AGH University of Science and Technology

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

Department of Telecommunications

Master's Thesis

Name and Surname	Grzegorz Gajewski, Jan Francisco Sanchez Duduś
Field of Study	Electronics and Telecommunications
Title	Mobile platform for smart shopping on the basis of Web 2.0.
Supervisor	Jarosław Bułat, PhD

Krakow, 2010

We hereby declare, aware of bearing the criminal responsibility, that this thesis is entirely the result of our own work, as stated in the introduction, except where otherwise indicated by references to other authors.

…........................ …........................

AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I ELEKTRONIKI

KATEDRA TELEKOMUNIKACJI

Praca dyplomowa
magisterska

Imię i nazwisko	Grzegorz Gajewski, Jan Sanchez Duduś
Kierunek studiów	Elektronika i Telekomunikacja
Temat pracy dyplomowej	Platforma mobilna do porównania cen w oparciu o Web 2.0.
Opiekun pracy	dr inż Jarosław Bułat

Kraków, rok 2010

Oświadczamy, świadomi odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonaliśmy osobiście i samodzielnie (w zakresie wyszczególnionym we wstępie) i że nie korzystaliśmy ze źródeł innych niż wymienione w pracy

…..........................   …..........................

Acknowledgement

I want to start by thanking my thesis supervisor, Jarosław Bułat, PhD, for your support and guidance throughout my work on this thesis.

To my mother, although you are not anymore in this world I will always keep you in my heart.

To my father, for all your support, love and wisdom. It would not have been possible for me to study in Poland without your help. If today, I am finishing my thesis is because of you.

To my sister, for always being there for me and for being so caring. Your encouragement always has helped me to go on, especially while doing my thesis. You are the best sister that one could have.

To my friend, Greg. I could not ask to have a better partner than you for doing this thesis.

To all my other friends, thanks for your friendship.

Jan Sanchez Duduś

Introduction

Barcodes appear on all products, from grocery products to electronic products. Different services could be built targeting barcodes and users. The main purpose of this project is to allow users to compare products between different stores.

Nowadays almost every person has his own cellphone and the use of smartphones is growing amongst phone users. For example only in the U.S. 53.4 million people owned smartphones during the three months ending in July, up 11 percent from the corresponding April period [1]. Also in the European Union (only taking into account U.K., France, Germany, Spain, Italy) as of July 2010 there were 60.8 million smartphone users, this number has grown 41 percent since July 2009 [2]. Targeting smartphone users with a service they could use at any time, anywhere would be ideal. That is why this project targets a certain group of smartphone users so they could compare different products everywhere.

ShopDroid was chosen as the name of the service for two reasons, one is that the main purpose of this is to use while users are shopping. The other reason was the choice of platform, Android.

The objectives and the scope of the ShopDroid project were:

- To develop a client-server architecture for price comparisons based on barcodes
- The system has to work on the basis of Web 2.0, with the content being mainly created by users, the content is composed of the prices, geolocation, reviews and other parameters related to a specific product
- The client has to be developed for smartphones that have a camera and access to the Internet (i.e. GPRS, EDGE, etc.)
- The server will act as a product database together with its metadata
- The system has to provide security, user identification and user credibility rating in order that the platform will still be useful even if malicious users to fake some data (like making a product undesirable)

- The final result of the thesis will be a working platform and a client developed for a popular, mobile operating system such as Symbian, iPhone, Android or Windows Mobile

At the time of designing ShopDroid there was only one similar solution available: *ShopSavvy*. There are many differences between these two projects. First of all, ShopSavvy does not work really well in Poland, due to not having a product and store database for this country. Secondly, ShopSavvy has some features that ShopDroid lacks, such as price alerts, however ShopDroid has a store locator feature that is not present in ShopSavvy. Finally, ShopDroid was designed to engage the users and therefore users can register an account, can add new products or new prices, can even add stores and product reviews. ShopSavvy lacks this social feature.

Another similar application that appeared later, was the Amazon application. The main differences are that Amazon does the barcode scanning in the server, and this takes longer. Also, that application is only for products sold by Amazon.

This thesis is divided into two main chapters:

- **Chapter 1 -** *System Overview:* provides a general description about the system and its components
- **Chapter 2 -** *Tools used to build ShopDroid*: describes which technologies and tools were used for this project, also about the platform choice
- **Chapter 3 -** *ShopDroid Project*: describes how does the application work, and how it was implemented, as well as all its features are described here. The process of barcode scanning is also explained.

Grzegorz Gajewski developed the following components:

- server functionality of the system
- barcode scanning in the client
- users authorization

Jan Sanchez developed the following components:

- manual product search
- store locator
- user's history

Chapter 1
System Overview

This chapter contains a general description of the system and its components. Also, the communication between the client and the server is described.

1.1 Objectives and Scope of ShopDroid

In general, the purpose this system is to allow users to scan barcodes and with that information be able to compare prices of products from different stores. This would be beneficial for users since they could save money using this system.

The main objective of this project was to make a client-server application. The client part should be an application for mobile devices, which would allow users to scan barcodes and be able to compare prices of different products. Barcodes should be detected automatically using a built-in camera.

The system would be user-oriented, with the users creating most of the content, like adding or changing product prices, and adding new stores. Besides being able to compare prices, users should be able to share their opinions about any product. This system should be safe from data vandalism such as adding fake data about products or prices, like in the case of a store wanting to attract more customers by editing other stores prices to be more expensive.

Users should be required to register an account with the service in order to be able to interact with the system. A points rating system should be implemented to track, which are the good users and which are malicious ones. There should be also the possibility for the users to report any wrong information or spam from other users. Also, a secure model should be implemented for user registration in order to avoid the stealing of user credentials.

The application was targeted to mobile devices because these devices have nowadays access to the Internet almost everywhere, so users would be able to check prices while shopping at any store. Also, thanks to the fact that almost every device is GPS (Global Positioning System) enabled, or have other geolocation features, the system can offer services that will be based on the user's location. That means if users are in Krakow, Poland they will only get information about prices in stores near their location. This would be helpful for users since they could see that a product they are buying is cheaper in some other store nearby.

The target country of this project is Poland, since the application is being developed in this country. However, in the future support for other countries could be added as long as enough products' data from those countries is obtained.

1.2 Use cases

Five use cases were considered when developing this project. For describing them it should be noted that the actors are any of the users of the system, with the exception of the administrator of the system.

The user is in a store and checking a product. Then the user scans the product barcode. The system will show the user the prices in nearby stores. Finally, the user decides whether to buy the product at the store the user is currently in, or in a nearby store if the product is cheaper.

Another case would be if the user is at home, and is thinking of buying a product. The user could either scan the product barcode (if the user has it already) or search manually the barcode or the name of the product. The system will return the prices of the product in nearby stores. Finally, the user will decide whether to buy the product or not, and if buying the product in which store.

The third user case is as follows: The user is searching for stores nearby. The system will show the user in a map where are the nearest stores located and the names of the stores. Then the user could decide a store where to go and the system should show the way to the user.

A fourth case would be when the user has already in the past searched for any products, or scanned barcodes, and the user wants to check again for prices for that product without scanning the barcode or searching again. They system will show the user the previously searched products and the user will select one of them. At the end, the system will show the user the prices of that product in nearby stores.

A fifth case would be when the user has used a product and wants to give some feedback about it. The user will find the product. The system will show the user the product information. The user will be able to add any opinion on the product. This opinion will be updated to the system and finally the user will be able to see the recently added opinion, and other users' opinions on the product.

1.3 System components

In the Chapter 2 the software components needed for the implementation will be described in details.

One of the main components of the system is the barcode scanning. This is a client component, and it is described in section 3.2. Locating nearby stores and showing them to the user requires the client to get from the server the information about the stores. Description about this component can be found on section 3.6.

Keeping track of the user's history is only done in the client. This is described in section 3.4. Another main component is showing the product data, like the name, a picture of it, prices in different stores as well as reviews if any. The client will provide the server with the barcode of the product and the server will return all the product data to the client. The client will then show this information to the user. This component is described in section 3.7.

If a product is not available then user is able to manually search by the product name, or the barcode if known. The way how this is implemented is described in section 3.5. Having a secure model to save the user's credentials is very important. This is implemented in the client as well as in the server, with the server holding the users data. This is described in more details in section 3.8. Also information about how the data is kept in the server is described in section 3.10.

1.4 Client-Server Communication

In order to minimize the server's load, the client needs to communicate with the server as little as possible, and when communicating with the server, the client has to get enough information for displaying to the user. All the queries from the client to the server are made using HTTP (Hypertext Transfer Protocol). The responses from the server are in XML (Extensible Markup Language).

The client will ask the server for a barcode, and the server will return the product information and records of several prices of the product in different stores. Information about the stores is returned as well, in order to avoid further queries from the client if the user chooses to view details about a certain store.

When searching products by name, the client will send a query to the server with the searched product name, and the server will return a list of products that match the product. If the user will select a product from the list, then the client will send a query to the server with the barcode of the selected product and the process described in the previous paragraph will take place.

Also when searching stores, the client will send to the server information about the user's location and the server will return the stores with information such as their coordinates in order to show them in a map. More detailed information is included in the response, since the user might want to check more details about a specific store. The number of stores returned should be limited to a certain radius around the user.

Finally when adding a new product, adding a new price, editing a existing price or adding a new store the client uploads this information to the server. Next time when client requests this kind of information, the new or updated data should already be shown.

Chapter 2
Tools used to build ShopDroid

2.1 Platform choice

Before starting this project, there was still a decision to be made, which mobile platform for a smartphone to choose. Four platforms were taken into consideration:

- Symbian
- Android
- IPhone OS (IOS)
- Windows Mobile

Windows Mobile had lots of different versions, which introduced compatibility issues. At the time there were also rumors that the platform was dying, and Microsoft releasing a completely new platform - Windows Phone 7 - confirmed this not long ago, which is not going to be backwards compatible at all.

One of the main reasons not to consider developing for the iPhone is that the iPhone SDK did not support at the time real time access to each single frame captured by the camera. Also at that time, Apple's developer's agreement, did not allow any open source applications, which make it hard to look for good material. The last reason was Apple's tight control on what is released on their App Store, which sometimes takes even several weeks or months to get an application approved.

Symbian was a good platform to develop for, but again in the time, it was getting new versions to the platform, and also there were plans to open source the platform (which came true), so there were doubts on the future of the platform and that is the main reason why this platform was not chosen. Also, having already developed for

Symbian during a course in the university made us want to develop for a completely new platform in order to gain new knowledge.

Android was the platform choice (Android logo is shown in Figure 2.1). There were several reasons for this choice. One of them was that the platform allows direct access to different hardware elements of the device, and in the case for this project; there was a need for real time access to each frame captured by the camera. The open nature of Android, being an open source project licensed mostly under Apache Software License [3] and based upon the Linux kernel too, made it a very good choice. Another reason was that there were plenty of materials and source examples and that publishing an application in the Android Market is easy and fast.

Figure 2.1 Android Logo[1]

The requirements for the device makers that want to have Android were also a reason. Since Android requires all devices to have a camera with auto-focus (which is needed for barcode scanning), to be GPS enabled (useful for geolocation features) and have access to the Internet.

At the time of choosing the platform Android had a quite brilliant future and it is confirmed now by seeing how fast has its share grown. Information technology research and advisory company Gartner forecasts that Android will be in the second place in worldwide market share by the end of 2010 with 17.7% only behind Symbian. They also predict that Android might challenge Symbian's top position by 2014 [4]. This analysis was made in August, 2010. Figure 2.2 shows these results.

[1] http://en.wikipedia.org/wiki/File:Android_logo.svg

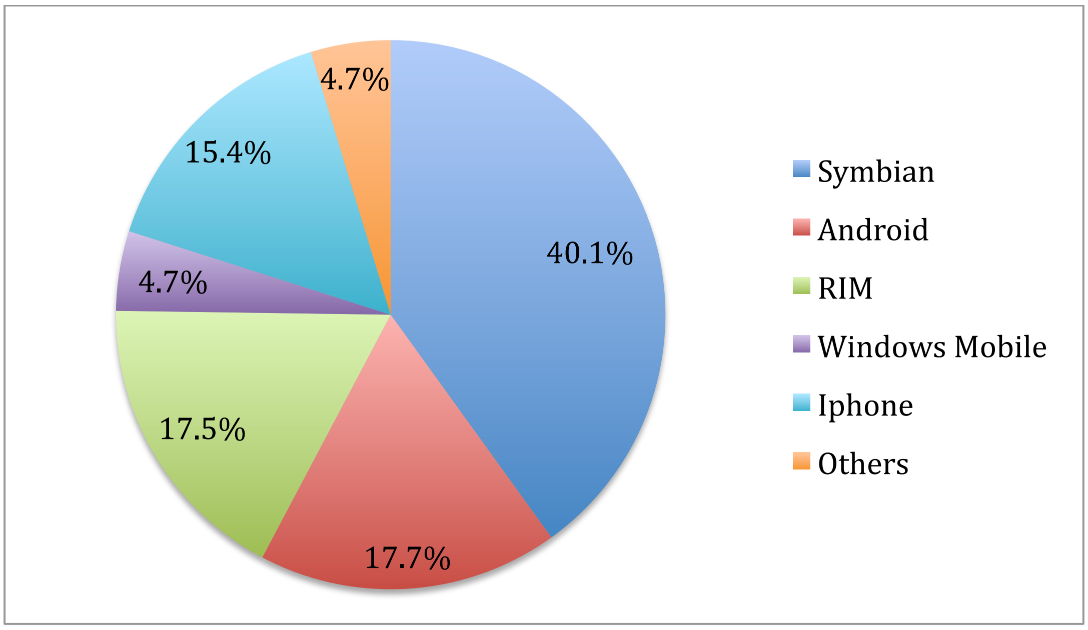


Figure 2.2 Mobile OS market share by end of 2010 by Gartner

2.2 Android Operating System

Writing Android applications is done in Java, since the platform itself runs on top of Java core libraries. However there is no Java Virtual Machine. Android has its own virtual machine called Dalvik. Developers write the application in Java, this code is compiled into Java bytecode (class file) and later recompiled into Dalvik bytecode, which can only be executed on a Dalvik Virtual Machine [5].

One interesting feature of the Android platform is the use of *Intents*. To understand them one should know the concept of *Activities* in Android. From the Android documentation:

An activity presents a visual user interface for one focused endeavor the user can undertake [6].

In general every screen in Android is an activity. Each activity has its own window to draw in. An application can be composed of one or more activities. These activities are activated through *intents*. From the documentation:

An Intent object, is a passive data structure holding an abstract description of an operation to be performed [7].

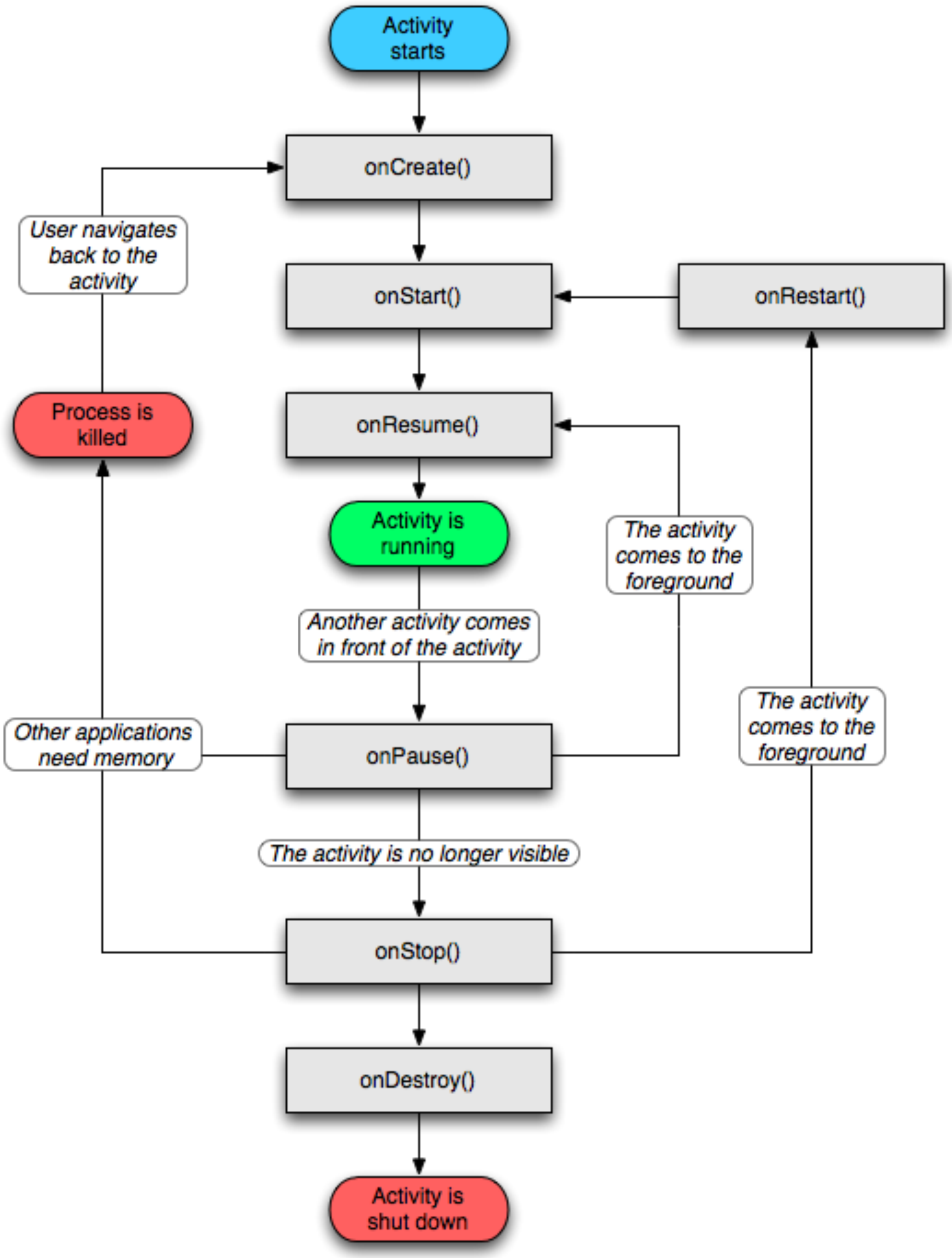
This allows a lot of flexibility in Android and makes it easy to share activities between applications. For instance it is easy to call the Android default image chooser or camera application to get a picture to the application being developed. Another example is when the application wants to show some directions on a map, or when a video needs to be opened. This activities are started one after the other so the user will not know that some parts are taken from another application. It all seems to him that all is run on the same application.

Android also provides a very good concept of the application life cycle. It is more correct to talk about activity life cycle, since during the life of an application; many activities might have been started and stopped. Figure 2.3 is presented to illustrate better this activity life cycle.

All this methods can be overridden to perform certain tasks. This is useful for certain applications that want to save some information when the application is being sent to the background (i.e. when the user opens another application), or when the application is being killed (i.e. the system might kill any application whenever it needs resources).

Developers publish their applications in the Android Market in order to distribute them. Almost all Android devices have access to this Market. The process is straightforward, first a developer account is needed (with a fee of 25 dollars), and then it is possible to publish any application. There is no review process for the applications published, and they appear almost instantaneously to the users. There are some limitations to this distribution model in the case of paid applications, since buying applications it is only available in a handful of countries.

An important part of any Android application is the *AndroidManifest.xml* file. This file, which all applications must have in their root directory, serves to control the application. You can specify which kind of permissions does the application need, like Internet permissions or camera permissions. These features will not work if not set in this Manifest file. This also allows the user to know which features does the application need, before installing it. The Android Manifest also allows changing the application's name and icon and setting the application's theme or background. It is possible to specify in the Manifest what kind of requirements are needed (like the screen size or accelerometer), with this, the application will only appear in the Android Market for the devices that meet this requirements. This Manifest file has many more functionalities that can be seen in the official documentation [8].



Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

Figure 2.3 Activity life cycle[1]

[1] http://d.android.com/images/activity_lifecycle.png found at http://d.android.com/guide/topics/fundamentals.html#lcycles

Localization is very well implemented in Android. String resources are stored into an xml file called *strings.xml* inside the *values* folder. These are the default string resources, and English is the default language. To add another languages, a new folder needs to be created by adding the appropriate suffix to *values*. For instance, for Polish, the folder *values-pl* needs to be created. Each of these folders needs to have their own strings file. Android will automatically load the proper language depending on the user's language settings. If the application does not have any string resources for the user's language, then it will fall back to the default one. Strings are defined as the following:

```
<string name="history">History</string>
```

This string is easily used in the application by calling *R.strings.history*. ShopDroid has implemented three languages: English, Polish and Spanish.

Android is available in different devices, which have different screen sizes and resolutions, and therefore different densities (spread of pixels across the height and width of the screen in dots per inch). The platform provides a way to deal with different screen sizes. When defining elements in the user interface, it is encouraged to use density independent pixels instead of pixels. This allows the platform to auto-scale the elements to different screen sizes. The conversion from dips to pixels is shown in (2.1).

$$dips = pixels \times \frac{160}{density} \tag{2.1}$$

where *density* - the density of the device's screen in dpi (dots per inch), *dips* - density independent pixels

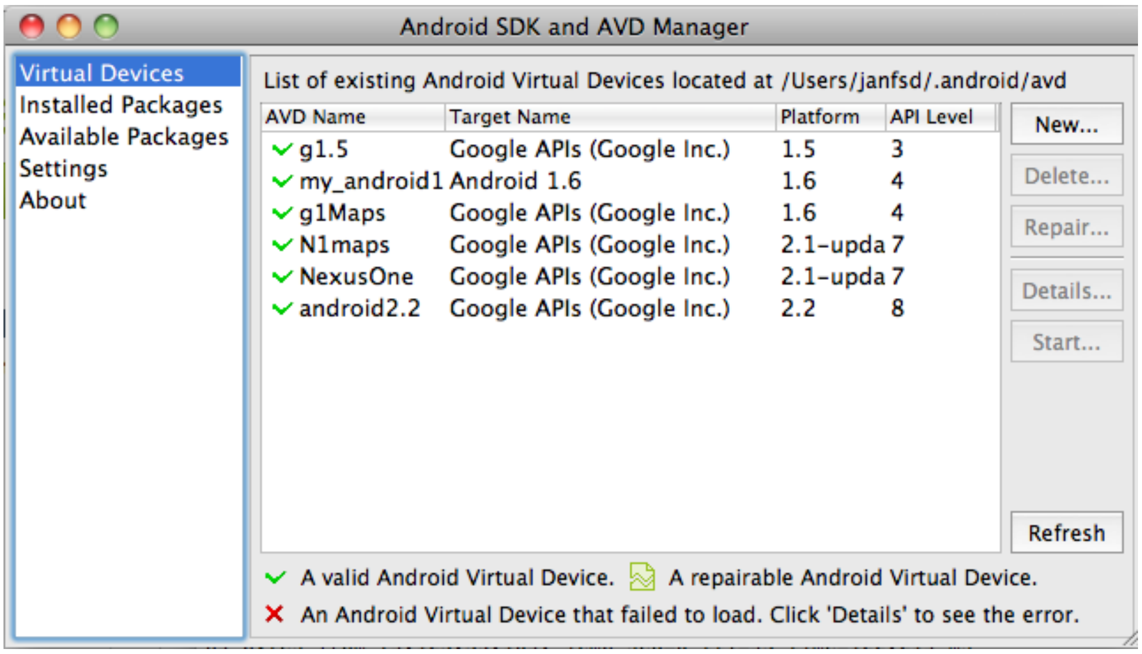
While this is quite handy for applications that implement default user interface controls, other applications that have their own elements, such as games, might not be able to use this option. Developers have also the option to restrict their application to one screen size type of devices, when publishing to the Android Market.

Android also allows Google Maps to integrate into the application; this makes the application more feature-rich. Other elements like the Web browser can be integrated also and provide additional possibilities for the application being developed.

2.2 Android Software Development Kit

Android SDK is a set of tools that are needed for development of Android mobile applications. There are also additional tools that while they are not essential for Android development, they make it easier.

First, there is the Android SDK and AVD (Android Virtual Devices) Manager, shown in Figure 2.4. It is used mainly for installing the different platform packages that exist for Android. This package includes the needed resources, compilers, and emulators. ShopDroid was successfully tested in all available platforms. To the date, there are five versions (1.5, 1.6, 2.0, 2.1, 2.2), while version 1.5 has been slowly disappearing due to manufacturer upgrades. Also, version 2.0 already disappeared because of version 2.1 being an update for it. Figure 2.5 shows how different versions have changed over time.



List of existing Android Virtual Devices located at /Users/janfsd/.android/avd

AVD Name	Target Name	Platform	API Level
✓ g1.5	Google APIs (Google Inc.)	1.5	3
✓ my_android1	Android 1.6	1.6	4
✓ g1Maps	Google APIs (Google Inc.)	1.6	4
✓ N1maps	Google APIs (Google Inc.)	2.1-upda	7
✓ NexusOne	Google APIs (Google Inc.)	2.1-upda	7
✓ android2.2	Google APIs (Google Inc.)	2.2	8

✓ A valid Android Virtual Device. A repairable Android Virtual Device.

✗ An Android Virtual Device that failed to load. Click 'Details' to see the error.

Figure 2.4 Screenshot of Android SDK and AVD Manager

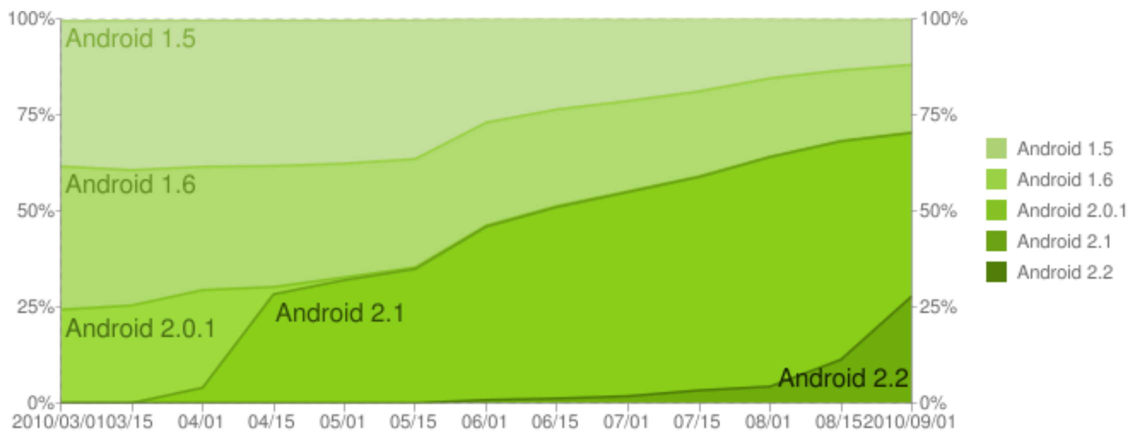


Figure 2.5 Historical Android platform distribution as of 16/09/2010[1]

Furthermore, this manager is useful for handling the emulators, in other words, it creates any emulator from a desired platform version, it can start emulators and it can delete them if wanted.

Android officially supports the Eclipse IDE for developing by providing a plugin for it. Even though it is possible to develop Android applications using other IDEs, Eclipse is the preferred way. This plugin nicely integrates with the manager and debugging tools for Android. A rusty interface editor is also included, but since it is limited in usefulness, it was not used for this project.

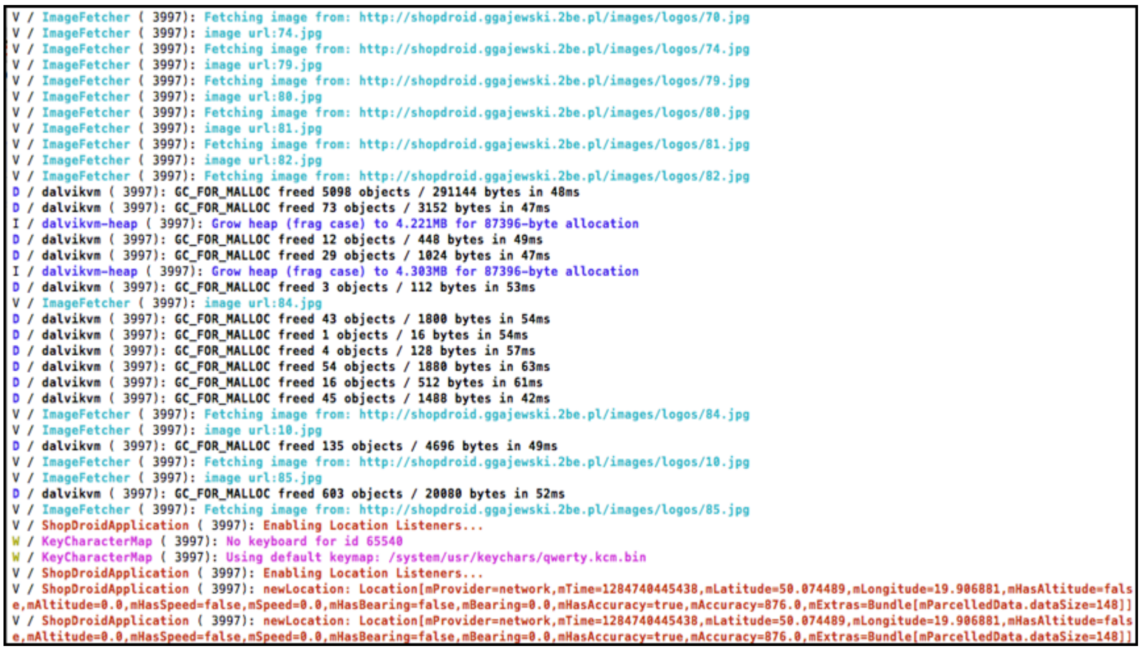
Logcat is probably the most useful tool; it dumps a log of all system messages, allowing seeing any exceptions or problems with the application. Inside the application it is possible to add manually logging code so it will get dumped too. Figure 2.6 shows Logcat in action.

Dalvik Debug Monitor Server (DDMS) is another useful tool. It was used mainly to take screenshots of the application, and to mock the location coordinates in the emulator.

Hierarchy Viewer is a tool that debugs the user interface. It shows in a more graphic way, how the structure of the interface was designed. With it, it is possible to get an idea on how some applications have implemented their user interface. Figure 2.7 shows this tool.

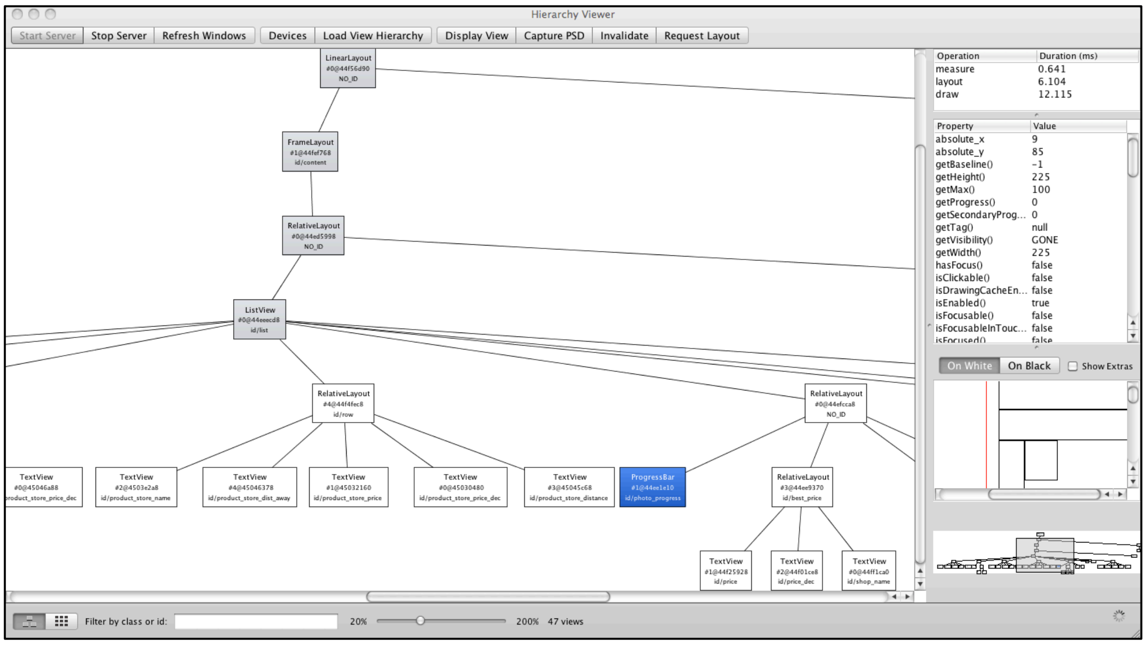
A testing tool called *Monkey* was also intensively used. This testing tool tries to stress out the application by simulating many taps on the screen what might bring the application down. If the application being developed passes this test, it means that it has good performance.

[1] http://developer.android.com/intl/zh-CN/resources/dashboard/platform-versions.html



```
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/70.jpg
V / ImageFetcher ( 3997): image url:74.jpg
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/74.jpg
V / ImageFetcher ( 3997): image url:79.jpg
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/79.jpg
V / ImageFetcher ( 3997): image url:80.jpg
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/80.jpg
V / ImageFetcher ( 3997): image url:81.jpg
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/81.jpg
V / ImageFetcher ( 3997): image url:82.jpg
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/82.jpg
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 5098 objects / 291144 bytes in 48ms
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 73 objects / 3152 bytes in 47ms
I / dalvikvm-heap ( 3997): Grow heap (frag case) to 4.221MB for 87396-byte allocation
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 12 objects / 448 bytes in 49ms
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 29 objects / 1024 bytes in 47ms
I / dalvikvm-heap ( 3997): Grow heap (frag case) to 4.303MB for 87396-byte allocation
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 3 objects / 112 bytes in 53ms
V / ImageFetcher ( 3997): image url:84.jpg
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 43 objects / 1800 bytes in 54ms
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 1 objects / 16 bytes in 54ms
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 4 objects / 128 bytes in 57ms
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 54 objects / 1880 bytes in 63ms
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 16 objects / 512 bytes in 61ms
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 45 objects / 1488 bytes in 42ms
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/84.jpg
V / ImageFetcher ( 3997): image url:10.jpg
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 135 objects / 4696 bytes in 49ms
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/10.jpg
V / ImageFetcher ( 3997): image url:85.jpg
D / dalvikvm ( 3997): GC_FOR_MALLOC freed 603 objects / 20080 bytes in 52ms
V / ImageFetcher ( 3997): Fetching image from: http://shopdroid.ggajewski.2be.pl/images/logos/85.jpg
V / ShopDroidApplication ( 3997): Enabling Location Listeners...
W / KeyCharacterMap ( 3997): No keyboard for id 65540
W / KeyCharacterMap ( 3997): Using default keymap: /system/usr/keychars/qwerty.kcm.bin
V / ShopDroidApplication ( 3997): Enabling Location Listeners...
V / ShopDroidApplication ( 3997): newLocation: Location[mProvider=network,mTime=1284740445438,mLatitude=50.074489,mLongitude=19.906881,mHasAltitude=fals
e,mAltitude=0.0,mHasSpeed=false,mSpeed=0.0,mHasBearing=false,mBearing=0.0,mHasAccuracy=true,mAccuracy=876.0,mExtras=Bundle[mParcelledData.dataSize=148]]
V / ShopDroidApplication ( 3997): newLocation: Location[mProvider=network,mTime=1284740445438,mLatitude=50.074489,mLongitude=19.906881,mHasAltitude=fals
e,mAltitude=0.0,mHasSpeed=false,mSpeed=0.0,mHasBearing=false,mBearing=0.0,mHasAccuracy=true,mAccuracy=876.0,mExtras=Bundle[mParcelledData.dataSize=148]]
```

Figure 2.6 Example of Logcat



Operation	Duration (ms)
measure	0.641
layout	6.104
draw	12.115

Property	Value
absolute_x	9
absolute_y	85
getBaseline()	-1
getHeight()	225
getMax()	100
getProgress()	0
getSecondaryProg...	0
getTag()	null
getVisibility()	GONE
getWidth()	225
hasFocus()	false
isClickable()	false
isDrawingCacheEn...	false
isEnabled()	true
isFocusable()	false
isFocusableInTouc...	false
isFocused()	false

Figure 2.7 Hierarchy Viewer in action

2.3 ZXing library

ShopDroid uses ZXing (Zebra crossing) library as barcode decoder. The3 main advantages of the ZXing project are: open source (Apache License v2.0) [9], support for UPC and EAN codes, Java language, active support and development for 5 years. This project is focused on using the built-in camera on mobile phones to photograph and decode barcodes on the device. Some barcode decoders take a picture of the barcode and send it to server, where the decoding process takes place, however the ZXing project performs all operations on device. It supports many code standards:

- UPC-A and UPC-E
- EAN-8 and EAN-13
- Code 39
- Code 93
- Code 128
- QR Code
- ITF
- Codabar
- RSS-14 (all variants)
- Data Matrix ('alpha' quality)
- PDF 417 ('alpha' quality)

The project can be found on the project's website [9]. Apache License v2.0 allows reusing and modifying published source code. The project has its own Android application called Barcode Scanner. Thanks to that application, the code could be tested before implementation. Barcode Scanner recognizes UPC and EAN codes quickly and has high efficacy.

2.4 Server requirements

The server is used as centralized data storage. That solution makes data operations easier. A data update is done only once on the server side and all clients can access the new data. In the scenario with the database on the client side, changes will require database synchronization or the client application to be reinstalled. The server requirements are:

- PHP support

- PostgreSQL database
- MySQL database
- HTTP Authorization header accepting

For the ShopDroid project a commercial server was used. Most of all hosting companies offer servers that fulfill those requirements. It is also easy and cheap to configure your own server, because all needed functionality is open-source and free to use.

2.5 PHP scripting language

PHP: Hypertext Preprocessor is a scripting language mostly used in web applications. Combining HTML tags with PHP helps to build dynamic web pages. In the ShopDroid project, PHP is used to provide user authorization, administrator panel and as a logic layer between client and database. The last feature is important as it helps to separate the client application code from the database type. In that case any database related changes would affect only the server side. That prevents from unnecessary updates for every single client application. Main advantages of PHP: fast, stable, secure, open-source, free to use [10].

2.6 Database choice

Choosing database to this project was a hard task. The first assumption made was that the database used in ShopDroid project should be fast and easily expandable as there will be a large amount of records. Other arguments, which were taken into account, are database stability and reliability. Being free to use and open source additionally increase database attractiveness. There are several databases which meets this requirement, some of them: SQL Server Express, Oracle XE, PostgreSQL, Firebird, MySQL. MySQL and PostgreSQL are popular among web developers what have direct impact on hosting offers. While almost all servers have MySQL, some of them have PostreSQL and it is really hard to find other systems.

PostgreSQL is an object-relational database management system (ORBDMS) [11]. This database was chosen to keep all products and stores related data. A big advantage of PostgreSQL is a license (PostgreSQL license, which is a MIT-style license [12]) that allows developers to use it even in closed projects for free. Other benefits: conformity with standards, good support, great possibilities, that might be useful in the future. If

MySQL is another object-relational database management system. Usually one database is enough for the project, however ShopDroid project uses the oauth-php library, which supports only the MySQL database. This fact was discovered in late development when part of the system was already working with PostgreSQL, so it was decided to use two databases system in ShopDroid. Additionally, storing user credentials in a separate database increases security level and system efficiency. The product and stores data is stored in a PostgreSQL database, and the users data is stored in a MySQL database.

2.7 XML Extensible Markup Language

XML - Extensible Markup Language simple and flexible text format, commonly used to exchange wide variety of data on the Web [13]. XML is a standard recommended and specified by W3 Consortium. Each XML script has to start with the XML declaration:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

ShopDroid uses XML format as a response to clients' HTTP requests. There are three types of responses: stores list, products list and product information. Stores list is used when the user is looking for nearby stores on the map, product list is used when the user is searching product by name, product information presents product, prices in different stores, user reviews. A XML document has to have exactly one root tag. A tag is a markup that begins with "<" and ends with ">". The content is surrounded with tags, the opening tag and the closing tag which additionally has "/" after "<". When the content is missing either the opening or closing tag, the XML document parsing process will stop and return an error. The content can be empty.

2.8 OAuth protocol

ShopDroid implements OAuth 1.0 open protocol for user authentication. The server - OAuth service provider is using oauth-php implementation [14], the client - consumer signpost library [15]. OAuth protocol is defined in RFC 5849: The OAuth 1.0 Protocol document [16]. The OAuth protocol does not provide any implementation itself. This protocol ensures high security level. Many big web companies like Google, Facebook, Twitter or Yahoo are using OAuth in their APIs what gives evidence of reliability of that solution.

There are several OAuth PHP implementations, but most of them include only the OAuth client. Oauth-php is a Consumer And Server library. It is a fully working and complete library with good documentation and support.

The library implements methods to:

- *verify incoming requests against the library*
- *sign outgoing requests, with curl support for actually doing the request*
- *sign requests with a body*
- *administrate consumer keys and tokens for multiple users (server and consumer side)*
- *log incoming and outgoing requests handled by the library (optionally in the database)* [14]

By default, the project is working with MySQL OAuth store. Authors of that project claims it has an extensible OAuth store. However, trials to make a different store i.e. using PostgreSQL database made changing a lot of undocumented source code. Finally, ShopDroid is using the default OAuth store.

As in the case of Service Provider there are a few OAuth Consumer JAVA implementations, but only one is Android ready - signpost. Signpost offers simple message signing. This project has a great community and documentation. Code is well tested and used in Android applications like Qype Radar.

Chapter 3
ShopDroid Project

In this chapter all the client and server components of the ShopDroid project will be described in details.

3.1. Main Screen

The main screen (shown in Figure 3.1) is the first activity that appears when the user launches the application and therefore it has to show the application's main functionalities. The planned functionalities were four:

- Barcode scanning
- History
- Manual barcode and product search
- Locating nearby stores

The user interface elements are defined in the layout file *main.xml*, while the source that makes this buttons function is defined in *ShopDroid.java*.

These four functionalities have to be quickly accessible by the user so it was decided to have a large button with an image for each of the functionalities. Having large buttons makes it very easy for the user to tap on them. Images are included inside the buttons so that the user can easily relate them to a certain function. Putting images inside a button can be down using the attribute *android:drawableTop* when defining the button properties in the layout file. A small phrase below the image is also shown in case the image is not clear enough.

A big empty space was left at the bottom of the screen to leave the future possibility of adding advertisement and monetizing the application.

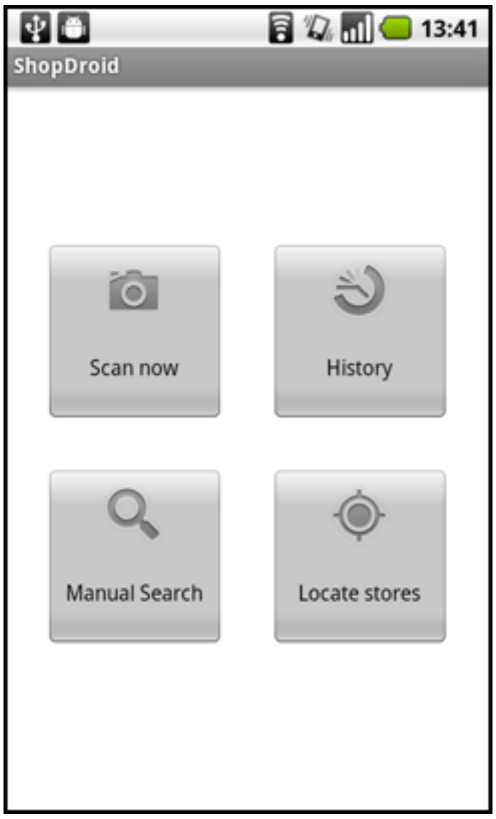


Figure 3.1 Main screen

Another thing to consider was the landscape view of the screen. Android automatically rearranges the user interface elements when switching from portrait to landscape. It is usually enough to design the interface for portrait mode, which is the default one. However, in some cases, as it was in this project's main screen, the automatic rearranging does not look good in landscape mode. Android leaves the option to define the landscape mode for a certain layout if wanted. It was only needed to create a layout file with the same name, put it on the *layout-land* folder and make the proper modifications. The differences of the screen in landscape mode without and with a landscape layout file can be seen in figure 3.2.

Google encourages the use of the Options Menu for the user to access certain application functions and settings. Tapping on the menu key on any Android device accesses this menu. For the main screen, the user can access the "settings" and "about" menu as shown in Figure 3.3.

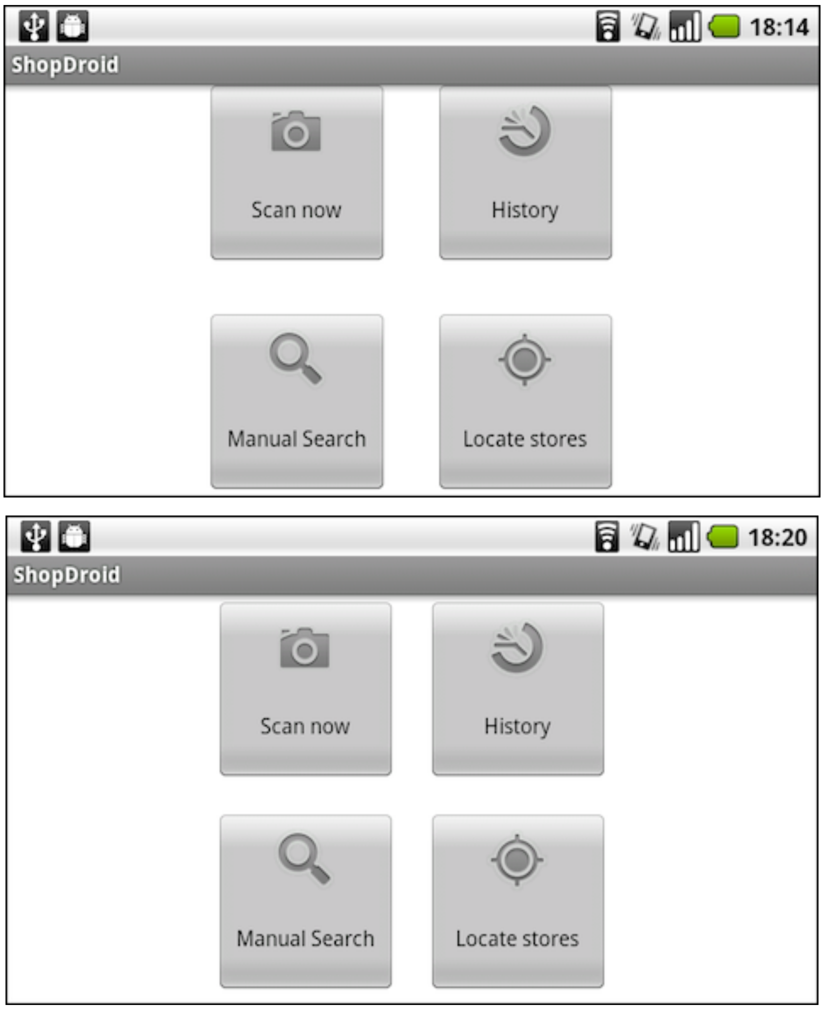


Figure 3.2 Differences in the main screen landscape mode. Upper screenshot: without a landscape layout file. Lower screenshot: with a landscape layout file.

3.1.1 Application settings

The settings screen, shown in Figure 3.4a, is useful to add certain options, like letting the user disconnect his ShopDroid account, or in other words to log off. Another option is to turn on the feedback once a barcode is scanned and decoded, the user can choose to have the device beep or vibrate or none. The last option in the application is an advanced option intended only for debugging purposes. Accessing the last option, brings the Location screen which is used for mocking the current location, useful for testing. This last screen is shown in Figure 3.4b.

These settings are defined in the *settings.xml* file inside the *xml* folder and in the *Settings.java* file. Android makes it possible to define a key for each option in xml (using the attribute *android:key*), which later on can be easily read from within the code.

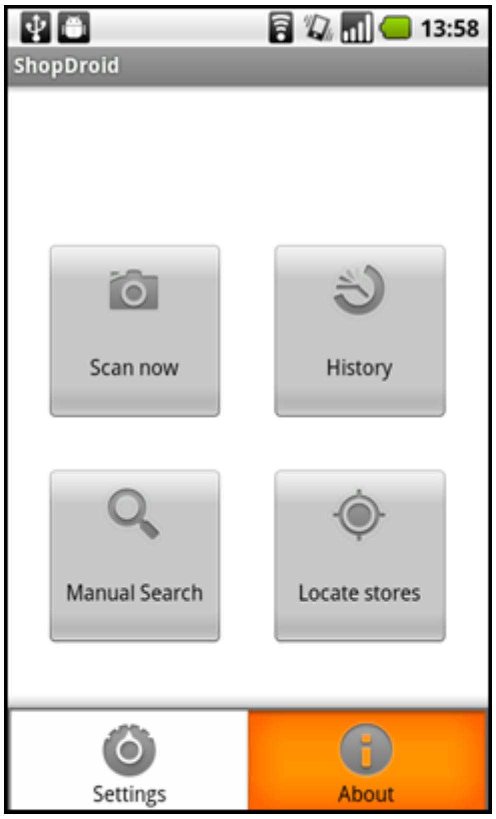


Figure 3.3 Main screen menu

In general, settings or preferences are just another activity, and to be more specific they are a `PreferenceActivity`. Use of the settings.xml file has to specified in code like this:

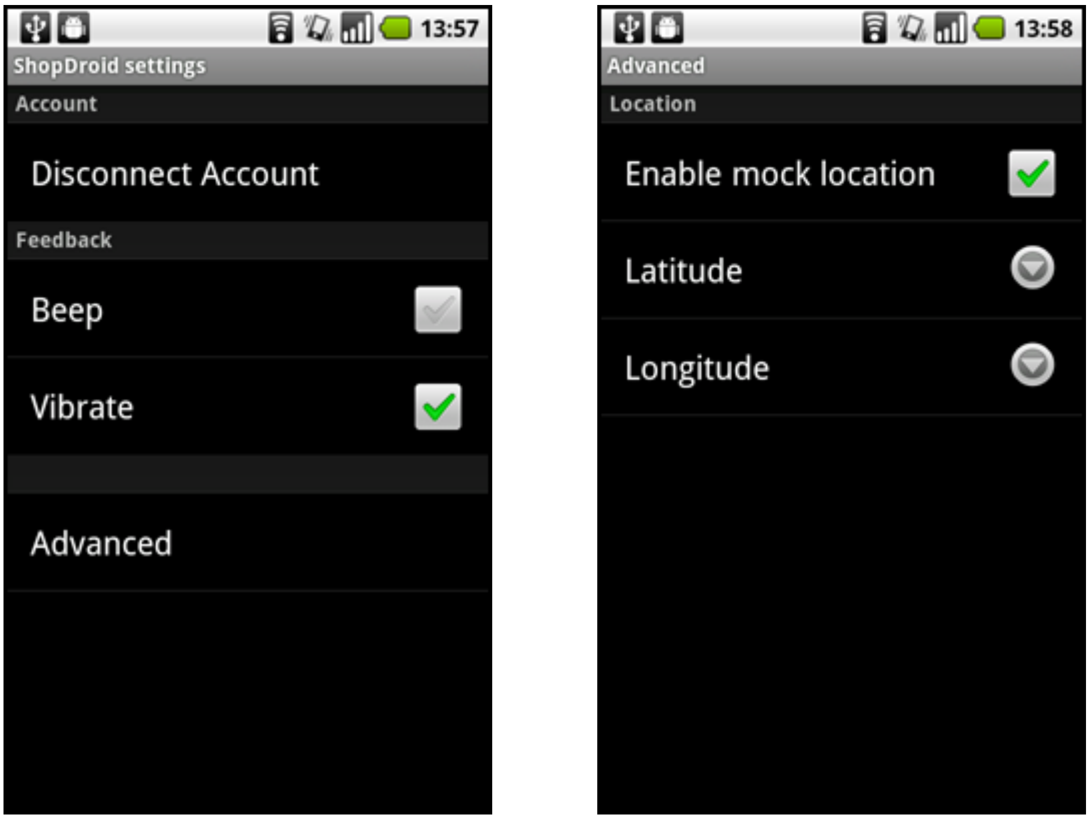
```
addPreferencesFromResource(R.xml.settings);
```

Usually this code should be enough, but in the case of the ShopDroid application some more code was needed for disabling and enabling the *Disconnect Account* option whether the user has already logged on or not.

3.1.2 About application

The About screen (shown in Figure 3.5) is useful for describing the application and giving credits. It is called in the code inside the main activity by creating an `AlertDialog`:

```
new AlertDialog.Builder(this).setTitle("A title")
     .setMessage("Some text").setPositiveButton("OK", null).show();
```



ShopDroid settings

Account

Disconnect Account

Feedback

Beep ☐

Vibrate ✓

Advanced

Advanced

Location

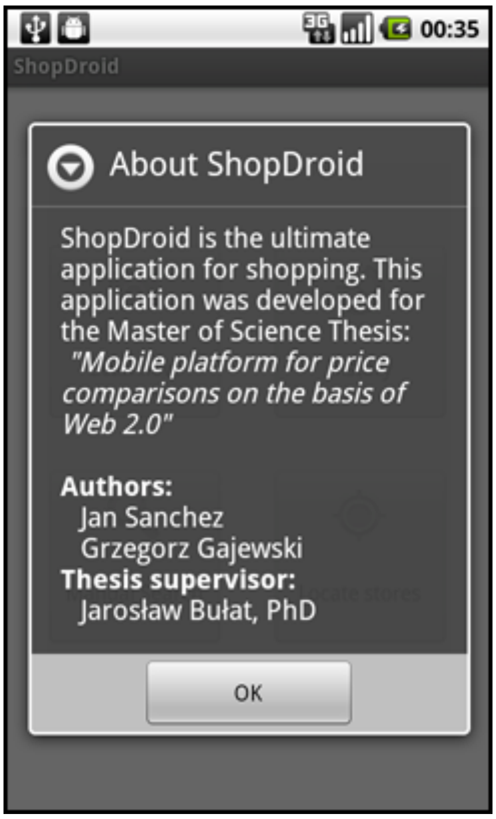
Enable mock location ✓

Latitude

Longitude

(a) (b)

Figure 3.4 a) Settings screen, b) Advanced settings



About ShopDroid

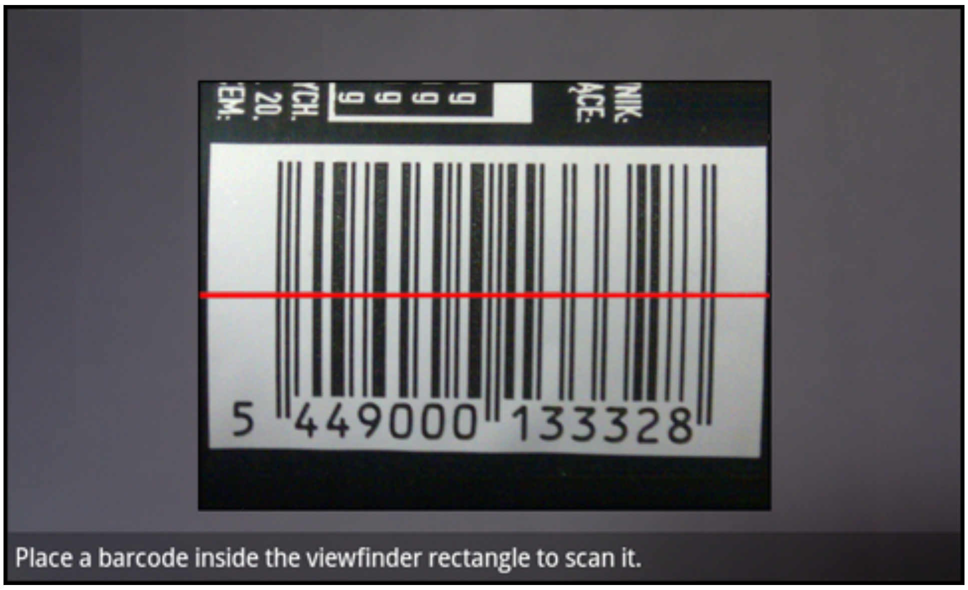
ShopDroid is the ultimate application for shopping. This application was developed for the Master of Science Thesis: *"Mobile platform for price comparisons on the basis of Web 2.0"*

Authors:
Jan Sanchez
Grzegorz Gajewski
Thesis supervisor:
Jarosław Bułat, PhD

Figure 3.5 About screen

3.2 Barcode scanning process

ShopDroid allows the user to scan barcodes using the phone-integrated camera. When the user chooses “Scan now” option from main menu capture activity is launched. The capture screen (Figure 3.6) displays the live camera view with a rectangle at the center of screen, transparent borders, help text, and a horizontally center red line which shows the user where to place the barcode.



Place a barcode inside the viewfinder rectangle to scan it.

Figure 3.6 Barcode scanning screen

During the scanning process yellow dots are drown when part of the code is recognized. Camera auto-focus runs in the loop helps improve image quality. Usually scanning process takes 5-7 sec, in good lightening conditions this time goes under 3 sec. If the code is detected the Product screen is showed to the user.

3.2.1 Barcodes

A barcode is a graphical representation of data easily readable by optical devices. There are many code standards, but only four are used for product tagging: EAN-13, EAN-8, UPC-A, UPC-E. The global, non-profit organization called GS1 has chosen these codes as GTIN (Global Trade Item Number). The cooperation between GS1 and manufacturers can guarantee worldwide unique product identification numbers. UPC family codes are currently used in Canada and USA while EAN codes are used in all the other countries. This chapter will describe the barcode decoding process with EAN-13 as example and only this code will be taken into consideration. [18, 19]

EAN codes

EAN (originally European Article Number, but now changed to International Article Number) can appear in two formats: EAN-13 (Figure 3.7) and EAN-8 (Figure 3.8). EAN-13 is commonly used for marking products, it encodes 13 digits, from which 12 are for the product number and 1 is for the checksum. EAN-8 (7 + 1 digits), which is a shortened version, is intended for small packages i.e. pocket tissues, chewing gums and elsewhere where a 13 digits long code does not fit.

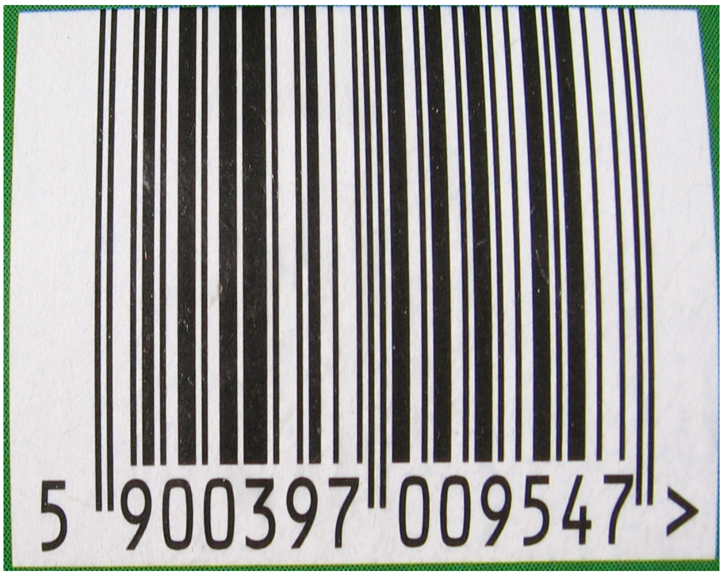


Figure 3.7 EAN-13 example

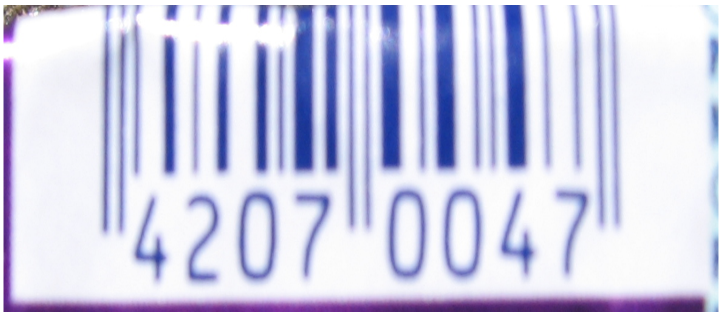


Figure 3.8 EAN-8 example

Code properties

Barcode properties are enumerated below:

- 1D - code readable in 1 dimension
- Continuous - no breaks between digits
- Numeric - encodes only digits in decimal notation
- Many-width - different bars and spacing height
- Checksum - additional digit for checksum

Digits meanings

- The GS1 Prefix, the first 2 or 3 digits usually identifying the country code where the manufacturer is registered (590 for Poland).
- The Company number, 4,5 or 6 digits, that GS1 assigns to the company depending on the number of different products lines.
- Item reference, from 2 up to 6 digits, provided by the manufacturer.
- Checksum, the very last digit.

Code structure

EAN family codes always start and end with a guard pattern, three bars: black, white and black, which are longer than the other bars, as shown on figure 3.9. The middle sign, which is build from five longer bars that alternate white and black, splits the code into two equal parts. Each module encoding a digit contains seven bars matching bits, the white bar is treated as binary ‘0’ and the black one as ‘1’. In table 3.1 the encoding of digits is shown. All digits are divided into three groups: the first digit, the digits from number two up to number seven and the last six digits. The first digit is encoded by parity of the following six digits, as those digits have two possible encodings, L-code with odd parity and G-code with even parity. The last six digits are encoded in R-code, which is a mirror reflection of G-code. Figures 3.10 - 3.12 show actual each digit encoding using bars, in accordance to L-code, G-code, R-code.

Table 3.1 EAN-13 Digits encoding[1]

Digit	L-code	G-code	R-code
0	0001101	0100111	1110010
1	0011001	0110011	1100110
2	0010011	0011011	1101100
3	0111101	0100001	1000010
4	0100011	0011101	1011100
5	0110001	0111001	1001110
6	0101111	0000101	1010000
7	0111011	0010001	1000100
8	0110111	0001001	1001000
9	0001011	0010111	1110100

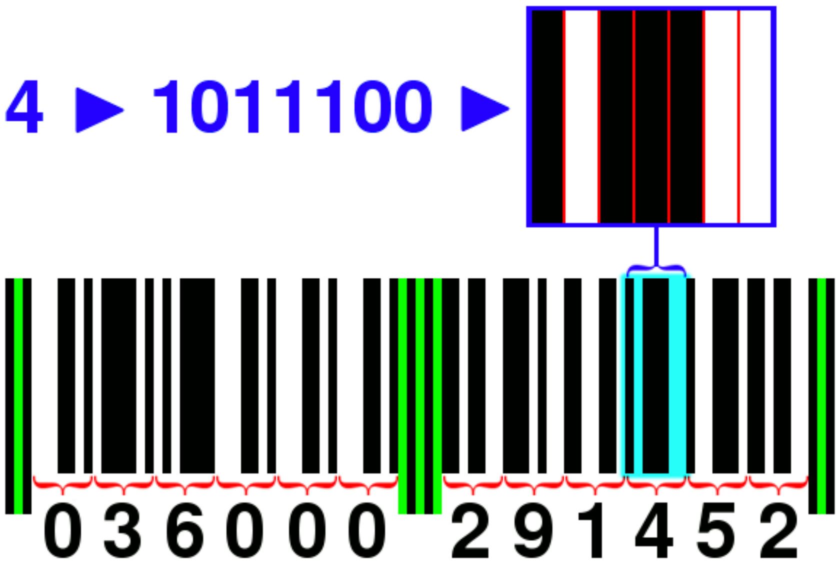


Figure 3.9 EAN-13 structure[2]

[1] http://en.wikipedia.org/wiki/European_Article_Number
[2] http://pl.wikipedia.org/w/index.php?title=Plik:UPC_EANUCC-12_barcode.png

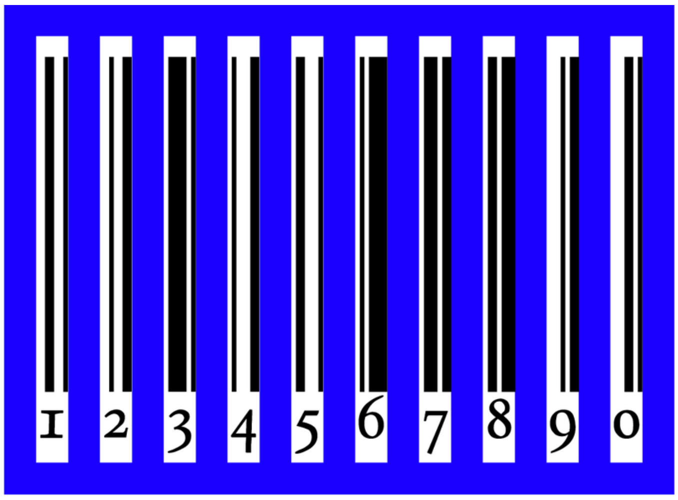


Figure 3.10 Digits encoded in L-code[1]

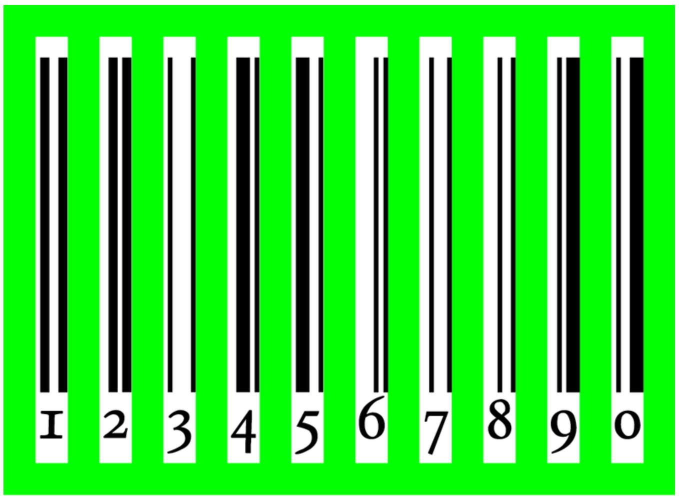


Figure 3.11 Digits encoded in G-code[2]

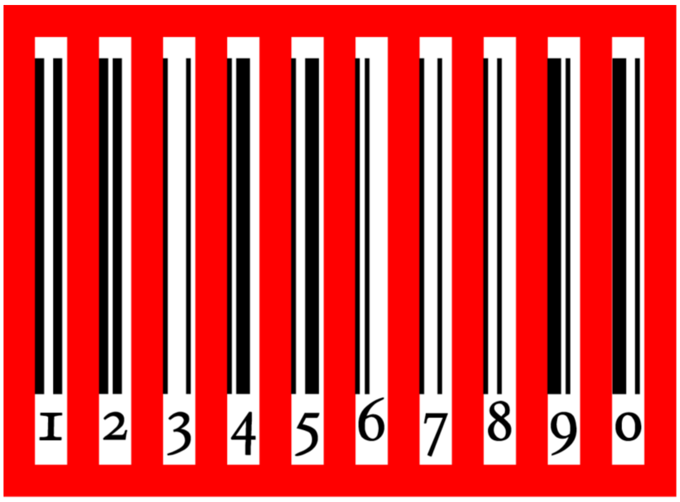


Figure 3.12 Digits encoded in L-code[1]

[1] http://en.wikipedia.org/wiki/File:EAN-L.png
[2] http://en.wikipedia.org/wiki/File:EAN-G.png

Table 3.2 shows how to encode the first digit using L-code, and G-code.

Table 3.2 EAN-13 First digit encoding. L - L-code, odd parity, G - G-code, even parity

Digit	**6 digits group**
0	LLLLLL
1	LLGLGG
2	LLGGLG
3	LLGGGL
4	LGLLGG
5	LGGLLG
6	LGGGLL
7	LGLGLG
8	LGLGGL
9	LGGLGL

EAN-13 Checksum calculation

The checksum needs to be calculated before printing the code. The checksum digit is the result of (3.1). If the result equals 10 then the checksum is also 0.

$$checksum = 10 - \left(\sum_{i=1}^{6} x_{2i} + \sum_{i=1}^{6} 3x_{2i-1}\right) mod\ 10 \qquad (3.1)$$

where x_i - code digit on i-th position

An exemplary calculation of checksum is shown below:

Lets assume that the code equals 590063500111.
Odd sum: 5 + 0 + 6 + 5 + 0 + 1 = 17.
Even sum: 3*9 + 3*0 + 3*3 + 3*0 + 3*1 + 3*1 = 42.

[1] http://en.wikipedia.org/wiki/File:EAN-R.png

Final sum: 59

Sum modulo ten: 59 mod 10 = 9

Checksum: 10 - 9 = 1

3.2.2 Decoding algorithm

The main class, `CaptureActivity`, controls the scanning process and displays a live camera preview. The decoding itself runs on another thread in order not to hang the user interface screen, what is more it make use of multicore architecture of future smartphones CPU without any code modification. `CaptureActivity` calls the `CameraManager` class to set up the camera. `CameraManager` is responsible for sending commands to the camera driver i.e. auto-focus request. The image taken by camera needs to be sharp to recognize the barcode and requires the camera to have autofocus. Almost all Android based smartphones fulfill that condition. `CameraManager` sets the camera resolution to be equal to the actual screen resolution. `CameraManager` also contains a method to calculate the framing rectangle that will be drawn to show the user where to place the barcode. The minimum size of this rectangle is 240x240 pixels and the maximum, 480x360. There is no need to take a larger image for decoding as it unnecessarily consumes lots of computing power. The image taken by the camera is in YCbCr420 format; frame is send to the decoding thread as a message. The `CaptureActivityHandler` class handles messages used to communicate with the decoding thread. The `DecodeThread` class tries to decode the barcode from a given image. Setting the `decodeFormats` variable to `CaptureActivity`. PRODUCT_FORMATS limits the code search to UPC-A, UPC-E, EAN-13, EAN-8, RSS14 formats. Decoding runs in a loop, when one decode fails a new one is started immediately. It creates `BinaryBitmap` and `HybridBinarizer` objects from the given image, which contains methods for image processing, and it passes a `BinaryBitmap` to `decodeWithState` method defined in the `MultiFormatReader` class. `MultiFormatReader` calls all selected Readers one by one to look up for the barcode. `OneDReader`, a basic class for all one-dimensional codes, calculates the height of a given image, then it examines the rows starting from the middle and moving outwards, above or below the middle alternately. The step depends on the `tryHarder` variable, if it has been set, the algorithm checks each row, if not, only 15 rows are checked with a step 1/16 of image height. By default, the `tryHarder` variable is unset. Tests showed that skipping rows does not remarkably reduce efficacy but it does increase scanning speed. To improve performance a simple sharpening is applied by calling the `getBlackRow` method. The body of that method can be found in `GlobalHistogramBinarizer` inherited by `HybridBinarizer`. It creates a histogram and searches the two tallest peaks: first for black and second for white color. If the difference between them is too small, that means that the image contrast is too little and the function throws an exception to avoid the risk of false positives. When peaks have been found, the valley that is low and closer to white is selected as a reference point. A bit array is created with a length that equals the image width. Pixels are compared to the black point; if the luminance is smaller then the appropriate bit is set in the array. When the whole array is filled, the code detection algorithm starts. Each row has two decoding attempts, the first, regular, and the second, reverse to handle decoding upside down barcodes. `UPCEANReader` contains methods and constants common to UPC and EAN family codes. It takes a prepared bit array row and tries to match it with the start guard pattern. To do that, the algorithm creates an integer counters array with the same length as the pattern length. Patterns are also stored in integer arrays, successive numbers are a sum of identical (white or black) bars in a row. For the start pattern (black, white, black bars), the array is {1,1,1}. To find it, three counters are created to count bits/pixels with the same color. When the counting is done, the `patternMatchVariance` method determines how close is the recorded sequence to the requested pattern. The measure to define that dependence is expressed as the ratio of total variance from expected pattern proportions across all pattern elements, to the length of the pattern (3.2). Value '0' means perfect match, to pass the value it needs to be lower than 0.42. This value was fixed by project authors after many experimentally trials.

$$t_v = \left(\sum_{i=0}^{p_l - 1} \left| c_i - p_i \frac{t_l}{p_l} \right| \right) \div t_l \tag{3.2}$$

where t_v - total variance, p_l - pattern length, c_i - value of i counter, p_i - value of i pattern, t_l - total length, all counters sum

Once the start pattern has been found, the algorithm checks if there is a quiet zone before the barcode, at least as big as the start pattern and it starts to decode the digits. First it attempts to decode six digits on the left. Each digit encoding has exactly four black/white fields, therefore it always ends with a bar color that is opposite to the first bar. To decode a digit four counters are needed, firstly it records the pattern and than tries to match it with any given pattern (for the left side, the number of patterns to check is 20 because of different parity), the digit with the smallest variance is returned. Calculation of the variance is done in the same way as for the start pattern and it needs to be lower than 0.42, otherwise a not found exception will be thrown. If six digits are positively matched, the algorithm determines the first digit as a result of parity as shown in table 3.2. The algorithm finds the middle guard bars range {1,1,1,1,1} and decodes the last six digits. While all digits are decoded correctly quiet zone after the barcode is checked if it is at least as big as the end pattern. The checksum is calculated as described in section 2.3, and the result must be equal to '0' to pass. If there were no errors a result is returned as a string containing 13 digits, if not another row is checked for the code existence. If all rows from image are checked without success algorithm takes new image and scanning starts again.

Classes required to decode a EAN-13 barcode [9]:

`CaptureActivity`

The barcode reader activity itself.

`CaptureActivityHandler`

Handles all the messaging that comprises the state machine for capture.

`CameraManager`

This object wraps the Camera service object and expects to be the only one talking to it. The implementation encapsulates the steps needed to take preview-sized images, which are used for both preview and decoding.

`DecodeThread`

Thread that does all the heavy lifting of decoding the images.

`BinaryBitmap`

This class is the core bitmap class used by ZXing to represent 1 bit data. Reader objects accept a BinaryBitmap and attempt to decode it.

`BitArray`

A simple, fast array of bits, represented compactly by an array of integers internally.

`MultiFormatReader`

A convenience class and the main entry point into the library for most uses.

`OneDReader`

Encapsulates functionality and implementation that is common to all families of one-dimensional barcodes.

`MultiFormatOneDReader`

As MultiFormatReader but limited to one-dimensional barcodes

`MultiFormatUPCEANReader`

A reader that can read all available UPC/EAN formats.

`UPCEANReader`

Encapsulates functionality and implementation that is common to UPC and EAN families of one-dimensional barcodes.

`EAN13Reader`

Implements decoding of the EAN-13 format.

3.2.3 Code detection problems

The ZXing algorithm detects correctly most product barcodes; it only requires good lightening conditions. However, there are some cases that it does not decode the barcode. The problem could appear when a manufacturer does not stick with the GS1 recommendations and use different colors with low contrast as shown on Figure 3.13.

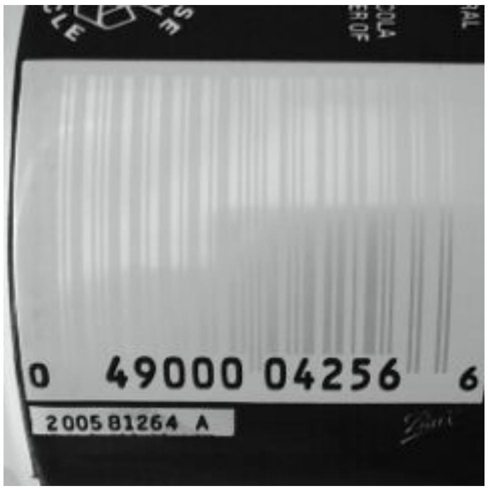


Figure 3.13 Low contrast barcode example[1]

Another problem concerns barcodes printed on a rounded surface (i.e. bottle) with a small radius (Figure 3.14). Bend on the barcode causes detecting problems because perspective changes spacing between bars. Fortunately, most of all manufacturers print the barcode on the bottle vertically; in that case it can be decoded without difficulty.

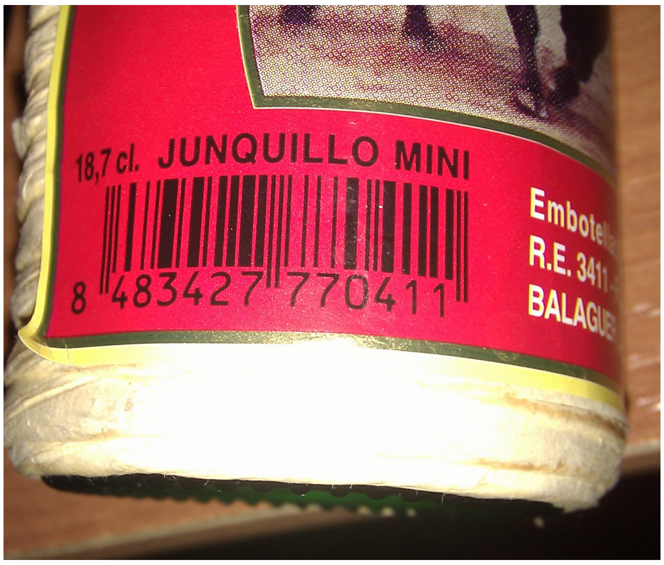


Figure 3.14 Barcode on rounded surface example

The last known problem can occur with barcodes located on flexible packages. Code deformation as shown in Figure 3.15 makes it impossible to decode.

[1] http://code.google.com/p/zxing/source/browse/trunk/core/test/data/benchmark/android-1/fail-1.jpg

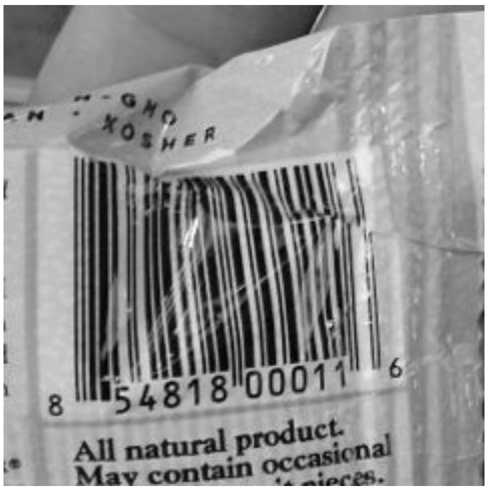


Figure 3.15 Deformed barcode example[1]

3.2.4 Android camera issue

Preview in the ZXing project is set to landscape mode by default. It is more comfortable to hold a phone in portrait mode while scanning barcodes, especially when a phone is hold with one hand. Due to hardware and system limitations it is impossible to force the camera preview to portrait mode at the time of writing this document. It is well known issue in the android developers community [20]. The ShopDroid application works around this issue by using a fake portrait mode. Camera preview is set to landscape mode. In `CaptureActivity`, an orientation listener has been implemented to handle orientation changes internally. To have a portrait mode illusion some ZXing's project methods needed to be modified. Because of the framing rectangle being different, the `getFramingRect` method in `CameraManager` class has been re-implemented for correct calculations. X/Y coordinates are changed in fake portrait mode as shown at Figure 3.16.

To correctly draw the rectangle, the `onDraw` method of the `ViewfinderView` class includes two different ways of drawing depending on the view. The status text is drawn on canvas rotated 90 degrees clockwise. The last problem to solve was camera frame data transition. There is a method called rotate data in the `CameraManager` class, which handles data rotation.

[1] http://code.google.com/p/zxing/source/browse/trunk/core/test/data/benchmark/android-1/fail-2.jpg

```
private byte[] rotateData(byte[] data, int width, int height) {

     int nw = height;
     byte[] newData = new byte[width * height];

     for (int y = 0; y < height; y++) {
          for (int x = 0; x < width; x++) {
               newData[nw * (x + 1) - y - 1] = data[y * width + x];
          }
     }
     return newData;
}
```

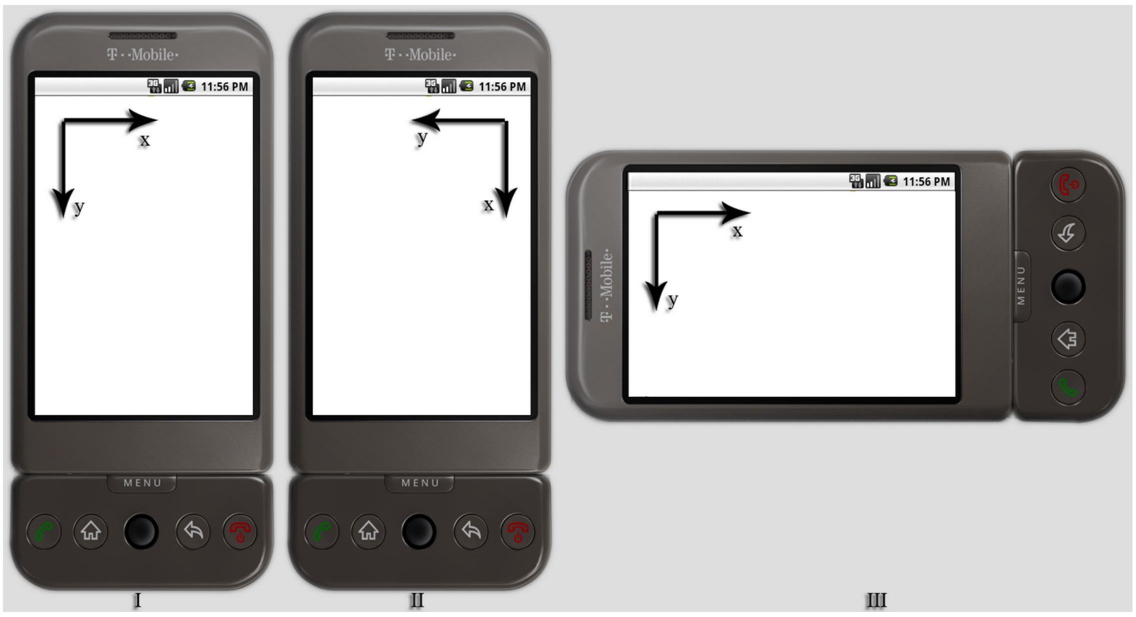


Figure 3.16 Android screen coordinates

3.3 Geolocation

Geolocation is the ability to identify the geographic coordinates, in other words the location, of the user's smartphone. This is an important feature of the ShopDroid application. It is needed in order to customize the user experience according to his location. In this case, the application will only show the product prices from the stores close to him.

The Android platform makes it possible to use two location providers, which have been successfully implemented in this application:

- GPS provider
- Network provider

3.3.1 GPS provider

All Android phones are GPS enabled. GPS would be the most accurate way of knowing the location of the user. It should be accurate to about 15 meters (this depends on each device). Getting a fix (that means that the device acquires the needed satellite signals, navigation data and calculates the position) with this provider might take some time - during tests, the fastest fix was in 5 seconds, while the slowest was around 2 minutes. Using this provider also requires that users have to be outdoors or close to a window if they are indoors. It also shows a GPS icon in the notification bar, indicating users that the GPS is active and being used by the application as shown in Figure 3.17.

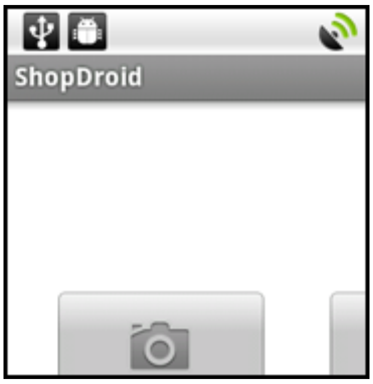


Figure 3.17 Gps icon notification

3.3.2 Network provider

When the better GPS provider does not work or it is not enabled in the phone (Android disallows an application to turn on the GPS for the user) then the application can use the Network provider, which can get the location coordinates from GSM (Global System for Mobile Communications) triangulation or if connected to a wireless network, it could get the location from the IP address of the network. It should be noted that this last option (getting the location from a wireless network) depends on Google's internal database, and the accuracy level of this may vary.

A location listener must be created per each provider. Having created the listeners, the application can request location updates. A problem aroused when implementing this in the application, was that creating and using the location listeners in an activity, made them unusable by other activities. With Android, each screen can be understood as an individual activity, with almost not reference at all to any previous or subsequent activity, only some parameters can be passed between activities, which are some primitive types and strings. Android FAQs (Frequently Asked Questions) provide a solution for this:

The android.app.Application is a base class for those who need to maintain global application state. It can be accessed via getApplication() from any Activity or Service. It has a couple of life-cycle methods and will be instantiated by Android automatically if your register it in AndroidManifest.xml [21].

The `Application` class is similar to a singleton; it lives throughout the entire life of the application, no matter which activity is in the foreground. With this functionality it seemed obvious to implement the location listeners inside the `Application` class. All Android applications provide a default `Application` class, but if need one can create a new one to add application specific features. This is implemented in the *ShopDroidApplication.java* file. The listeners are created as soon as the application is started, inside the `onCreate()` method of the application. Every activity can get the current location by calling `getLocation()` method:

```
((ShopDroidApplication) this.getApplication()).getLocation();
```

This returns a `Location` object, from which the latitude and longitude values can be obtained.

3.3.3 Avoiding Location updates when the application is in the background

Enabling the location listeners makes them active throughout the life of the application, unless disabling them manually. This can cause some battery drainage, which is not positive for the user. This can be seen as soon as the user switches the application, or goes to the device's home screen, the GPS icon will still be shown in the notification area, indicating that it is active and that it might request new location updates. This is also true for the Network provider listener, which doesn't show any icon in the status bar, but nonetheless it will still be active.

In order to avoid this, the listeners must be disabled when the application is paused. Unfortunately Android's `Application` class does not provide any way to check if the application was paused. This can only be done per each activity. So in ShopDroid, each activity enables these listeners, which were already created once the application started, whenever resuming, inside the activity's `onResume()` method. This method will even be called the first time the activity is created. Also, these listeners are disabled whenever any activity goes into the background, this is implemented inside the activity's `onPause()` method.

For debugging and testing purposes, an option to mock the location was implemented. ShopDroid's `Application` class will check on start, if the mock location option is activated, and if so, then it will return the mock location to whatever activity that requests the location. By doing like this, none of the activities code needed to be modified.

3.4 History

One of the ShopDroid application's features is to save the users history and allow them to check any found products as quickly as possible, so that they would not need to scan or search the barcode again. Android, like most modern mobile operating systems, provides support for SQLite. Using a SQLite database does not produce additional overhead for the system, data can be read from the database extremely fast. While writing several records to the database might incur in performance decrease, this application only inserts one record each time it needs to add a new history item. This is the reason why SQLite suits best the needs of this application.

The database is created thanks to the constructor of `SQLiteOpenHelper`. This constructor also accepts the database version as a parameter, which is useful when upgrading the database to alter, drop or create tables. A table *history* was created for this feature using the following SQL statement:

```
CREATE TABLE history( _id INTEGER PRIMARY KEY AUTOINCREMENT, product TEXT NOT NULL, barcode TEXT UNIQUE NOT NULL);
```

Table 3.3 shows in more details the table created for the history feature.

Table 3.3 History table details.

Column	Type	Integration relations	Comment
_id	integer	primary key, auto-increment	auto-increment primary key, identification number
product	text	not null	product name
barcode	text	unique, not null	barcode of the product, it is unique to avoid duplicates

All this functionality is implemented in *HistoryManager.java* inside the *com.janandgreg.shopdroid.util* source package. `HistoryManager` objects are created in two screens: the history screen, and the product screen.

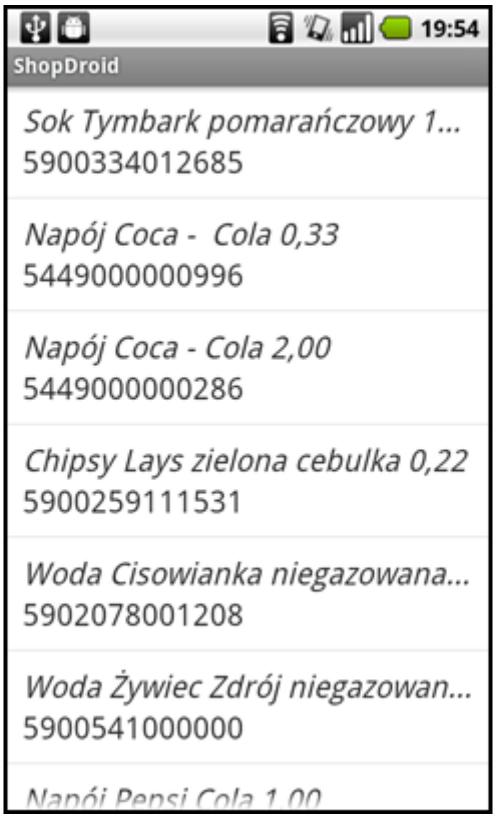
In the product screen, a new entry will be added to the table if the product has been found. It should be noted, that the barcode column is defined as unique, so there will not be any duplicates. There is also a limit for the number of records which can be easily changed. When adding a new entry, `HistoryManager` checks the number of records with the following SQL statement:

```
SELECT COUNT(*) FROM history
```

It will check the value returned and if it is more than the maximum number of records allowed, it will proceed by checking the oldest record and deleting it. The oldest record is checked with the following statement:

```
SELECT MIN(_id) FROM history
```

The History screen shows users all of their successfully found products that they tried to search or scan. An example screen is shown in Figure 3.18.



Sok Tymbark pomarańczowy 1...
5900334012685

Napój Coca - Cola 0,33
5449000000996

Napój Coca - Cola 2,00
5449000000286

Chipsy Lays zielona cebulka 0,22
5900259111531

Woda Cisowianka niegazowana...
5902078001208

Woda Żywiec Zdrój niegazowan...
5900541000000

Napój Pepsi Cola 1,00

Figure 3.18 History screen

This screen is implemented in *History.java*. This view uses the ListView widget. To load all the history records it uses HistoryManager's `fetchAll()` method. That method returns a Cursor object with all the table records, which can be iterated over. Also, a layout must be defined for a single item in the database. Once the Cursor is returned, a `SimpleCursorAdapter` is created, that uses the newly created cursor and the layout defined as parameters to show the entries in the ListView widget.

Users can also delete any single entries when long pressing the one they want to delete (shown in Figure 3.19). It is also possible for users to delete all the history by selecting that option when pressing the menu key.

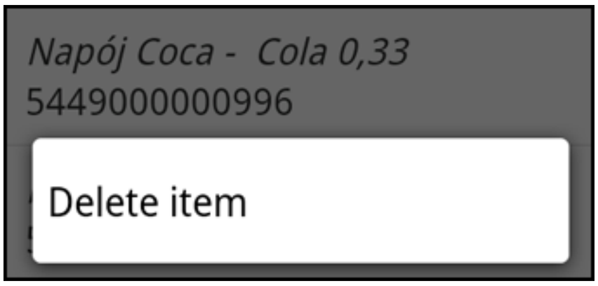
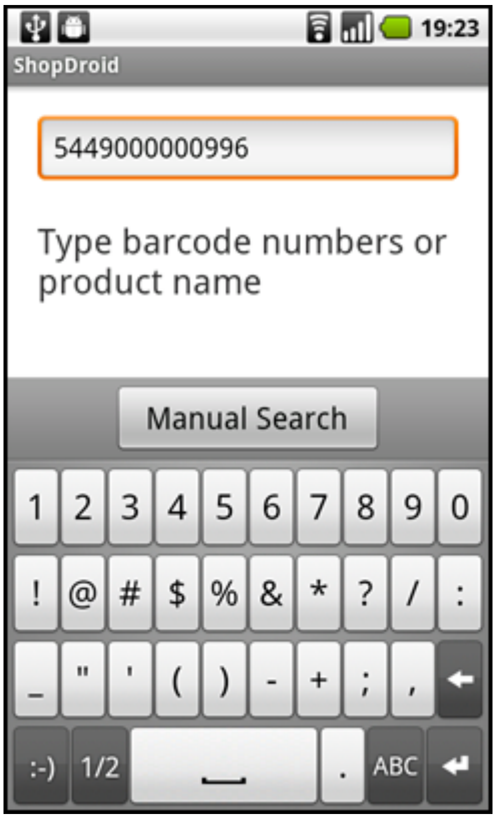

Napój Coca - Cola 0,33
5449000000996

Figure 3.19 Delete history item

3.5 Manual Search

Users have the option to manually type the barcode of the product, or the product name itself. This can be the case if the barcode for some reason cannot be decoded. Also if users do not want to scan the barcode or they do not have access to the product, they can manually search for the name of the product. This is shown in Figure 3.20. All of this is coded in *ManualSearch.java.*



5449000000996

Type barcode numbers or product name

Figure 3.20 Manual Search screen

The input from the user is taken as a string, and the application automatically recognizes if it is of numeric type or not. It does by using this method:

```
Long.parseLong(String input)
```

This method throws a `NumberFormatException` if the input has something else than numbers. So if it does not throw this exception, the user's input is used as a barcode by the next activity, which will be the Product screen. This is the exact same result as if the user would scan the barcode and the barcode would be decoded by the application. This is shown in Figure 3.21.

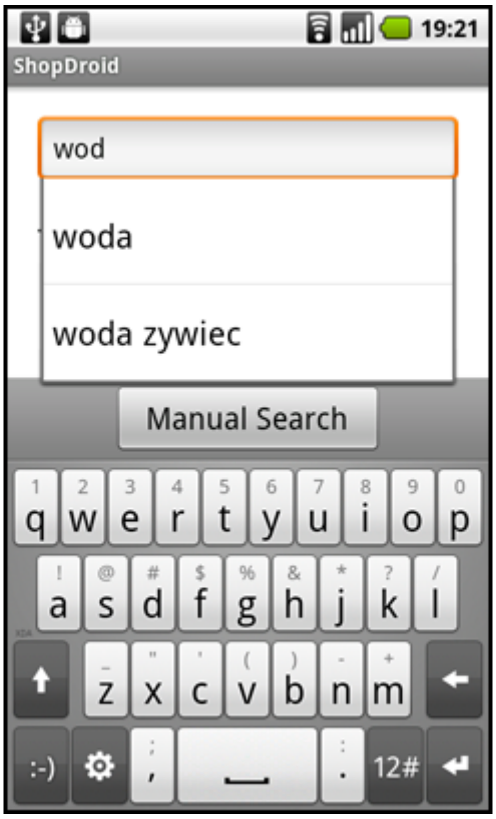


Figure 3.21 Searching products by name

However, if it throws a `NumberFormatException`, then the process is quite different. It does not go directly to the same Product activity, but to another one, which searches in the server database for all records that have the name of the product just searched.

3.5.1 Searching by product name

This subsequent screen for searching by product name is shown in Figure 3.22. This is coded in the *ProductSearchByName.java* file.

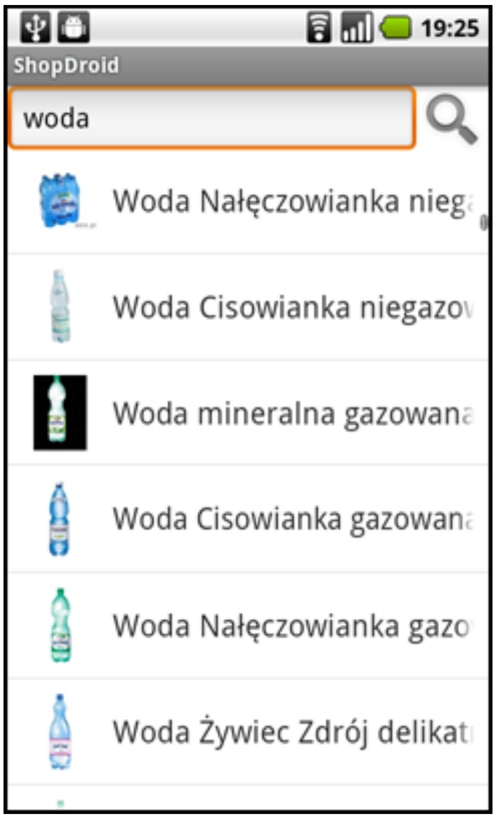


Figure 3.22 Product results when searching by name

A simple GET request to the server returns the results in XML format. The XML structure is presented in Table 3.4. The elements are self-explanatory and do not need any additional comment.

This screen also has a small input box in the top in order to let the user search again without moving back to the previous screen. The list of products is basically a ListView, with a custom adapter, which is in the *ProductAdapter.java* file inside the *adapters* package. This adapter was written with efficiency in mind, so that the interface will not freeze on the user, even if there are plenty of records.

The adapter will load first the product names that are currently on the user's screen. Just after that it will spawn threads to download and show the thumbnails for each product. This only occurs for the elements that can be seen in the screen, once the user scrolls down, then new thumbnails will be downloaded. However it will not download a new thumbnail if it has already been downloaded previously. There is also a special thread spawned to download the thumbnails of the products that are not seen in the current screen. This thread will stop when the user is scrolling down the list in order to avoid the lagging of the interface. It will continue after a period of time of idling, that is the user not touching the screen at all.

Table 3.4 Products list XML structure, *Violet* means that it is a variable

```
<?xml version="1.0" encoding="UTF-8" ?>
<search_result>
 <product>
   <id>product id</id>
   <name>product name</name>
   <image>file name</image>
   <barcode>product barcode</barcode>
 </product>
</search_result>
```

3.5.2 Search History

When the user starts typing the name of the product, suggestions might appear as a dropdown as shown in Figure XX. Whenever a user tries to search a new product, the name will be added to a SQLite database, which is the same as the one for the general history. The table for this is created with the following SQL statement:

```
CREATE TABLE search_history( _id INTEGER PRIMARY KEY AUTOINCREMENT, product TEXT NOT NULL, word TEXT UNIQUE NOT NULL);
```

Table 3.5 shows in details the table created for this feature.

The widget that allows having this kind of dropdown list is the AutoCompleteTextView widget. This has the option to set an adapter from which it loads the records of the table and displays them to the user.

Table 3.5 Search history table details

Column	Type	Integration relations	Comment
_id	integer	primary key, auto-increment	auto-increment primary key, identification number
word	text	unique, not null	product name

3.6 Store locator

Users have also the option to get the stores nearest to their current location. This is presented in an interactive way showing a map, which is provided by Google, and in that map the stores are shown as small dots. An example of this is shown in Figure 3.23.

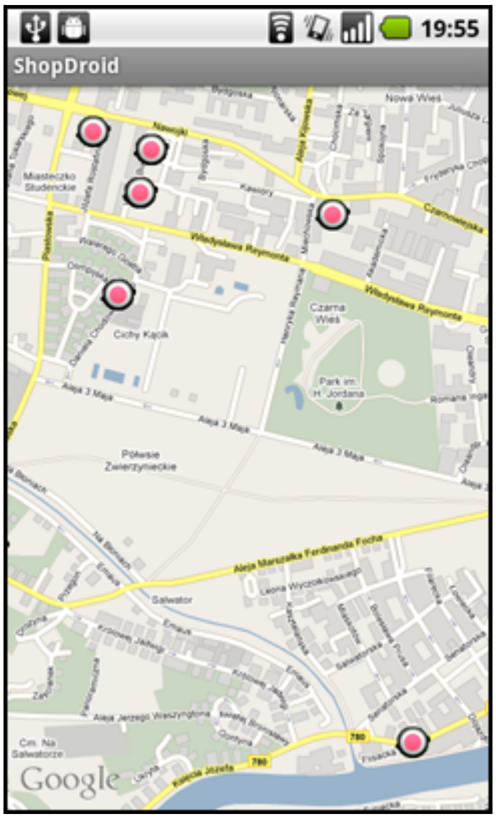


Figure 3.23 Store locator

The code for this feature is in the *StoreLocator.java* file inside the source folder. ShopDroid gets the values from the server, using a simple GET request. An example of this request is shown below:

```
http://shopdroid.ggajewski.2be.pl/getstores.php?&lat=50.10784&long=19.95241&maxl
ong=19.95241&maxlat=50.10784&minlong=19.84942&minlat=50.00811
```

Latitude and longitude are taken from the device's current location (either from GPS or network provider). The number of stores returned is limited by the *maxlat* (maximum latitude)*, maxlong* (maximum longitude)*, minlat* (minimum latitude) and *minlong* (minimum longitude) values. This is done to limit the size of the XML response, and also to show the stores to the user faster. This limits are two screens more for each side, that means that the user will be able to scroll up, down, right or left, for two more screens (which is the horizontal or vertical length of the map). After the limit is passed in any direction, then a new request is send to the server with new values.

Android can easily return the latitude and longitude spans of the current visible area. These values may vary depending on the device, because of different screen sizes. Once these values are obtained, the next step is to obtain the latitude and longitude of the map center, which can also be obtained from the system. Finally, the latitude or longitude values are added respectively to the span values, which are multiplied by 2.5 (This is half the vertical or horizontal length of the map plus two times that length). An example of this procedure is shown in (3.3).

$$lat_{max} = lat_c + 2.5 \times lat_s \tag{3.3}$$

where lat_c - latitude of the center of the map, lat_s - latitude span of the map, lat_{max} - maximum latitude

The XML structure of the response is presented in Table 3.6.

Table 3.6 Stores list XML structure for stores list. *Violet* means that it is a variable

XML structure	Notes
<?xml version="1.0" encoding="UTF-8" **?>**	XML declaration
<stores>	root opening tag
<store>	store opening tag, could be more than one
<store_id>*store id***</store_id>**	
<storename>*store name***</storename>**	
<address>*store address***</address>**	
<city>*city where store is located***</city>**	
<zipcode>*store zip code***</zipcode>**	
<lat>*store location latitude***</lat>**	
<long>*store location longitude***</long>**	
<distance>*distance to store***</distance>**	distance is calculated from a given location
<opening>*store business hours***</opening>**	format: [opening hour]-[closing hour]; “-1” when store is closed
<phone>*store phone***</phone>**	
<logo>*file name***</logo>**	file name for store logo/picture image
<logosrc>*store logo source***</logosrc>**	
</store>	store closing tag
</stores>	root closing tag

3.6.1 Calculating distance

The distance between the user’s location and each store is calculated in the server side. It does not calculate the distance for all the records in the server database; it is limited by the maximum and minimum latitude and longitude values. This is made to reduce the server load.

A great-circle distance is used as an approximation for the distance between two locations. In order to explain in more details this calculation, the concept of the great-circle needs to be explained.

A great circle is a section of a sphere which contains a diameter of the sphere. The shortest path between two points on a sphere, also known as an orthodrome, is a segment of a great circle [22].

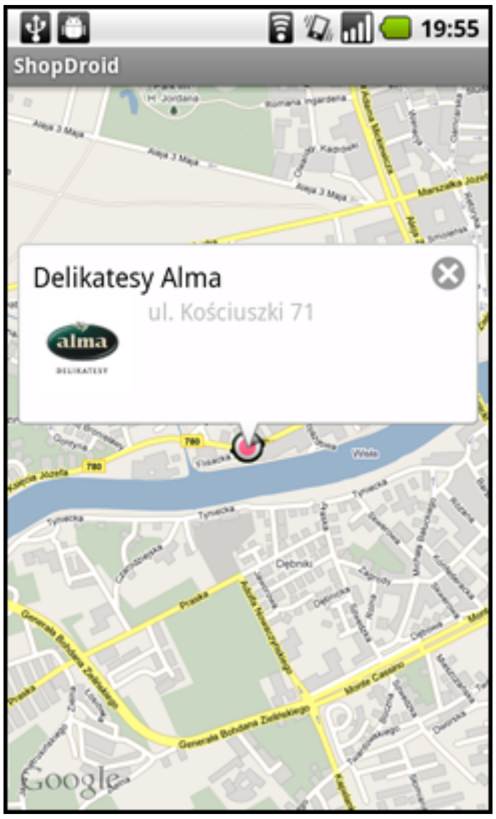
Given two points with latitude and longitude in radians, A (lat_A, $long_A$) and B (lat_B, $long_B$), the radius of a sphere R, the distance between these points d is calculated using (3.4).

$$d = \cos^{-1}(\sin lat_A \sin lat_B + \cos lat_A \cos lat_B \cos(long_B - long_A))R \tag{3.4}$$

The Earth is not a sphere, so in order to use this expression, a spherical approximation of the Earth is used, and the mean radius of it is 6371 km.

3.6.2 Store Details

Users can tap on any store dot to bring a bubble dialog with some more details about the store, like the store name, address and the store logo whenever available, this is shown in Figure 3.24. The fetching of the store logos works in a similar way to the one of the manual search of products. In this case, a thread is spawn as soon as the xml response is parsed, that starts downloading the logos of all the stores included in the xml response. When the bubble dialog of a particular store pops up, the logo fetched shows too.



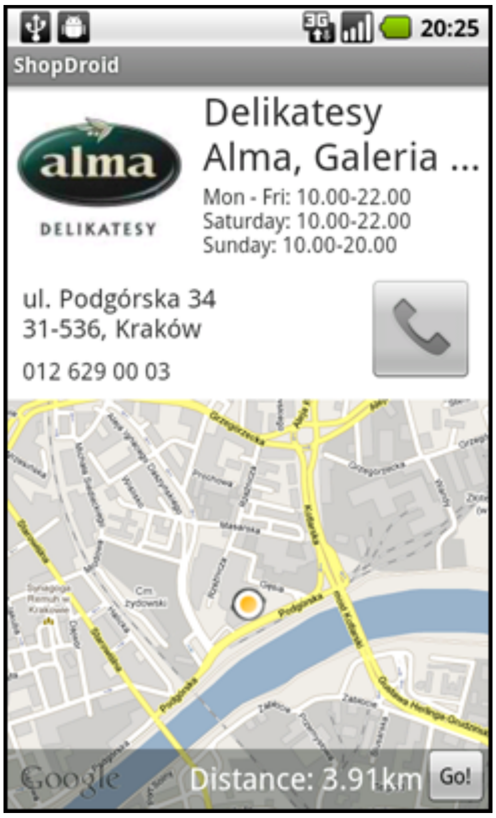
Delikatesy Alma
ul. Kościuszki 71

Figure 3.24 Bubble dialog showing some store details

Users have also the possibility to tap on the bubble dialog to bring a new screen with many more details of the selected store. The new screen, as shown in Figure 3.25, presents the user the open hours of the store (if available), the phone number of the store, which the users can call straight way by pressing the call button. There is also a small map embedded which shows the location of the store, and a "Go!" button which takes the user to the Google Maps application and shows him the directions on how to get to the store from his current location. In this case, ShopDroid will pass to the Google Maps application the user's coordinates as the source address and the store's coordinates as the destination address. This is done via Intent as shown below:

```
Intent intent = new Intent(android.content.Intent.ACTION_VIEW,
"http://maps.google.com/maps?saddr=USER_LAT,USER_LONG&daddr=STORE_LAT,
STORE_LONG");
```

Android will correctly detect the URL (Uniform Resource Locator) and will launch the Google Maps application. The code for the screen with the store's information is in the *StoreDetails.java* file.



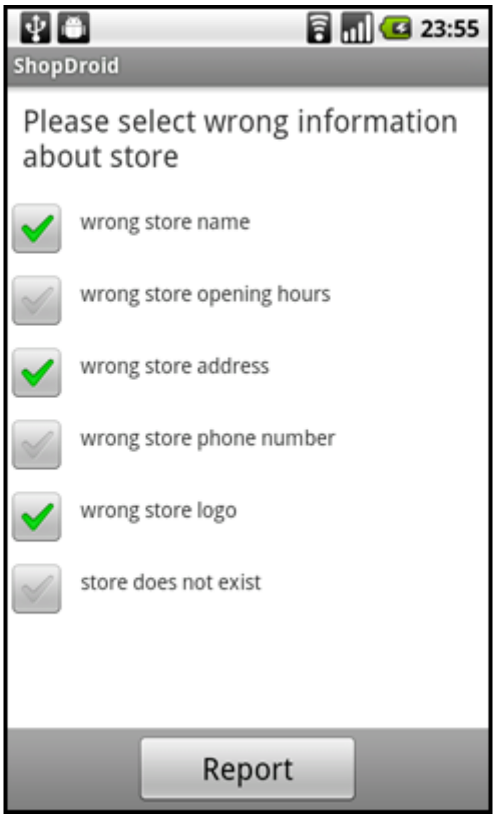
Delikatesy Alma, Galeria ...
Mon - Fri: 10.00-22.00
Saturday: 10.00-22.00
Sunday: 10.00-20.00
ul. Podgórska 34
31-536, Kraków
012 629 00 03
Distance: 3.91km

Figure 3.25 Store details screen

3.6.3 Reporting wrong information

Users can also report any wrong information presented within the store details screen, this is shown in Figure 3.26. They can report the following wrong items of the store:

- name
- opening hours
- address
- phone number
- logo
- if it does not exist at all



Please select wrong information about store

[✓] wrong store name
[ ] wrong store opening hours
[✓] wrong store address
[ ] wrong store phone number
[✓] wrong store logo
[ ] store does not exist

Figure 3.26 Reporting store wrong information

Users need to be logged in the service in order to be able to report wrong information. However, all the wrong reports must be confirmed by an administrator (more on this on section 3.9).

3.7 Product screen

The main feature of ShopDroid is price comparison. The user can see the product name, and prices in different stores as the result of a successful barcode scanning or manual product search. There is one condition for this: the product must exist in the ShopDroid database. The `Product` class, whose code can be found in Product.java file, controls the product screen. This screen contains several information about products such as: product name, product picture, nearest store name with product price (if available), image button "like this product", total number of users who clicked on like button, user reviews (if available). Figure 3.27 shows an example product screen containing prices in two different stores, and one product review.

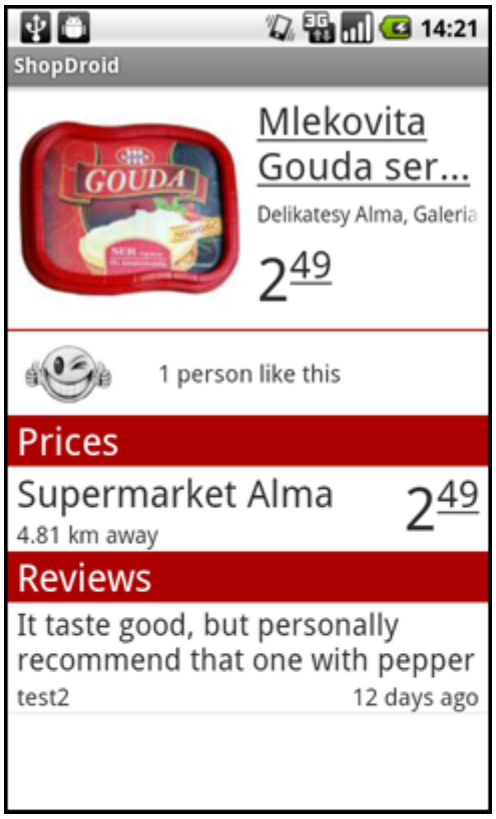


Figure 3.27 Example product screen

At the beginning of Product Activity it connects to the ShopDroid server to download the product related information. This connection is realized in an asynchronous task using HTTP POST method. The request contains these values: product barcode, number of prices shown limit, reviews limit, location latitude and longitude (only if location is available). Additionally if the user has connected his account, the request is signed using the OAuth protocol. To keep the product screen clearer, the number of prices and reviews is limited. The maximum number of prices presented is five and the maximum number of reviews is three. If there are more a "View more..." button appears at the end of the list. The ShopDroid server queries the database for a product with the given barcode, if the product cannot be found it returns a XML file presented below:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<error>404</error>
```

In the case that the product exists in the database, PHP executes another queries to the database. One of the queries returns a list of prices in different stores limited by the stores limit parameter, the other query returns a list of user reviews limited by reviews limit. If the request is signed then the user ID is obtained and the script checks in the database if the user has selected the "like button" before. The prices query has two versions depending on that if the user location is known. The ShopDroid project uses a dedicated script written in PHP to create XML responses. It simply takes an array of data and puts it between the appropriate tags. Table 3.7 shows XML structure for the product details.

After receiving the product XML from the server, the Product activity parses it and displays it on the screen. Another asynchronous task is run to download the product picture (if there is any).

3.7.1 XML Parsing

Android provides two different ways to parse xml documents. These are:

- Simple API for XML (SAX)
- Document Object Model (DOM)

These two parsers work exactly the same as in Java. The main difference is that DOM uses more memory, since it loads first the entire document into the memory, while SAX takes less memory but it takes more time to implement. A basic test was made to test the speed performance on Android. 105 complex records, using the same structure as the prices XML structure, were parsed. Each of the tests was ran ten times to help factor out any other events that might be going on with the system. The average results are shown in Figure 3.28.

Table 3.7 Product details XML structure

XML structure	Notes
<?xml version="1.0" encoding="UTF-8" ?>	
<product>	root opening tag
<id>*product id***</id>**	
<name>*product name***</name>**	
<image>*product image***</image>**	
<imgsrc>*product image source***</imgsrc>**	
<userlike>*user like***</userlike>**	1 if user clicked product like button, otherwise 0
<peoplelike>*people liking product***</peoplelike>**	A total number of users which have clicked on product like button
<added_by>*user name***</added_by>**	name of the user who added product
<store>	store opening tag, almost like for store list, except for the price
<store_id>*store id***</store_id>**	
<storename>*store name***</storename>**	
<address>*store address***</address>**	
<city>*city where store is located***</city>**	
<zipcode>*store zip code***</zipcode>**	
<lat>*store location latitude***</lat>**	
<long>*store location longitude***</long>**	
<price>*price***</price>**	product price in that store
<distance>*distance to store***</distance>**	
<opening>*store business hours***</opening>**	
<phone>*store phone***</phone>**	
<logo>*file name***</logo>**	
<logosrc>*store logo source***</logosrc>**	
</store>	
<review>	review opening tag
<review_id>*review id***</review_id>**	
<author>*user name***</author>**	name of the user who wrote that product review
<date>*review date***</date>**	minutes, hours, days, months or years since review adding time
<text>*review text***</text>**	
</review>	
</product>	

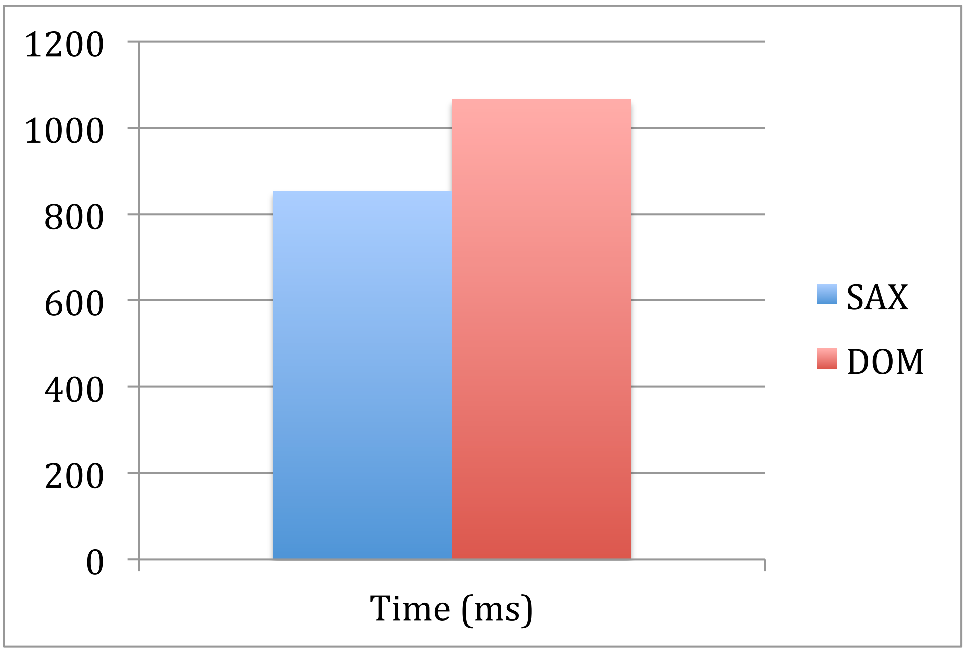


Figure 3.28 Chart showing XML parsing results

On average, the SAX parser completed the tests within 854 ms, whereas the DOM parser completed them within 1066 ms. This tests were taken in the Android device: Nexus One, with the following features [23]:

- CPU: 1 GHz Qualcomm QSD 8250 Snapdragon ARM
- Memory: 512 MB RAM

These results show that SAX parsing on Android is a bit faster than DOM parsing. In addition to be faster it uses less memory, and that is the reason why SAX parsing was chosen for this project. All the parsers implemented are inside the *parsers* folder.

3.7.2 Multiple Lists

Trying to have two or more lists in Android presents some scrolling issues, especially when all the elements do not fit the screen and they need to be scrolled. The problem appears because the system will not know which the user wants to scroll, either the entire screen or one of the lists. To resolve this issue, an external library (*CWAC-MergeAdapter*) was used.

This library implements a custom adapter, to which other adapters can be added. This custom adapter uses only one `ListView` widget, thus making the scrolling issues disappear. Not only adapters can be added, also views, like the top part describing the product, can be added.

3.7.3 Product prices

Prices are parsed from the xml response, which are sorted by proximity to the user (if the user location is known). By sorting this way, the first result will be most likely the store where the user is at the moment, if the store is in the database. The rest of the results are sorted by price, from cheapest to most expensive, in the client side.

ShopDroid conveniently does not show any currency symbol because the prices will depend on the location of the user, and thus will be in the user's currency. This avoids any kind of currency conversion, and the server will always list the prices in the right currency, although it does not show the currency symbol.

The first result (Figure 3.29) is shown at the top part of the product screen, together with the product name and picture. This is to give more emphasis to that result. If the user taps on this result, the store details screen will appear.

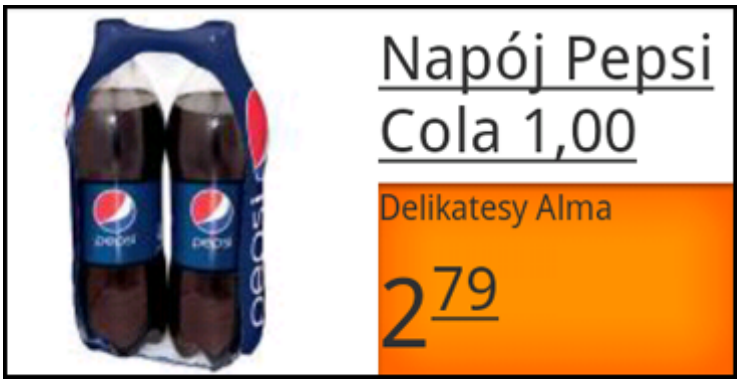


Figure 3.29 Price result of the first store

The rest of the results appear below the "like it" part. These results are part of a custom adapter, which is implemented in the *ProductAdapter.java* file. This adapter allows the rest of the prices to be loaded into the main `ListView` of the product screen. An example of an individual price result is shown in Figure 3.30. The user can also tap on any of these results to be taken to the store details screen.

The elements of each result are the store name, the price and the distance to the store. The distance is calculated in the same way as described previously in section 3.6.1. However the maximum and minimum latitude and longitude are calculated in another way. There is a default radius threshold of 70 km (this can be changed by specifying the value in the client).

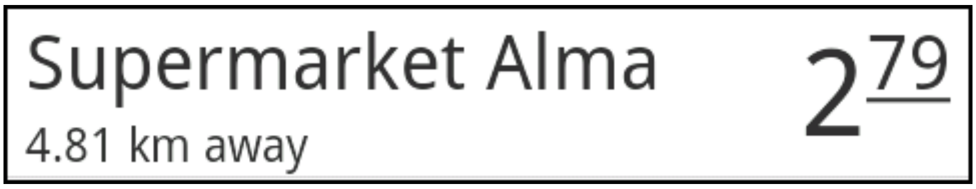


Figure 3.30 Example of the price result

An approximation of the maximum and minimum latitudes to be searched can be found using (3.5)

$$lat_m = lat_u \pm \frac{r}{R} \times \frac{180}{\pi} \tag{3.5}$$

where *lat*$_m$ - maximum or minimum latitude, *lat*$_u$ - current latitude of the user, *r* - radius threshold in km (70 km by default), *R* - mean radius of the Earth (6371 km)

Calculating the maximum and minimum longitudes looks like (3.6)

$$long_m = long_u \pm \frac{r}{R \cos lat_u} \times \frac{180}{\pi} \tag{3.6}$$

where *long*$_m$ - maximum or minimum longitude, *long*$_u$ - current longitude of the user, *lat*$_u$ - current latitude of the user

This will be the first cut of the SQL query, in order not to calculate the distance for every record. Once the distance is calculated for the records that obtained from the first cut, the distance is checked again against the threshold radius, and the final results are obtained.

If there are more than five results, then a footer “View All” appears in the prices list. This takes the user to another screen, which contains all the prices for a product. This is implemented in the *PricesAll.java* file, and is much more simpler because it only contains one `ListView`. Each element of the list looks exactly the same as in the product screen, and the same adapter is used to load the elements.

3.7.4 Product reviews

Below the price results, reviews are shown if available for the product. This is shown in Figure 3.31. The reviews are also obtained from the xml response and loaded to the `ListView` in a similar way to the prices results; another custom adapter is implemented in the *ReviewsAdapter.java* file. Whenever there are more than three reviews, a footer "More" is shown, and this takes the user to a new screen which shows all the reviews available for that product. The code for this new screen is in the *ReviewsAll.java* file.

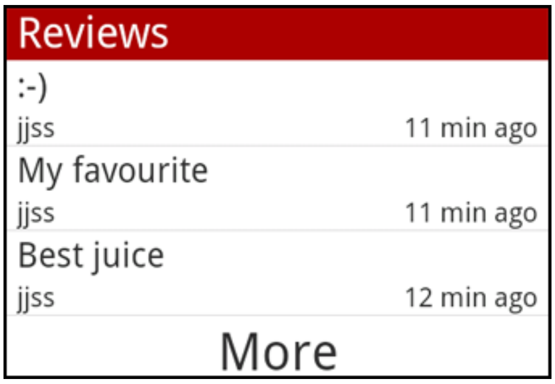


Figure 3.31 Example of reviews

Each review shows also the author of the review and how much time ago the review was posted. Any registered user can mark any review as spam if that is the case by long-clicking on any review and a menu will appear as shown in Figure 3.32. This also requires an administration to confirm.

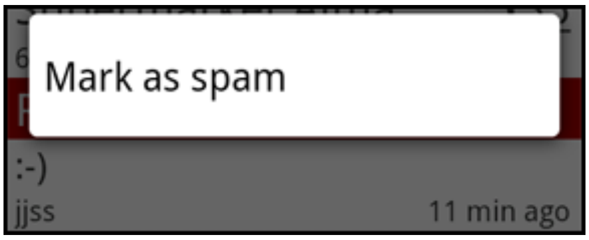
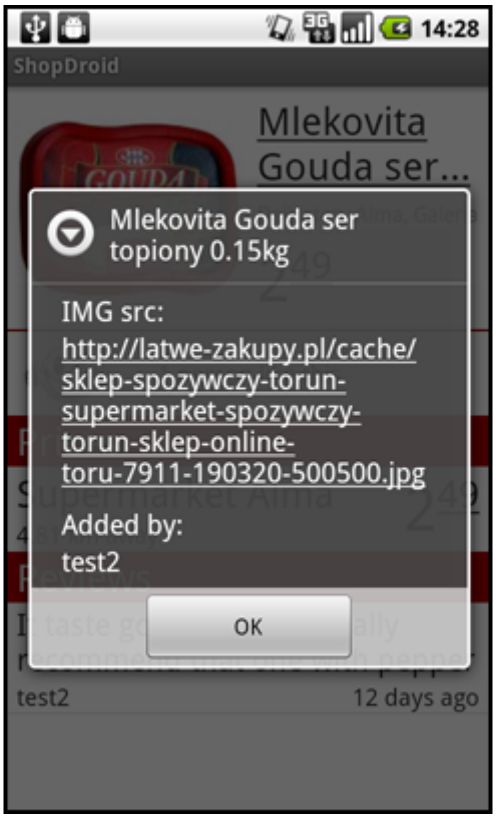


Figure 3.32 Marking a review as spam

3.7.5 About product

The About Product screen is useful for describing product information such as: full product name and volume, product image source, and who added this product. It is showed in a Dialog widget (Figure 3.33). Product image source can be web URL when the picture was downloaded from a web page or the user name when the picture was taken and uploaded by a user.



Mlekovita Gouda ser topiony 0.15kg

IMG src:
http://latwe-zakupy.pl/cache/sklep-spozywczy-torun-supermarket-spozywczy-torun-sklep-online-toru-7911-190320-500500.jpg

Added by:
test2

Figure 3.33 About product screen.

3.7.6 Reporting wrong information

Users can also report any wrong information about the product screen (Figure 3.34); this is similar to reporting wrong information about store screen. They can report the following wrong items of the product:

- name
- volume

Only these values can be reported because they are not editable by users (only administrators can change those values). In the future users could report other information if products would have more information available (like nutrition facts).

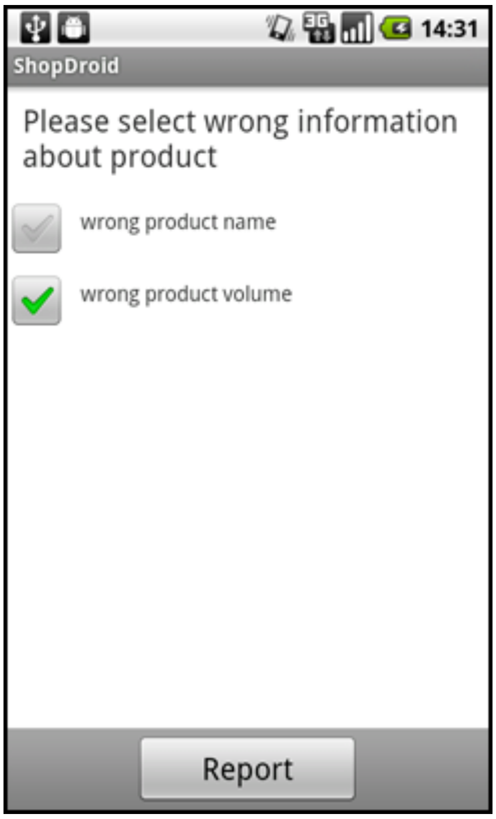


Figure 3.34 Reporting wrong information for a product

3.7.7 Registered user actions

Some actions in the Product screen are restricted to registered users - users who successfully connected their accounts with ShopDroid. These actions are: changing the product picture, editing the product price, adding a product review, adding a product price, adding a new product, adding a new store, like product button. The reason of this restriction is to protect product information from spammers or dishonest competition. Spammers can add reviews not related with product or products/stores that actually do not exist. Dishonest competition can write bad product reviews or lower the prices. The protection mechanism will be described later in this document.

Add/change product picture

Each product can have one picture; if it is not set a “no image” icon is displayed (Figure 3.35).

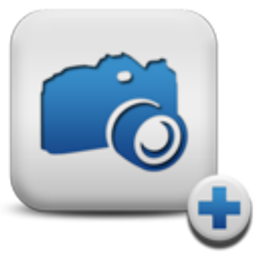
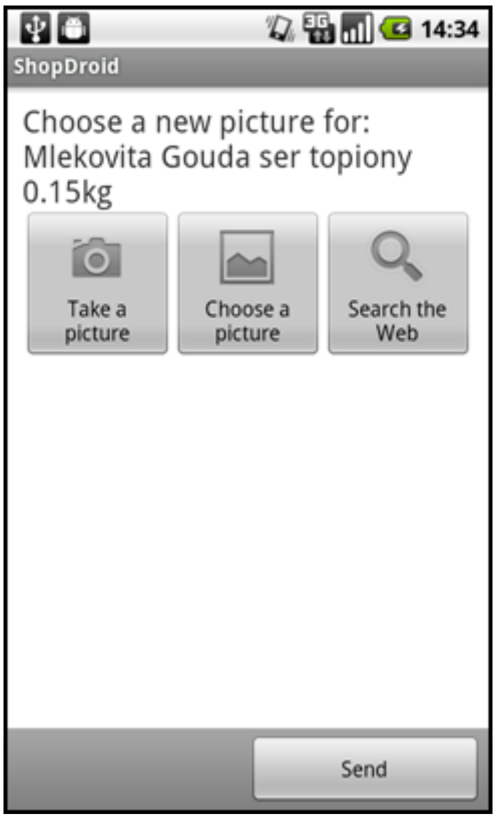


Figure 3.35 No product image icon

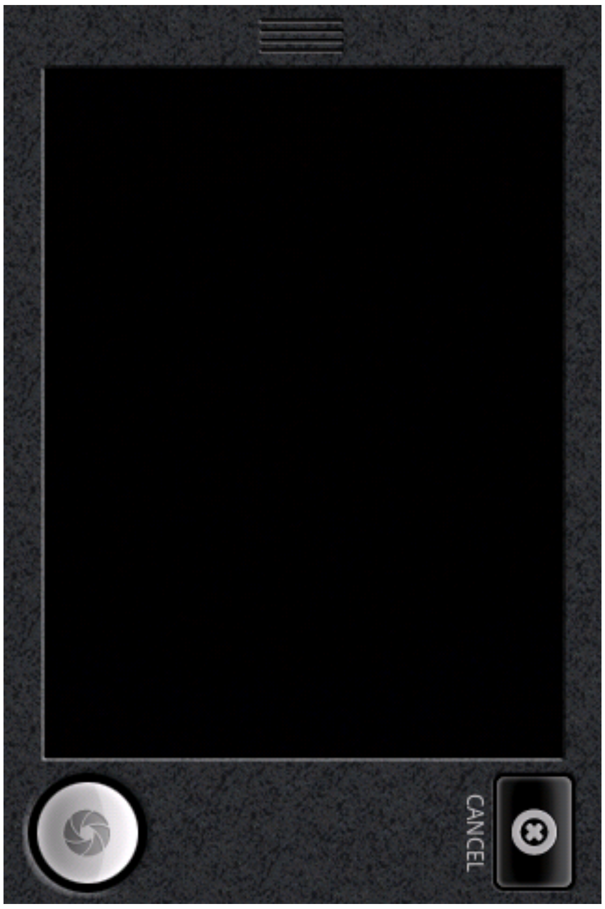
When users perform a long click on a product picture/no image icon they can change/add the product picture. A menu pops up with the option “Add a new Photo” and after selecting it, the picture upload menu shows up (Figure 3.36).

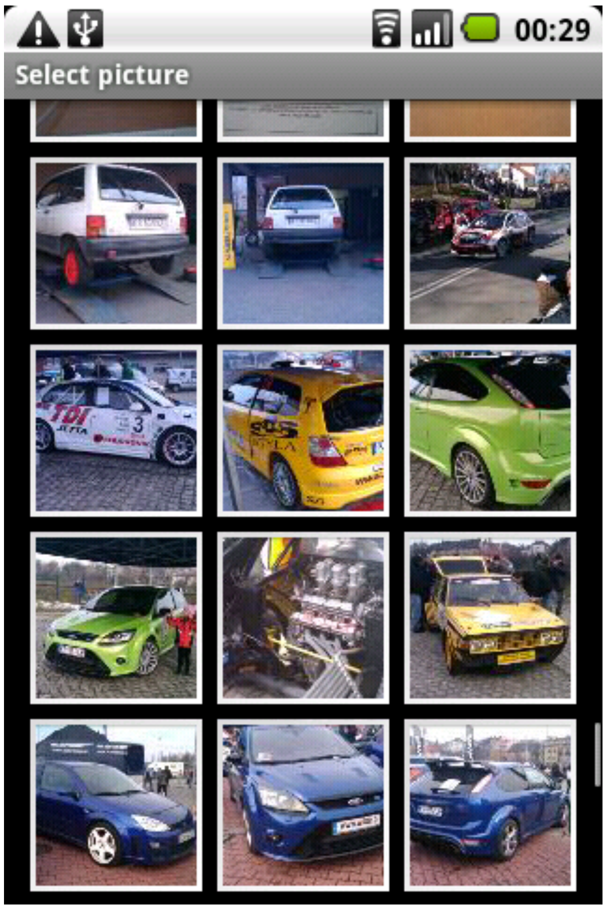


Choose a new picture for:
Mlekovita Gouda ser topiony
0.15kg

Figure 3.36 Picture upload menu example.

This menu is a part of the `AddProduct` activity, which will be described in details later. Users have three options to upload a new photo: Take the picture themselves, choose the picture from the phone gallery, search the picture in the Web using Google image search engine. The first two options run a native android application via intents. Intents can be used to launch an activity, but also an Intent is a mechanism in Android OS which allows other application to do some work and return a result. This feature is very useful as ShopDroid does not have to implement all the taking pictures code itself, but only run the native Camera application to take a picture and return it into application. Figure 3.37a shows taking a picture screen and Figure 3.38b shows choosing the picture from gallery action.





(a) (b)

Figure 3.37 a) Taking picture screen, b) Choosing picture from gallery screen.

Last option - search picture of the Web implements a Web browser, which opens the Google image search site with results for the product name (Figure 3.38). Users have ability to change the search phrase, browsing result pages, but they cannot leave this page. The code for this can be found in *SearchImages.java*, it detects when users click on the picture and returns a picture web URL.

Once the picture is selected, the user is able to hit the send button. The picture is uploaded in an asynchronous task and an uploading dialog is displayed on the screen. If the picture was successfully send to the server, a screen is displayed with a message -

"Picture successfully added. Thank you" and a return button. If a problem appeared, an error message is shown to the user. In that case, users can either try to upload the picture again or return back to the Product screen. On the server side the picture is resized and the database is updated.

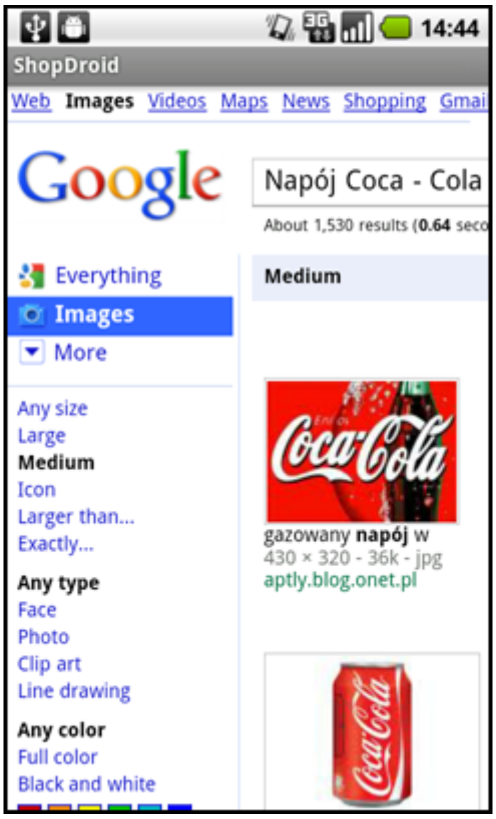


Figure 3.38 Picture Web search screen

Like button

The user has the ability to "like" a product by clicking on the image button. This system can measure product popularity with a total number of users "liking the product". There where three options considered to measure product popularity: stars rating in a scale of 0-5, buttons like/dislike, like button. The last option was implemented, as it is the simplest system in use so it has the highest probability that user will use it. On the right side of the like image button is a text, which represents the total number of the users who likes that product. The text includes whether or not the user likes that product and correct declension. Figure 3.39 presents two different like button states.

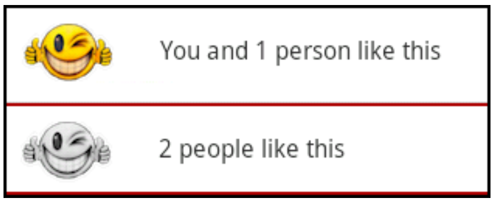


Figure 3.39 Like line examples.

Edit price

ShopDroid project allows registered users to edit the product price. Nowadays many stores have special offers, discounts and other promotions that result in frequent price changes. Editing price is simple and fast. To change the price, the user has to long click on the store/price row, and the menu Edit price will come up (Figure 3.40). Choosing it opens a dialog (Figure 3.41) where the user is asked for new price. OK button confirms it, and the price is uploaded to the server, where PHP scripts check OAuth sign and update the product price inside the database.

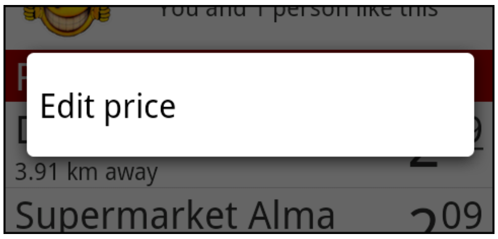


Figure 3.40 Edit price menu

Add review

Registered users can write product reviews. It is a useful addition, which allows users to share feelings about the product. Composing review screen (Figure 3.42) is accessible from the product options menu. The review text is limited to 500 characters. The send button uploads the review to the server, where it is saved to the database. Again, the message has to be signed with a valid OAuth signature. After the successful review submission, the review text is automatically added to the product view without the need of re-downloading product information.

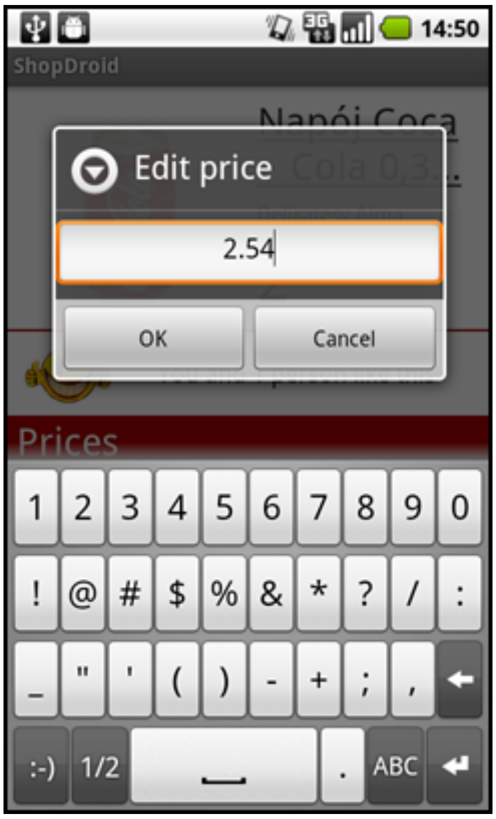


Figure 3.41 Edit price dialog

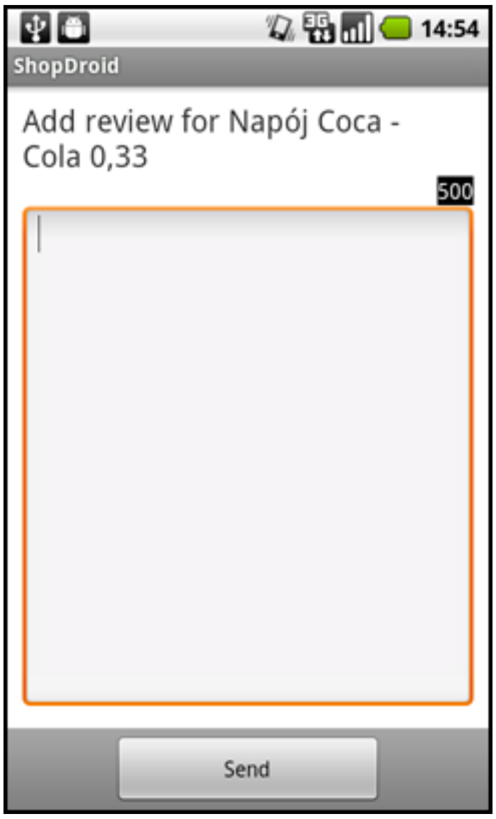
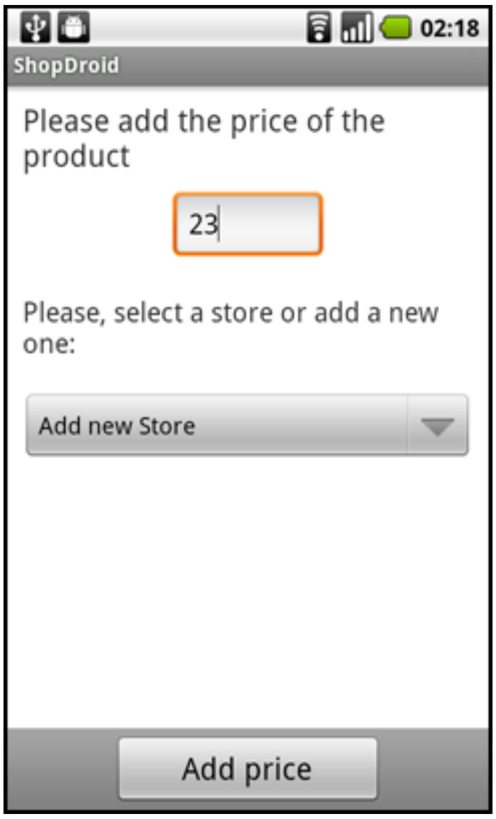
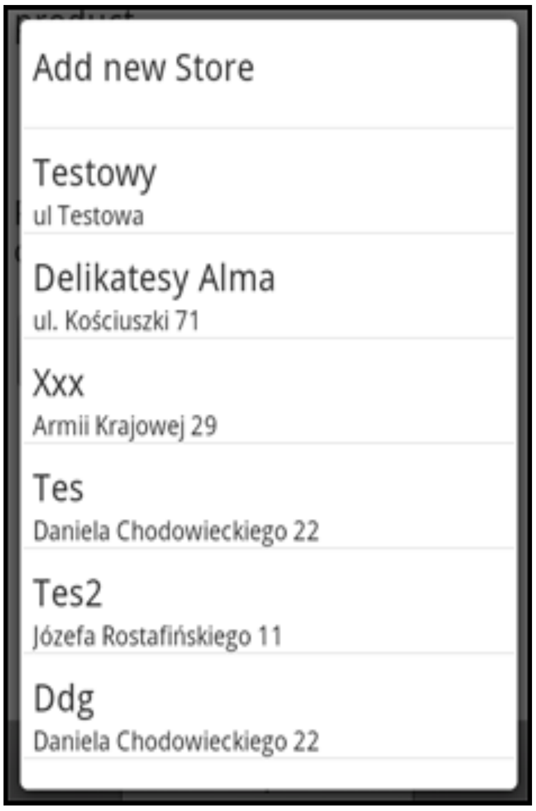


Figure 3.42 Review composing screen

Adding a new price

Users can also add a new price of a store that does not show in the price results. Users can add a new price by selecting the option from the menu. This will take the user to a new screen as shown in Figure 3.43a. In this new screen the user can input the price and select an existing store by clicking on "Add new Store". This will pop up a list with the stores that are in the user's proximity (see Figure 3.43b). However, if none of the stores in the list are the intended ones, or there is no store nearby, the user can choose to add a new store.


02:18
ShopDroid
Please add the price of the product
23
Please, select a store or add a new one:
Add new Store
Add price



(a) (b)

Figure 3.43 a) Adding a new price, b) Stores list near the user's location

After the user adds the new price, and if an existing store is selected, a confirmation message will appear and the user will be return to the previous (Product) screen.

Adding a new store

To add a new store, users have to go through seven simple steps. The first step is adding the name of the store. Adding the city where the store is located is the next step. ShopDroid will try to get the city from the users' location, so they will not need to write it. This functionality is provided by Android, which uses Google's database. The third step is adding the address. Again, this will also be obtained from the users' location (however this might not be very accurate) and users will only need to manually add the address if incorrect, or if it could not be obtained from the system. These first three steps are mandatory and users will not be able to get to the next steps if this information is not provided.

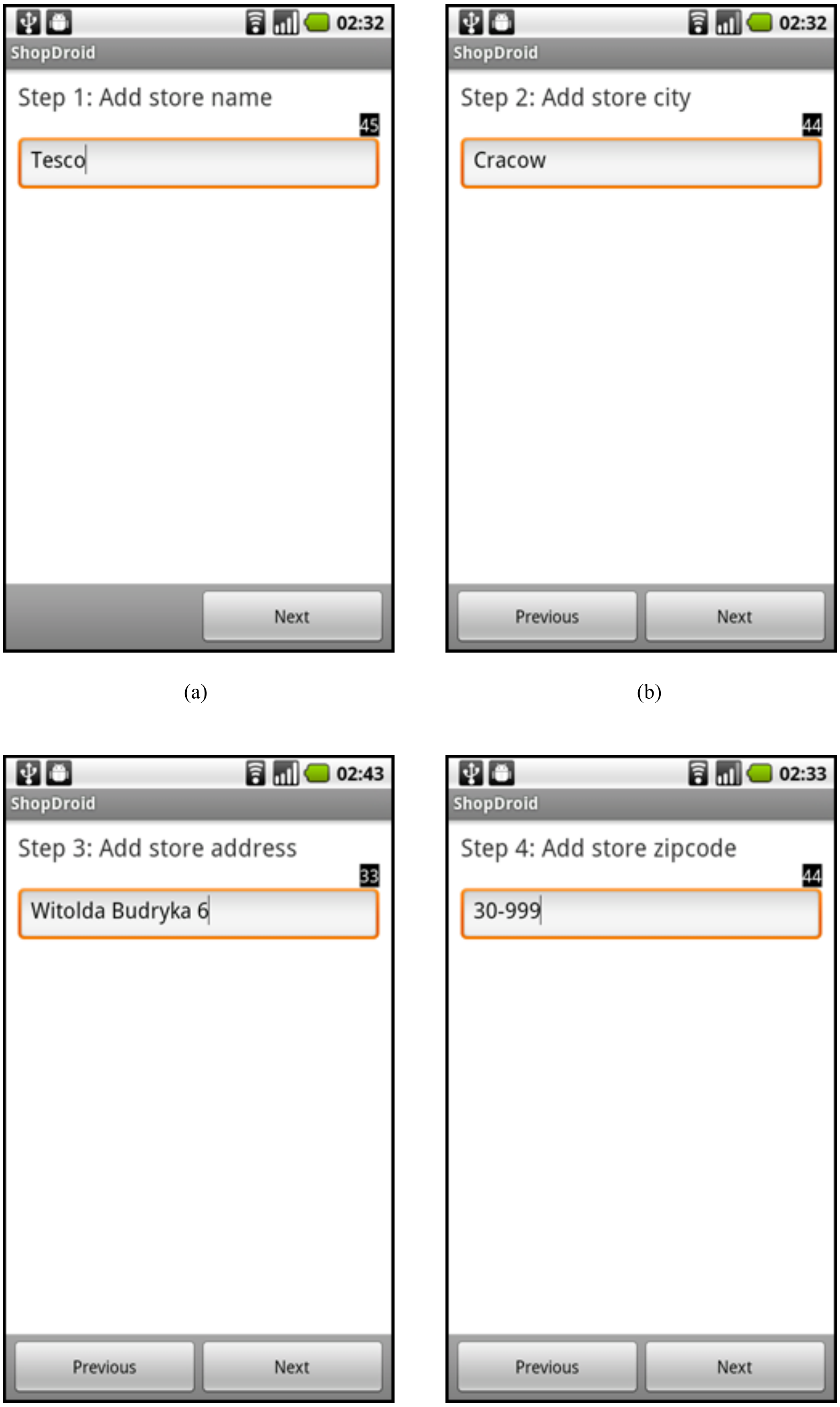
The coordinates of the store added depend on the address provided by the user and not the user's location. The server then will check the address with Google's geocoding service using the following query:

```
http://maps.google.com/maps/api/geocode/xml?address=STREET,+CITY,+COUNTRY
&sensor=true
```

This query will return an xml response, which will have the geographic coordinates of the searched address.

Next steps are optional; users might just skip them without providing any information. Adding the zip code of the store is the fourth step. This too might be obtained from the users' current location. For the first four steps there is a character counter that shows how many more characters users can still input, the limit by default is 50 characters. The fifth step is to add a phone number. Users are limited to input numbers, in order not to get wrong type of information into the database. Selecting the opening hours of the store is the next step. There are two modes for this, a simple one and an advanced one. The simple mode shows only three options: Monday till Friday, Saturday and Sunday and the advanced mode allows the user to select the opening hours for every day of the week. The last step is to add a picture of the store. This is exactly the same as adding or editing the picture of the product and is described earlier in this section.

These steps are implemented as one activity in the *AddStore.java* file. There is a `ViewFlipper` widget that allows the application to change between views within the same activity. The steps are shown in Figure 3.44a – 3.44g.



Step 1: Add store name

45

Tesco

(a)

Step 2: Add store city

44

Cracow

(b)

Step 3: Add store address

33

Witolda Budryka 6

(c)

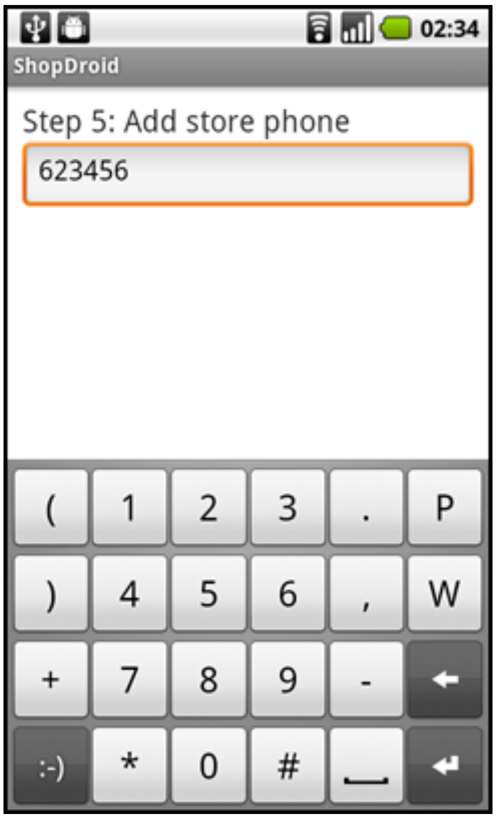
Step 4: Add store zipcode

44

30-999

(d)

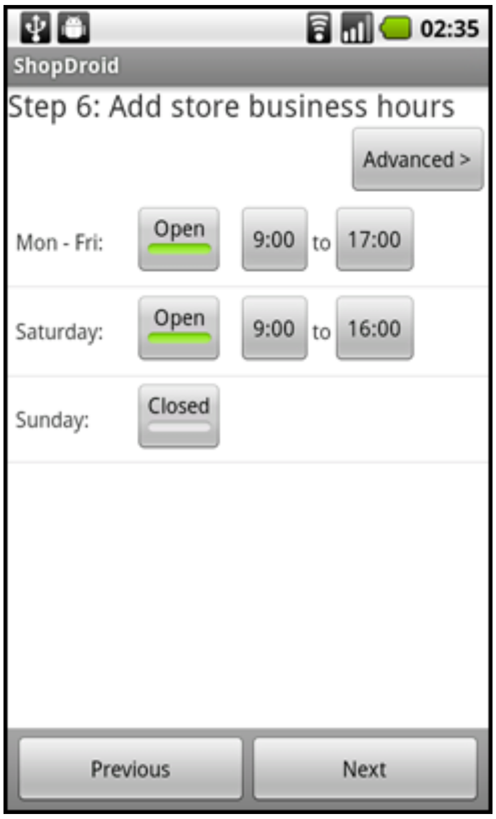
Figure 3.44 a,b,c,d) Steps 1-4 for adding a new store



Step 5: Add store phone

623456

(e)



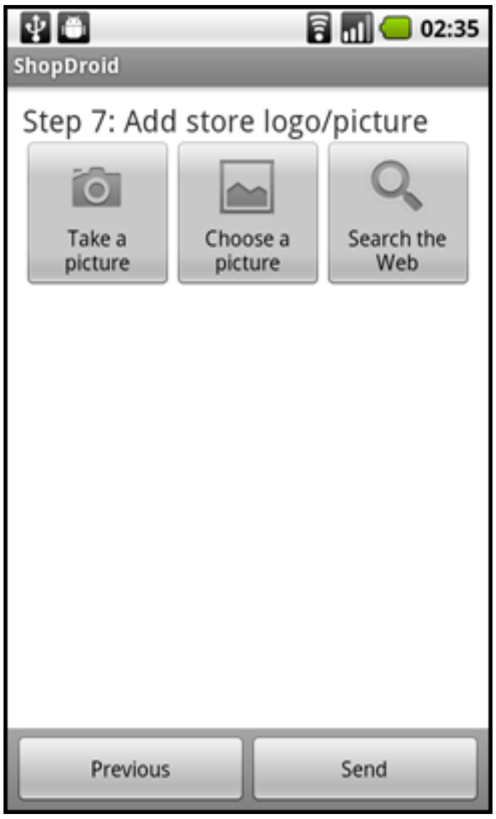
Step 6: Add store business hours

Mon - Fri: Open 9:00 to 17:00

Saturday: Open 9:00 to 16:00

Sunday: Closed

(f)



Step 7: Add store logo/picture

(g)

Figure 3.44 e,f,g) Steps 5-7 for adding a new store

3.7.8 Adding a new product

A registered user has the ability to add a new product into the ShopDroid database. This function is available when the product was scanned or manually searched by barcode and does not exist in the database yet. In this case the "Product Not Found" screen appears with two options in the menu: refresh and add product (Figure 3.45).

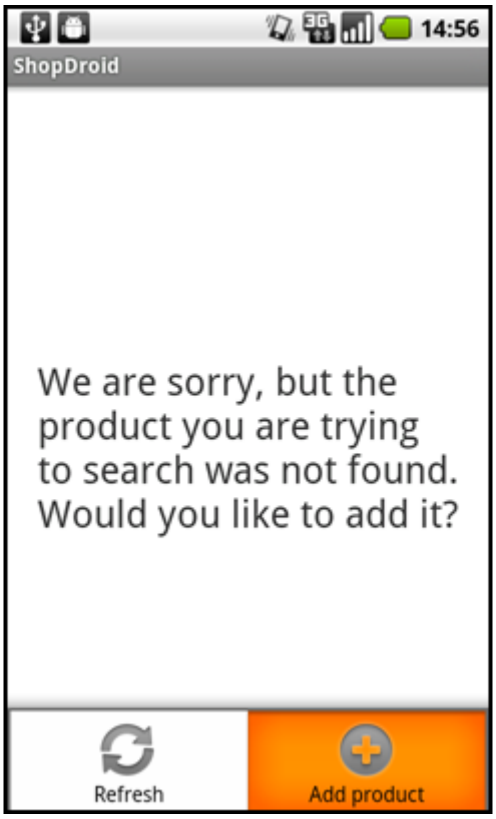
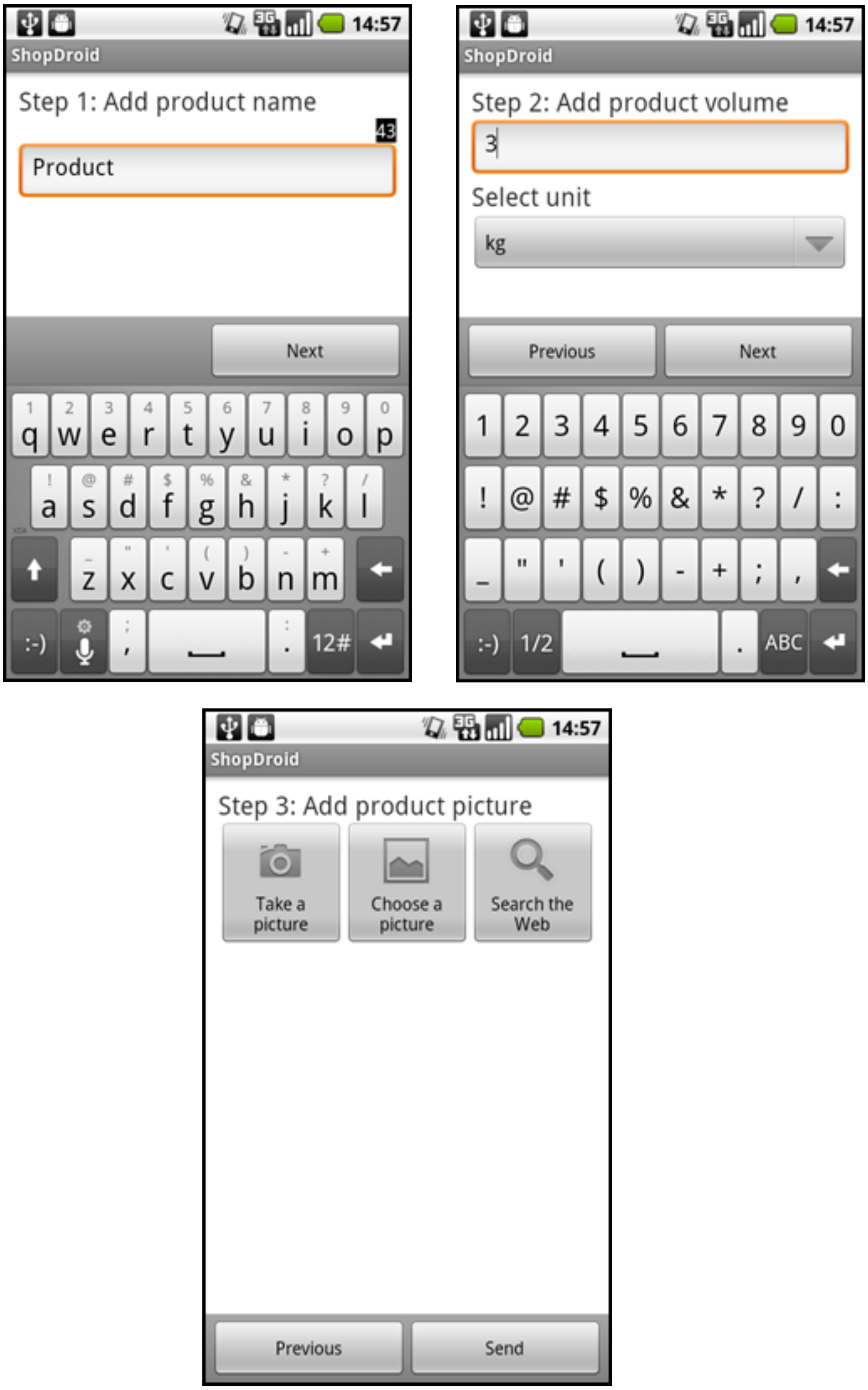

We are sorry, but the product you are trying to search was not found. Would you like to add it?

Figure 3.45 Product not found screen

Adding a product is similar to adding new store described before, but simpler and shorter. The user has to go through three steps. The first step is adding the name of the product. In the second step, the user has to fill the product volume and choose a proper unit from the list (kilograms, liters or pieces). At the moment, ShopDroid only uses the metric system for the product units, in future revisions other measurement systems could be taken into account. These two steps are obligatory. The last optional step is to choose a picture for the product, which is exactly the same as adding or editing the picture of the product and it is described in section 3.7.7. All steps are shown in Figures 3.46.



ShopDroid

Step 1: Add product name

43

Product

ShopDroid

Step 2: Add product volume

3

Select unit

kg

ShopDroid

Step 3: Add product picture

Figure 3.46 Steps for adding a new product

3.8 Users

As mentioned before a user needs to be registered to get the full ShopDroid experience. It helps to protect product information and reviews from spam robots or human with bad intentions. User reputation points system helps to detect bad users. To have really good price comparisons it is important to keep objectivity and eliminate users who provide bad/fake information. There are several authentication scenarios, but the system has to be secure and easy to use so the OAuth protocol was chosen.

3.8.1 OAuth authentication

Official OAuth specification describes the authentication process as follows [24]:

OAuth authentication is the process in which users grant access to their protected resources without sharing their credentials with the Consumer. OAuth uses tokens generated by the Service Provider instead of the user's credentials in protected resources requests. The process uses two token types:

Request Token:
Used by the Consumer to ask the user to authorize access to the protected resources. The user-authorized Request Token is exchanged for an Access Token.

Access Token:
Used by the Consumer to access the protected resources on behalf of the user. Access Tokens may limit access to certain protected resources. Service Providers should allow users to revoke Access Tokens. Only the Access Token shall be used to access the protect resources.

OAuth Authentication is done in three steps (Figure 3.47)*:*

1. *The Consumer obtains an unauthorized Request Token.*
2. *The user authorizes the Request Token.*
3. *The Consumer exchanges the Request Token for an Access Token.*

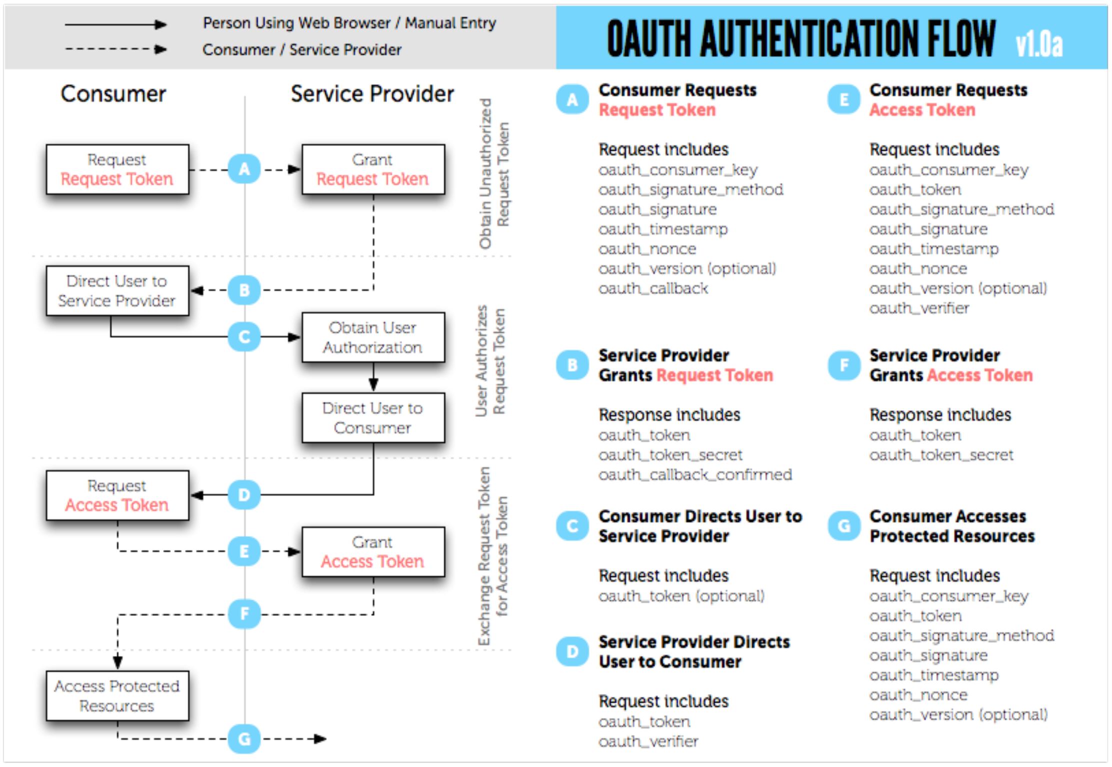


Figure 3.47 OAuth authentication flow[1]

OAuth nonce and timestamp

To prevent reply attacks the OAuth protocol defines nonce and timestamp combination. Nonce is uniquely generated for each request, it helps Service Provider verify that the request has never been made before. Timestamp is the number of seconds since January 1, 1970 00:00:00 GMT. Timestamp needs to have higher value than a previous request with the same nonce [24].

OAuth request signing

ShopDroid uses HTTP POST method to send OAuth parameters described in Table 3.8.

[1] http://oauth.net/core/diagram.png

Table 3.8 OAuth signed request parameters [24]

oauth_Consumer_key:	The Consumer Key.
oauth_token:	The Access Token.
oauth_signature_method:	The signature method the Consumer used to sign the request.
oauth_signature:	The signature
oauth_timestamp:	Timestamp
oauth_nonce:	Nonce
oauth_version:	OPTIONAL. If present, the value must be 1.0. Service Providers must assume the protocol version to be 1.0 if this parameter is not present. Service Providers' response to non-1.0 value is left undefined.

Sample ShopDroid application request parameters:

```
oauth_token="ca97f23e482cce2b6ac9683d2ce0656b04c758fc6"
oauth_Consumer_key="ba7c4903f1961c356886e4eb7141691a04bf54957"
oauth_version="1.0"
oauth_signature_method="HMAC-SHA1"
oauth_timestamp="1282773022"
oauth_nonce="-1446670766868377449"
oauth_signature="G%2F1XgGKTm7m7QUsWsgqwcHUhmoI%3D"
```

ShopDroid Service Provider supports all signatures methods specified by OAuth: HMAC-SHA1, RSA-SHA1, PLAINTEXT, ShopDroid client uses the HMAC-SHA1 method to sign requests. Signing a request is a process which encodes the Consumer secret and the token secret into the oauth_signature variable. During the signature process only oauth_signature parameter can be modified. HMAC-SHA1 method calculates the digests value as a base64-encoded string, using the signature base string as the text, Consumer secret and token secret separated by an ‘&’ character as the key. The OAuth specification defines the signature base string as follows:

The Signature Base String is a consistent reproducible concatenation of the request elements into a single string [24].

Verifying signature

To verify request’s HMAC-SHA1 signature the Service Provider takes parameters provided by the Consumer and compares the oauth_signature value with the calculated signature as described previously using the Consumer secret and token secret stored in the database.

OAuth Server Provider implementation problem

The project uses 2be.pl hosting. There was a problem with HTTP Authorization header - the PHP script could not access that header [25]. It occurs when PHP is installed and working as CGI. The solution is as simple as creating a .htaccess file:

```
RewriteEngine on
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

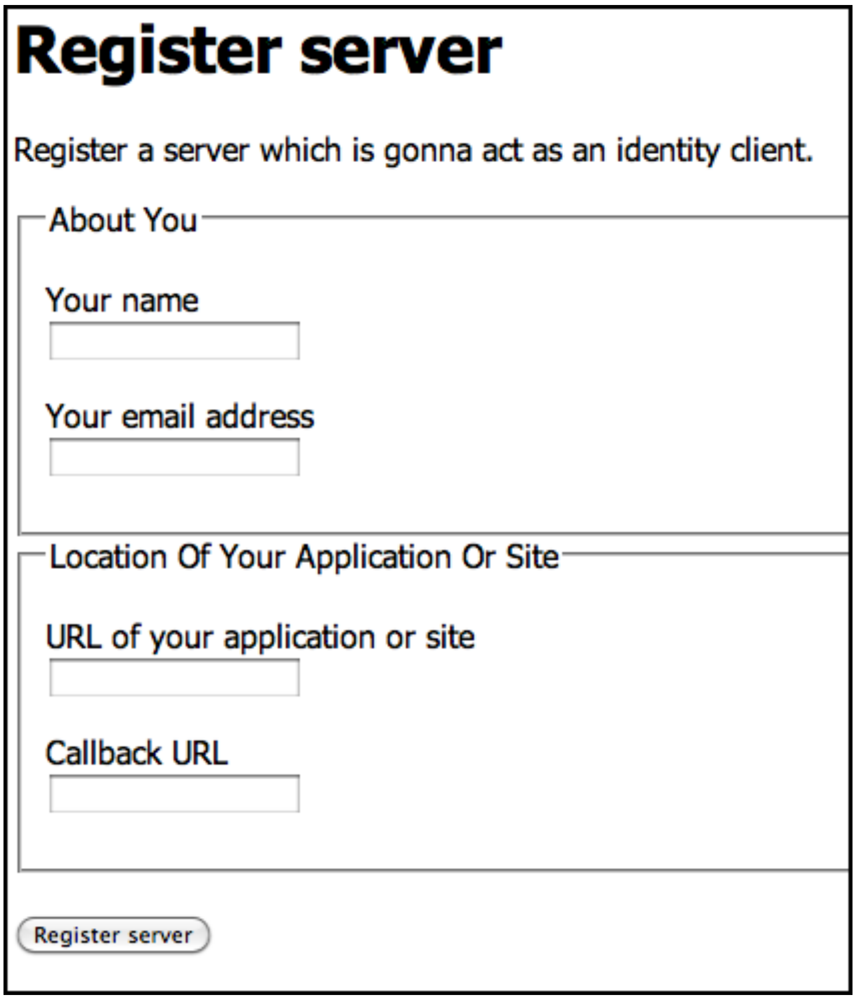
The above script rewrites HTTP Authorization header so it can be accessible by reading a $_SERVER variable in the PHP script. This solution does not have any influence on the OAuth protocol security level.

OAuth protocol security threads

One of main security threads is secure tokens and secrets exchange. The OAuth specification does not provide any mechanism, only suggests using transport-layer mechanisms such as TLS or SSL. Currently ShopDroid uses hosting 2be.pl server, which supports neither TLS nor SSL, which means token and secret transmissions stay unsecured from eavesdropping. The production server should have support for SSL or TSL.

3.8.2 Connecting user account in ShopDroid

To connect the ShopDroid application with the server (Service Provider) the Consumer key and secret are needed. It can be obtained by going to *http://shopdroid.ggajewski.2be.pl/oauth/register.php* (administrator privileges are required) and filling the registration form as shown on Figure 3.48.



Register server

Register a server which is gonna act as an identity client.

About You

Your name

Your email address

Location Of Your Application Or Site

URL of your application or site

Callback URL

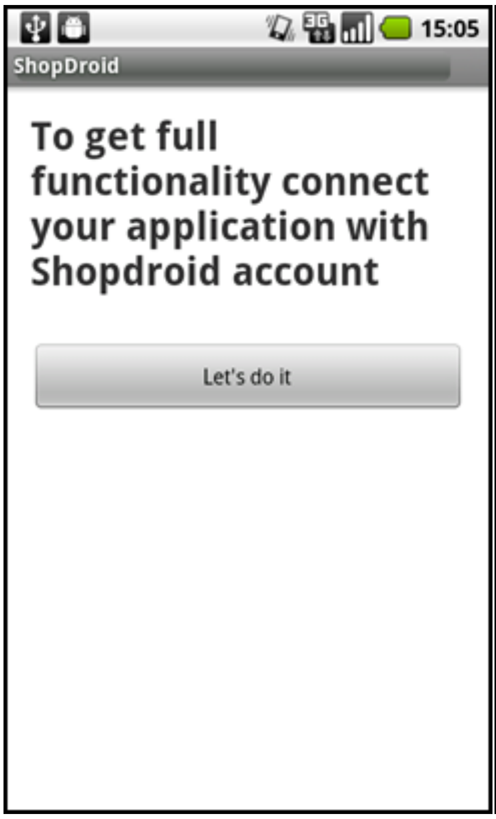
Register server

Figure 3.48 Register application form

The Consumer key and secret are generated only once for the application and hard coded into the ShopDroid source code. Authentication is done in three steps as told before:

1. User clicks on “Let’s do it” button (Figure 3.49a), application retrieves the Request Token from the server *http://shopdroid.ggajewski.2be.pl/oauth/ request_token.* The Request Token is valid for 60 minutes.
2. The Web browser opens the login site *http://shopdroid.ggajewski.2be.pl/ oauth/logon* (Figure 3.49 b). After successfully login, the user is redirected to the authorization site (Figure 3.49c) *http://shopdroid.ggajewski.2be.pl/ oauth/authorize* to authorize the Request Token and the callback URL *shopdroid://done*.
3. The application recognizes the callback and exchanges the authorized Request Token with the Access Token *http://shopdroid.ggajewski.2be.pl/ oauth/access_token*. The access Token is saved into the application’s `SharedPreferences`.

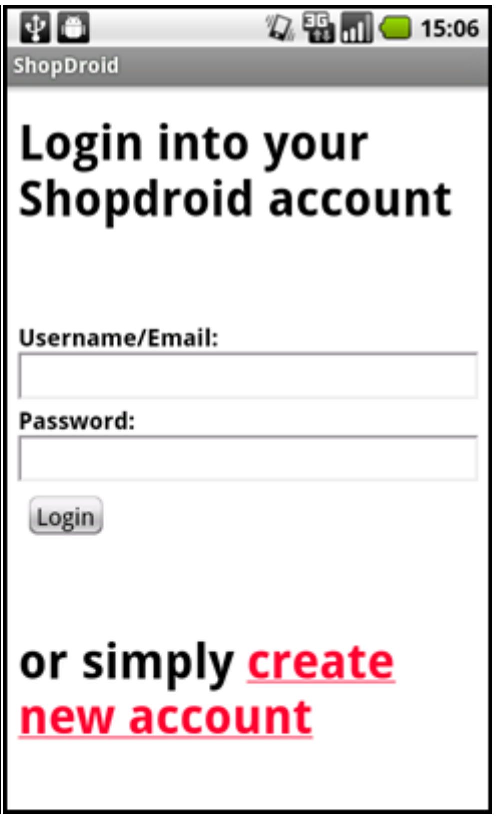
Authentication has to be done only once, when the Access Token is saved in the application all the server requests are signed and the user can be easily identified. To erase the token from the application memory, the user has to go to settings and choose disconnect account. `SharedPreferences` is a good place to keep the Access Token, as it can be accessible only from the owner’s application. When users do not have a ShopDroid account, they can easily create a new one by clicking on the “Create new account” link in the login page. Figure 3.50 shows the new user registration form. All of the above mentioned websites are optimized to be displayed on mobile phones.


15:05

ShopDroid

To get full functionality connect your application with Shopdroid account

Let's do it


15:06

ShopDroid

Login into your Shopdroid account

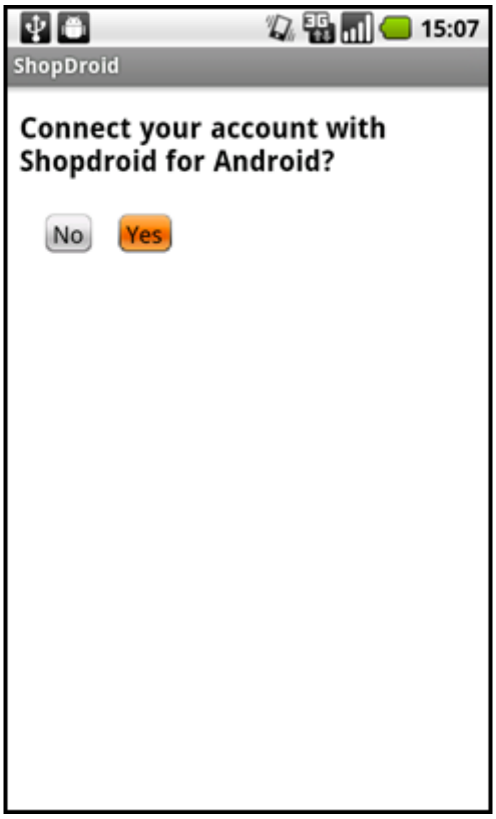
Username/Email:

Password:

Login

or simply create new account

(a) (b)


15:07

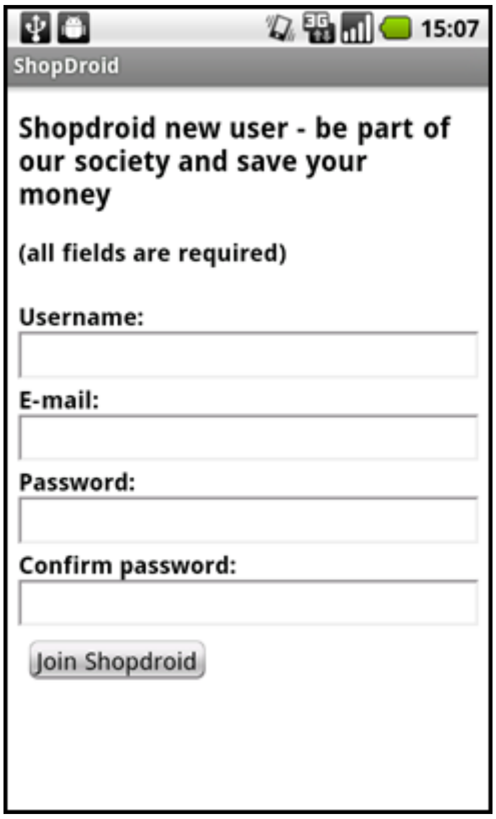
ShopDroid

Connect your account with Shopdroid for Android?

No Yes

(c)

Figure 3.49 a) ShopDroid connect account screen, b) ShopDroid login website, c) ShopDroid authorize Request Token website


ShopDroid

Shopdroid new user - be part of our society and save your money

(all fields are required)

Username:

E-mail:

Password:

Confirm password:

Figure 3.50 ShopDroid register new account website

3.8.3 OAuth database structure

The OAuth server database consists of three tables: oauth_server_registry, oauth_server_nonce, oauth_server_token.

oauth_server_registry - table containing seventeen fields, used to store information about the registered Consumer. Includes the Consumer key and Consumer secret used for verification of incoming requests, detailed information shown in table A.1.

oauth_server_token - table containing ten fields, used to store information about request/Access Tokens. Includes user identification number which is returned when verification is successful, detailed information shown in table A.2.

oauth_server_nonce - table containing five fields, used to store the timestamp/nonce combination, detailed information shown in table A.3.

3.8.4 ShopDroid User reputation points system

User reputation points are a simple system used for quick detection and blocking users with bad intentions. Each user after the registration has zero points. Users with points lower than zero are subject to be deleted by the administrator. Table 3.9 presents scoring for different user actions.

Table 3.9 ShopDroid User actions and points

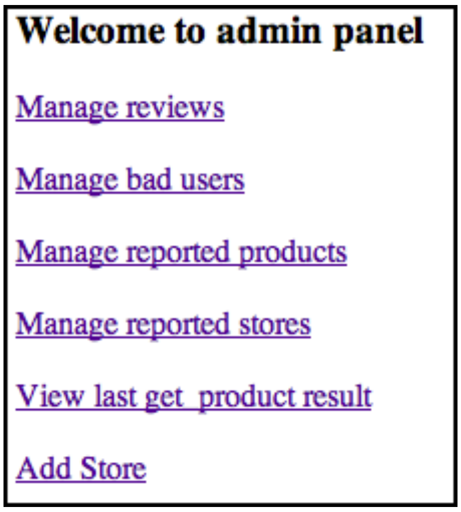
User action	Points
Add new product	+5
Add new store	+5
Edit product price	+2
Like product	+1
Review product	+7
Bad/spam review deleted by admin	-15

3.9 Administrator module

ShopDroid has a simply administrator module to manage stores, products, users and products reviews. The administrator module is the website: http://shopdroid.ggajewski.2be.pl/admin. Only users with administrator privileges can access this website after login. Figure 3.51 shows the main administrator page.

All links on the main page are self-explanatory. “View last get_product result” is a link to the xml file - dump of last “get_product” response - useful for debugging.

Manage reviews is a page displaying all reviews which was reported by ShopDroid users as spam sorted by the number of reports (Figure 3.52). Each review can be simply deleted or marked as good by selecting the corresponding checkbox and clicking on the submit button.



Welcome to admin panel

Manage reviews

Manage bad users

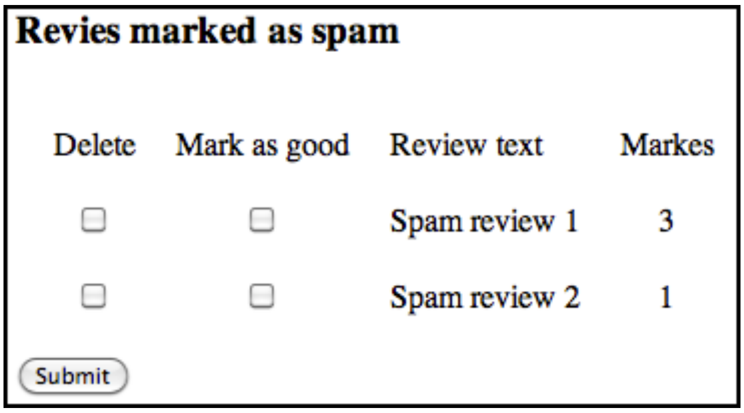
Manage reported products

Manage reported stores

View last get_product result

Add Store

Figure 3.51 Administrator module main page



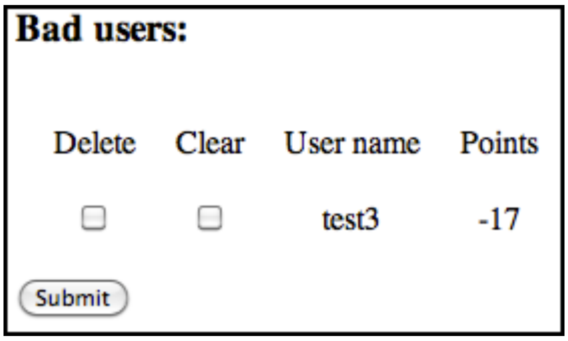
Revies marked as spam

Delete	Mark as good	Review text	Markes
☐	☐	Spam review 1	3
☐	☐	Spam review 2	1

Submit

Figure 3.52 Administrator module manage reviews page

Manage bad users page lists all users whose points score is below zero (Figure 3.53). User accounts can be deleted or points can be reset to zero.



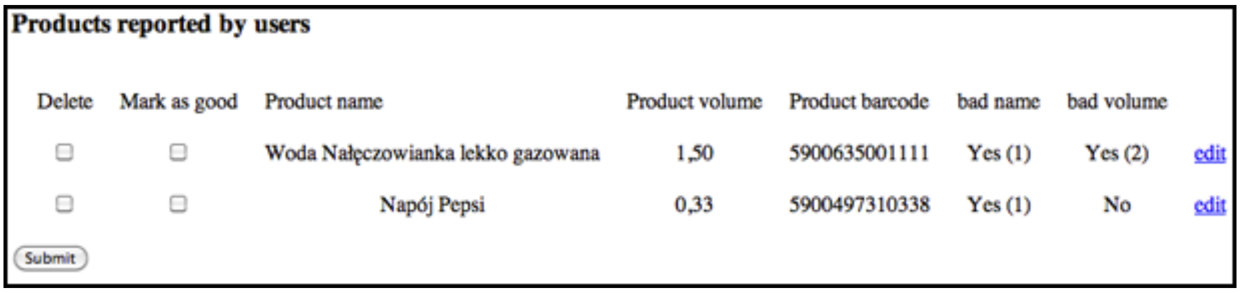
Bad users:

Delete	Clear	User name	Points
☐	☐	test3	-17

Submit

Figure 3.53 Administrator module manage bad users page

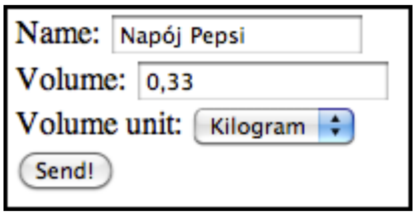
Manage reported products page shows a list of products which description was reported as wrong. In columns “bad name” and “bad volumes” numbers in brackets is the report’s total count (Figure 3.54). The product can be deleted, marked as good or edited (Figure 3.55).



Products reported by users

Delete	Mark as good	Product name	Product volume	Product barcode	bad name	bad volume	
☐	☐	Woda Nałęczowianka lekko gazowana	1,50	5900635001111	Yes (1)	Yes (2)	edit
☐	☐	Napój Pepsi	0,33	5900497310338	Yes (1)	No	edit

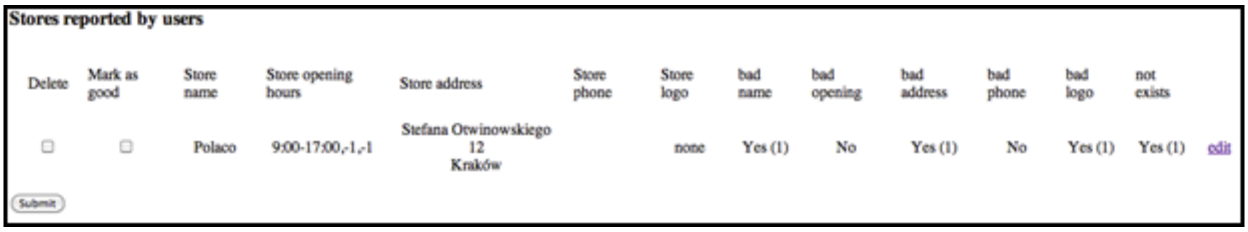
Figure 3.54 Administrator module manage reported products page



Name: Napój Pepsi
Volume: 0,33
Volume unit: Kilogram

Figure 3.55 Administrator module edit product page

Manage reported stores page is equivalent to the manage reported product page for stores (Figure 3.56).



Stores reported by users

Delete	Mark as good	Store name	Store opening hours	Store address	Store phone	Store logo	bad name	bad opening	bad address	bad phone	bad logo	not exists	
☐	☐	Polaco	9:00-17:00,-1,-1	Stefana Otwinowskiego 12 Kraków		none	Yes (1)	No	Yes (1)	No	Yes (1)	Yes (1)	edit

Figure 3.56 Administrator module manage reported stores page.

Add store is a form page to add new stores (Figure 3.57).

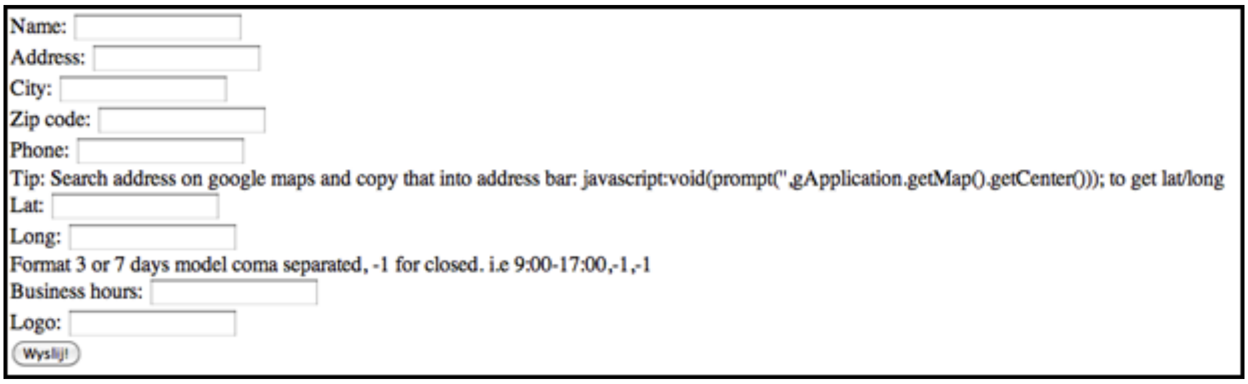

Name:
Address:
City:
Zip code:
Phone:
Tip: Search address on google maps and copy that into address bar: javascript:void(prompt('',gApplication.getMap().getCenter())); to get lat/long
Lat:
Long:
Format 3 or 7 days model coma separated, -1 for closed. i.e 9:00-17:00,-1,-1
Business hours:
Logo:

Figure 3.57 Adding new store

3.10 Database

As mentioned before the ShopDroid system uses two separate databases. The PostgreSQL database stores all products and user data, the MySQL database OAuth tokens.

The ShopDroid database consists of ten tables called: admins, likes, prices, products, reported_products, reported_stores, reviews, reviews_spam, stores, users. Each table contains an auto-increment primary key to ensure uniqueness, speed-up records searching. A more detailed description for each table is shown below.

products - table containing seven fields, storing all product information, detailed information shown in table A.4.

stores - table containing eleven fields, storing all store information, detailed information shown in table A.5.

prices - table containing five fields, storing products prices in different stores, detailed information shown in table A.6.

users - table containing seven fields, saving all user related information, detailed information shown in table A.7.

admins - table containing two fields, used to give administrator privileges to users, detailed information shown in table A.8.

likes - table containing three fields, saving information about the user liking the product, detailed information shown in table A.9.

reported_products - table containing four fields, used to store user reports about wrong product information, detailed information shown in table A.10.

reported_stores - table containing eight fields, used to store user reports about wrong store information, detailed information shown in table A.11.

reviews - table containing five fields, storing user product reviews, detailed information shown in table A.12.

reviews_spam - table containing three fields, used to store user reports about product review (marked as spam), detailed information shown in table A.13.

3.11 Handling multiple screen resolutions

There exist several different Android devices in the market, which can have different screen resolutions. Therefore, when developing ShopDroid this was taken into account. For instance, when designing the user interface, none of the size of the elements is specified in pixels, which would lock the application only for certain resolutions. Instead of pixels, density independent pixels (as described in section 3.3) were used. This allows different elements to scale accordingly.

This was not the only technique used; the position of the interface elements is defined relatively. For instance, a new element can be positioned to the right or below another element. Also it is possible to make the size of an element to occupy the rest of what is left of free space of the screen.

Android also provides the option to set manually the size of any graphic resource such as icons. Inside the *res* folder, it is possible to add more *drawable* folders depending on the needs. This will make the platform to choose the correct graphic resource depending on the device's screen resolution. If none are present, Android will auto-scale the graphic resource. ShopDroid uses this feature for the application icons, and some graphics such as the ones that appear in the screen for locating stores.

In some cases, when the elements do not fit the screen without the need of scrolling, it is possible to wrap all the elements inside a `ScrollView` widget, which will allow the user to scroll and be able to see all the elements of the screen. This widget is not necessary if a `ListView` widget is used. ShopDroid uses `ListView` widgets in several parts of it.

Conclusions

The project was successfully realized. A working system based on barcodes where users can compare product prices was built. Also users can interact with the system by adding new products, changing prices, or writing their opinions about a particular product. So this is a system that allows the users to create its content. A good security model was implemented to keep users credentials safe and to avoid data vandalism.

The client was made targeting the Android Operating System. The application is fully working and uses Android devices' geolocation features; as well it uses the camera for barcode scanning. The application was thoroughly tested and performed quite well, without freezing the user interface.

All this thesis objectives were fulfilled and even additional features were implemented, such as locating stores and having a history of the searched products. Unfortunately, there was not enough time to deploy the application (publishing it into the Android Market) and test it in a wider audience.

Getting a products database in Poland was hard to do. Most of the stores do not have any information about their products in their web sites. Only one store - *Alma* - was found to have this important information. Therefore, to start a service like this, would require close cooperation from several stores, and also having very engaged users that could get this information into the service.

Developing for Android was not that easy. Some of the documentation is lacking or not very clear. Also supporting all current versions of Android is problematic and time consuming, one needs to start an emulator instance for each version. Supporting the earliest Android version, which is 1.5, makes sometimes the user interface to behave differently that how it is supposed to work. Some workaround needed to be made. Hopefully, these legacy versions will disappear in the future and make the task much easier if the updates not introduce any new problems.

For future releases some features could be added. One of these features could be adding information about nutrition facts to the product screen, so users could know which products are healthy for them and which not. Also, product ingredients could be added, this would be extremely helpful for users with any allergies. Another feature that could be added would be price alerts. With this kind of feature, users could set alerts for a specific product and a price threshold for the product, so that whenever the price for this product will go lower than the threshold set, the user will get an alert about it. Making a shopping list could be another option. This would allow users to choose before shopping what products they need to buy and they could see in which store they would spend the least money.

In the future one could think about monetizing the application. If there are plenty of users, having some advertisement showing in certain parts of the application could be a good source of revenue. Also, the advertisement could be better targeted towards each user (advertising companies could pay more for this), since the application will know the users' location and which products they want to buy. Another option could be having two versions: a free version and a paid version, which would have additional features. If the users like the free version, they might be tempted to buy the paid one.

Also, a web site could be built for the service, letting any user access the service from any browser. Having some kind of integration with a social service such as Facebook could be a great idea too.

A total of 58 stores are in the database. The number of products, which have at least one price, is 9405 and the total number of prices (most products have more than one price) is 118660.

References

[1] ComScore press release
http://comscore.com/Press_Events/Press_Releases/2010/9/comScore_Reports_July_2010_U.S._Mobile_Subscriber_Market_Share

[2] ComScore press release
http://comscore.com/Press_Events/Press_Releases/2010/9/European_Smartphone_Market_Grows_41_Percent_in_Past_Year

[3] Android Open Source Project
http://source.android.com/source/licenses.html

[4] Gartner press release
http://www.gartner.com/it/page.jsp?id=1434613

[5] Android Developer's Guide
http://developer.android.com/intl/zh-CN/guide/basics/what-is-android.html

[6] Android Developers' Guide
http://d.android.com/guide/topics/fundamentals.html#appcomp

[7] Android Developer's Guide
http://d.android.com/guide/topics/intents/intents-filters.html

[8] Android Developer's Guide
http://developer.android.com/intl/zh-CN/guide/topics/manifest/manifest-intro.html

[9] ZXing Open Source Project
http://code.google.com/p/zxing/

[10] Official PHP manual
http://www.php.net/manual/en/

[11] W.J. Gilmore *Beginning PHP and PostgreSQL 8*

[12] PostrgreSQL database
http://www.postgresql.org/about/licence

[13] Extensible Markup Language (XML)
http://www.w3.org/XML/

[14] OAuth Consumer And Server Library For PHP
http://code.google.com/p/oauth-php/

[15] Simple OAuth message signing for Java
http://code.google.com/p/oauth-signpost/

[16] RFC 5849 - The OAuth 1.0 Protocol

[17] Android Developer's Guide
http://developer.android.com/intl/zh-CN/guide/topics/data/data-storage.html#pref

[18] Wikipedia
http://en.wikipedia.org/wiki/Barcode

[19] Roger C. Palmer *The Bar Code Book: Fifth Edition*

[20] Android Issues
http://code.google.com/p/android/issues/detail?id=1193

[21] Android Resources
http://developer.android.com/intl/zh-CN/resources/faq/framework.html#3

[22] Eric W. Weisstein *CRC Concise Encyclopedia of Mathematics*, Second Edition p. 1236
[23] Official Nexus One phone specifications http://www.google.com/phone/static/en_US-nexusone_tech_specs.html
[24] OAuth Core 1.0 specification http://oauth.net/core/1.0
[25] Web Hosting Articles http://www.besthostratings.com/articles/http-auth-php-cgi.html

Appendix A

Database Tables

The structure of the server database tables is shown here.

Table A.1 Table “oauth_server_registry” OAuth database

Column	Type	Integration relations	Comments
osr_id	int(11)	primary key, not null	auto-increment primary key
osr_usa_id_ref	int(11)	*default - NULL*	user identification number
osr_Consumer_key	varchar(64)	not null, unique	Consumer key
osr_Consumer_secret	varchar(64)	not null	Consumer secret
osr_enabled	tinyint(1)	default - 1, not null	0 - Consumer disabled, 1 - Consumer enabled
osr_status	varchar(16)	not null	Consumer status
osr_requester_name	varchar(64)	not null	Consumer name
osr_requester_email	varchar(64)	not null	Consumer e-mail
osr_callback_uri	varchar(255)	not null	Consumer callback url

osr_application_uri	varchar(255)	not null	Consumer’s application url
osr_application_title	varchar(80)	not null	Consumer’s application title
osr_application_descr	text	not null	Consumer’s application description
osr_application_notes	text	not null	Consumer’s application additional notes
osr_application_type	varchar(20)	not null	Consumer’s application url
osr_application_commercial	tinyint(1)	default - 0, not null	if Consumer’s application is commercial
osr_issue_date	datetime	not null	Consumer registration date
osr_timestamp	timestamp	default - CURRENT_TIMESTAMP, not null	timestamp

Table A.2 Table “oauth_server_token” OAuth database

Column	Type	Integration relations	Comments
ost_id	int(11)	primary key, not null	auto-increment primary key
ost_osr_id_ref	int(11)	foreign key - osr_id from oauth_server_registry, not null	Consumer registry identification number
ost_usa_id_ref	int(11)	not null	user identification number
ost_token	varchar(64)	not null, unique	token
ost_token_secret	varchar(64)	not null	token secret
ost_token_type	enum('request', 'access')	*default - NULL,*	token type: “request” or “access”
ost_authorized	tinyint(1)	default - 0, not null	0 - tokn not authorized, 1 - token authorized
ost_referrer_host	varchar(128)	not null	referrer host
ost_token_ttl	datetime	default - 9999-12-31 00:00:00, not null	token expiration date, default for Access Token if infinity
ost_timestamp	timestamp	CURRENT_TIMESTAMP, not null	timestamp

Table A.3 Table “oauth_server_nonce” OAuth database

Column	Type	Integration relations	Comments
osn_id	int(11)	primary key, not null	auto-increment primary key
osn_Consumer_key	varchar(64)	not null, unique	Consumer key
osn_token	varchar(64)	not null, unique	token
osn_timestamp	bigint(20)	not null, unique	timestamp
osn_nonce	varchar(80)	not null, unique	nonce

Table A.4 Table “products” ShopDroid system database

Column	Type	Integration relations	Comment
product_id	integer	primary key, not null	auto-increment primary key, product identification number
product_name	text	not null	product name
product_imgsrc	character varying(255)		source of the product image, can be null if there is no product image uploaded, user name if image was taken by user or web url if image was found on a website
product_volume	character varying(10)		product package volume number
product_volume_unit	character varying(2)		product package volume unit. Three units are allowed: l - liter, kg - kilograms, p - piece
product_barcode	bigint	unique	product barcode number which can be found on package
product_added_by	integer	foreign key - user_id from users	user identification number who added product

Table A.5. Table “stores” ShopDroid system database

Column	Type	Integration relations	Comment
store_id	integer	primary key, not null	auto-increment primary key, store identification number
store_name	text		store name
store_address	text		store address
store_lat	double precision		store location latitude
store_long	double precision		store location longitude
store_city	character varying(60)		store location city name
store_logo	character varying(255)		source of the store image, can be null if there is no store image uploaded, user name if image was taken by user or web url if image was found on a website
store_zipcode	character varying(8)		store location zip code
store_phone	character varying(20)		store phone number
store_opening	text		store business hours
store_added_by	integer	foreign key - user_id from users	user identification number who added store

Table A.6 Table “prices” ShopDroid system database

Column	Type	Integration relations	Comment
id	integer	primary key, not null	auto-increment primary key
product_id	integer	foreign key - product_id from products, not null	product identification number
store_id	integer	foreign key - store_id from stores, not null	store identification number
price	real	not null	price for the product with given product_id in the store with given store_id
price_added_by	integer	foreign key - user_id from users, not null	user identification number who added price

Table A.7 Table “users” ShopDroid system database

Column	Type	Integration relations	Comment
user_id	integer	primary key, not null	auto-increment primary key, user identification number
user_name	character varying(20)		user name
user_email	character varying(255)	not null	user email
user_password	character varying(32)	not null	user password
user_registration_date	timestamp with time zone	default - now(), not null	user registration date and time
user_loc	character varying(5)		user localization code, to letters for language two capital for localization i.e en-GB
user_points	integer	default - 0	user points in ShopDroid system

Table A.8 Table “admins” ShopDroid system database

Column	Type	Integration relations	Comment
admin_id	integer	primary key, not null	auto-increment primary key
user_id	integer	foreign key - user_id from users, not null	user identification number

Table A.9 Table “likes” ShopDroid system database

Column	Type	Integration relations	Comment
like_id	integer	primary key, not null	auto-increment primary key
product_id	integer	foreign key - product_id from products, not null	product identification number
user_id	integer	foreign key - user_id from users, not null	user identification number

Table A.10 Table “reported_products” ShopDroid system database

Column	Type	Integration relations	Comment
reported_product_id	integer	primary key, not null	auto-increment primary key
product_id	integer	foreign key - product_id from products, not null	product identification number
reported_product_name	integer	not null	number increased by one when user reports wrong product name
reported_product_volume	integer	not null	number increased by one when user reports wrong product volume

Table A.11 Table “reported_stores” ShopDroid system database

Column	Type	Integration relations	Comment
reported_store_id	integer	primary key, not null	auto-increment primary key
store_id	integer	foreign key - store_id from stores, not null	store identification number
reported_store_name	integer	not null	number increased by one when user reports wrong store name
reported_store_opening	integer	not null	number increased by one when user reports wrong store business hours
reported_store_address	integer	not null	number increased by one when user reports wrong store address
reported_store_phone	integer	not null	number increased by one when user reports wrong store phone number
reported_store_logo	integer	not null	number increased by one when user reports wrong store logo image
reported_store_not_exist	integer	not null	number increased by one when user reports that store does not exist

Table A.12. Table “reviews” ShopDroid system database

Column	Type	Integration relations	Comment
review_id	integer	primary key, not null	auto-increment primary key, review identification number
user_id	integer	foreign key - user_id from users, not null	user identification number - review author
review_text	text	not null	review content
review_date	timestamp with time zone	default - now(), not null	review submission time and date
product_id	integer	foreign key - product_id from products, not null	product identification number

Table A.13. Table “reviews_spam” ShopDroid system database

Column	Type	Integration relations	Comment
review_spam_id	integer	primary key, not null	auto-increment primary key
review_id	integer	foreign key - review_id from reviews, not null	review identification number
review_marked	integer	not null	number increased by one when user mark review as spam