a collection when needed.

Object Identity Object identity plays an important role in a data-driven iPhone application. In order to provide context to changes in the user interface it is necessary to know why and how the underlying data was changed. As previously mentioned a server side object is serialized into a JSON string, which is then parsed by the client application before being persisted as a native domain object in local Core Data store. However, the data store may already contain the "same" object with obsolete data.

When updating the data store with fresh data there are two options. The first is to discard the old data completely in favor inserting new objects created from the response data. The second option is to find the corresponding existing object and update it with the new data.

The first option is straightforward, but introduces some issues. One problem is that it is impossible to tell apart the objects that have changed and the ones that remained unchanged on the server side. This means they cannot be treated differently, for example by only updating the changed ones. Another problem that arises in the case of Core Data is the ability to give context to data updates in the user interface, for example by using animations. In a typical iOS application the user interface reacts to changes in the model layer based on the type of changes that occur. One such example is a simple listing of objects. An update event triggered by the model layer causes a list entry to update its data in-place. In contrast, a deletion followed by an insertion into the data store would fully remove the entry from the list and then insert a new one with fresh data. This communicates to the user that the object was replaced, which is different from the actual cause; a simple data field update.

Because of the problems stated above, there is a need to be able to determine the *identity* of an object, remote or local. The mapping mechanism of PodioKit requires each mappable target class to conform to an Objective-C protocol called *PKMappableObject*. It defines a class method `identityPropertyNames` that requires the conforming class to implement this method and return an instance of `NSArray` containing the property names used to determine object identity.

The object mapper uses these property names to lookup the corresponding attribute mapping in the object mapping to extract the property value from the response data. It then dynamically constructs an identity predicate and uses the object repository to look up the existing object. If the object does not yet exist, a new instance is created; otherwise the existing one is updated.

`identityPropertyNames` returns an array since in some occasions a single property is not sufficient to determine object identity. If `nil` is returned from this method, all response objects are considered new and the existing objects not in the response collection are deleted.

Performance A consequence of the dynamic mapping strategy used in PodioKit is an added performance overhead. As with any library or framework introducing additional abstraction layers, maintainability takes precedence over highly optimized code. The primary reason for the added performance overhead is the generic style of the mapping process, where very few assumptions are be made about the structure of the response data. Instead, the mapping process relies on inspection and lookup of the mapping definitions. PodioKit also dynamically constructs database queries in the form of *NSPredicate* instances to determine object identity. These queries are more generic that they would be if constructed on a case-by-case basis and therefore they take longer to evaluate.

To assess the performance of iPhone applications, the Xcode development tools include a companion application called *Instruments*. Using Instruments one can gather a vast number of metrics in real-time while the application is running in the simulator or on a device. Some of these metrics include execution time for individual methods, Core Data performance and memory allocations.

Instruments is of help when identifying the main performance bottlenecks in the Podio iPhone application as a whole, and PodioKit in particular. The measurements show that during the mapping process, the most time is spent accessing the SQLite database through Core Data.

Optimizing Core Data After observing execution during the mapping process, it became clear that a considerable amount of time is spent evaluating the rather complex database queries constructed. PodioKit expresses object identity using *NSPredicate* instances created dynamically by the object mapper. Querying the database is inherently slower than accessing in-memory data, and there are a number of optimizations that have been used to both avoid and decrease the execution time of database queries. These include:

- **Use of indices** Core Data model schemas offer the ability to define indices on entity attributes. Since the underlying storage for Core Data is SQLite the benefit is the same as with a regular database index, namely faster lookup. The approach used in PodioKit is to create indices for any attribute that is used to determine object identity.
- **Conditional predicate creation** Depending on the domain object iden-

tity properties, one can choose to create different format identity predicates. For example, some object types might have a single identity property, while others have more.

- **Avoid database access** Making a round trip to the database is quite fast, but accessing in-memory objects is faster. Therefore, one could identify cases where going to the database to find an existing object is not needed. One such case is when updating child relationships of top-level object types. For example, if a task object is to be updated and is found in memory, its *files* relationship is accessible through that parent property and thus, there is no need to access the database to map each of the underlying file objects. Instead, one can look for the object in the relationship collection first and only access the database if the file object was not found.

Optimizing the Mapping Process

There are a number of ways in which one can improve the runtime performance of the mapping process. Although method dispatching and memory allocation is fast in Objective-C, in great numbers they both contribute to a significant performance overhead.

- **Minimize call depth** Objective-C is a dynamic language and therefore determining the actual function pointer to execute for a given method call required runtime inspection of the class hierarchy. Reducing the depth and number of method calls is one way to achieve a small performance gain.
- **Use the right data structure** Objective-C native collection objects have, as is the case with all data structures, different strengths and weaknesses [49]. These characteristics affect lookup time, iteration performance and inclusion testing etc.
- **Use autorelease pools** Autorelease is a feature of the Objective-C runtime where decrementing an object's retain count can be delayed until the end of the run loop. This makes memory management easier but has some side affects. The mapping process creates a large amount of autoreleased objects that are not deallocated until the operation finishes. This often impacts memory usage and general performance. Cocoa provides a way to define scopes of autoreleased objects by using autorelease pools. When a pool is deallocated, the `release` method is called for all autoreleased objects created within that autorelease pool. This decreased the time autoreleased objects remain in memory and, thus, improves performance.

Performance Measurements

Table 1 shows measurements performed on physical devices attached to a Mac computer. The timing data shows the time it takes to instantiate and map

the activity stream response data to native objects, including the full hierarchy of relationships and properties. Two scenarios are used. *Full stream* refers to mapping of a page, i.e. the 20 most recent stream events, while clearing all old stream data. *Stream offset* refers to the paging behavior used for fetching and additional 20 stream events while keeping any previously cached page of the stream.

	Full stream (s)		Stream with offset (s)	
Model	Old method	PodioKit	Old method	PodioKit
iPhone 4S	0.765	1.542	0.704	1.332
iPhone 4	1.232	4.46	1.21	2.453
iPhone 3GS	1.66	5.331	1.735	4.401

Table 1: Time comparison for mapping the activity stream response data using the old mapping strategy and PodioKit respectively

Table 1 shows that there is a significant added performance overhead when using PodioKit. For the iPhone 4S, PodioKit roughly doubles the time to map and store the response data compared to the old method. For the iPhone 4 and iPhone 3GS the mapping time is even slightly worse.

5.3 Assessment of Results

The approach used in PodioKit to dynamically map remote data attributes to native objects is not as customizable as performing data mapping on a case-by-case basis as was the case with the previous mapping strategy. On the other hand, the advantages are high cohesion and reduced duplication by the introduction of a number of key concepts for defining object mappings, while keeping the actual definitions simple and hiding the complexity of the mapping process on the library level. This approach improves overall maintainability, reusability and testability because of its modularity and the separation of concerns like network request management, object mapping and persistence logic. However, because of the sometimes hard to map structure of the API response, there is still a need to be able to use the response data itself as input to the mapping process. Fortunately, Objective-C blocks make this possible as they can be used to attach behavior to a mapping definition to be evaluated during mapping.

From a performance perspective, PodioKit does not perform as well as the previous implementation during object mapping, as shown by Table 1. The primary reason is that the previous implementation handled object mapping on a case-by-case basis and could therefore provide a very optimized implementation for each API operation and its response. Additional abstractions layers such as PodioKit affect performance negatively in exchange for modularity and high cohesion between its subcomponents. High cohesion in turn has many desirable side effects such as usability, testability, maintainability and reusability among others [50, 51].

The tradeoff to be made is that of determining what the acceptable performance impact is in exchange for the aforementioned benefits. In the case of the Podio iPhone application, the threshold for acceptable performance is not a fixed number but is instead defined by the user-perceived performance. The numbers in table 1 has shown to be acceptable both from a user's perspective and for the following additional reasons:

- **Mapping Complexity** Table 1 shows the measurements of mapping the 20 most recent objects in the Podio activity stream. This is the most complex mapping operation in the Podio iPhone application since the server response contains the most relationships and sub-entities of any other API operation used by the application. As PodioKit is used throughout the application, this is considered the worst-case scenario in terms of data mapping. In many other places of the application only a single object or far less complex data is returned from the server, decreasing mapping times significantly.
- **Advancements in Mobile Computing** The advancement in computing power available in mobile devices as demonstrated by the chipsets used in both the iPhone 4 and the iPhone 4S shows that performance will become less of a problem with time. Also, because of the limited number of iOS device models in the market, it is fairly easy to get some insight in how the application performs on each of them.
- **Further Optimization** PodioKit can most likely be optimized further. The initial implementation focused on conceptual simplicity and proof of concept. More can be done in terms of identifying and optimizing the data mapping process.

An analogy can be made with the performance of user interface rendering. The first couple of iOS devices, namely the original iPhone and the iPhone 3G had rather limited memory and CPU power. Thus, a lot of techniques were developed by the community to optimize the rendering of complex view hierarchies to improve the user experience. These optimizations came at the cost of maintainability and the inability to utilize the system provided mechanisms for interface layout such as automatic resizing of views. With the iPhone 4 and 4S, the benefit of these optimizations are now less obvious and one can now write interface code the way that was intended by the frameworks and still get good performance.

6 Conclusion

In this thesis project, we have investigated the issues experienced with the existing Podio iPhone application. We used these findings to identify common problem and challenges in mobile application development in terms of performance, user experience and maintainability. We concluded that the hardest part

with developing a mobile application is related to request management and local data caching due to the environmental variations mobile devices are subjected to.

We also looked at how an existing product or service can be rethought and restructured to provide the best mobile experience possible by acknowledging the restrictions but also the advantages, such as context awareness and inherent mobility, of mobile devices. We suggested a method for determining the feature set of mobile applications based on identifying lightweight and heavyweight actions, and the accessibility of these features through the use of gesture-based actions.

Finally, our research was used in practice to implement a robust and flexible request and data management library, PodioKit, for the Podio iPhone application. PodioKit is a standalone static library linked to the main Podio application project at compilation time. It takes a dynamic approach to mapping remote data returned as JSON strings from the API to native Objective-C object types, capable of being persisted using a local Core Data store. Object mappings can be customized and nested to create a mapping hierarchy to fit any client application's data model, regardless of the type or lack of persistence strategy. PodioKit also encapsulated much of the Podio API operations through a set of native API classes. This removes the need to redefine these calls in a client application and makes it suitable as an addition to the Podio family of open sourced client libraries.

The research was also used to bring new features such as organizations, spaces and apps to the iPhone application. A large part of realizing these features included reconsidering the user experience and interface design for the iPhone.

6.1 Discussion

When working with the Podio API it becomes clear that it has been developed with the Podio front-end Ruby on Rails application in mind. This is noticeable through the operation interfaces available and the structure of the response data. For example, many translation strings used in the interface are composed by the front-end and are not provided by the API. Instead the API returns simple enumeration values used by the front-end to compose these strings. This works well when the front-end and API can be deployed at the same time and are always consistent, but for other client applications who are independent from the development of the API it can become a cumbersome task to keep the client consistent with the API. Many would perhaps argue that the correct way to handle this is to have a properly versioned API to guarantee compatibility. The Podio API is however not versioned and for good reasons. To support the rapid development speed at Podio and the frequency with which new features are deployed, maintaining compatibility with older API versions would become a rather big task in itself. Perhaps this is something that can be improved in the future, but at the current state of Podio as a product this is not feasible.

Because mobile applications has entered the consumer space fairly recently, at least on a broad scale, it has been quite difficult to find relevant and formal research in the field, a fact acknowledged by Wasserman [16]. Perhaps more have been written internally in the enterprise space, but the research available does not seem to be as extensive as in many other software engineering fields. Additionally, there seems to be a lot of varying opinions in the developer community and no well-defined best practices or de facto standards on how to deal with some of the issues presented in this thesis.

6.2 Future Work

In the future, more time could be spent on optimizing PodioKit and identifying its bottlenecks. More work could also be done in collaboration with the Podio developers to make the Podio API more friendly and optimized for third party application developers and hopefully more Cocoa-based applications can be build using PodioKit.

The Podio Android application is still lacking many of the key features available in the iPhone application. Given the growth of the Android platform, collaboration with the Android developers is necessary to improve the application to support the same features as on the iPhone. Such a task could provide additional insight around issues related to mobile application development.

Podio already offers a number of API client libraries, available as open source software. In the same way PodioKit could be open sourced for others to use in Cocoa related projects. Any opportunity for Podio customers to use the API and build their own custom solutions would ultimately benefit the platform and the company as a whole.

References

[1] International Data Corporation (IDC), Smartphones Outstrip Feature Phones for First Time in Western Europe as Android Sees Strong Growth in 2Q11, Says IDC, http://www.idc.com/getdoc.jsp?containerId=prUK23024911 (September 2011).

[2] Podio ApS, Podio, https://company.podio.com (November 2011).

[3] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Ph.D. thesis, University of California, Irvine (2000).

[4] Podio ApS, Getting started with Podio API, https://developers.podio.com (November 2011).

[5] Introducing JSON, http://www.json.org (October 2011).

[6] OAuth 2.0, http://oauth.net/2 (November 2011).

[7] Ruby on Rails, http://rubyonrails.org (November 2011).

[8] podio-rb, https://github.com/podio/podio-rb (November 2011).

[9] Android, http://www.android.com (October 2011).

[10] Apple Inc., App Store Review Guidelines, http://developer.apple.com/appstore/guidelines.html (December 2011).

[11] Open Handset Alliance, http://www.openhandsetalliance.com (November 2011).

[12] Apple Inc., iOS Technology Overview, http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/IPhoneOSOverview/IPhoneOSOverview.html (December 2011).

[13] J. J. P. C. Rodrigues, M. Oliveira, B. Vaidya, New Trends on Ubiquitous Mobile Multimedia Application, EURASIP Journal on Wireless Communications and Networking (2010) 1–11.

[14] J. H. Christensen, Using RESTful web-services and cloud computing to create next generation mobile applications, Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications OOPSLA 09 (2009) 627–633.

[15] J. Dehlinger, J. Dixon, Mobile Application Software Engineering: Challenges and Research Directions, Proceedings of the Annual Workshop on Software Engineering and Mobile Application Development 2011 (2011) 27–30.

[16] A. I. Wasserman, Software Engineering Issues for Mobile Application Development, Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10 (2010) 397–400.

[17] Apple Inc., iOS Human Interface Guidelines - Human Interface Principles, http://developer.apple.com/library/IOS/#documentation/UserExperience/Conceptual/MobileHIG/Principles/Principles.html (October 2011).

[18] R. Benbunan-Fich, A. Benbunan, Understanding user behavior with new mobile applications, The Journal of Strategic Information Systems (2007) 393–412.

[19] P. I. Lei, A. K. Wong, The Multiple-Touch User Interface Revolution, IT Professional, Volume 11, Issue 1 (2009) 42–49.

[20] L. Brichter, User Interface Mechanics - US Patent 2010/0199180 A1, http://www.patentlens.net/patentlens/patent/US_20100199180 (August 2010).

[21] The WebKit Open Source Project, http://www.webkit.org (November 2011).

[22] Appcelerator, http://www.appcelerator.com (November 2011).

[23] Nitobi, PhoneGap, http://phonegap.com (November 2011).

[24] Apple Inc., Core Animation Programming Guide, http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html (November 2011).

[25] TIOBE Software BV., TIOBE Programming Community Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (November 2011).

[26] T. Mikkonen, A. Taivalsaari, Apps vs. Open Web: The Battle of the Decade, Proceedings of the Annual Workshop on Software Engineering and Mobile Application Development 2011 (2011) 22–26.

[27] H. Gopal, Resource-Aware Mobile Device Application, Dr. Dobb's Journal (2006) 10–16.

[28] Apple Inc., Concurrency Programming Guide - Migrating Away from Thread, http://developer.apple.com/library/ios/#documentation/General/Conceptual/ConcurrencyProgrammingGuide/ThreadMigration/ThreadMigration.html (October 2011).

[29] Ben Copsey, All-Seeing Interactive, ASIHTTPRequest, http://allseeing-i.com/ASIHTTPRequest (October 2011).

[30] Apple Inc., Core Data Programming Guide - Introduction to Core Data Programming Guide, http://developer.apple.com/library/mac/#documentation/cocoa/conceptual/coredata/cdprogrammingguide.html (October 2011).

[31] JSONKit, https://github.com/johnezang/JSONKit (December 2011).

[32] Cocoanetics Blog, JSON versus PLIST, the Ultimate Showdown, http://www.cocoanetics.com/2011/03/json-versus-plist-the-ultimate-showdown (March 2011).

[33] Apple Inc., Notification Programming Topics, http://developer.apple.com/library/IOS/#documentation/Cocoa/Conceptual/Notifications/Introduction/introNotifications.html (October 2011).

[34] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[35] Apple Inc., Cocoa Fundamentals Guide - Cocoa Design Patterns, http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html (October 2011).

[36] Apple Inc., Building a Server-driven User Experience (Video), http://developer.apple.com/videos/wwdc/2010/?id=117 (June 2010).

[37] M. Fowlers, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002.

[38] Apple Inc., Core Data Programming Guide - Persistent Store Features, http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CoreData/Articles/cdPersistentStores.html (October 2011).

[39] Apple Inc., Core Data Programming Guide - Concurrency with Core Data, http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/coredata/Articles/cdConcurrency.html (October 2011).

[40] Apple Inc., Core Data Programming Guide - Core Data Performance, http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/coredata/Articles/cdPerformance.html (October 2011).

[41] Apple Inc., Network Apps for iPhone OS - Part 1 and 2 (Video), http://developer.apple.com/videos/wwdc/2010/?id=207 and http://developer.apple.com/videos/wwdc/2010/?id=208 (June 2010).

[42] Apple Inc., Key-Value Coding Programming Guide, http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/KeyValueCoding/Articles/KeyValueCoding.html (October 2011).

[43] Apple Inc., Objective-C Runtime Programming Guide, http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html (October 2011).

[44] Apple Inc., Blocks Programming Topics, http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html (October 2011).

[45] P. Dix, Service-Oriented Design with Ruby and Rails, Addison-Wesley, 2010.

[46] Apple Inc., Archives and Serializations Programming Guide, http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Archiving/Archiving.html (October 2011).

[47] Apple Inc., Core Data Model Versioning and Data Migration Programming Guide, http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreDataVersioning/Introduction/Introduction.html (October 2011).

[48] RestKit, http://restkit.org (January 2012).

[49] M. Gallagher, NSArray or NSSet, NSDictionary or NSMapTable, http://cocoawithlove.com/2008/08/nsarray-or-nsset-nsdictionary-or.html (December 2011).

[50] M. B. Linda Badri, F. Toure, Exploring Empirically the Relationship between Lack of Cohesion and Testability in Object-Oriented Systems, Communications in Computer and Information Science, 2010, Volume 117 (2010) 78–92.

[51] S. D. Bart Du Bois, A. Taivalsaari, Refactoring - Improving Coupling and Cohesion of Existing Code, Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04) (2004) 144–151.