Android client application development for existing CRM solution

Martin Kusyn

Bachelor's Thesis
May 2013

Degree Programme in Software Engineering
School of Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s) KUSYN, Martin	Type of publication Bachelor's Thesis	Date 2013-05-05
	Pages 74	Language English
		Permission for web publication (X)
Title Android client application development for existing CRM solution		
Degree Programme Software Engineering		
Tutor(s) PELTOMÄKI, Juha		
Assigned by RAYNET s.r.o.		

Abstract

This bachelor's thesis focuses on the development of client application for Android operating system for mobile devices. The final application serves as a client for RAYNET Cloud CRM, an existing CRM solution.

The first part of the thesis contains brief introduction of RAYNET s.r.o. , the creator of RAYNET Cloud CRM, introduction of the CRM system itself and brief description of its functionality and implementation. Requirements of the mobile client application are discussed and its user interface, behavior and control are suggested.

Resolving of the application requirements brought several issues that have to been solved or bypassed. These issues include several problems related to incompatibility between different versions of Android operating system and also introduces several advanced techniques of Android programming and describes used libraries.

The major part of the thesis is dedicated to implementation of the application itself. Particular functionality of most important UX components is described as well as authentication model, data obtaining workflow and usage of accounts and services.

The thesis demonstrates the development of Android client application that is compatible with several generations of Android system. The last chapter summarizes the results of the application development and points out its most problematic parts and also discuss the future development of application.

Keywords
Android, RAYNET Cloud CRM, Java, AJAX, REST, Mobile application

Miscellaneous

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 Motivation

The currently existing mobile client for RAYNET Cloud CRM is limited in many ways and due to the fact it is built as mobile web application, it does not provide user experience on the advanced level that RAYNET Cloud CRM's users expect.

The goal was to create a native Android application that will allow users to work with most important contact and business information, when the computer is not accessible. It is important that application should be easy to use and allow users to synchronize contacts and planned activities with the mobile device, so they can be available even without an internet connection - typically this would be a case when a user is on the business trip to his/her customer.

It is also significant to provide good users experience and allow user to work with the application nearly the same way they are used to work with the system itself.

1.2 Company introduction

RAYNET s.r.o. is a company founded in 2004, which is mostly concerned with CRM system development. Having the experience with custom solutions development, the company decided to create a new main product - RAYNET Cloud CRM, which is user friendly solution accessible to a wide range of customers and flexible for further development or custom modifications.

The company mostly operates in Central Europe and still tries to expand to other markets. In 2009 RAYNET was awarded with DELOITTE - CE Technology Fast - 'Rising Stars' category award for the fourth place in central Europe region. In 2011, RAYNET was awarded once again, this time with DELOITTE - CE Technology Fast 50 award as the thirteenth most progressive technology company in central Europe region.

The company also owns ISO 9001 certificate as an evidence of keeping high standards in quality of service provided to customers.

1.3 RAYNET Cloud CRM introduction

RAYNET Cloud CRM is the company's main product. The main idea of RAYNET Cloud CRM is to create a state of art CRM solution, which is easy to use for both - its users and its developers.

The most important thing was to create a modern application, which will use modern, yet stable technologies and will be easy to modify to handle the request of customers looking for a more advanced custom solution.

To fulfill these needs, the application structure had to be well designed and appropriate technologies had to be used on the project. The most essential approaches in RAYNET Cloud CRM project were AJAX and RIA.

1.4 RIA - Rich Internet Application

The idea about RIA is to move client-server applications from desktop to web browser and allow users to use them without installing the client locally. Using the RIA two different approaches can be faced - applications that uses the browser plug-ins and applications based on JavaScript and AJAX technology.

Plug-in based applications depend on preinstalled components integrated into the browser, e.g. Adobe Flash or Microsoft Silverlight. The disadvantage of this approach is obvious - the plug-in must be supported and installed in the browser. If the plug-in cannot be installed, it will not be possible to run the application.

The other approach is to develop the application based purely on JavaScript and AJAX technology. In these days, there is a large amount of already existing and stable JavaScript frameworks, which made the development of dynamic, single window application much easier. These frameworks are generally well optimized to work in major web browsers and take the responsibility for handling differences in the browser, and often brings UX elements to supply standard desktop application forms and widgets. Another advantage of the JavaScript and AJAX approach is the possibility of lazy-loading the code currently needed. The client application can be started very quickly and it provides the basic functionality without downloading the whole application code. Another code is downloaded in the moment, when required.

> *Significant drawback of applications run in the browser is the complete isolation of such applications. The original intent of client JavaScript implementation in browsers was to provide better user interface and effects, it was not designed to emulate the complete desktop application. For security reasons the JavaScript interpreter runs in a secure sandbox with limited access. The client application cannot interact with other parts of the operating system it runs in. The example consequence of such isolation is the inability to directly access files on the client's computer. (Stříž 2011, 8)*

1.5 AJAX - Asynchronous JavaScript and XML

The basic idea of a RIA application is to provide desktop-like client-server application inside a web-browser environment, without reloading the page of the client application. To make this possible, it is necessary to perform asynchronous requests and have ability to update elements of the client application without rendering it again.

> *The need of reloading and re-rendering whole pages lead to emergence of AJAX, which stands for Asynchronous JavaScript and XML. AJAX is a technology that allows client side (web page) to be more interactive and responsive. (Garret 2005)*

Asynchronous requests require web browser support, which is included in all major modern browsers. *"Programmatically, everything is done through the special XMLHttpRequest (XHR) object, that allows to open connection and asynchronously, by registering a callback function, wait for an answer." (Stříž 2011, 8)*

Modern JavaScript frameworks usually provides an interface to execute AJAX by encapsulating XHR object and covering possible problems and different XHR processing in browsers, and also provide the methods how to work with the application's DOM.

1.6 Server-side implementation

1.6.1 Spring framework

Server-side of RAYNET Cloud CRM is based on Spring Framework.

> *Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. The main feature of Spring is the ability to compose the application out of intercommunicating POJOs (Plain Old Java Objects). Key principles used in the Spring Framework that allow easy composition of applications are Inversion of Control (IoC) / Dependency Injection (DI) and Aspect Oriented Programming (AOP).*
> *(Stříž 2011,12)*

1.6.2 Inversion of Control / Dependency Injection

Inversion of Control is a paradigm, that shifts creation of child object from caller object to the IoC container, usually provided by framework. Since all the dependencies are based strictly on interfaces, the components code does not know anything about actual implementation of the object behind the interface, which makes the code more flexible.

Dependency injection is a concrete implementation of Inversion of Control pattern that allows removing hard-coded dependencies and makes possible to change them during the run-time. Spring framework provides the IoC container that handles dependency injection.

> *In Spring Framework, the components are registered into the container in a declarative manner, either from an XML file or by annotating classes with Java annotations. The absence of central implementation that wires everything together in procedural manner allows for reusable and clear implementations. (Stříž 2011, 13)*

1.6.3 Aspect Oriented Programming

> *Aspect Oriented Programming (AOP) is a paradigm that allows the developer to implement aspects, the cross-cutting concerns such as logging, auditing, security or transaction management into the system without modification to the business logic.*
>
> *Spring provides the AOP framework that transparently creates dynamic proxy classes. These proxies invoke aspects around components with business logic.*

Aspects are, same as components, configured by an XML document or annotations. Many predefined aspects exist inside the Spring Framework for various tasks. New aspects can also be introduced should the need arise. (Stříž 2011, 13)

1.6.4 System architecture

As an enterprise application, RAYNET Cloud CRM is divided into the system of separated tiers and layers, where the tiers represents the physical level of division and the layers represent logical levels of divisions. Every tier can be run on different machine or machines.

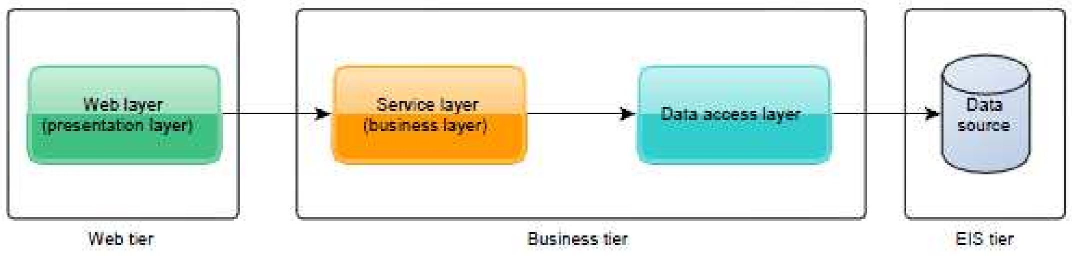


FIGURE 1 - System's tiers and layers (Stříž 2011, 30)

Web layer (Presentation layer) represents data to the client and retrieves user's input. It can provide data to the user in the form of HTML page or pure JSON data.

Service layer (Business layer) contains the business logic of the application. It defines the entities used in the system and functions that cover the work with them.

Data access layer creates a level of abstraction in order to allow having various data sources with different access methods. Data access layer provides Service layer the interface with no information about methods used to obtain data. This allows developer to change the data source without no need for Service layer modification.

Web layer

For this work, it is important to understand the web layer, its controllers, and its connection to the service layer. Web layer provides adapter between the client application that runs in the browser and the applications business logic defined by the service layer.

RAYNET Cloud CRM's web layer is based on Model-View-Controller architectural pattern and is built on Java Servlets and Spring MVC technologies.

Model is represented by service layer that contains the main functionality to work with entity objects.

Views contain the actual presentation of the logic. RAYNET Cloud CRM architecture uses two common types of Views - the first is delegated to the JSP page and is used to provide HTML pages such as login page, index page or error pages, and the second is JSON view. JSON view is used to serialize the model into the JSON, which is returned as a result to the AJAX request.

Controllers serve as the entry points to the application. Every user actions is handled by the browser application as an AJAX request that is processed and dispatched to the specific controller method according to the request's URL address.

> *Generally, the controller is responsible to get or put data to the model (represented by the service layer in this case) and delegate the presentation to the appropriate view. (Stříž 2011,33)*

Record Controller, in the terminology of RAYNET Cloud CRM, is the controller that handles the CRUD operations and passes the request data to the appropriate record service.

1.7 Front-end implementation details

RAYNET Cloud CRM's FE client is built on the JavaScript and AJAX approach. The whole application is built as single-page application, with lazy-loading of the currently needed code.

To make the client application as stable and reliable as possible, RAYNET created its own JavaScript framework based on the existing framework ExtJS 3.0. The main differences RAYNET's client framework differs from the original ExtJS framework in many ways. It brings its own UX components and widgets, but it also has more complex inner structure.

The framework has its own methods to perform lazy-loading of code and allows combining components based on different versions of ExtJS framework using sandboxing inside the client application itself. This feature was used in analytics module, which uses the charts based on the ExtJS 4.0 framework. The charts code and required ExtJS 4.0 classes are encapsulated in its own namespace and provides the interface which allows communicating with the rest of the application.

Client application workflow

When users try to access their instance of RAYNET Cloud CRM, the first thing to run is a check if the user is still logged in or not. In the case that a user is logged in (e.g. after page refresh), the index page is prepared and returned to him/her, otherwise the login page is displayed. The generation of the pages is provided by JSP Views on Web layer's controllers.

Since RAYNET Cloud CRM web client is built as a single page RESTful application, the index page contains only links to included styles, core script of the application's framework and user information and settings.

1.8 Available APIs

RAYNET Cloud CRM is accessible via two APIs. Standard web API, that serves as an entry point for web application and REST API with limited functionality for third-party applications.

Standard web API uses the Spring Security check to authenticate the client application against the server, when the communication is initialized. Later communication is authenticated using the session id.

REST API uses BasicAuth for authentication and the authentication string have to be sent with each request.

2 MOBILE CLIENT

2.1 Motivation

Providing the mobile client for the application gives more comfort to RAYNET Cloud CRM users during the business trips. The idea is to provide users with the lightweight application containing the functionality which is easily accessible even on the small touch-screen devices.

Creating complementary mobile client, several different approaches are possible. The first one is to create a scalable web application, which would work on both - traditional desktops and small screen devices. This approach however is very limiting for both types of devices.

The second approach is to create a separated mobile web application, which would run in mobile device's browser. This approach brings the opportunity to create an application purely designed for small touch-screen devices, however the functionality is still limited by abilities of integrated web-browser.

The third approach is to create an application using one of the available mobile frameworks like PhoneGap, Appcelerator, Rhodes or Sencha Touch. These frameworks allow compiling the application for several different mobile platforms, bringing their own HTML5 based UX components and often allow using some basic phone functionality.

The advantage of this approach is obvious. It is giving programmers the option to write the code once and run it on different kind of mobile operating systems. The problem is that the targeted platforms are different, which leads to compromises in both - functionality and design. Application based on mobile framework always has a different design than one which is standard on target platforms. This fact is confusing for the users and the applications are usually not very successful.

The fourth approach is to create a native application for a targeted platform. This approach allows creating the application according to common platform guidelines and taking advantage of all functionality that the platform provides. Developing the application for several platforms, this approach is the most complicated and time consuming, but in the end it brings the best results.

2.2 Selected functionality

A mobile client is not designed to provide all the functionality of a web client. It should provide only the functionality that makes sense to use on a mobile device, when the user needs the quick access in a business trip. It is assumed that users on the way to customer will mostly need to check the contact details, addresses or planned agenda. These are the primary tasks and the mobile client should make the work with records as easy as possible - for example allows user to directly dial the phone number of contact or open the navigation with the address of a company.

The mobile client is dedicated to slightly different use, so it is not necessary to include work with all entities of full web client.

2.2.1 Lead

Lead is the temporary record, which is supposed to be later converted to Person, Company or Business case. It is supposed to be created, when a user is contacted by a person or company interested in business cooperation, but it is not clear how to categorize it yet.

In the mobile client it should be possible to access the list of Lead records, display the detail of record, save the contact information to the device's contact list, or create a new record.

2.2.2 Company

Since the RAYNET Cloud CRM is B2B oriented product, the Company entity is the basic contact entity. The company allows users to store the basic information about the firms, including the contacts and a list of addresses.

In case of Company entity, mobile client should also allow displaying list of records, detail, save contact information or create new record. It should also allow users to work with a list of company addresses or a list of employees.

2.2.3 Person

Person entity figures in the RAYNET Cloud CRM as an employee of the company, or possibly multiple companies. From a user point of view, Person entity allows storing contact information, relationships between the person and companies or some additional information.

Beside the basic functionality, the mobile client should allow user to synchronize Person records as contacts in device.

2.2.4 Business case

Business case is the entity covering business activity between user and other business subjects. It gives the user an overview of offers, orders, offered products or status of negotiations with business subject.

Working with business cases is more complex and it will not be very comfortable to perform these operations on a small screen of mobile devices. It is assumed, that users would choose the full web client to work with business cases, so the mobile client should provide only basic operations with business case, without allowing user to create or modify records.

2.2.5 Activity

Activity is an abstract entity, which is realized by Email, Letter, Meeting, Phone Call, Task or Event. An activity entity is meant to be planned and scheduled using the application's calendar. Activity is linked to Person, Company, Lead or Business Case, or it could be planned as personal. Activity entity also stores the list of participants.

Different implementations of Activity entity provide different options to users. For example Email entity allows users to store an email message, which would be sent to participants of the activity, or which was received from a business subject. To simplify this operation, there also exists the "Email assistant", which allows user to import email messages simply by forwarding them to the e-mail address of the assistant.

Working with activities is a crucial feature of a mobile client. Besides the basic functionality, mobile client should allow users to synchronize activities with build-in calendar of device, export the activities to calendars of user's Gmail accounts, send pre-prepared messages by email or perform calls to the clients.

2.3 UX design

RAYNET CRM Touch application provides users with two basic views for each entity - *ListView* and *DetailView*. *ListView* displays a list of available entity records and allows user to apply available filters. *DetailView* displays detailed values of record.

2.3.1 General application View layout

All the views in the application (excluding *LoginViews*) use the general Layout. The general Layout defines two components - the *Sliding menu drawer* and content area. Both components are built as *Fragments*. (See fig. 2)

Content area contains the *View* specific content - this should be a list of available entity records, entity record's detail or other content such as user's dashboard.

The *Sliding menu drawer* contains the application's main menu that allows user to navigate inside the application.

Opening the *Sliding menu drawer* can be done in two different ways - by clicking on the menu button in the view's *Main Action Bar*, or using the *Swipe gesture* on the left edge of the screen. To close the *Sliding menu drawer*, users can use the opposite direction *Swipe gesture* or click the visible part of content area on the left.

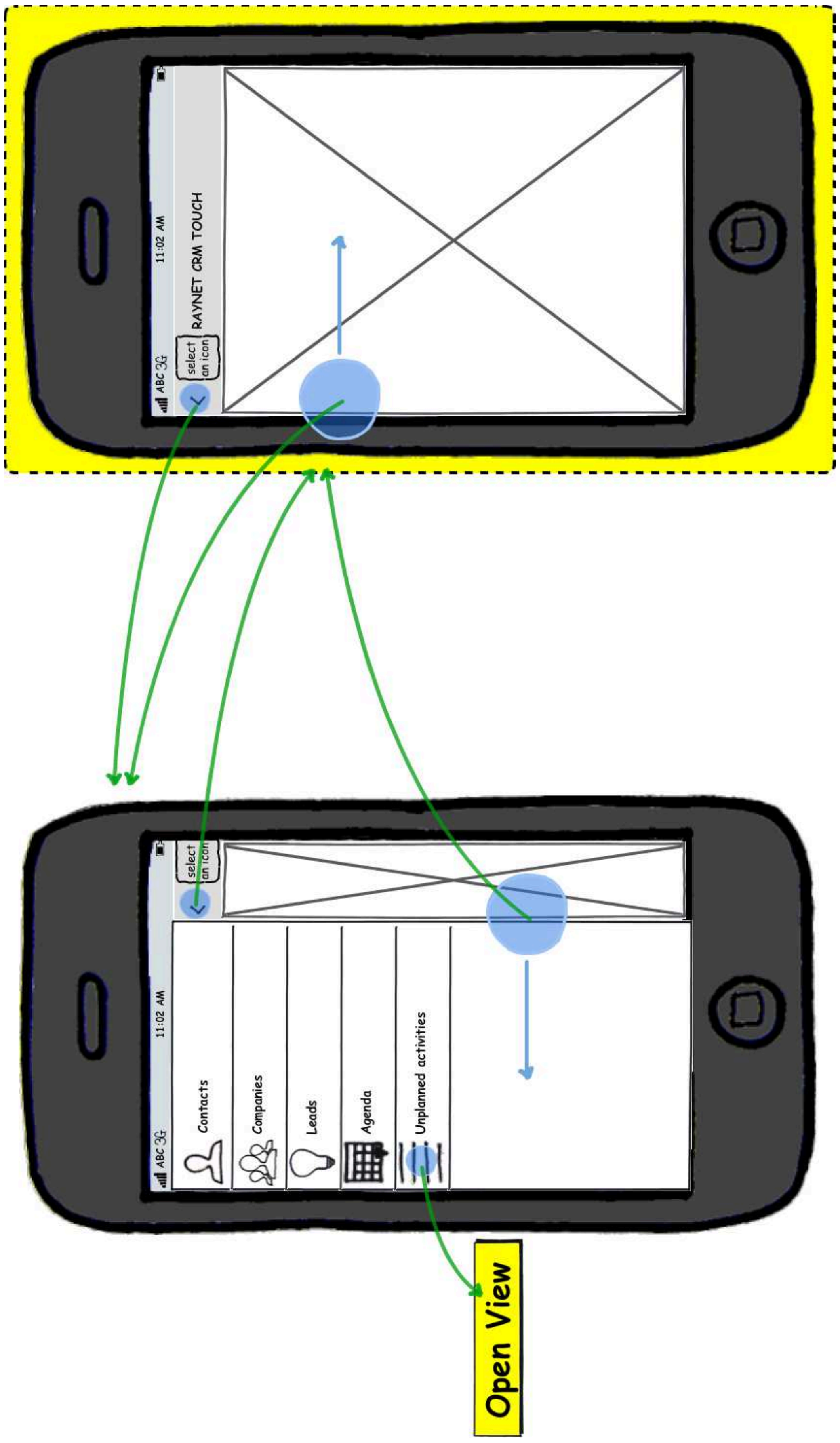


FIGURE 2 - General application View layout

2.3.2 General ListView layout

ListView layout extends general View layout and defines the basic behavior of entity *ListViews*. The content area of the *ListView* contains two main components - *ListFragment* and *FilterFragment*. (See fig. 3)

ListFragment contains the list of an entity's records. The list is filtered by filters specified by *FilterFragment*.

When the *ListItem* is clicked, the according *DetailView* is opened and a detail of the selected record is displayed.

If the user performs a Long press gesture on the *ListItem*, the record's *QuickView* is displayed and Action Mode menu replaces the *Main Action Bar*. *QuickView* contains more detailed records information, such as contact information. *Action Mode* menu contains the list of most used actions that can be performed with a record, such as dialing-up the phone number from records contacts.

FilterFragment contains entity specific filters that can be set to filter out the list's records. When users set the filters and confirm them by pressing the *"Set filters" button*, *ListFragment* is displayed and a new request is performed to acquire the filtered list of records. The fact that filtering is active is indicated in *FilterFragment's* tab header.

Switching between the *ListFragment* and *FilterFragment* can be done by clicking on the according tab, or by performing the *Swipe gesture* on the body of content area.

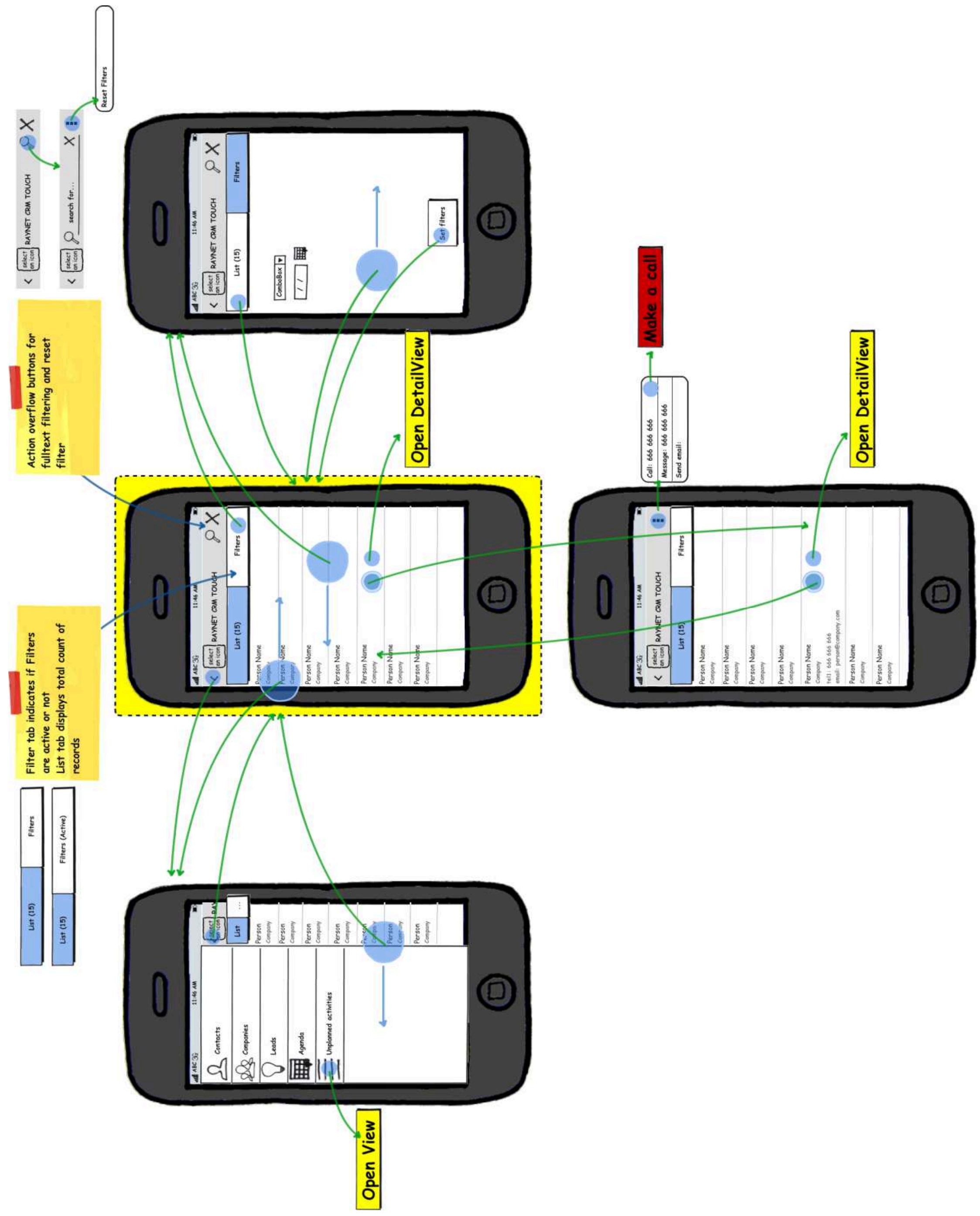


FIGURE 3 - ListView layout

2.3.3 General DetailView layout

The *DetailView* layout is based on the general *View* layout and contains two components - *DetailFragment* and *HistoryListFragment*. (See fig. 4)

HistoryListFragment contains the list of records context history - the information about linked activities, relationships or business cases.

DetailFragment is a container component that holds several info *subfragments* displaying different record's information. *Subfragments* are reusable units that can be used to display the information on different places in application.

Subfragments can also contain the *fields* - active units that react on click action. On the click action, the field opens the default intent activity based on its type
- e.g. *EmailContactField* calls the intent to prepare new email with filled-out email address. If there are more possible actions that can be done with the record, the context menu is displayed and user can choose the next action
- e.g. *PhoneContactField* can be used to dial the contact's phone number, or to prepare new SMS using the contact's phone number.

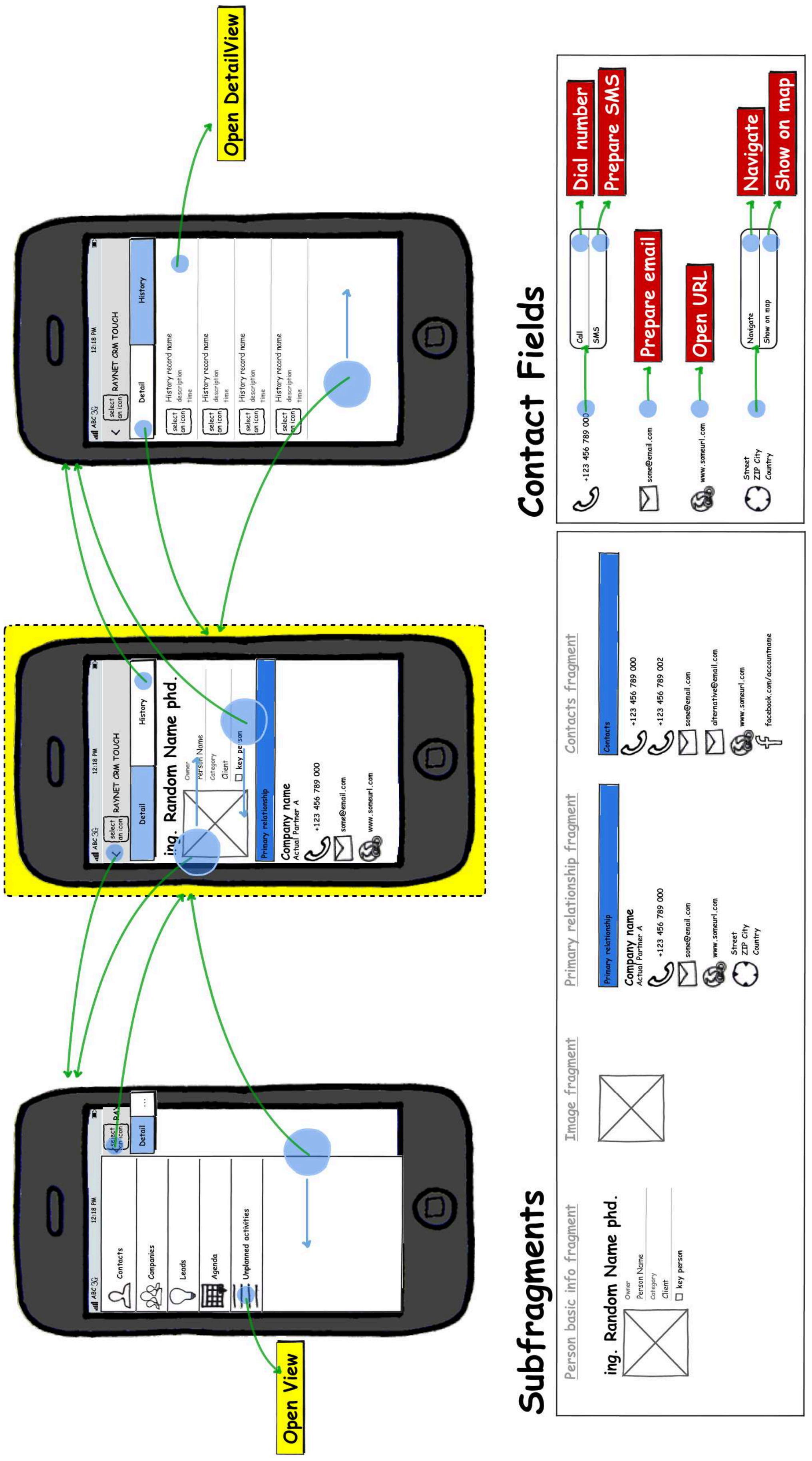


FIGURE 4 - DetailView layout

2.3.4 LoginView layout

RAYNET CRM Touch application contains two entry points. The first one is responsible for authorizing users when opening the application and the second one is responsible for authentication user for the *Account Authenticator Service*. Both entry points have specific requirements and need to be handled by their specific *LoginViews*. (See fig. 5)

Logging into application can be done in two ways - by selecting *Account* stored in *AccountManager*, or by filling-out the login form. For these different approaches custom *Fragments* were created.

AccountListFragment provides the list of application's *Accounts* stored in *Account Manager*, *AccountLoginFragment* contains the login form fields.

The application login screen contains both *AccountListFragment* and *AccountLoginFragment*. If the *Account Manager* does not contain any RAYNET CRM Touch application *Accounts*, the *AccountListFragment* visibility is set as gone.

If a user chooses the logging into application using the *AccountLoginFragment*, user can choose if the credentials will be stored in the *Account Manager* by checking the *"Store as Account" checkbox*.

Account Authenticator Service's login screen contains only *AccountLoginFragment* and since the credentials are always saved into *AccountManager*, the *"Store as Account" checkbox.*

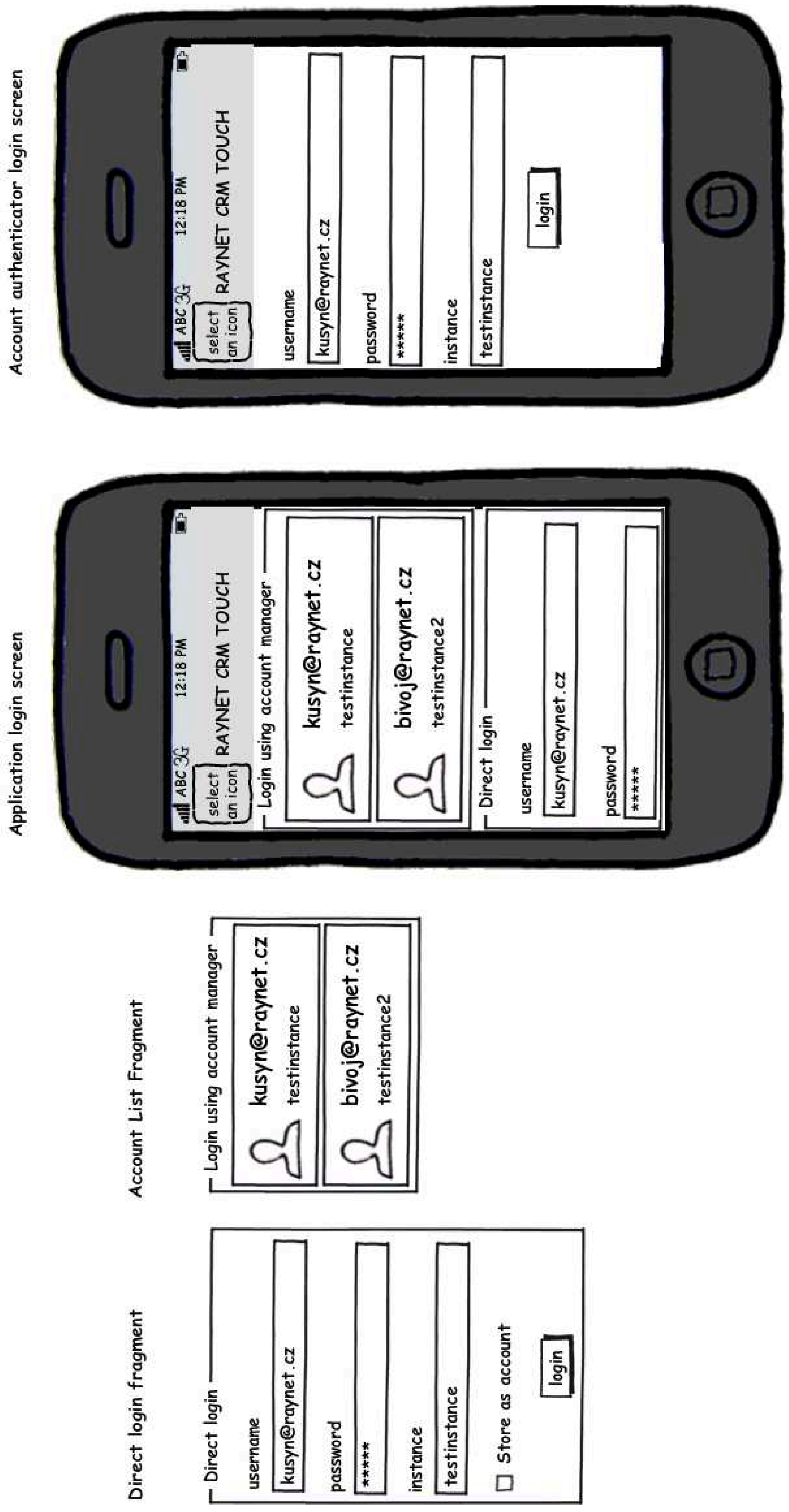


FIGURE 5 - LoginView layout

3 ANDROID DEVELOPMENT

3.1 Design

Designing the application's User Interface, it is always good to follow the platform's design guidelines. Using the platform specific elements and patterns in correct context simplifies the application to users, who are already used to work with these elements.

> *"Most developers want to distribute their apps on multiple platforms. As you plan your app for Android, keep in mind that different platforms play by different rules and conventions. Design decisions that make perfect sense on one platform will look and feel misplaced in the context of a different platform. While a "design once, ship anywhere" approach might save you time up-front, you run the very real risk of creating inconsistent apps that alienate users. Consider the following guidelines to avoid the most common traps and pitfalls." (Pure Android)*

3.1.1 ActionBarSherlock library

One of the difficulties during Android application development is dealing with different versions of Android system. Despite the fact that Android API contains a support library that allows programmers to use new features on older Android versions, it does not really allow to develop Android 4.x application that will run on devices with Android 2.x. The problem in this case is, that support library does not contain Android's 4.x Holo theme pack, which is essential for creating an application that follows current UX standards. Older Android themes did not support one of the most used features of today's Android UI - *Action bars*.

Of course it is possible to create Android 2.x full compatible application and run it on version 4.x, or simply do not support the older versions. Neither of the solutions can provide both - market penetration and good user experience.

The nonexistence of an optimal solution lead to creation of non-commercial ActionBarSherlock library, that extends original Android's support library. *"ActionBarSherlock is an extension of the support library designed to facilitate the use* *of the action bar design pattern across all versions of Android with a single API." (ActionBarSherlock)*

> *The library will automatically use the native action bar when appropriate or will automatically wrap a custom implementation around your layouts. This allows you to easily develop an application with an action bar for every version of Android from 2.x and up. (ActionBarSherlock)*

3.1.2 SlidingMenu library

One of popular present UX elements is a sliding menu. It allows users to open the menu simply by dragging the edge of the screen, which is very comfortable and shifts user experience a little bit further.

However this functionality is not a part of the Android API, so it is necessary to implement the functionality, or use SlidingMenu library that covers it.

> *SlidingMenu is an Open Source Android library that allows developers to easily create applications with sliding menus like those made popular in the Google+, YouTube, and Facebook apps. Feel free to use it all you want in your Android apps provided that you cite this project and include the license in your app. (SlidingMenu)*

One of the advantages of SlidingMenu library is the possibility to link it with the ActionBarSherlock library and use the sliding menu feature on Android 2.x devices.

3.1.3 ViewPagerIndicator library

Android 3.x version and above provides *ViewPager* component that allows users to switch between the contained fragments using the Swipe gesture. From a user's point of view it is very convenient.

ViewPagerIndicator library provides sets of components to indicate which page is currently selected by *ViewPager* component and to represent it to the user in many different forms including tabs, icons or titles.

ViewPagerIndicator library is also compatible with ActionBarScherlock library and Android 2.x compatible.

3.2 Using Android intents

RAYNET Cloud CRM mobile client is supposed to open other installed applications to provide users a possibility to work with some detail of application's detail outside the application itself - e.g. writing an email to a person from a contact list. Opening the other application can be done by using a hardcoded link or using the Android's intent API. However, Android guidelines recommend using intents.

> *Do not hardcode links to other apps. In some cases you might want your app to take advantage of another app's feature set. For example, you may want to share the content that your app created via a social network or messaging app, or view the content of a weblink in a browser. Do not use hard-coded, explicit links to particular apps to achieve this. Instead, use Android's intent API to launch an activity chooser which lists all applications that are set up to handle the particular request. This lets the user complete the task with their preferred app. For sharing in particular, consider using the Share Action Provider in your action bar to provide faster access to the user's most recently used sharing target. (Pure Android)*

> *Intents are asynchronous messages which allow Android components to request functionality from other components of the Android system. Intents can be used to signal to the Android system that a certain event has occurred. Other components in Android can register to this event via an intent filter. (Vogel 2013)*

On example below is showed the example usage of calling an implicit intent for sending the pre-prepared email message. "*Implicit intents specify the action which should be performed and optionally data which provides data for the action.*" (Vogel 2013)

```
Intent intent = new Intent(Intent.ACTION_SENDTO,
        Uri.parse("mailto:"+StringUtils.join(emailAddressList, ",")
                                + "?body=" + URLEncoder.encode(emailText)));
startActivity(intent);
```

When the code is executed, the list of installed applications that are hooked to the *ACTION_SENDTO* intent is displayed to user to choose the application, which should be used for further work (in case that users did not set default a application for that kind of requests) and after that, intent data are passed to the selected application.

For opening an activity within the application itself, explicit intents are used. *Explicit intents explicitly define the component which should be called by the Android system,* *by using the Java class as identifier.* (Vogel 2013) The code below illustrates opening the *DetailView* of Company's owner by passing *ownerId* to *PersonDetailView* class.

```
Intent detailIntent = new Intent(view.getContext(),PersonDetailView.class);
detailIntent.putExtra("recordId", CompanyDetailView.this.ownerId);
startActivity(detailIntent);
```

3.3 Using AccountManager

Android platform allows to store application's login details in device. This simplifies logging into the application, and also allows using synchronization services to synchronize application contacts or calendar events with built-in applications.

Extending AbstractAccountAuthenticator class

To allow users to use the *AccountManager*, several steps must be taken. *AccountManager* expects the data to be passed in a specific way. Account authenticator class needs to be created and it has to extend *AbstractAccountAuthenticator* class and override its abstract functions.

Most important function to implement is **AbstractAccountAuthenticator.addAccount** function, which is responsible for retrieving user's credentials, authenticating them and then storing them using Account Manager. It is programmer's responsibility to implement the code for these procedures.

Retrieving user's credentials can be done in various ways including exotic procedures such as biometric scan, however, the most common is to display login form to user and obtain the credentials from it.

Authentication of credentials can be done by connecting to the server and checking if the connection was authorized, or connecting to the application's database and checking if there exist local account - in case of purely local application.

When the credentials are retrieved, it is time to store them locally. This is done by function **AccountManager.addAccountExplicitly**

```
final Account account = new Account(mUsername, your_account_type);
mAccountManager.addAccountExplicitly(account, mPassword, extra_info);
```

While storing the credentials using the *AccountManager*, credentials should be protected.

> *It is important to understand that AccountManager is not an encryption service or a keychain. It stores account credentials just as you pass them, in plain text. On most devices, this isn't a particular concern, because it stores them in a database that is only accessible to root. But on a rooted device, the credentials would be readable by anyone with adb access to the device.*
>
> *With this in mind, you shouldn't pass the user's actual password to AccountManager.addAccountExplicitly(). Instead, you should store a cryptographically secure token that would be of limited use to an attacker. If your user credentials are protecting something valuable, you should carefully consider doing something similar. (Creating a custom Account Type)*

A suitable solution for a situation when secure token cannot be retrieved from server is to implement one's own password encryption procedure. The simplest way how to achieve this is to use one of the symmetric key algorithms (e.g. TripleDES) to encode the password before passing it to the **AccountManager.AddAccountExplicitly** function. The key can be stored as private static variable in encryption service. When the account data is to be retrieved from Account Manager and passed to the application, the password would be decoded again.

This procedure will make the password more complicated to break. To retrieve the encryption key it would be necessary to decompile an application's binary code.

3.4 Creating Authenticator Service

> *Account authenticators need to be available to multiple applications and work in the background, so naturally they're required to run inside a Service. We'll call this the authenticator service.*
>
> *Your authenticator service can be very simple. All it needs to do is create an instance of your authenticator class in **onCreate** and call **getIBinder** in **onBind**. (Creating a custom Account Type)*

Authenticator service has to be registered in the manifest.xml file and the **AccountAuthenticator** intent must be declared, so that the system can recognize it correctly and allow user to store application's account and allow it to use its available synchronization services.

```
<service
  android:name="cz.raynet.raynetcrm_touch.service.CrmAccounAuthenticateService"

  android:exported="true"
  android:process=":auth" >
  <intent-filter>
      <action android:name="android.accounts.AccountAuthenticator" />
  </intent-filter>

  <meta-data
      android:name="android.accounts.AccountAuthenticator"
      android:resource="@xml/authenticator" />
</service>
```

Also the resources for the service should be defined. That is important for a correct representation of the account in system.

```
<account-authenticator xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="cz.raynet.raynetcrm_touch.account"
    android:icon="@drawable/ic_launcher"
    android:smallIcon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:accountPreferences="@xml/account_preferences"/>
```

3.4.1 Content providers

When the synchronization of data between the application and system is needed (e.g. contact synchronization), the best practice is to use Android's *Content Providers.*

> *Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process. (Content Providers)*

Programmers can use the default *contents providers* to implement a way to connect an application and a system. Android covers most common system entities (e.g. contacts, calendars, multimedia, user dictionary) with its own content providers, however, it is also possible to write custom content provider for the application.

> *A content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database. A row*

> *represents an instance of some type of data the provider collects, and each column in the row represents an individual piece of data collected for an instance. (Content Provider Basics)*

Retrieving data in an application from a system using the content provider is done by executing the ***query*** method of *ContentResolover* class.

Following code example is from *Content Provider Basics*

```
// Queries the user dictionary and returns results
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    mProjection,                        // The columns to return for each row
    mSelectionClause                    // Selection criteria
    mSelectionArgs,                     // Selection criteria
    mSortOrder);                        // The sort order for the returned rows
```

Based on *Content URI*, the specific Content provider's ***query*** method is called and parameters are passed. Comparing ***query*** method to the relational database query, the *Content URI* works as table identifier.

All content resolvers that are used in application needs to be listed in the manifest file to obtain necessary permissions.

> *The ContentResolver.query() client method always returns a Cursor containing the columns specified by the query's projection for the rows that match the query's selection criteria. A Cursor object provides random read access to the rows and columns it contains. Using Cursor methods, you can iterate over the rows in the results, determine the data type of each column, get the data out of a column, and examine other properties of the results.*
> *(Content Provider Basics)*

If there are no results matching the query's selection criteria, an empty cursor is returned.

3.4.2 Sync Providers

"Sync providers are services that allow an Account to synchronize data on the device on a regular basis." (Writing an Android Sync Provider) To create *sync provider*, it is necessary to create service, that returns a subclass of *AbstractThreadedSyncAdapter* in service's ***onBind*** method.

Sync provider is linked to the authenticator service and from user's point of view; the sync provider is located under account settings in Account and Synchronization menu option. When the synchronization request is initiated, account object is passed to the ***onPerformSync*** method of implemented *AbstractThreadedSyncAdapter* subclass.

onPerformSync method is responsible for whole synchronization process, including connecting to the server and authorizing clients account. Synchronization can be done in several ways. The simplest situation is when the device only mirrors the records on the server end can't edit them. Easiest way is to delete all the account's synchronized records and replace them with actual ones.

In situation that records can be edited on device, it is necessary to implement a correct way to synchronize records. In this case, it also can come to pass that the record was edited on both sides - in the devices and also on server. In that case it is necessary to decide which record will be preferred, or allow user to choose the record. However, demanding user to choose the preferred record is not very user friendly in the case, when user sets automatic synchronization option - it brings issues on both sides, technology and user experience.

3.4.3 Jackson library

Jacskon library is Java JSON-processor library. Library itself doesn't depend on any other package beyond JDK, contains JSON parser and JSON generator and supports data binding.

4 IMPLEMENTATION

4.1 UX Components

The main UX components are based on several abstract class that provides the basic functionality and describes the basic behavior of the activity.

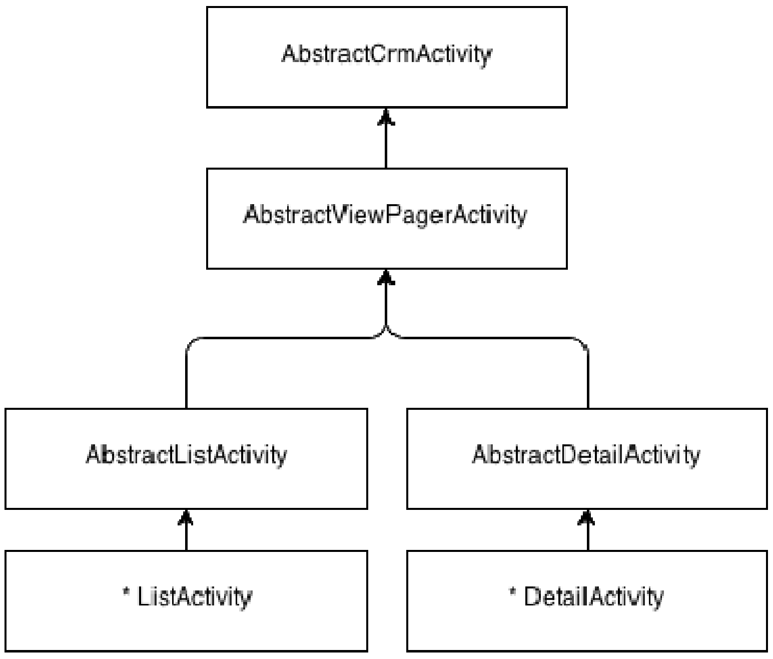


FIGURE 6 - Simplified inheritance model

The basic abstract class is *AbstractCrmActivity* extending the *SlidingFragmentActivity* from *SlidingMenu* library.

AbstractCrmActivity defines the abstract **init** method that is supposed to be called when the necessary checks are performed.

```
abstract void init();
```

onCreate method sets the main menu fragment in the sliding menu drawer and its behavior.

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setBehindContentView(R.layout.menu_frame);
        FragmentTransaction t = this.getSupportFragmentManager().beginTransaction();
```

```
       mFrag = new MainMenuFragment();
       t.replace(R.id.menu_frame, mFrag);
       t.commit();

       SlidingMenu sm = getSlidingMenu();
       sm.setShadowWidthRes(R.dimen.shadow_width);
       sm.setShadowDrawable(R.drawable.shadow);
       sm.setBehindOffsetRes(R.dimen.slidingmenu_offset);
       sm.setFadeDegree(0.35f);
       sm.setTouchModeAbove(SlidingMenu.TOUCHMODE_MARGIN);

       getSupportActionBar().setDisplayHomeAsUpEnabled(true);
}
```

switchContent defines the behavior of opening the *SlidingMenu*

```
public void switchContent(Fragment fragment) {
    mContent = fragment;
    getSupportFragmentManager()
    .beginTransaction()
    .replace(R.id.content_frame, fragment)
    .commit();
    getSlidingMenu().showContent();
}
```

handleException method is defined to inform the user that requested action didn't performed correctly.

```
public void handleException(Exception e){
    e.printStackTrace();
    Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();
}
```

In **onStart** performs the checks for internet connection availability and ensures that user is logged into the application. If the checks are passed, the init method is called.

```
@Override
protected void onStart() {
    super.onStart();
    if(!isNetworkAvailable()){
          this.startActivityForResult(
                    new Intent(android.provider.Settings.ACTION_WIFI_SETTINGS),0);
          return;
      }

    if(!AuthenticationProvider.getAuthenticationProvider().isSet()){
          this.startActivity(new Intent(this, LoginActivity.class));
        return;
    }

      init();
}

private boolean isNetworkAvailable() {
    ConnectivityManager connectivityManager =
        (ConnectivityManager) this.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo activeNetworkInfo = connectivityManager.getActiveNetworkInfo();
    return activeNetworkInfo != null;
}
```

AbstractViewPagerActivity extends the *AbstractCrmActivity* and works as a basic class for *Activities* that use multiple *Fragments*.

ArrayList of fragments *mFragments* is defined as well as *ViewPager* itself and the *TabPageIndicator*.

```
protected ArrayList<Fragment> mFragments = new ArrayList<Fragment>();
protected TabPageIndicator mTabPageIndicator;
protected ViewPager mViewPager;
```

onCreate method sets the *ViewPager* as the content view of the sliding menu frame and initializes the *page indicator* component.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.fragment_view_pager);

    mTabPageIndicator = (TabPageIndicator) findViewById(R.id.indicator);

    mViewPager = (ViewPager) findViewById(R.id.pager);
    mViewPager.setId("VP".hashCode());
    mViewPager.setAdapter(new PagerAdapter(getSupportFragmentManager()));
    mViewPager.setCurrentItem(0);

    mTabPageIndicator.setViewPager(mViewPager);

}
```

Important part of *AbstractViewPagerActivity* is implementation of *FragmentPagerAdapter* responsible for switching the Fragments inside *ViewPager* component.

```
public class PagerAdapter extends FragmentPagerAdapter {

    public PagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public int getCount() {
        return mFragments.size();
    }
    @Override
    public CharSequence getPageTitle(int position) {
        return getTabTitle(position);
    }

    @Override
    public Fragment getItem(int position) {
        return mFragments.get(position);
    }

}
```

4.1.1 ListViews

To create the *ListViews* according to the design conception described in chapter 2.3.2 - ListView design, it was necessary to solve several issues first - the general structure of the list *Activities*, behavior of the subcomponents, communication between *Activity* and its subcomponents, filtering logic and filter structuring.

General ListView structuring

Every entity list view is implemented by its own *Activity* that extends abstract class *AbstractListActivity*. *AbstractListActivity* extends *ViewPagerActivity*, implements the common methods and defines abstract methods that are required to be implemented by subclasses.

Every subclass should define two basic Fragments - Fragment that contains list of entity records itself, and Fragment that contains set of the filters that can be applied to the entity list.

```
abstract AbstractListFragment getListFragment();
abstract FilterListFragment getFilterFragment();
abstract void setStandardFilters();
abstract int getTitleResource();
```

To handle filtering, *AbstractListView* defines methods **setFilters** and **resetFilters**. **setFilters** method's argument is a *Map* containing a set of *CriteriaObject* objects that would be applied to list. Newly set filters are applied to the set of filters that are already stored, the indication of filtering is enabled, and the method for retrieving new list data from the *ListActivity's ListFragment* is called.

```
public void setFilters(Map<String, CriteriaObject> filters) {
    mFilters.clear();
    mFilters.putAll(mStandardFilters);
    mFilters.putAll(filters);
    setFilterActive(true);
    getListFragment().reset();
    getListFragment().loadData(
                        new ArrayList<CriteriaObject>(mFilters.values()),
                        mFulltext);
    mViewPager.setCurrentItem(0);
}
```

resetFilter first cleans the currently used filters *Map* and loads standard set of filters, then set the filter indication off and calls the *ListFragment's* method to retrieve data.

```
public void resetFilters() {
    mFilters.clear();
    mFilters.putAll(mStandardFilters);
    mFulltext = null;
    resetFullText();
    setFilterActive(false);
    getListFragment().reset();
    getListFragment().loadData(
                        new ArrayList<CriteriaObject>(mFilters.values()),
                        mFulltext);
    mViewPager.setCurrentItem(0);}
```

Adding the fulltext search component brought the compatibility issue. Current version of ActionBarScherlock library does not correctly implement the *SearchView* component, so it's not working properly on Android 4.x. On the other side, standard Android 4.x *SearchView* component is not supported on Android 2.x.

In this case it was necessary to create two different XML menu definition files - first one for Android 2.x with *SearchView* component from ActionBarScherlock library, and second one for Android 4.x using the standard *SearchView* component. Aside of that, both XML menu definition files are identical.

Using the different *SearchView* component also brought the need to specify different *OnQueryTextListeners* with same function.

Fulltext SearchView component is set in **onCreateOptionsMenu** method.

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
@Override
public boolean onCreateOptionsMenu(Menu menu) {

    int currentapiVersion = android.os.Build.VERSION.SDK_INT;
    if (currentapiVersion >= android.os.Build.VERSION_CODES.ICE_CREAM_SANDWICH){
        MenuInflater inflater = getSupportMenuInflater();
        inflater.inflate(R.menu.person_list_menu, menu);
        mSearchViewContainer = menu.findItem(R.id.menu_search);
        SearchView sv = (SearchView) mSearchViewContainer.getActionView();
        sv.setOnQueryTextListener(new SearchView.OnQueryTextListener() {
            @Override
            public boolean onQueryTextSubmit(String query) {
                setFullText(query);
                return true;
            }

            @Override
            public boolean onQueryTextChange(String newText) {
                return false;
            }
        });
    } else {
        MenuInflater inflater = getSupportMenuInflater();
        inflater.inflate(R.menu.person_list_menu_support, menu);
        mSearchViewContainer = menu.findItem(R.id.menu_search);
        com.actionbarsherlock.widget.SearchView sv =
                                      (com.actionbarsherlock.widget.SearchView)
                                        mSearchViewContainer.getActionView();

        sv.setOnQueryTextListener(
             new com.actionbarsherlock.widget.SearchView.OnQueryTextListener() {
            @Override
            public boolean onQueryTextSubmit(String query) {
                setFullText(query);
                return true;
            }

            @Override
            public boolean onQueryTextChange(String newText) {
                return false;
            }
        });
    }
    return true;
}
```

Fulltext search is handled by methods *setFullText* and *resetFullText*. This method sets the *mFulltext* property and reloads the list data. Fulltext search and filtering work independently - it is possible to search in the filtered records using the fulltext.

```
public void setFullText(String fullText) {
    mFulltext = fullText;
    setFilterActive(true);
    getListFragment().reset();
    getListFragment().loadData(
                          new ArrayList<CriteriaObject>(mFilters.values()),
                          mFulltext);
    mViewPager.setCurrentItem(0);
}
```

resetFullText method resets the value of the *mFullText* view property and resets the *SearchView* component. Since the *SearchView* components are not compatible, it is necessary to use duplicate code on different *SearchView* component depending on the used version of Android operating system.

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public void resetFullText() {
    mFulltext = null;

    int currentapiVersion = android.os.Build.VERSION.SDK_INT;
    if (currentapiVersion >= android.os.Build.VERSION_CODES.ICE_CREAM_SANDWICH){
        SearchView sv = (SearchView) mSearchViewContainer.getActionView();
        sv.setQuery("", false);
    } else {
        com.actionbarsherlock.widget.SearchView sv =
                                      (com.actionbarsherlock.widget.SearchView)
                                      mSearchViewContainer.getActionView();
        sv.setQuery("", false);
    }

    mSearchViewContainer.collapseActionView();
}
```

AbstractListActivity also contains methods for updating the titles in the *TabPageIndicator* component. **setListTotalRecordCount** method adds the information about total count of actually found entity records with currents filters applied, with paging information excluded. When the page title is updated, *TabPageIndicator* component is notified about this change.

```
public void setListTotalRecordCount(int count){
    mListTotalRecordCount = count;
    mPageTitles.set(0, String.format("entity (%s)", count));
    mTabPageIndicator.notifyDataSetChanged();
}
```

Function of **setFiltersActive** method is similar. Filter tab page title is updated according to the status of applied filters and then *TabPageIndicator* is notified.

```
public void setFilterActive(boolean active){
    mFilterActive = active;
    String title = active?"filters (active)":"filters";
    mPageTitles.set(1, title);
    mTabPageIndicator.notifyDataSetChanged();
}
```

Users can reset the filters using the menu button. Handling the click on the menu buttons is done by the **onOptionItemSelected** method. Based on the menu item's id, the particular action is executed. In the case of *Reset filters button*, the action is to call reset filter methods in both - *ListActivity* and *FilterFragment*.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case R.id.reset_filters:
        resetFilters();
        getFilterFragment().resetFilters();
        return true;
    default:
        return false;
    }
}
```

AbstractListActivity overrides the ***onCreate*** method to apply the requested behavior of the list *Activities*. The title of the activity is set, **setStandardFilters** method implemented by subclass is called, *ListFragment* and *FilterFragment* are added to the *ViewPager* list of fragments, *TabPageIndicator* is notified and *listFragment* is requested to load the data with default set of filters.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setTitle(getString(getTitleResource()));
    setStandardFilters();

    AbstractListFragment lf = getListFragment();

    mFragments.add(lf);
    mFragments.add(getFilterFragment());
    mTabPageIndicator.notifyDataSetChanged();

    lf.loadData(new ArrayList<CriteriaObject>(mFilters.values()));
}
```

Entity ListActivity example

Each entity *ListActivity* extends *AbstractListActivity* and specifies the details of *ListActivity* implementation. The *AbstractListActivity* abstract methods are implemented.

```
private AbstractListFragment mListFragment = new PersonListFragment();
private FilterListFragment mFilterFragment = new FilterListFragment();

@Override
AbstractListFragment getListFragment() {
    return mListFragment;
}

@Override
FilterListFragment getFilterFragment() {
    return mFilterFragment;
}
```

```
@Override
int getTitleResource() {
    return R.string.person_list_activity_name;
}

@Override
  void setStandardFilters() {
}
```

General ListFragment structuring

General behavior of list *Fragments* is determined by *AbstractListFragment*, an abstract class that extends *ListFragment*.

AbstractListFragment defines abstract methods **taskBackgroundFn** and private for *listAsyncTask - AsyncTask* subclass defined in body of the class, **getDetailClass** method, that returns the Class of View used for displaying entity detail, **getActionModeCallback** method to obtain a callback for handling *ActionMode* menu, and *mapItem* method defining the mapping of the entity record to the *ListView* item.

```
abstract JsonNode taskBackgroundFn(String...parameters)
              throws CrmConnectionException,
                     CrmConnectorException,
                     OnlineDataConnectorException;
abstract ActionMode.Callback getActionModeCallback();
abstract Class<?> getDetailClass();
abstract View mapItem(JsonNode data);
```

onCreateView method is overriden to define several *ListFragment* specific features. The *mProgressBar* property is defined, *ListView* is defined and the *onScrollListener* and *OnItemLongClickListener* are set, list's adapter is created and set and in the end progress bar is set to the *Fragment's* list view as the footer view and the height is adjusted to fill the whole container.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                                                    Bundle savedInstanceState) {
    mActivity = getActivity();

    mProgressBar = inflater.inflate(R.layout.list_progress_bar, null);
    mListView = (ListView) inflater.inflate(mViewResource, null);
    mListView.setOnScrollListener(mOnScrollListener);
    mListView.setOnItemLongClickListener(mOnItemLongClickListener);

    mAdapter = new ListItemAdapter(mActivity);
    setListAdapter(mAdapter);

    mListView.addFooterView(mProgressBar);
    setProgressBarHeight();

    return mListView;
}
```

The above mentioned *OnItemLongClickListener* is responsible for calling the **setSelectedView** method in charge of displaying the panel with extra record information hidden in the list's row, and also for displaying the *ActionMode* menu with most commonly used entity record actions.

```
private OnItemLongClickListener mOnItemLongClickListener =
                                                new OnItemLongClickListener() {

  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  @Override
  public boolean onItemLongClick(
                      AdapterView<?> parent, View view, int position, long id) {

      setSelectedView(view);

      mSelectedItem = mAdapter.getItem(position);
      if (mActionMode != null) {
          mActionMode.invalidate();
      } else {
          mActionMode = ((SherlockFragmentActivity) mActivity)
                                      .startActionMode(getActionModeCallback());
      }

      return true;
  }
};
```

setSelectedView method hides the panel with extra record information of last selected record's View, then sets new record's View as selected and displays it's extra record information panel.

```
protected void setSelectedView(View view) {
    if (mSelectedView != null) {
        mSelectedView.findViewById(R.id.row_extra).setVisibility(View.GONE);
    }

    mSelectedView = view;
    mSelectedView.findViewById(R.id.row_extra).setVisibility(View.VISIBLE);
    mListView.invalidateViews();
}
```

AbstractListFragment provides the **loadData** method for loading and reloading data according to the filters and fulltext search that are set by the container *Activity*. Loading the data is done by calling the **execute** method of *listAsyncTask* class.

```
public void loadData(List<CriteriaObject> filters, String fulltext) {
    mFilters = filters;
    mFulltext = fulltext;

    new listAsyncTask().execute();
}
```

listAsyncTask is a subclass of *AsyncTask* prepared for usage in entity's *FragmentList* components. Since it is not possible to use the *HTTP request* on *MainThread* in Android, *AsyncTask* encapsulates handling of the threads and allows programmers to simply define the actions that have to be done before the request is done (**onPreExecute** method), calling the request itself (**doInBackground** method) and how to process the result of request (**onPostExecute** method).

doInBackground method of *listAsyncTask* calls the **taskBackgroundFn** method implemented by a subclass of *AbstractListFragment*. **doInBackground** is running in the separated thread, so it is not possible to simply propagate the exceptions to the UI. One of the possibilities is to use **runOnUiThread** method of Activity, which needs an instance of *Runnable* interface that will be executed. It was decided to create the *AsyncTaskResultObject<T>* that contains the result of *taskBackgroundFuncion*, or the *Exception* in the case that any problem occurred.

Propagating the *Exception* to the UI is then ensured by **onPostExecute** method, which is running in the main thread and its possible call the **handleException** method of current instance of *AbstractCrmActivity*.

In the case that no exceptions occurred, the result is processed and the particular records views are created using subclass of *AbstractListFragment* subclass's implementation of **mapItem** method.

```
protected class listAsyncTask extends
                      AsyncTask<String, Void, AsyncTaskResultObject<JsonNode>> {

  @Override
  protected void onPreExecute() {
      taskPreExecuteFn();
      if (mListView != null) {
          mListView.addFooterView(mProgressBar);
          setProgressBarHeight();
      }

  }

  @Override
  protected AsyncTaskResultObject<JsonNode> doInBackground(String... parameters) {

      AsyncTaskResultObject<JsonNode> result =
                                              new AsyncTaskResultObject<JsonNode>();
      try {
          result.setData(taskBackgroundFn(parameters));
      } catch (Exception e) {
          result.setException(e);
      }
      return result;
  }

  @Override
  protected void onPostExecute(AsyncTaskResultObject<JsonNode> result) {
      JsonNode data = result.getData();
      Exception e = result.getException();

      if (e != null) {
          ((AbstractCrmActivity) mActivity).handleException(e);
          return;
      }

      if (data.isMissingNode()) {
          return;
      }

      mTotalCount = data.path("totalCount").asInt();
      if (mActivity instanceof AbstractListActivity) {
          ((AbstractListActivity) mActivity).setListTotalRecordCount(mTotalCount);
      }

      for (JsonNode record : data.path("data")) {
          mAdapter.add(record);
          mItemViews.add(mapItem(record));
      }

      mListView.removeFooterView(mProgressBar);

      if (mStart + mLimit >= mTotalCount) {
          mListView.setOnScrollListener(null);
      } else if (mStart == 0) {
          mListView.setOnScrollListener(mOnScrollListener);
      }
  }
}
```

ListItemAdapter extends *ArrayAdapter* and overrides its **getView** method to allow usage of pre-prepared record views. Standard behavior of the *ArrayAdapter* is to keep in the memory only the visible views and two not directly visible views that are about to show when user starts to scroll up or down. The views are not replaced, only their data is switched.

That fact caused the problems with rendering the items with open extra info panel. All the record views are now saved in the memory and just passed to the adapter through **getView** function. The downside of this solution is larger memory consumption.

```
public class ListItemAdapter extends ArrayAdapter<JsonNode> {

    public ListItemAdapter(Context context) {
        super(context, 0);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        return mItemViews.get(position);
    }
}
```

Standard action when the list item is clicked is defined by **onListItemClick** method. Id of the selected record is obtained and set to the new *detailIntent Intent* object as the extra information. The entity's *DetailActivity* is obtained from via **getDetailClass** method, which is implemented by a subclass, and passed to the constructor of Intent. In the end, entity's *DetailActivity* is started.

```
@Override
public void onListItemClick(ListView lv, View v, int position, long id) {
    JsonNode item = mAdapter.getItem(position);

    int recordId =
      JsonUtils.getJsonNode(item, CommonEntityPath.ID.toString()).asInt();

    Intent detailIntent =
      new Intent(getActivity().getApplicationContext(), getDetailClass());
    detailIntent.putExtra("recordId", recordId);
    startActivity(detailIntent);
}
```

To handle the need of paging, the *OnScrollListener* was implemented. **onScroll** method is checking, if the index of currently visible item did not overstep the limit for new paging request. In praxis this works in the way, that when the user is closing to the end of currently loaded records, the request for more record is called and usually finished before user gets to the end.

The *OnScrollListener* is set and removed in **onPostExecute** method of above described *listAsyncTask* class, so it is active only when the record list is not fully loaded.

```
protected OnScrollListener mOnScrollListener = new OnScrollListener() {
  @Override
  public void onScrollStateChanged(AbsListView view, int scrollState) {}

  @Override
  public void onScroll(AbsListView view, int firstVisibleItem, i
                              nt visibleItemCount, int totalItemCount) {
      if (firstVisibleItem + visibleItemCount > mStart + mLimit / 2) {
          mStart += mLimit;
          new listAsyncTask().execute();
      }
  }
};
```

Entity ListFragment example

Each entity's *ListFragment* extend *AbstractListFragment* and specifies the details of *ListFragment* implementation. The *AbstractListFragment* abstract methods are implemented.

taskBackgroundFn calls the method responsible to obtain correct data from the server.

```
 @Override
JsonNode taskBackgroundFn(String... parameters) throws
                                        CrmConnectionException,
                                        CrmConnectorException,
                                        OnlineDataConnectorException {
    return WebApiDataProvider.getPersonList(mFilters, mFulltext, mLimit, mStart);
}
```

getDetailClass method returns the Class that is used to display the detail of the entity's record.

```
@Override
Class<?> getDetailClass() {
    return PersonDetailActivity.class;
}
```

mapItem method returns the list's item view with mapped data.

```
@Override
View mapItem(JsonNode data) {

    View view = LayoutInflater.from(mActivity).inflate(getItemLayout(), null);

    String personName = JsonUtils.getJsonNode(data,
                            PersonListPath.FULL_NAME_WITHOUT_TITLES).asText();
    TextView title = (TextView) view.findViewById(R.id.row_title);
    title.setText(personName);

    String companyName = JsonUtils.getJsonNode(data,
                    PersonListPath.PRIMARY_RELATIONSHIP_COMPANY_NAME).asText();
    TextView subtitle = (TextView) view.findViewById(R.id.row_subtitle);
    subtitle.setText(companyName);

    JsonNode tel1 = JsonUtils.getJsonNode(data, PersonListPath.TEL1);
    JsonNode email = JsonUtils.getJsonNode(data, PersonListPath.EMAIL);

    LinearLayout extraLayout = (LinearLayout) view.findViewById(R.id.row_extra);

    if (tel1.isTextual()) {
        String tel1text = tel1.asText();
        TextView tv = new TextView(mActivity);
        tv.setText(getString(R.string.tel1) + tel1text);
        extraLayout.addView(tv);
    }

    if (email.isTextual()) {
        String emailText = email.asText();
        TextView tv = new TextView(mActivity);
        tv.setText(getString(R.string.email) + emailText);
        extraLayout.addView(tv);
    }

    return view; }
```

getActionModeCallback method returns instance of implemented *ActionMode.Callback* interface.

```
@Override
Callback getActionModeCallback() {
    return mActionModeCallback;
}
```

Instance of *ActionMode.Callback* is responsible for handling most important *ActionMode* menu events. **onCreateActionMode** menu method can set the menu items according to the *.xml menu resource file. Since *ActionMode* menu in RAYNET CRM Touch should be variable according to the available records information, the author of the thesis decided to set the items in the **onPrepareActionMode** method, which is called every time the *ActionMode* is shown. **onCreateActionMenu** returns Boolean that indicates is the action mode was created or not.

```
private enum ActionCode{CALL,SMS,EMAIL};
protected ActionMode.Callback mActionModeCallback = new ActionMode.Callback(){
    private final HashMap<ActionCode, String> mDataMap =
                new HashMap<PersonListFragment.ActionCode, String>();
    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        return true;
    }
```

In **onPrepareActionMenu**, first the menu and map with actions extra information are cleared, and then the menu items are added in dependency on available fields in record item. At the same time the extra information for the actions is put into the *Map*.

```
    @Override
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {

        menu.clear();
        mDataMap.clear();

        JsonNode tel1 = JsonUtils.getJsonNode(mSelectedItem,
                                    PersonListPath.TEL1.toString());
        JsonNode email = JsonUtils.getJsonNode(mSelectedItem,
                                    PersonListPath.EMAIL.toString());

        if(tel1.isTextual()){
            String tel1text = tel1.asText();
            menu.add(Menu.NONE,ActionCode.CALL.ordinal(),
              Menu.NONE, getString(R.string.call)+ tel1text);
            mDataMap.put(ActionCode.CALL, tel1text);

            menu.add(Menu.NONE,ActionCode.SMS.ordinal(),
              Menu.NONE, getString(R.string.sms)+tel1text);
            mDataMap.put(ActionCode.SMS, tel1text);

        }

        if(email.isTextual()){
            String emailText = email.asText();
            menu.add(Menu.NONE,ActionCode.EMAIL.ordinal(),
              Menu.NONE, getString(R.string.email)+emailText);
            mDataMap.put(ActionCode.EMAIL, emailText);
        }

        return true;
    }
```

onActionItemClicked function is handling the click events of the menu items. New intent is created according to the selected action and then the responsible activity is started.

```
    @Override
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {

        if(item.getItemId() == ActionCode.CALL.ordinal()){
            Intent intent = new Intent(Intent.ACTION_CALL,
              Uri.parse("tel:"+mDataMap.get(ActionCode.CALL)));
            startActivity(intent);
        }
        if(item.getItemId() == ActionCode.SMS.ordinal()){
            Intent intent = new Intent(Intent.ACTION_VIEW,
              Uri.parse("sms:"+mDataMap.get(ActionCode.SMS)));
            startActivity(intent);
        }
        if(item.getItemId() == ActionCode.EMAIL.ordinal()){
            Intent intent = new Intent(Intent.ACTION_SENDTO,
              Uri.parse("mailto:"+mDataMap.get(ActionCode.EMAIL)));
            startActivity(intent);
        }

        return false;
    }
```

onDestroyActionMode is a method called when the *ActionMode* component is dispatched. In this case it sets the value in container *ListFragment* to null, so it cannot be referenced anymore.

```
    @Override
    public void onDestroyActionMode(ActionMode mode) {
        mActionMode = null;
    }
};
```

General FilterFragment structuring

General structure of *FilterFragments* is defined by *AbstractFilterFragment* class. *FilterFragment* contains *LinearLayout* container *mFiltersPane* and a *Button mApplyBtn*. The filter fields are stored in the *mFilterFields* List. In the **onCreateFunction** the Fragment's layout is inflated, the basic elements are obtained and the *mApplyButton's OnClickListener* is set.

The action of the *mApplyButton* is to send the filters to container implementation of *AbstractListActivity*, which propagates the filters further to its *ListFragment*.

```
@Override
public View onCreateView(LayoutInflater inflater,
                              ViewGroup container,
                              Bundle savedInstanceState) {

    mContext = getActivity();
    mView = inflater.inflate(R.layout.list_filter, null);

    mInflater = inflater;
    mApplyBtn = (Button) mView.findViewById(R.id.apply_btn);
    mApplyBtn.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            ((AbstractListActivity) getActivity()).setFilters(mFilters);
        }
    });
    mFiltersPane = (LinearLayout) mView.findViewById(R.id.filtersListLayout);

    initFilters();
    return mView;
}
```

onCreateView method also calls the **initFilters** method that obtains the *List* of *filterFields* from *AbstractListFragment* subclass using its implementation of abstract **getFilterFields** method.

When the *filterFields* are obtained, their views are prepared and added to the *mFilterPane* layout using the *FragmentTransaction*.

```
public void initFilters() {

    FragmentTransaction ft;
    ft = getFragmentManager().beginTransaction();
    mFilterFields = getFilterFields();

    for (IResetable filterField : mFilterFields) {
        ft.add(mFiltersPane.getId(), (Fragment) filterField);
    }

    ft.commit();
}
```

setCriteria method provides the filter fields the way to pass it's *CriteriaObject* to the *FilterFragment's* collection of filters.

```
public void setCriteria(String criteriaName, CriteriaObject criteria) {
    mFilters.put(criteriaName, criteria);
}
```

To remove particular *CriteriaObject* from the collection of filter, there was implemented the **removeCriteria** method.

```
public void removeCriteria(String criteriaName) {
    mFilters.remove(criteriaName);
}
```

resetFilters method serves to reset all the *filterFields* to its standard value.

```
public void resetFilters() {
    for (IResetable filterField : mFilterFields) {
        filterField.reset();
    }
}
```

Entity FilterFragment example

Particular implementation of *AbstractFilterFragment* is supposed to define only the **getFilterFields** method that defines used *filterFields*.

```
@Override
protected List<IResetable> getFilterFields() {

    ExternalSelect personCategory = new ExternalSelect();
    personCategory.setArguments(
            new PersonCategoryAsyncTaskLoader(mContext),
            this,
            CodeListPath.ID,
            CodeListPath.CODE01,
            "category",
            "Category");

    ExternalSelect owner = new ExternalSelect();
    owner.setArguments(
            new OwnerAsyncTaskLoader(mContext),
            this,
            PersonDetailPath.ID,
            PersonDetailPath.FULL_NAME_WITHOUT_TITLES,
            "owner",
            "Owner");

    CheckBoxField openBc = new CheckBoxField();
    openBc.setArguments(
            "With open Business Case",
            "openBusinessCase",
            false,
            new CriteriaObject("openBusinessCase",OPERATOR.CUSTOM, null),
            this);

    CheckBoxField invalid = new CheckBoxField();
    invalid.setArguments(
            "Show invalid",
            "invalid",
            false,
            new CriteriaObject("rowInfo.rowAccess",OPERATOR.CUSTOM, "all-rows"),
            this);

    List<IResetable> filterFields = new ArrayList<IResetable>();
    filterFields.add(personCategory);
    filterFields.add(owner);
    filterFields.add(openBc);
    filterFields.add(invalid);

    return filterFields;
}
```

FilterField

All the *FilterFields* implements the *IResetable* interface, which specifies only the **reset** method. This method is supposed to set the value of filter field to its default value.

```
public interface IResetable {
    void reset();
}
```

FilterField itself extends the *Fragment* and it is only necessary to pass its *CriteriaObject* to the container *FilterFragment*, when the filter is active. The implementation itself can be simple like in the case of *CheckBoxField* that just pass the prepared *CriteriaObject* using the *OnCheckedChangeListener*.

```
public class CheckBoxField extends Fragment implements IResetable{

    protected String mTitle;
    protected String mMapping;
    protected boolean mDefault = false;
    protected CriteriaObject mCriteria;
    private AbstractFiltertFragment mParent;
    protected CheckBox mCheckBox;

    public void setArguments(String title, String mapping, boolean defaultValue,
                            CriteriaObject criteria,
                            AbstractFiltertFragment filterFragment){
        mTitle = title;
        mMapping = mapping;
        mDefault = defaultValue;
        mParent = filterFragment;
        mCriteria = criteria;
    }

    OnCheckedChangeListener onCheckedChangeListener = new OnCheckedChangeListener(){
        @Override
        public void onCheckedChanged(CompoundButton buttonView, boolean isChecked){
            if(isChecked){
                mParent.setCriteria(mMapping, mCriteria);
            }else{
                mParent.removeCriteria(mMapping);
            }
        }
    };

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                              Bundle savedInstanceState) {
        mCheckBox = new CheckBox(getActivity());
        mCheckBox.setText(mTitle);
        mCheckBox.setOnCheckedChangeListener(onCheckedChangeListener);
        reset();
        return mCheckBox;
    }

    @Override
    public void reset(){
        mCheckBox.setChecked(mDefault);
    }
}
```

In some cases it can be necessary to obtain filter values from the server first, like in the case of *ExternalSelectField*.

ExternalSelectField class implements *GeneralListArrayAdapter*, that is using the Loader to retrieve the data from server and then to process it.

```
public class GeneralListArrayAdapter extends ArrayAdapter<SimpleListItem>
                                           implements LoaderCallbacks<JsonNode>{

    public GeneralListArrayAdapter(Context context, int loaderId, Bundle bundle){
        super(context, 0);

        getLoaderManager().initLoader(loaderId, bundle, this).forceLoad();
    }

    public View getView(int position, View convertView, ViewGroup parent) {
        if (convertView == null) {
            convertView = LayoutInflater.from(getContext())
                                                    .inflate(mItemLayout, null);
        }
        TextView title = (TextView) convertView.findViewById(android.R.id.text1);
        title.setText(getItem(position).tag);
        return convertView;
    }

    @Override
    public Loader<JsonNode> onCreateLoader(int arg0, Bundle arg1) {
        return mLoader;
    }

    @Override
    public void onLoadFinished(Loader<JsonNode> loader, JsonNode response) {
        for (JsonNode record : response.get("data")) {
            String name = JsonUtils.getJsonNode(record, mTagPath).asText();
            Long id = JsonUtils.getJsonNode(record, mIdPath).asLong();
            this.add(new SimpleListItem(name, id));
        }
    }

    @Override
    public void onLoaderReset(Loader<JsonNode> arg0) {

    }

}
```

In **onCreateView** method, there is an *AlertDialog* specified. This *AlertDialog* is using the above mentioned *GeneralListArrayAdapter* to obtain the list of records from the server and to display them in the form of simple dialog with list and two buttons.

The positive button closes the dialog and do no action. Negative button serves to reset the filter value to the default.

```
@Override
public View onCreateView(LayoutInflater inflater,
                              ViewGroup container, Bundle savedInstanceState) {
    Context c = getActivity();

    mAdapter = new GeneralListArrayAdapter(c, 0, null);

    AlertDialog.Builder builder = new AlertDialog.Builder(c);
    builder.setTitle(mTitle)
                  .setAdapter(mAdapter, new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int position) {
           SimpleListItem item = mAdapter.getItem(position);

           mParent.setCriteria(mMapping, new CriteriaObject(
                                 mMapping, CriteriaObject.OPERATOR.EQ, item.id));
           mSelectedTV.setText(item.tag);
        }
    }).setPositiveButton("Cancel", null)
    .setNegativeButton("Clear", new DialogInterface.OnClickListener() {

        public void onClick(DialogInterface dialog, int which) {
            mParent.removeCriteria(mMapping);
            mSelectedTV.setText("---");
        }
    });
    mDialog = builder.create();

    View view = inflater.inflate(mLayout, null);
    mTitleTV = ((TextView) view.findViewById(android.R.id.text1));
    mSelectedTV = ((TextView) view.findViewById(android.R.id.text2));
    mTitleTV.setText(mTitle);
    mSelectedTV.setText("---");
    view.setClickable(true);
    view.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            mDialog.show();
        }
    });

    return view;
}
```

Loader to be used is passed to the *ExternalSelectField* in **setArguments** method. This allows to use the filed for different variations of filtering requests.

```
public void setArguments(Loader<JsonNode> loader,
                            AbstractFiltertFragment filterFragment,
                            IEntityPath idPath, IEntityPath tagPath,
                            String mapping, String title) {
    mLoader = loader;
    mIdPath = idPath;
    mTagPath = tagPath;
    mMapping = mapping;
    mTitle = title;
    mParent = filterFragment;
}
```

Loaders in this case represents the subclass of *AsyncTaskLoader* that handles the asynchronous requests in background thread in similar way the *AsyncTask* do.

4.1.2 DetailViews

General DetailActivity

DetailActivity is a container *Activity* that holds all the parts of detail record together. All the *DetailActivities* implementations are based on *AbstractDetailActivity* abstract class extending the *ViewPagerActivity*.

AbstractDetailActivity defines two abstract methods - **getDetailFragment** for setting the implementation of *DetailActivity* and **getEntity** for obtaining the entity type for correct initialization of *HistoryFragment*.

```
abstract AbstractDetailFragment getDetailFragment();
abstract EntityEnum getEntity();
```

Also, the default *ViewPage* page titles are being hold in the list *mPageTitles* and the record id is being hold in *mRecordId* variable.

```
protected List<String> mPageTitles = Arrays.asList("entity", "context");
protected Long mRecordId;
```

In the overriden **onCreate** method the record id is obtained from *extras Bundle*.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Bundle extras = getIntent().getExtras();
    mRecordId = extras.getLong("recordId");
}
```

The init method sets the content *Fragments* and notifies the *PageIndicator* about this change.

```
@Override
void init() {
    HistoryFragment hf = new HistoryFragment();
    hf.setArguments(mRecordId, getEntity());
    AbstractDetailFragment df = getDetailFragment();
    df.setArguments(mRecordId);

    mFragments.add(df);
    mFragments.add(hf);

    mTabPageIndicator.notifyDataSetChanged();
}
```

The **getTabTitle** defines the default function to obtain the *ViewPager* page title.

```
@Override
public String getTabTitle(int position) {
    String title = mPageTitles.get(position);
    return title;
}
```

Entity DetailActivity example

The concrete implementation of *DetailActivity* defines the abstract methods from *AbstractDetailActivity* to define the correct entity types.

```
@Override
AbstractDetailFragment getDetailFragment() {
    return new PersonDetailFragment();
}

@Override
EntityEnum getEntity() {
    return EntityEnum.PERSON;
}
```

In **onCreate** method the corresponding titles are set to the *mPageTitles* list.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mPageTitles.clear();
    mPageTitles.add(getResources().getString(R.string.detail));
    mPageTitles.add(getResources().getString(R.string.context));
}
```

General DetailFragment

DetailFragment is a container Fragment for the record information *subfragments* displaying the actual data. *DetailFragment's* responsibility is to obtain the correct data and pass them to the *subfragments*.

All *DetailFragments* are subclasses of *AstractDetailFragment* abstract class. For obtaining the data corresponding to the entity type the *AbstractDetailFragment* defines the **mainTaskBackgroundFn** method, which is meant to be run in the **doInBackground** method of its own subclass of *AsyncTask* - *MainAsyncTask*.

For initializing the correct *subfragments* the abstract method **processData** is defined.

```
abstract JsonNode mainTaskBackgroundFn() throws CrmConnectionException,
                                                CrmConnectorException,
                                                OnlineDataConnectorException;
abstract void processData(JsonNode data);
```

The *MainAsyncTask* is executed in the the **onStart** method. **onStart** method is called in the case that container *Activity* is started or returned, so the data are obtained with every displaying of the *Activity*.

In **onStart** method also all current the *Fragments* are removed, so they can be added again when the response is processed.

```
@Override
public void onStart() {
    super.onStart();
    new MainAsyncTask().execute();

    FragmentTransaction ft;
    for (Fragment f : mFragments) {
        ft = getFragmentManager().beginTransaction();
        ft.remove(f);
        ft.commit();
    }
    mFragments.clear();
}
```

MainAsyncTask extends the *AsyncTask* class and defines the behavior of obtaining the data from server.

In **onPostExecute** method the obtained data are passed to the abstract **processData** method where are the correct *Fragments* initialized, and then the *Fragments* are attached to the container layout.

```
protected class MainAsyncTask extends
                      AsyncTask<String, Void, AsyncTaskResultObject<JsonNode>> {

  @Override
  protected void onPreExecute(){}

  @Override
  protected AsyncTaskResultObject<JsonNode> doInBackground(String... parameters){

      AsyncTaskResultObject<JsonNode> result =
                                    new AsyncTaskResultObject<JsonNode>();
      try {
          result.setData(mainTaskBackgroundFn());
      } catch (Exception e) {
          result.setException(e);
      }
      return result;
  }

  @Override
  protected void onPostExecute(AsyncTaskResultObject<JsonNode> result) {
      JsonNode data = result.getData();
      Exception e = result.getException();

      if (e != null) {
          ((AbstractCrmActivity) mActivity).handleException(e);
          return;
      }

      if (data.isMissingNode()) {
          return;
      }

      processData(data.path("data"));

      FragmentTransaction ft;
      for (Fragment f : mFragments) {
          ft = getFragmentManager().beginTransaction();
          ft.add(R.id.detail_view_main_layout, f);
          ft.commit();
      }

  }
}
```

Entity DetailFragment example

Concrete implementation of *DetailFragment* just needs to define abstract methods of *AbstractDetailFragment*. **processData** method defines the used *subfragments* for displaying the record's information.

```
@Override
JsonNode mainTaskBackgroundFn() throws CrmConnectionException,
                                       CrmConnectorException,
                                       OnlineDataConnectorException {
    return WebApiDataProvider.getPersonDetail(mRecordId);
}

@Override
void processData(JsonNode data) {
    PersonBasicInfoFragment pbif = new PersonBasicInfoFragment();
    pbif.setArguments(data);
    mFragments.add(pbif);

    PrimaryRelationshipFragment prf = new PrimaryRelationshipFragment();
    prf.setArguments(
            JsonUtils.getJsonNode(data,PersonDetailPath.PRIMARY_RELATIONSHIP));
    mFragments.add(prf);

    PersonContactsFragment pcf = new PersonContactsFragment();
    pcf.setArguments(data);
    mFragments.add(pcf);

}
```

Record information subfragments

All the record information *subfragments* are subclasses of *AbstractSubfragment* abstract class. *AbstractSubfragment* class defines several helping methods to generate different kinds of simple *Views* like *TextView*.

```
public TextView generateTextView(IEntityPath path){
    JsonNode node = JsonUtils.getJsonNode(mData, path);
    return generateTextView(node.isTextual()?node.asText():"");
}
```

Concrete implementation of *Subfragment* is responsible for displaying the records information. For this purpose also other *Fragments* or *Subfragments* can be used.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {

    View v = inflater.inflate(R.layout.default_subfragment_layout, null);
    mLayout = (LinearLayout) v.findViewById(R.id.subfragment_body);
    mLayout.setId(R.id.primary_relationship_fragment_mlayout);

    mLayout.addView(generateLargeTextView(RelationshipPath.COMPANY_NAME));
    String state =
        JsonUtils.getJsonNode(mData, RelationshipPath.COMPANY_STATE).asText();
    mLayout.addView(generateTextView(state));

    JsonNode tel = JsonUtils.getJsonNode(mData, RelationshipPath.COMPANY_TEL1);
    if (tel.isTextual()) {
        PhoneContactField pcf = new PhoneContactField();
        pcf.setArguments(tel.asText());

        getFragmentManager().beginTransaction().add(mLayout.getId(), pcf).commit();
    }

    return v;
}
```

ContactFields

ContactFields are special kind of simple *Fragments* that reacts on user action like click. All *ContactFields* are based on *AbstractContactField* abstract class extending *Fragment*.

AbstractContactField class defines the internal variables and also the abstract methods to obtain the icon drawable resource id and to handle the click action.

```
protected String mContactString;
protected TextView mView;

abstract protected int getIcon();
abstract protected void onContactClick(View v);
```

The basic *onClickAction* listener is defined.

```
protected OnClickListener mOnClickListener = new OnClickListener() {

    @Override
    public void onClick(View v) {
        onContactClick(v);
    }
};
```

In **onCreateView** method the basic behavior of *ContactField* and the information to display is set.

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {

    mView = new TextView(getActivity());
    mView.setText(mContactString);

    Drawable img = getActivity().getResources().getDrawable(getIcon());
    mView.setCompoundDrawablesWithIntrinsicBounds(img, null, null, null);
    mView.setCompoundDrawablePadding(10);
    mView.setTextAppearance(getActivity(), android.R.style.TextAppearance_Large);

    mView.setOnClickListener(mOnClickListener);

    return mView;
}
```

Particular implementation of *ContactField* is responsible mainly for defining the action performed on the click of *ContactField's* body. This can be simple action as opening the email application using intent with prefilled contact data like on example below.

```
@Override
protected void onContactClick(View v) {
    Intent intent =
          new Intent(Intent.ACTION_SENDTO, Uri.parse("mailto:"+mContactString));
    startActivity(intent);
}
```

HistoryFragment

HistoryFragment extends the *AbstractListFragment*, however, it changes lots of its functionality. For example there is no need for filtering, no showing of extra data records and opening the *DetaiView* of selected record depends on its entity.

The **onCreateView** method disables *onItemLongClick* listener of *Fragment's ListView* and calls **loadData** method with no filters or fulltext set.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                                                    Bundle savedInstanceState) {
    View v = super.onCreateView(inflater, container, savedInstanceState);
    mListView.setOnItemLongClickListener(null);
    loadData(null, null);

    return v;
}
```

taskBackgroundFn method calls the *WebApiDataProvider's* **getHistoryList** method for the current record.

```
@Override
JsonNode taskBackgroundFn(String... parameters) throws CrmConnectionException,
                          CrmConnectorException, OnlineDataConnectorException {
    return WebApiDataProvider.getHistoryList(mEntity, mRecordId, mLimit, mStart);
}
```

Record's view is rendered according to its entity. Different icons and data are displayed for different entities.

```
@Override
View mapItem(JsonNode data) {
    View view = LayoutInflater.from(mActivity).inflate(getItemLayout(), null);
    TextView titleTV = (TextView) view.findViewById(R.id.row_title);
    TextView subtitle1TV = (TextView) view.findViewById(R.id.row_subtitle);
    TextView subtitle2TV = (TextView) view.findViewById(R.id.row_subtitle_2);
    ImageView iconIV = (ImageView) view.findViewById(R.id.icon);

    String entityName = data.path("_entityName").asText();

    EntityEnum e = EntityEnum.enumMatch(entityName);

    switch (e) {
```

```
    case EMAIL:
    case EVENT:
    case MEETING:
    case PHONE_CALL:
    case LETTER:
    case TASK:
        titleTV.setText(
            JsonUtils.getJsonNode(data, CommonEntityPath.TITLE).asText());
        subtitle1TV.setText(
            JsonUtils.getJsonNode(data, CommonEntityPath.STATUS).asText());
        break;
    case RELATIONSHIP:
        titleTV.setText(
            JsonUtils.getJsonNode(data, CommonEntityPath.COMPANY_NAME).asText());
        subtitle1TV.setText(
            JsonUtils.getJsonNode(data, CommonEntityPath.TYPE).asText());
        break;
    default:
        break;

    }

    iconIV.setImageResource(e.getIconResource());
    subtitle2TV.setText(
            JsonUtils.getJsonNode(data, CommonEntityPath.CREATED_AT).asText());

    return view;
}
```

The list item click action is handled by **onListItemClick** method. Depending on the entity of selected item the according *DetailView* is opened.

```
@Override
public void onListItemClick(ListView lv, View v, int position, long id) {
    JsonNode item = mAdapter.getItem(position);

    Class<?> detailClass =
            EntityEnum.enumMatch(item.path("_entityName").asText()).getClass();

    if(detailClass == null){
        return;
    }

    int recordId = JsonUtils.getJsonNode(item, CommonEntityPath.ID).asInt();

    Intent detailIntent =
            new Intent(mActivity.getApplicationContext(), detailClass);
    detailIntent.putExtra("recordId", recordId);
    startActivity(detailIntent);
}
```

4.1.3 Login screens

Using the synchronization services brought the need of using the login screen separately from the application. It is possible to implement one login screen, that would meet the needs of both login scenarios, however, that would limit the flexibility of the login screen. On this point the author of the thesis decided to create two separated login *Activities*, both based on common login form *Fragment*.

Account authenticator login screen uses only the basic *login form Fragment* (*AccountLoginFragment*), while the applications login screen also uses the list of accounts that are already stored in device's *AccountManager* (*AccountListFragment*).

To allow login application to use the *Fragments*, both of the login activities implements interface *ICredentialsReceiver*, which contains method **receiveCredentials(Bundle credentials)** that is responsible for receiving and processing credentials obtained from used fragments.

Method **receiveCredentials** is called from the fragments after the defined action is executed. In the case of *AccountLoginFragment*, it is a click action of login button. **onClick** method first obtains the credentials filled by user to the login form's *TextViews* and store them in the *credentials Bundle*, which is then passed to the container activity's **receiveCredentials** method.

```
public void onClick(View view) {
       Bundle credentials = new Bundle();
       credentials.putString(Constants.CRM_ACCOUNT_USERNAME,
                                   tb_username.getText().toString());
       credentials.putString(Constants.CRM_ACCOUNT_INSTANCE,
                                   tb_instance.getText().toString());
       credentials.putString(Constants.CRM_ACCOUNT_PASSWORD,
                                   tb_password.getText().toString());
       ((ICredentialsReceiver)getActivity()).receiveCredentials(credentials);
}
```

In the case of *AccountListFragment*, the action is mapped to the click on the item of account list. *AccountListFragment* stores the list of application's *Accounts* that are obtained from *Account Manager* in fragments **onAttach** method, and when the click action is performed, *credentials Bundle* is obtained from the selected *Account* and passed to the container *Activity's* **receiveCredentials** method.

4.2 Accounts and Services

4.2.1 Obtaining and storing credentials in Account Manager

For the purpose of adding application's *Accounts* to the *AccountManager*, the author created the *CrmAccounAuthenticateService* according to an approach described in the chapter 3.3 - Using AccountManager.

addAccount method of inner *CrmAccountAuthenticator* class creates the *Bundle* that defines the *Activity* that should be used for obtaining the credentials from the user. In this case it is *LoginSimpleActivity*.

```
public Bundle addAccount(AccountAuthenticatorResponse response, String accountType,
       String authTokenType, String[] requiredFeatures, Bundle options)
       throws NetworkErrorException {

    Bundle reply = new Bundle();

    Intent i = new Intent(mContext, LoginSimpleActivity.class);
    i.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE, response);
    reply.putParcelable(AccountManager.KEY_INTENT, i);

    return reply;
}
```

Login *Activities* for *Account* authentication usually inherit from *AccountAuthenticatorActivity*, which is a helper class containing the methods for a correct respond to the account authenticator instance. However, this class is not capable of working with *Fragments*.

At this point the author decided to incorporate the *AccountAuthenticatorActivity* methods in own class - *LoginSimpleActivity*, that supports *Fragments*. The code involved in the class is identical with the code from *AccountAuthenticatorActivity*, which extends basic Android's *Activity*.

```
private AccountAuthenticatorResponse mAccountAuthenticatorResponse = null;
private Bundle mResultBundle = null;

public final void setAccountAuthenticatorResult(Bundle result) {
mResultBundle = result;
}

protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    mAccountAuthenticatorResponse = getIntent().getParcelableExtra(
                      AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE);

    if (mAccountAuthenticatorResponse != null) {
        mAccountAuthenticatorResponse.onRequestContinued();
    }
}

public void finish() {
    if (mAccountAuthenticatorResponse != null) {
        if (mResultBundle != null) {
            mAccountAuthenticatorResponse.onResult(mResultBundle);
        } else {
            mAccountAuthenticatorResponse.onError(
              AccountManager.ERROR_CODE_CANCELED, canceled");
        }
        mAccountAuthenticatorResponse = null;
    }
    super.finish();
}
```

LoginSimpleActivity contains *AccountLoginFragment* and for the purpose of obtaining the credentials it implements *ICredentialsReceiver* interface and overrides it is method **receiveCredentials**.

```
public void receiveCredentials(Bundle credentials) {
    LoginTask t = new LoginTask(this);
    t.execute(credentials);
}
```

When this method is called, it creates the instance of *LoginTask* (subclass of *AsyncTask*) and passes the *credentials Bundle* to it. **doInBackground** method of *LoginTask* checks validity of the credentials and then adds the *Account* to the account manager explicitly.

```
@Override
public Boolean doInBackground(Bundle... params) {

    Bundle credentials = params[0];

    if (checkCredentials(credentials)) {
        String accountName = String.format("%s / %s",
              credentials.getString(Constants.CRM_ACCOUNT_USERNAME),
              credentials.getString(Constants.CRM_ACCOUNT_INSTANCE));

        Account account = new Account(accountName, Constants.CRM_ACCOUNT_TYPE);
        AccountManager am = AccountManager.get(mContext);
        if (am.addAccountExplicitly(account, null, credentials)) {
            Bundle result = new Bundle();
            result.putString(AccountManager.KEY_ACCOUNT_NAME,
                                  account.name + " / instancename");
            result.putString(AccountManager.KEY_ACCOUNT_TYPE,
                                  account.type);
            setAccountAuthenticatorResult(result);
            return true;
        }
    }

    return false;
}
```

If the credentials are confirmed by a server, new Account is created and added to the account manager. When adding the Account using **addAccountExplicitly**, the password is set as null, and the credentials Bundle is used for storing the credentials. Password field is ready for possible future storing a local security PIN in future.

If the *Account* is successfully added, result Bundle is created, set as Activity's result and sent back to the *CrmAccountAuthenticator* in **finish** method. Otherwise the error is sent.

4.2.2 Contact synchronization service

Contact synchronization is processed by *CrmSyncContactsService*, a subclass of *Service*. The class implements *CrmSyncAdapter* extending the *AbstractThreadedSyncAdapter*.

```
private static class CrmSyncAdapter extends AbstractThreadedSyncAdapter {

    private Context mContext;

    public CrmSyncAdapter(Context context) {
        super(context, true);
        mContext = context;
    }

    @Override
    public void onPerformSync(Account account,
                              Bundle extras,
                              String authority,
                              ContentProviderClient provider,
                              SyncResult syncResult) {
        try {
            CrmSyncContactsService.performSync(mContext,
                                               account,
                                               extras,
                                               authority,
                                               provider,
                                               syncResult);
        } catch (OperationCanceledException e) {
        }
    }
}
```

This adapter is bind to the service in the **onBind** method.

```
@Override
public IBinder onBind(Intent intent) {
    IBinder ret = null;
    ret = getSyncAdapter().getSyncAdapterBinder();
    return ret;
}

private CrmSyncAdapter getSyncAdapter() {
    if (sSyncAdapter == null)
        sSyncAdapter = new CrmSyncAdapter(this);
    return sSyncAdapter;
}
```

Synchronization itself is performed by **performSync** method, which obtains the data from server using the RAYNET Cloud CRM REST API. When all the data are obtained, service checks if the group exists and starts to add the Person records into it using the **addPerson** binding method.

```
private static void performSync(Context context,
                                Account account,
                                Bundle extras,
                                String authority,
                                ContentProviderClient provider,
                                SyncResult syncResult
                               ) throws OperationCanceledException {
    android.os.Debug.waitForDebugger();
    mContentResolver = context.getContentResolver();

    AccountManager am = AccountManager.get(context);
    String username = am.getUserData(account, Constants.CRM_ACCOUNT_USERNAME);
    String password = am.getUserData(account, Constants.CRM_ACCOUNT_PASSWORD);
    String instance = am.getUserData(account, Constants.CRM_ACCOUNT_INSTANCE);

    AuthenticationProvider
              .getAuthenticationProvider()
              .setCredentials(username, password, instance);

    try {
        int p_start=0;
        int p_limit=25;
        int total_count = -1;

        List<JsonNode> responseList = new ArrayList<JsonNode>();
        while(total_count < 0 || p_start <= total_count){

            List<Object> paging = new ArrayList<Object>();
            paging.add(p_start);
            paging.add(p_limit);

            JsonNode n = RestApiDataProvider.getPersonList(paging);
            if(total_count == -1){
                total_count = n.path("totalCount").asInt();
            }
            responseList.add(n);
            p_start += p_limit;
        }

        long personGroupId = ensureSampleGroupExists(context, account, "Persons");

        for(JsonNode personList : responseList){
          for (JsonNode person : personList.path("data")) {
            ArrayList<ContentProviderOperation> operationList =
                                new ArrayList<ContentProviderOperation>();
            operationList.addAll(addPersonContact(account,person,personGroupId));
            mContentResolver.applyBatch(ContactsContract.AUTHORITY, operationList);
          }
        }

    } catch (Exception e) {
        throw new OperationCanceledException(e);
    }
}
```

4.3 Authentication

Credentials of currently logged user are stored in the singleton class *AuthenticationProvider*.

Class defines several private variables. *currentInstance* contains the currently used *AuthenticationProvider* instance. *instanceName*, *j_username* and *j_password* contains actual user's credentials. *base64Authentication* is *Base64* encoded credentials string used for authorizing application against REST API. *isSet* indicates if the credentials were set to the *AuthenticationProvider* instance.

```
private static AuthenticationProvider currentInstance;

private String instanceName;
private String j_username;
private String j_password;

private String base64Authentication;

private boolean isSet = false;
```

All of the above mentioned private variables have primitive public getters, except for *currentInstance*. This variable is accessible via **getAuthenticatoProvider** method, which creates new instance in the case that *currentInstance* hasn't been initialized yet.

```
public static AuthenticationProvider getAuthenticationProvider() {
    if (currentInstance == null) {
        currentInstance = new AuthenticationProvider();
    }
    return currentInstance;
}
```

Credentials are set to the *AuthenticationProvider* using the **setCredentials** method. When setting the credentials are being set, also the *base64Authentication* string is generated.

```
public void setCredentials(String username, String password, String instanceName){
    this.instanceName = instanceName;
    this.j_username = username;
    this.j_password = password;
    this.isSet = true;
    this.base64Authentication = Base64.encodeToString(
                                        (username + ":" + password).getBytes(),
                                        Base64.NO_WRAP);
}
```

4.4 Obtaining data

Data obtaining workflow is processed by two classes. Data provider class serves as a gateway for obtaining the data from *CrmConnector* class, which realizes the http request execution.

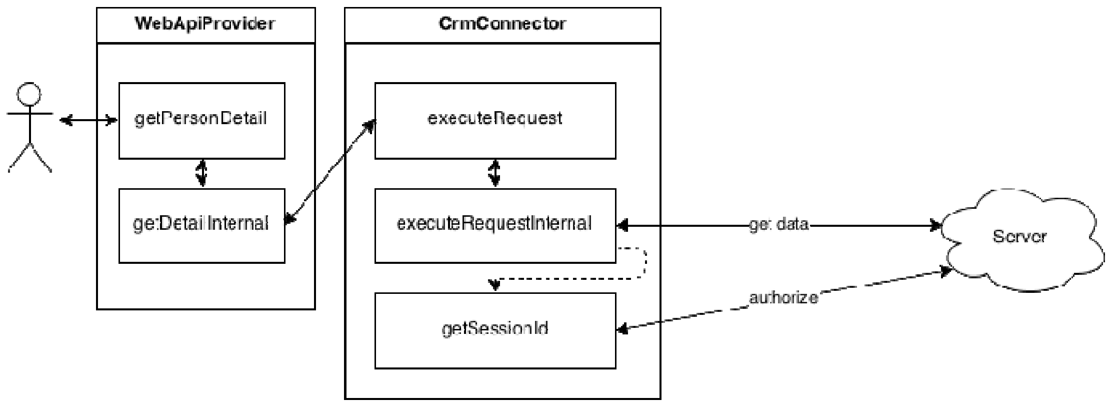


FIGURE 7 - Data obtaining workflow

UX component can request data from *data provider* passing appropriate data to responsible provider method - e.g. calling the **getPersonDetail** from *WebApiProvider*.

```
WebApiDataProvider.getPersonDetail(mRecordId);
```

getPersonDetail encapsulates the calling of **getDetailInternal** method common for most of the standard detail requests.

```
public static JsonNode getPersonDetail(Long id) throws CrmConnectionException,
                                                    CrmConnectorException,
                                                    OnlineDataConnectorException {
    return getDetailInternal(null, EntityEnum.PERSON.getString(), id, null);
}
```

getDetailInternal prepares the JSON serialization of *SearchCommand* helper objects, calls the **executeRequest** from *CrmConnector* to obtain the data and tries to parse the JSON response into the *JsonNode* object from Jackson library.

```
private static JsonNode getDetailInternal(String url,
                                          String entityName,
                                          Long id,
                                          SearchCommand searchCommand
                                         ) throws CrmConnectionException,
                                                  CrmConnectorException,
                                                  OnlineDataConnectorException  {

    Writer sw = new StringWriter();
    try {
        fObjectMapper.writeValue(sw, searchCommand);
    } catch (Exception e) {

    }
    String searchCommandJSON = sw.toString();
    String requestUrl = (url != null) ? url : CrmUrl.DETAIL_SPECIAL_URL.getUrl();

    String response =
      CrmConnector.executeRequest(requestUrl, entityName, id, searchCommandJSON);

    JsonNode rootNode;
    try {
        rootNode = fObjectMapper.readTree(response);
    } catch (Exception e) {
        throw new OnlineDataConnectorException(e);
    }

    return rootNode;
}
```

Public **executeRequest** of *CrmConnector* prepares the correct request URL and calls the internal method **executeRequestInternal** to obtain the data.

```
public static String executeRequest(String requestUrl,
                                    String entityName,
                                    Long id,
                                    String requestPayload
```

```
                                ) throws CrmConnectionException,
                                         CrmConnectorException {

    List<Object> args = new ArrayList<Object>();
    if (entityName != null) {
        args.add(entityName);
    }
    if (id != null) {
        args.add(id);
    }

    AuthenticationProvider ap = AuthenticationProvider.getAuthenticationProvider();
    String responseText = executeRequestInternal(formatUrl(
                                                  requestUrl,
                                                  args,
                                                  ap.getInstanceName()),
                                                  requestPayload);
    return responseText;
}
```

executeRequestInternal executes the data request itself. *HttpPost* object is prepared and executed by *DefaultHttpClient* instance. In the case that the server declined the request, **getSessionId** method is called to authenticate request and obtain session id from server, and then repeats the request.

```
synchronized private static String executeRequestInternal(String requestUrl,
                                                          String requestPayload
                                                  ) throws CrmConnectionException,
                                                           CrmConnectorException {

    HttpPost request = new HttpPost(requestUrl);
    String responseText = null;

    if (requestPayload != null) {
        StringEntity se = null;

        try {
            se = new StringEntity(requestPayload);
        } catch (UnsupportedEncodingException e) {
            throw new CrmConnectorException(e);
        }
        se.setContentType(new BasicHeader(HTTP.CONTENT_TYPE, "application/json"));
        request.setEntity(se);
    }

    StringBuilder builder = new StringBuilder();

    HttpResponse response;
    try {
        response = fHttpClient.execute(request);
    } catch (Exception e) {
        throw new CrmConnectorException(e);
    }
    StatusLine statusLine = response.getStatusLine();
    int statusCode = statusLine.getStatusCode();

    switch (statusCode) {
    case 200:
        if (response.getHeaders("X-RAYNETCRM-Login").length > 0) {
            try {
                response.getEntity().consumeContent();
            } catch (IOException e) {
                throw new CrmConnectorException(e);
            }
            getSessionId();
            return executeRequestInternal(requestUrl, requestPayload);
        } else {
            HttpEntity entity = response.getEntity();
            InputStream content;
            try {
                content = entity.getContent();
            } catch (Exception e) {
                throw new CrmConnectorException(e);
            }

            BufferedReader reader =
                            new BufferedReader(new InputStreamReader(content));
            String line;
            try {
                while ((line = reader.readLine()) != null) {
                    builder.append(line);
                }
            } catch (IOException e) {
                throw new CrmConnectorException(e);
            }
            responseText = builder.toString();
        }
        break;
    default:
        throw new CrmConnectionException(statusCode);
    }

    return responseText;
}
```

getSessionId method performs an authentication check against the server. If the authentication process fails at any point, the appropriate exception is thrown out.

```
synchronized private static void getSessionId() throws CrmConnectorException,
                                                       CrmConnectionException {

    AuthenticationProvider ap = AuthenticationProvider.getAuthenticationProvider();

    HttpPost request = new HttpPost(
                        formatUrl(CrmUrl.J_SPRING_SECURITY_CHECK.getUrl(),
                        null,
                        ap.getInstanceName()));

    List<BasicNameValuePair> formparams = new ArrayList<BasicNameValuePair>();
    formparams.add(new BasicNameValuePair("j_username", ap.getUsername()));
    formparams.add(new BasicNameValuePair("j_password", ap.getPassword()));
    UrlEncodedFormEntity postEntity = null;
    try {
        postEntity = new UrlEncodedFormEntity(formparams, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        throw new CrmConnectorException(e);
    }

    request.setEntity(postEntity);

    StringBuilder builder = new StringBuilder();
    HttpResponse response;
    try {
        response = fHttpClient.execute(request);
    } catch (Exception e) {
        throw new CrmConnectorException(e);
    }
    StatusLine statusLine = response.getStatusLine();
    int statusCode = statusLine.getStatusCode();

    switch (statusCode) {
    case 200:

        HttpEntity entity = response.getEntity();
        InputStream content;
        try {
            content = entity.getContent();
        } catch (Exception e) {
            throw new CrmConnectorException(e);
        }
        BufferedReader reader = new BufferedReader(new InputStreamReader(content));
        String line;
        try {
            while ((line = reader.readLine()) != null) {
                builder.append(line);
            }
        } catch (IOException e) {
            throw new CrmConnectorException(e);
        }
        String responseText = builder.toString();
        responseText.toString();
        if (response.getHeaders("X-RAYNETCRM-Login").length > 0) {
            throw new CrmConnectionException(401);
        };
        break;
    default:
        throw new CrmConnectionException(statusCode);
    }
}
```

5 CONCLUSION

This thesis describes the development of an Android client application and successfully demonstrates the planned functionality on a working prototype. The problems that occurred during the prototype application can be separated into two categories.

The first category contains the problems caused by the attempt to make application compatible with older versions of Android operating system. Most of these compatibility problems were solved by using the ActionBarSherlock library, however, in several cases the compatibility issues cannot be solved or bypassed without reducing or modifying the functionality for older versions of Android
- e.g. synchronization with Google calendar.

The second category of issues is limited by the current server side implementation of RAYNET Cloud CRM. These limits influenced mostly the authentication process and synchronization services. These issues are currently bypassed, however, they could be solved in a slightly more elegant way with some minor changes on the server-side.

The fact that it is not possible to obtain the information about user using the RAYNET Cloud CRM standard web API led to the need to use the RAYNET Cloud CRM REST API for this single request. This problem can be resolved by creating the server controller that returns the user information on standard security check request.

Using the synchronization services, application have to use the RAYNET Cloud CRM REST API to obtain the data because of security checks on RAYNET Cloud CRM standard web API. Using the standard web API, users can be logged in using only one session id. In the case of synchronization services that request data automatically, this could lead to logging off users from the desktop computer while working in the system.

Even though RAYNET Cloud CRM REST API is able to provide the most important data for synchronization services, it is limited in many ways - e.g. it is not possible to download images using it. This fact is not essential, but it can significantly improve user experience.

This issue can be solved by allowing the mobile client to connect to the server using different TCP port than web client does.

The following development of the application will focus on extending the described functionality and UX tuning.

REFERENCES

ActionBarSherlock. Referenced 2013-03-19 from ActionBarSherlock: http://actionbarsherlock.com/

Content Provider Basics. Referenced 2013-03-15 from Android Developer: http://developer.android.com/guide/topics/providers/content-provider-basics.html

Content Providers. Referenced 2013-03-15 from Android Developer: http://developer.android.com/guide/topics/providers/content-providers.html

Creating a custom Account Type. Referenced 2013-03-15 from Android Developer: http://developer.android.com/training/id-auth/custom_auth.html

Garrett, J. (2005) *Ajax: A New Approach to Web Applications*. Referenced 2013-03-25 from Adaptive Path:
http://www.adaptivepath.com/ideas/essays/archives/000385.php.

Pure Android. Referenced 2013-03-10 from Android Developer: http://developer.android.com/design/patterns/pure-android.html

SlidingMenu. Referenced 2013-03-19 from jfenstein10 SlidingMenu: https://github.com/jfeinstein10/SlidingMenu

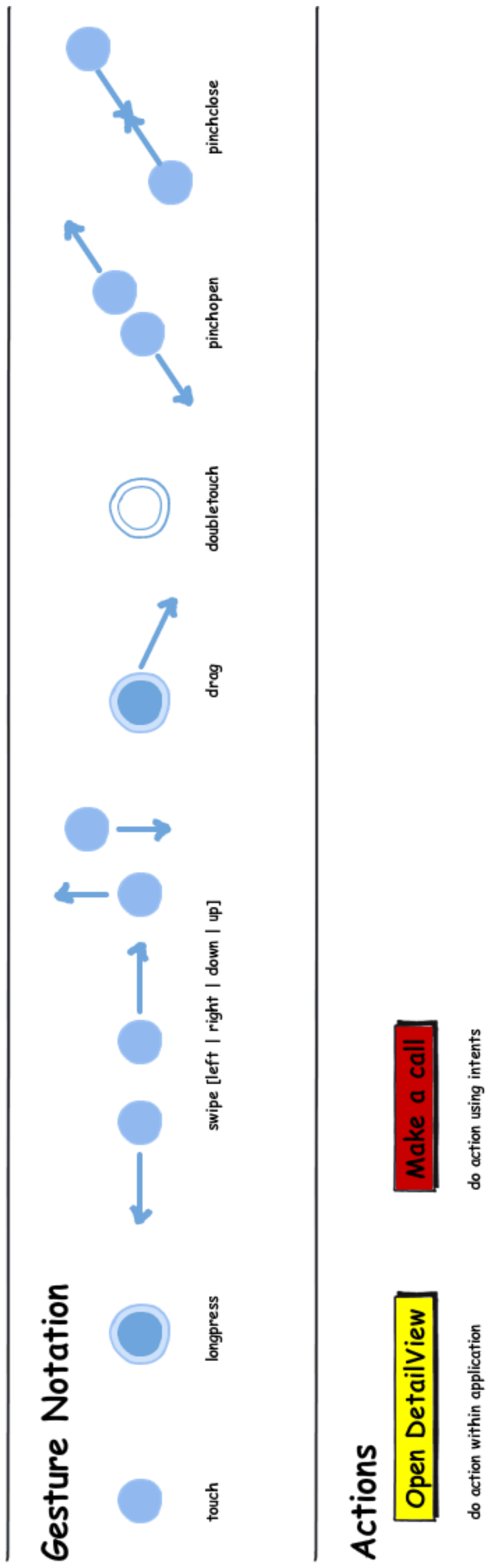
Stříž, M. (2011) *Platform for Rich Internet Applications development*. Master's thesis FIT VUT Brno

Vogel, L. (2013) *Android Intents - Tutorial*. Referenced 2013-03-18 from Vogella: http://www.vogella.com/articles/AndroidIntent/article.html

Writing an Android Sync Provider. Referenced 2013-03-15 from Did You Win Yet: http://www.c99.org/2010/01/23/writing-an-android-sync-provider-part-2/

APPENDICES

Gestures notation and actions


Gesture Notation
touch
longpress
swipe [left | right | down | up]
drag
doubletouch
pinchopen
pinchclose
Actions
Open DetailView
do action within application
Make a call
do action using intents