



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Distributed
Computing*



Smart Shopping

Semester project

Denitsa Dobрева, Dimitrios Gkounis, Konstantinos Karvounis

`dobrevad@student.ethz.ch`, `dgkounis@student.ethz.ch`,
`kostas213@student.ethz.ch`

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zurich



Supervisors:
Jochen Seidel,
Prof. Dr. Roger Wattenhofer

August 31, 2012

Abstract

Even though technology penetration in everyday life increases constantly there still is the burden of how to manage your personal shopping history and store all receipts. Still all shopping bills are in the old fashioned paper format which makes it difficult to analyze the information. Inspired by this the Smart Shopping project aims to build an innovative and user-friendly Android application which helps the users to keep track of all bought items. The application furthermore performs an analysis on the stored user data and gives suggestions for products which the user most probably would like to buy in the next days. The application also incorporates the capability to browse reduction offers from favorite shops and include them into the next shopping list.

Contents

| | |
|---|-----------|
| Abstract | i |
| 1 Introduction | 1 |
| 1.1 Description | 1 |
| 1.2 Organization | 2 |
| 2 Architecture of the application | 3 |
| 3 Client | 4 |
| 3.1 Design | 4 |
| 3.2 Implementation | 5 |
| 3.3 Tesseract and Leptonica | 7 |
| 3.3.1 Training Tesseract | 7 |
| 3.3.2 JNI | 7 |
| 3.4 Receipt Recognition | 8 |
| 3.5 Corrections and Predictions | 10 |
| 3.5.1 Corrections Algorithm | 10 |
| 3.5.2 Predictions | 11 |
| 3.6 Reductions | 12 |
| 4 Database | 13 |
| 4.1 Remote database | 13 |
| 4.2 Local database | 15 |
| 4.3 Synchronization | 15 |
| 5 Server | 18 |
| 5.1 Communication with the client | 19 |
| 5.2 Connection to the database | 21 |

| | |
|--------------------------------------|------------|
| CONTENTS | iii |
| 6 Web Interface | 22 |
| 6.1 GUI and Functionality | 23 |
| 6.2 OpenID | 26 |
| 7 Future Work and Conclusion | 27 |
| A Building Tesseract | A-1 |
| B Training Tesseract | B-1 |
| C Web Application Development | C-1 |
| D Web Application Deployment | D-1 |

Introduction

1.1 Description

The purpose of this semester project is to automate the process of storing receipts and making use of this information in a lightweight way. For the scanning we use an existing OCR library, namely Tesseract and extend it with an overlay interface supporting recognition for different receipt layouts. The OCR engine does not always perform an errorless text recognition. Therefore, the project introduces a collaborative word correction. The idea is that when a product is introduced for the first time to any user the user can correct manually the name. From then on the product will be presented in its corrected version to any other user.

Further, we take into consideration that smartphones offer great connectivity capabilities. Based on this we built a three-tier system where part of the functionality and storage is pushed to a remote application and a database tier. Moreover, the system is partially distributed. Every user has a copy of his or her shopping history and can synchronize with the shared remote server when there is internet connectivity.

To get a better perspective on the main use cases of the Android application we refer to figure 1.1. One of the usage options is to scan a receipts. This makes a call to the OCR engine and image analysis, which is described in details later. Users can also create shopping lists to help them with the shopping. A list can be created using suggestions by the application or simply adding manually any other product. All scanned receipts and created shopping lists can further be browsed and edited. Users can keep track of their money balance using the application and reset the balance at any time. Without the need to refer to any third party websites or application users can check the reduction offers of their favorite shops and add the desired ones to a shopping list.

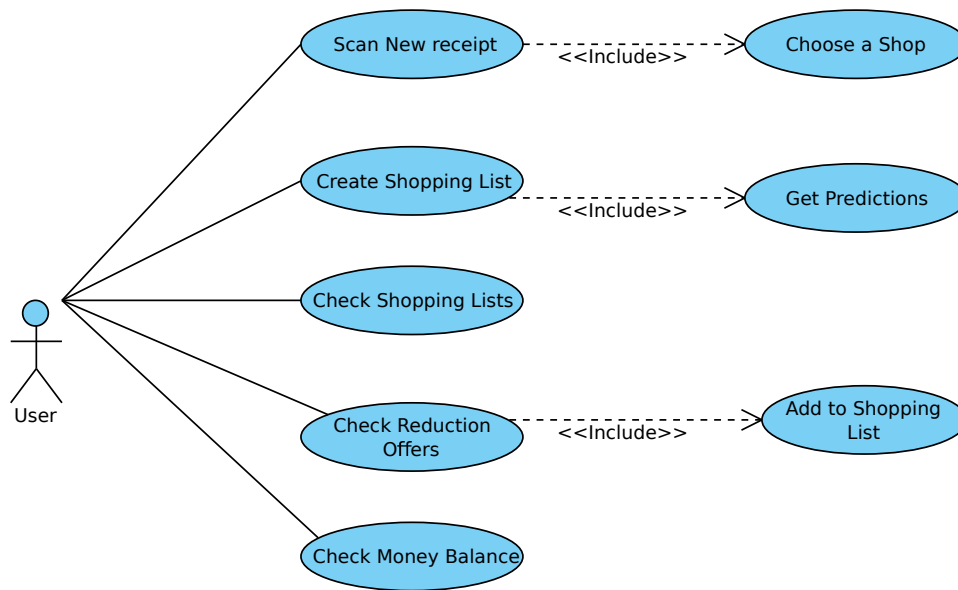


Figure 1.1: Use Case Diagram of the Android application

1.2 Organization

Firstly we describe the architecture of the project as a whole (Chapter 2). In Chapter 3 we describe thoroughly the architecture of the client and its functions. These functions include the recognition of the receipts, the correction and prediction algorithms and the search for reduction offers. Next we give a description about the databases used in the project, their schemas and the synchronization procedure used for communication between them. In Chapter 5 we analyze the architecture of the Server and its services. In Chapter 6, we describe the web interface of the Server and its functionality. Finally we present some ideas about future extensions of the project and draw conclusions about our experience constructing the system.

Architecture of the application

The project consists of two subprojects: an Android application (Client) and a web service (Server). Both parts contain a database for data storage and communicate with each other to exchange information between the databases.

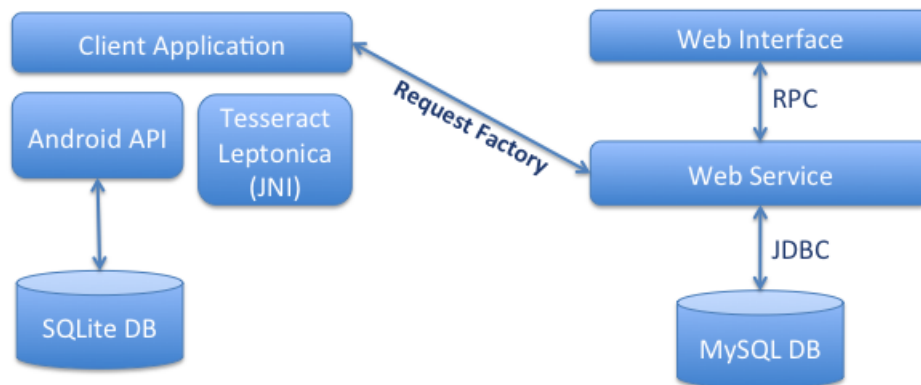


Figure 2.1: Architecture diagram

The Android application is developed based on the Android API Level 10 and it is compatible with devices running at least Android 2.3.3. Also it uses an OCR library (Tesseract) and an image processing library (Leptonica). The two libraries are compiled via Android NDK and the application accesses them via JNI. There is finally a SQLite database for local data storage.

The web server runs on an Apache Tomcat server. It consists of a web service based on Google Web Toolkit (GWT). The web service connects to a MySQL database via Java DataBase Connectivity (JDBC) API. GWT provides two communication means for the web service: RequestFactory and RPC. The former is used for the communication with the client application and the latter for the communication with the web interface.

3.1 Design

The Client has a modular architecture. The motivation to design it in a loosely coupled way is to support easy future extension. A separate component supports every main feature where on top of everything we build an intuitive and user-friendly GUI.

On Figure 3.1 are presented the main components of the Client architecture. The purpose of the Reductions Collector is to supply the data for the reduction offers. It is implemented as an interface which gives the possibility to support easily any additional shop using the parsing methods. For the moment we do support Lidl and Denner.

The next important component in the Client architecture is the Receipt Recognition. It has a special place there because it handles the main functionality of the application and namely the receipt analysis. This component is build again around an interface that allows extension in the number of shops. The Receipt Recognition uses the two modules: Tesseract and Leptonica. Leptonica is used for image enhancement together with Tesseract to give better results on the recognition. More details on this follow in the next sections.

Corrections and Predictions are both based on algorithms we run on the local database. The Corrections is used to improve the results from the Receipt Recognition. It uses the corrections which the user or other users have done before so that it can avoid future errors in the text recognition. The Predictions component performs an analysis on the user behavior by means of how often and what kind of products does the user buy. It makes suggestions about next purchases. This improves the application usability by leaving less work for the user.

To offer an offline usage of the application and higher performance by limited interaction with the server we introduce a local database. This keeps a copy of the user's data. The database is synchronized with the server when Internet connection is detected. The synchronization is done by the Sync module.

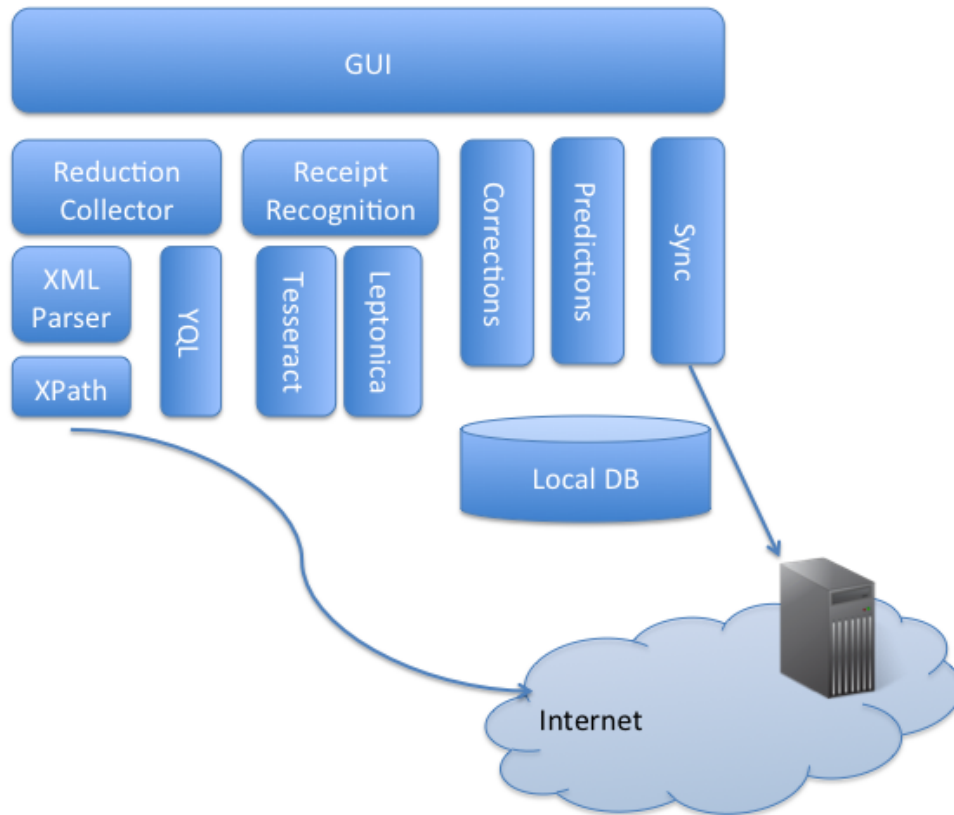


Figure 3.1: Architecture Design of the Client

The Sync is used to keep the local and the remote databases updated. Only the new and altered entries are sent to the remote database so that the server has an up-to-date copy of the user's local database. In the opposite direction, the server sends on demand the new product entries from other clients. These are used by the correction algorithm.

3.2 Implementation

In this section we describe the connections between the separate packages in the Client project. On Figure 3.2 we show a Package Diagram of the Client implementation. Classes for the Tesseract Engine and Leptonica reside in the `com.googlecode` package with nested packages `tesseract.android` and `leptonica.android`. These packages are responsible for image processing. Therefore we separate them logically from the other code.

The shared package contains all definitions of objects exchanged between the

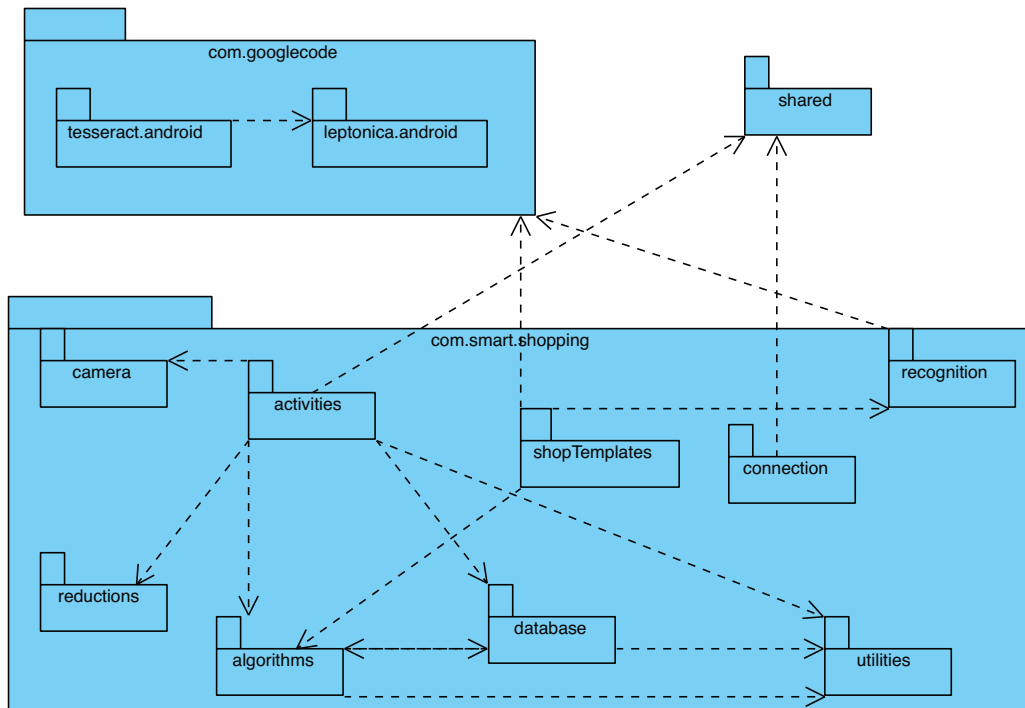


Figure 3.2: Package Diagram of the Client

Client and the Server which are used for the synchronization. It resides on the Server side but is linked from the Client project in order to avoid copying of code.

The `com.smart.shopping` package contains the rest of the client logic implemented in nested packages. All Android Activities reside in the `activities` package. It serves most for presentation purposes where the other packages implement the functionality or deliver the data.

`ShopTemplates` together with the `recognition` package and the `com.googlecode` contain the implementation of the image processing for the supported shops. The `shopTemplates` package defines the rules for the image recognition and links both `recognition` and `com.googlecode` for the invocation of the image processing.

3.3 Tesseract and Leptonica

3.3.1 Training Tesseract

We use version 3.01 of Tesseract. It has support for a variety of languages but the recognition of those languages requires that the text font is among the supported ones otherwise the results are bad. To improve the text recognition we trained for specific fonts. To do that one has to first build Tesseract. For more information please check with Appendix A.

The training on its own is a challenge. The process is very tedious since most of it has to be done manually. The process has two major steps:

- Create the training documents
- Train Tesseract

To create the training documents there are two options: one can either create them from text files if the font of the receipt is known in advance or train with images of the receipts. Since we have no information about the exact font of the receipt we trained Tesseract using the second option. However, training was not successful because scanned receipts have a special layout with big spaces between the columns and the text is not homogenous. Further, to get good results we need quite big data sample which was another issue. Therefore the current version of the project is using the language files provided with Tesseract. However, it would be a good extension to improve this as a future work. For more details about the training procedure please see Appendix B: Training Tesseract.

3.3.2 JNI

The Tesseract engine is written in C and C++ therefore we need something that wraps it and makes Java access to the API possible. The standard way to do that is the Java Native Interface (JNI) which enables Java calls to the natively-compiled Tesseract and Leptonica APIs. Since there are already existing solutions for that we decide to make use of the most appropriate one and focus on the domain of the project.

The `tess-two`¹ project is an open source project that is implemented on top of Tesseract and Leptonica. It uses the compiled libraries together with JNI and offers a Java based API. We use it as a library project into the Android application. It serves as a bridge to Tesseract and Leptonica and gives us access to all native methods we need to use for image enhancement and text recognition.

¹<https://github.com/rmtheis/tess-two>

3.4 Receipt Recognition

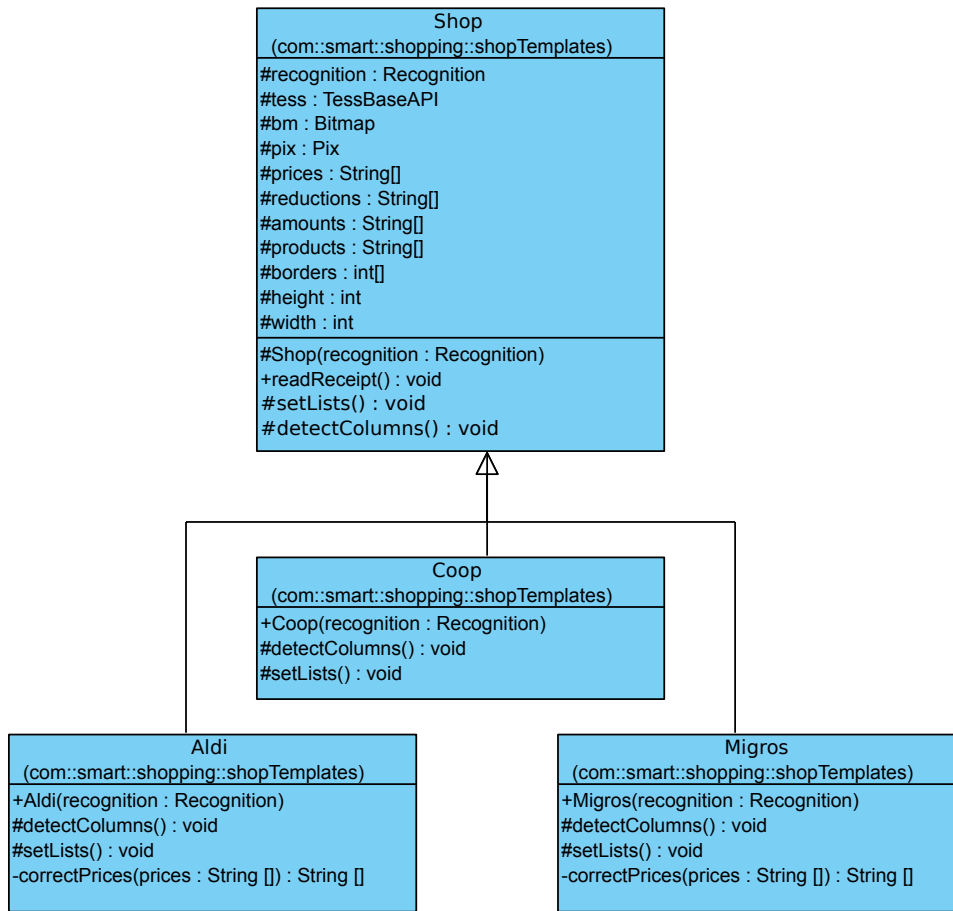


Figure 3.3: Sample Receipt from Coop Figure 3.4: Sample Receipt from Aldi

Every shop has its own receipt template and if we run Tesseract on the entire image the output of the recognition is above 50% incorrect. To avoid this we add another layer on top of the OCR. This layer adds a template for every supported shop.

We see from Figure 3.3 and Figure 3.4 a sample shopping receipt from Coop and Aldi. These two shops are among the supported ones. The difference in the layout between the receipts is huge in terms of information ordering. Therefore we cannot rely on automatic column recognition that will detect the separate columns and give us the type with respect to product name, price, etc. As we see from the images Coop has a dedicated column for every information type whereas Aldi has only two columns: a dedicated one for the prices and the first one containing everything else. If we add all different shops to the picture we will see that it is almost impossible to automatically distinguish the information.

We approach this problem by handling each specific receipt layout in a dedicated class. This resolves into a hierarchy, where we have an abstract class Shop, which hides the concrete shop implementation for the rest of the source code. In detail, we need two methods from each child - `detectColumns()` and `extractShoppingList()`. `detectColumns()` handles the specific receipt layout, and `extractShoppingList()` performs a shop specific content extraction from the parsed layout. Figure 3.5 shows the class hierarchy of the supported shops. This hierarchy provides an easy way to add more shops to the application in the future. One only needs to introduce a new implementation of the abstract Shop class with the above mentioned methods. Moreover, a factory class is introduced to completely isolate the creation of the different shop implementations from the

Figure 3.5: Class Diagram of package `com.smart.shopping.shopTemplates`

rest of the program logic.

Let us now describe the concept of the algorithm for detecting the text columns of a receipt. Before that we need to clarify that for every shop template this algorithm has a tweaked implementation which accounts for the different column number, minimum expected space between columns, etc. The algorithm tries to define the virtual borders between the separate columns. To do that it analyses the image from top to bottom and defines for each line of width one pixel whether it is part of a text column or an empty space. If it finds enough white vertical lines in sequence we conclude that there can be drawn a virtual border. To visualize the process we show screenshots from the application. On Figure 3.6 the user is prompted to crop once the image with all product items and prices. The cropped area is defined with the orange rectangle. Figure 3.7 shows the result of the columns detection, which is hidden from the user. The virtual borders would look like the dashed green lines added to the screenshot.



Figure 3.6: Screenshot from the Android app: Scanning a receipt

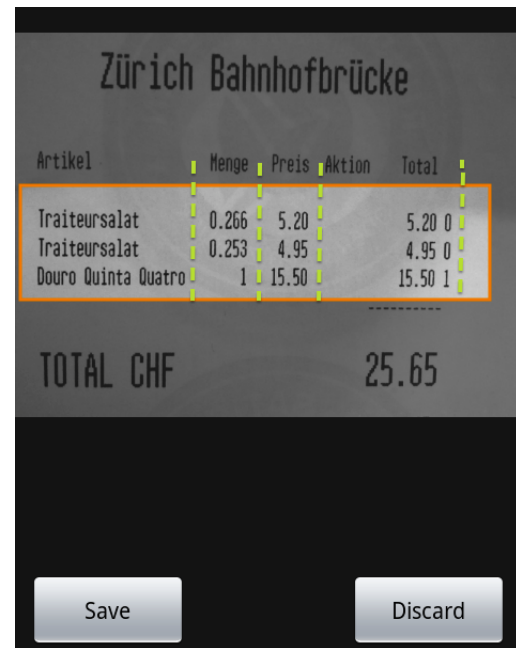


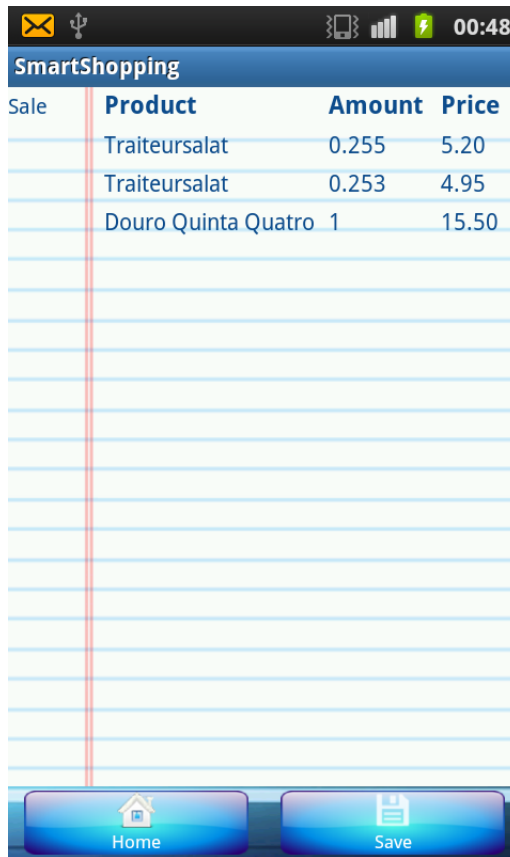
Figure 3.7: Screenshot from the Android app: Virtual Borders

The actual output to the user is shown on Figure 3.8. This process improves the recognition for two reasons. First, Tesseract performs bad on the receipt template but works good on a homogenous text as when we restrict the recognition to one single column. Second, mixing together alphabet characters and numbers has also shown to be a drawback for the performance of Tesseract.

3.5 Corrections and Predictions

3.5.1 Corrections Algorithm

The corrections are needed to avoid errors that Tesseract repeats in the text recognition. Because of problems with the font recognition it tends to make the same mistakes for particular letters or words. After the OCR is finished we execute a word correction based on previous corrections by users. In the local Database we keep the entries for all scanned words and corresponding corrections if any. Upon synchronization we add to the local Database all pairs of scanned and corrected words from other users. The algorithm for the corrections is as follows. For every scanned word we search in the Database for the best match given a level of tolerance. The best match is the word which is closest to the current one. The distance is computed using the Damerau-Levenstein distance.



| Sale | Product | Amount | Price |
|------|---------------------|--------|-------|
| | Traiteursalat | 0.255 | 5.20 |
| | Traiteursalat | 0.253 | 4.95 |
| | Douro Quinta Quatro | 1 | 15.50 |

Figure 3.8: Screenshot from the Android app: Scanned Receipt View

The scanned word is replaced by the corrected version of the best match if it has a distance below the predefined tolerance level. In the implementation of the corrections we compute a list of words that fulfills the tolerance criteria and is sorted based on the distance value. The purpose is to go further and show a list with possible corrections so that the user can decide on the optimal one. In the current version of the application we take the best match and replace the scanned word. Given the current size of the products table this gives optimal results because it does not require any intervention by the user. However, when there are many and similar product entries in the database this might not be the case.

3.5.2 Predictions

The predictions are another important part of the Client project. This component performs an analysis on the user shopping behavior based on previous purchases. We need to model the past purchases and the expected ones. To ap-

proach this problem we use linear regression. For every product bought at least once the time of all previous purchases is taken into account and extrapolated using linear regression. The product is included in the next list of suggestions if the time for the future purchase fits within a predefined range of time. We set the range to $[-10, +10]$ days of the current moment. We assume that products that had to be bought earlier are actually bought but not scanned by the user or are not relevant any more. This way if the user is periodically buying particular product he/she will get a suggestion to include this into his/her new shopping list.

3.6 Reductions

For the Reductions we follow the same approach as with the Receipt Recognition. To support easy extension for different shop types we define an interface `Reductions` with a single method `loadReductions()`. This interface can be implemented by any shop type where the implementation of the method will be website specific. To get the reduction offers the application craws the website and extracts structured data. For proof of concept we implement the reduction collection for Denner and Lidl. We choose exactly these two shops because their websites require different technologies to extract the data.

For Denner we use the Yahoo! Query Language (YQL) Web Service. It enables applications to query data from different sources across the Internet including Web content using a familiar SQL like syntax. Moreover, it transforms the data and can deliver it in a JSON format. This makes it a good solution for our case. We call the YQL Web Service with a simple HTTP Get on the URL and pass our YQL statement as a parameter to the query. For the statement we use a CSS selector for HTML which fetches for us the desired elements with the specified class or attribute. The output is a JSON object containing all the information for the reduction offers as product name, old and new price, URL of the picture, etc.

For Lidl we approach the task in a different way. The problem there is that with YQL we cannot access the image address for each reduction offer. Therefore we combine the old fashioned XML parsing together with YQL.

Database

The application handles different types of data. Firstly, there is the information obtained from the recognition of the receipts. Secondly, the application has to save and organize the retrieved information of the receipts into lists for every user and to provide basic sharing functions with other users. Finally, we need to avoid data duplication and to be able to find the information we need quickly.

For these reasons, the application uses a local database on the phone to save the data. The database is a SQLite database. The Android Database class (`android.database` package)¹ provides the creation and the connectivity to the database.

Furthermore, there is a remote database. The application uses the remote database to synchronize the local data. The remote database also provides a way to share data between different devices and users. It is an MySQL database and the client communicates via a web service. In the next chapter, we describe in more detail the structure of the server.

4.1 Remote database

There are six primary data types for the application: users, purchases, products, lists, groups and scans. Each of these types corresponds to a table in the database. There is also a class for each data type in the application. The `as1_users` table contains information about the user of the device where the application runs. Also it may contain data for other users when it is necessary. During the synchronization, the client downloads from the server the information of other users only if the user of the device has shared information with another user. In addition for each type exists a corresponding class. These classes are used for data exchange between the client and the server.

The schema of the remote database can be viewed in the following figure:

¹<http://developer.android.com/reference/android/database/package-summary.html>

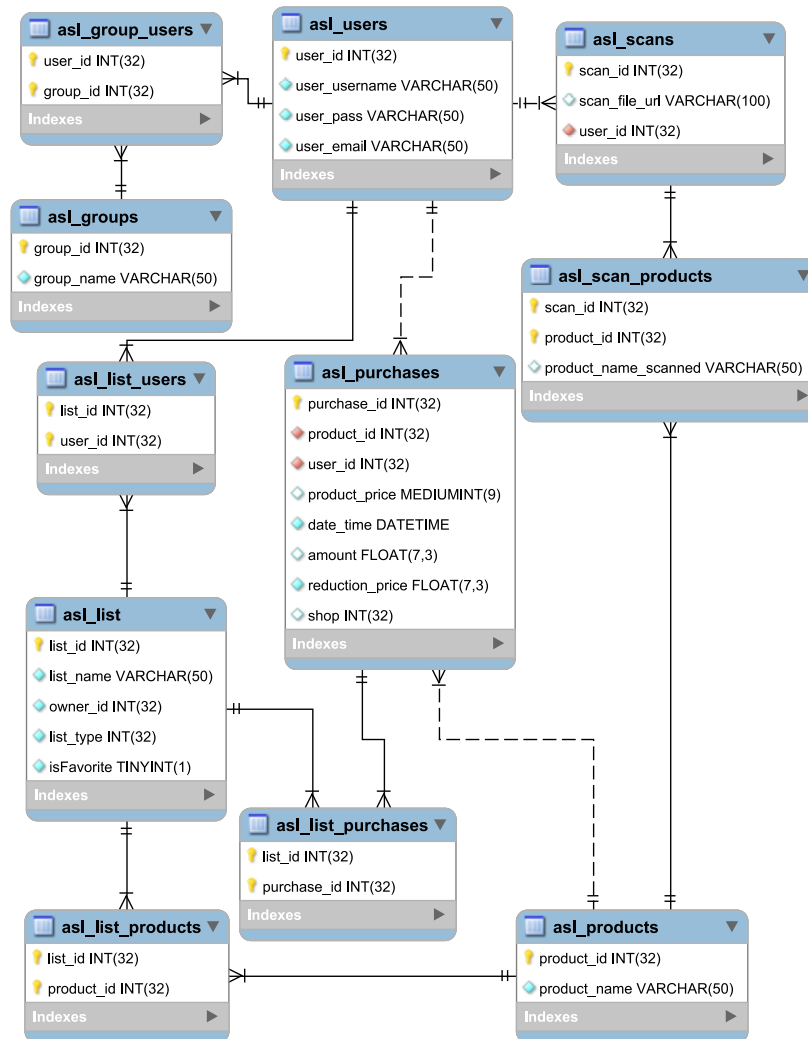


Figure 4.1: Remote database schema

The `asl_purchases`, `asl_products` and `asl_scans` table contains the information obtained after the receipt recognition. Specifically, the `asl_products` table saves the product names after they are saved by the user. The `asl_purchases` table saves the information obtained from the receipt for every single product. By using two separate tables for products and purchases, data duplication is avoided: as long as the users use the application, it becomes more probable that a product name is already scanned. The `asl_lists` tables helps to organize the receipts into lists and also gives the ability to store new custom lists. Furthermore, `asl_scans` is used to save data for every scanning of a receipt and `asl_groups` to create groups between users, where every member of the group is able to see the information of every other member.

In addition to these tables, there are four tables that connect some of the primary tables. The table `asl_list_users` is used to identify the users that have access to a specific list. The data from this table can be used to share lists between users. The `asl_list_purchases` table stores the information about which purchases belong to a list and `asl_group_users` is used to connect users and groups. Finally, `asl_scan_products` identifies which products belong to a scan.

All of the tables are common to the two databases. The local databases adds some fields into every table in order to reference the data on the server and to check for new data. Another difference between the two databases is the support of foreign keys. The remote database supports them, but the local databases does not. The application code does not create references to non-existing data in order to eliminate problems during the communication between the databases.

4.2 Local database

The schemas of the two databases are similar between them, as they share the same table structure with some differences in the fields. Both databases contain the same type of data in each table. The local database has to contain extra information about the editing of its records (insertion or update of data by the user). The remote database contains the data of all users, hence the local database has also to contain a reference to the equivalent data of the remote database. Finally, having an (almost) identical schema between the two databases was beneficial during the development of the application. A significant portion of the code written for one of the databases, can be easily adapted for the other database.

The local database contains the tables described in the previous section. The difference that each table contains two extra fields: `isAltered` and `server_id`. The application sets `isAltered` as `true` when it inserts or updates a record. During the synchronization, the application sends to the server only the records that have changed (`isAltered=true`). `server_id` contains the ID of the corresponding server record. This way the server knows which records to compare when it receives data from the client.

The schema of the local database is displayed in figure 4.2.

4.3 Synchronization

The synchronization allows the application to connect to a remote server in order to backup or share data between different devices and users. In the server resides

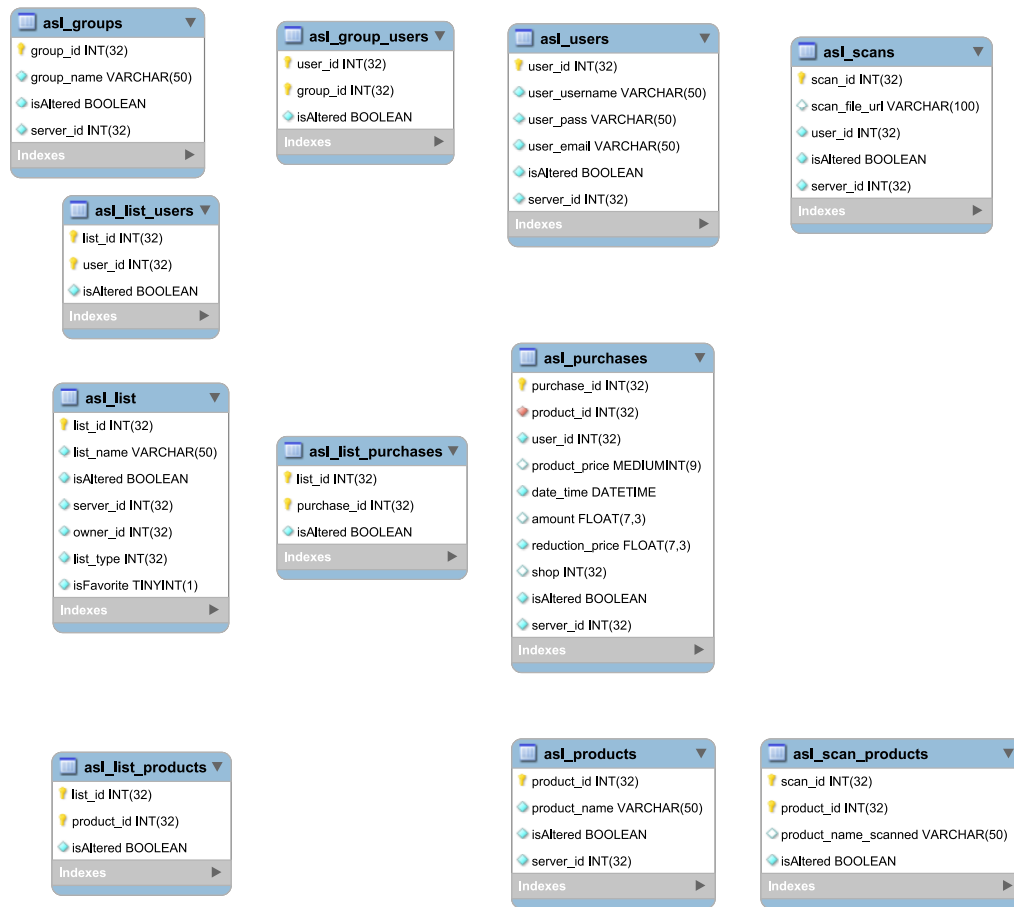


Figure 4.2: Local database schema

a web service and a database. The application uses the web service as a middle layer to connect to the database.

The synchronization algorithm was implemented in a way to minimize data traffic between the client and the server. The client sends to the server only the data that have changed from the last synchronization and the server sends back to the client only the data that need to be updated or inserted into the database. In this way the data transferred are small enough even for large databases. A schematic view of the synchronization procedure can be viewed in 4.3.

The client requests from the database the newly inserted or updated records. Then it encapsulates the data in a single object (instance of a class specifically made for the synchronization) and sends it to the web service of the server. The communication between the client and the web service of the server is asynchronous due to the use of Google Web Toolkit on the server. For this reason, it is convenient to send only one object to the server in order to avoid waiting for

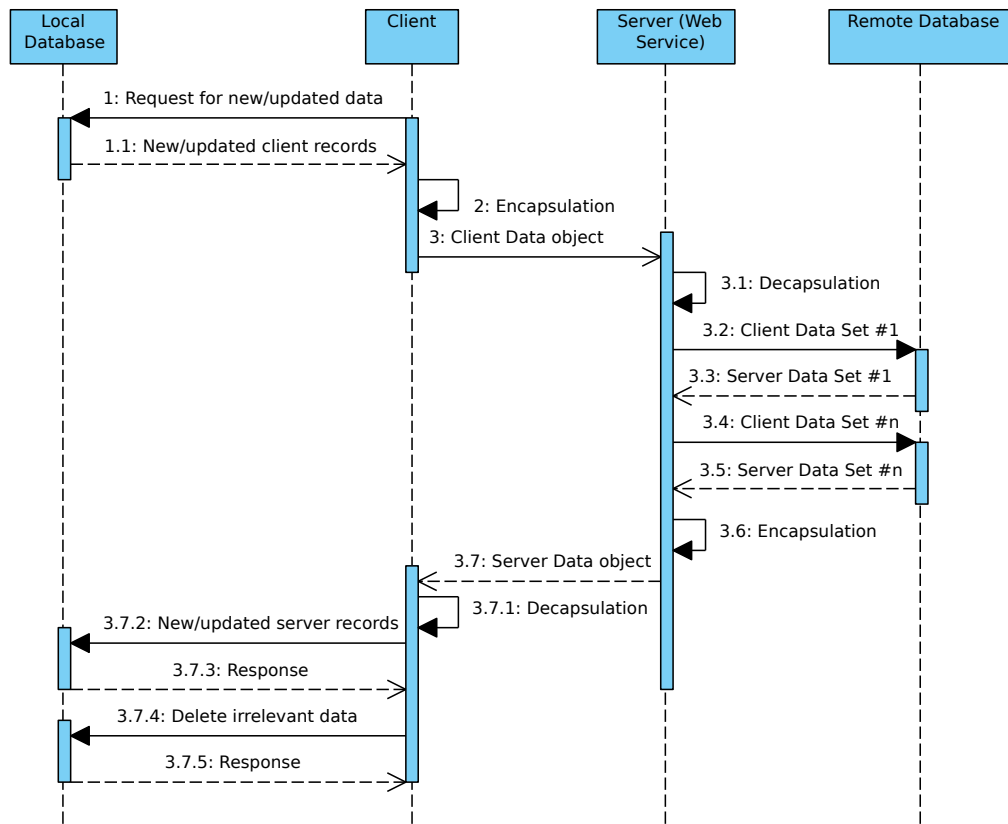


Figure 4.3: Client-Server communication during the synchronization

specific data (due to dependencies) and having to manage multiple asynchronous responses. The server then decapsulates the data and compares them with the remote database. For each table of the database the services sends the client data to the database and gets back the records that need to be forwarded to the client. The web service encapsulates the server data into a single object and forwards it to the client. The client updates its local database with the server data. At last it deletes the information that are irrelevant or unnecessary to the user. This last feature is implemented for future use. Then if a user stops sharing of a list to an another user, then during the sync, all information that are not longer shared, are deleted. Keeping the local database as small as possible is beneficial for the performance of the application.

To encapsulate remote database access in our Android application, we develop a web application. The web application receives the requests from the Android application on the mobile device and passes them to the database. Then, the web application gets the response from the database and returns it to the Android device (Figure 5.1).

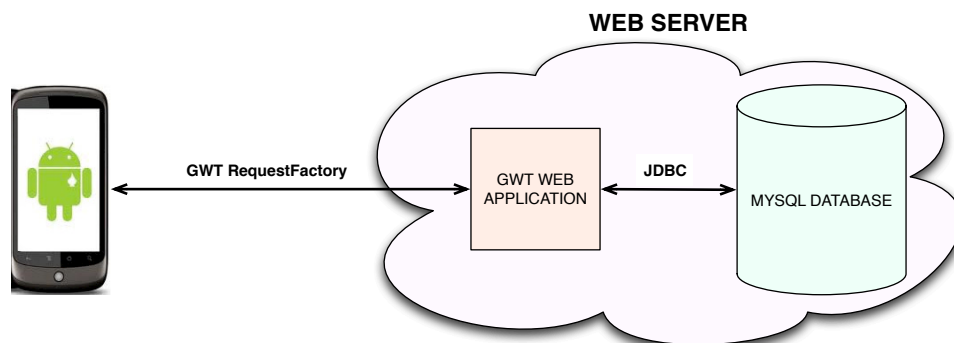


Figure 5.1: Client-Server Communication

To develop the web application, we use the Java-based Google Web Toolkit (GWT)¹. We use this framework because we want to develop a web application in the same programming language as the Android application, which is developed in Java programming language. This also enables us to share database-specific code between client and server, as discussed in the previous chapter.

By using GWT, there are two ways for enabling client-server communication. The one uses the GWT-RPC framework² of GWT and the other uses Request-Factory³. The GWT-RPC is the way for making Remote Procedure Calls (RPC)

¹<https://developers.google.com/web-toolkit/>

²<https://developers.google.com/web-toolkit/doc/1.6/DevGuideServerCommunication>

³<https://developers.google.com/web-toolkit/doc/latest/DevGuideRequestFactory>

when using GWT and RequestFactory is an alternative of GWT-RPC. To connect Android to the web application, we use RequestFactory. This is because RequestFactory has a more data-centric approach than GWT-RPC which focuses more on services. The GWT-RPC is used to develop the web interface of the web application, which is discussed in the next chapter. The web application uses the Java DataBase Connectivity (JDBC)⁴ API to enable access to the MySQL database.

5.1 Communication with the client

As we discussed, we use the RequestFactory of GWT to enable client-server communication between the Android smartphone and the web application respectively. In order to exchange data between client and server side, one has to define the data entities in each side. RequestFactory doesn't use the same data entities both in the client and the server side. At first, we define the data entities in the server-side. These entities are just Plain Old Java Objects (POJOs)⁵ that persist the data from the remote database. To be specific, classes like AslUsers, AslGroups, AslProducts etc. in the `com.smart.shopping.database` package of the shared folder of our web project set up the server-side entities. Then, we define their representatives in the client-side. Each client-side entity is mapped to a server-side entity. Each client-side entity has the same name as its corresponding server-side followed by the name Proxy, e.g. the `AslUsersProxy` interface in the client-side is mapped to the `AslUsers` class in the server-side (Figure 5.2). All proxies are interfaces which their corresponding classes implement in the server-side. The client uses the proxies in the client side to process data that comes from the server by using methods that proxies contain.

To exchange data between the client and the server, we use an interface between them. The client invokes methods that the server executes. We implement these methods, for example to access to the remote database, in the server side and specifically where the web application service, the `ShoppingService`, resides. We also define the same methods, but we don't implement them, in the client side. The methods definitions reside in an interface which is the representative of the web service in the client-side, the `ShoppingServiceProxy` interface. The proxy of the service is contained in an interface which extends `RequestFactory`, the `SmartShoppingRequestFactory`. The client first creates a communication interface between the client and the server using this `RequestFactory` interface. Then he can make the `RequestFactory` calls to the server by just invoking the methods that the proxy of the service defines. These calls are asynchronous, so the client doesn't wait for each call to complete before proceeding to the next call. The client-server interaction is outlined in a diagram in Figure 5.3.

⁴http://en.wikipedia.org/wiki/Java_Database_Connectivity

⁵http://en.wikipedia.org/wiki/Plain_Old_Java_Object

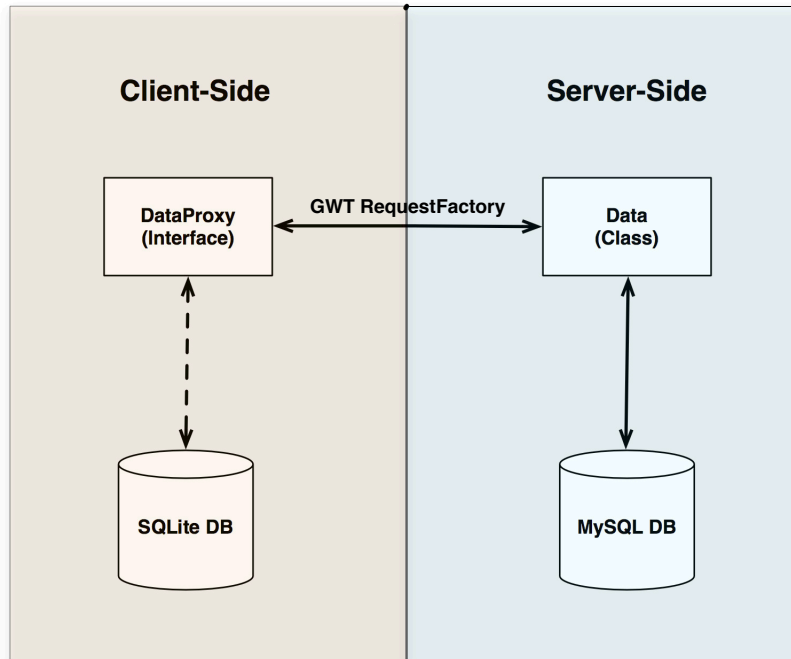


Figure 5.2: Client and Server Side Data Entities

We use a shared folder in our web project so the Android project can be linked to some parts of code of the web project. The shared folder is a way to combine both projects and enable client-server communication between an Android application and a GWT web application. Thus, as we explained in previous paragraphs, we have to put the RequestFactory interface and the proxy interfaces in the shared folder. They reside in the `com.smart.shopping.client` package of the shared folder. There are two more packages present in the shared folder. The first one, `com.smart.shopping`, contains the IP address of the web server that hosts our web application. We put it in the shared folder so that both the Android application and a web client can access to the web application. The other one, `com.smart.shopping.database`, contains classes that both the Android and the web application use but each one for a different purpose. The Android application hosts a local database and these classes are used for persisting the data of this database, while the web application uses these classes to represent the data entities in the server-side. This was done to avoid writing an additional set of the same classes to enable the intended functionalities.

Client-server communication in our application happens only when synchronization between the local and the remote database takes place. More details about what data are exchanged between client and server can be found on the synchronization section of the report.

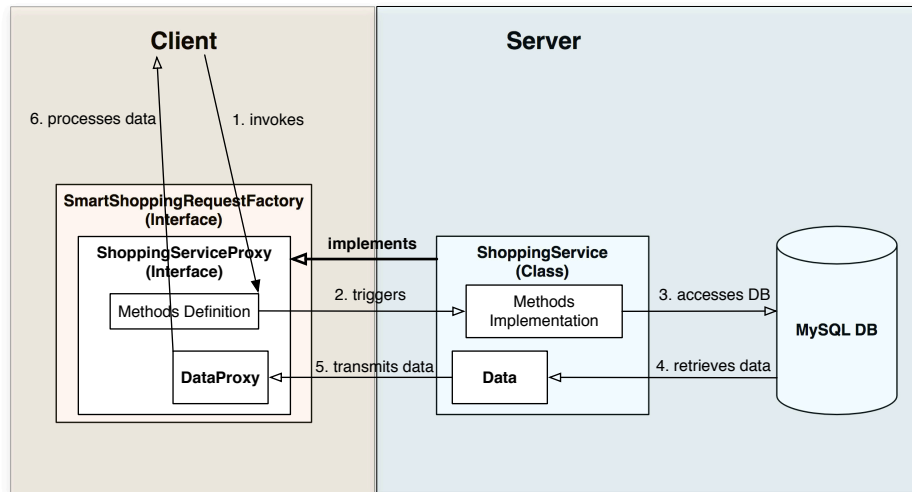


Figure 5.3: Client-Server Interaction

5.2 Connection to the database

To connect to the database from the server-side of the web application, we use the Java DataBase Connectivity (JDBC)⁶ API. As we explained earlier, we use a set of classes to persist the data that are exchanged in every server-database interaction. The names of the classes come from the corresponding database tables in which we use them. For example, the `AslUsers` class holds the data of the `AslUsers` table of the remote database. These classes contain variables with the same types and names as the columns of the corresponding database tables. They also contain getters and setters regarding their variables. We use such a class to store data to or retrieve data from all columns of a table of the database. The JDBC technology allows us to access to the MySQL database using SQL queries. We have implemented many kinds of SQL queries to all tables of the remote database for the purposes of our application. To be able to access to the database using this API, it is necessary we put a `mysql-connector.jar` file in the `war/WEB-INF/lib` folder of the web project (unless the server has such a `.jar` file already installed) and to declare the same `.jar` file as a library in the build path of our web project.

⁶http://en.wikipedia.org/wiki/Java_Database_Connectivity

Web Interface

As part of our semester project, we also develop a website that will allow users to have almost the same functionalities as when using our Android application but in a more convenient way (Figure 6.1). We integrate the OpenID standard^{1,2} to our web application to authenticate users using Google as our OpenID identity provider. After having signed in using their Google accounts, users can check their shopping lists, create shopping lists and check how much money they have spent shopping. They can create a shopping list easier when using this web interface than creating one when using their smartphones. They can also check more clearly what they have bought than trying to check all their purchases on their smartphones. This website aims at improving the user experience when using our application.

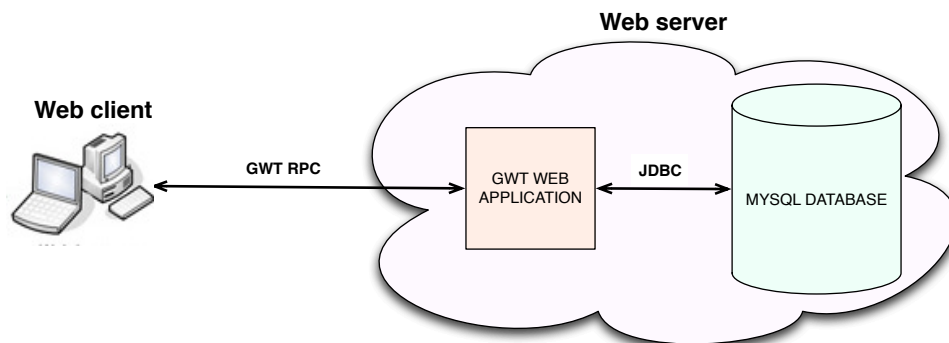


Figure 6.1: Using the Web Interface

¹<http://openid.net/>

²<http://en.wikipedia.org/wiki/OpenID>

6.1 GUI and Functionality

When a user visits <http://smart-shopping.ethz.ch/SmartShoppingServer> using a browser, he is directed to the website of our application. A login page appears on screen (Figure 6.2), displaying the logo and the name of our application on top of the page and below these a message is present prompting the user to sign-in using his Google account. Next to this message there is the sign-in button. When the user pushes this button, the browser is redirected to Google login page. Thus, Google authenticates the user and informs our application about his identity. More details about this standard and the authentication process are discussed in the following section.



Figure 6.2: Login Page

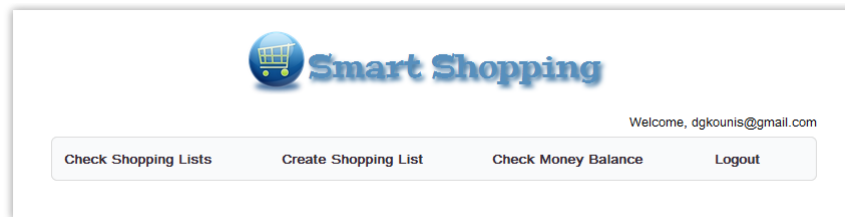


Figure 6.3: Home Page

After the user has logged in, a different web page appears on screen (Figure 6.3). The logo and the name of our application on top of the page are present and in the same position like in the login page. Below these and on the right of the page, a welcome message exists which is followed by the email of the user. Below these, there is a tab menu. The user can check the shopping lists he has created, create a new shopping list, check the money he has spent. He can also logout from this web page by choosing the corresponding tab. If he does that, his browser loads the login page again.

The user can check his shopping lists by pushing the appropriate tab of the menu (Figure 6.4). Then, below the menu and on the left side of the page, a set of all the shopping lists created by the user appears on screen and their corresponding entries on the right side. Each entry of a shopping list contains the name, the amount and the price of a product.

The user can also create a shopping list. After choosing the appropriate tab of the menu, two buttons appear on screen allowing the user to add a new

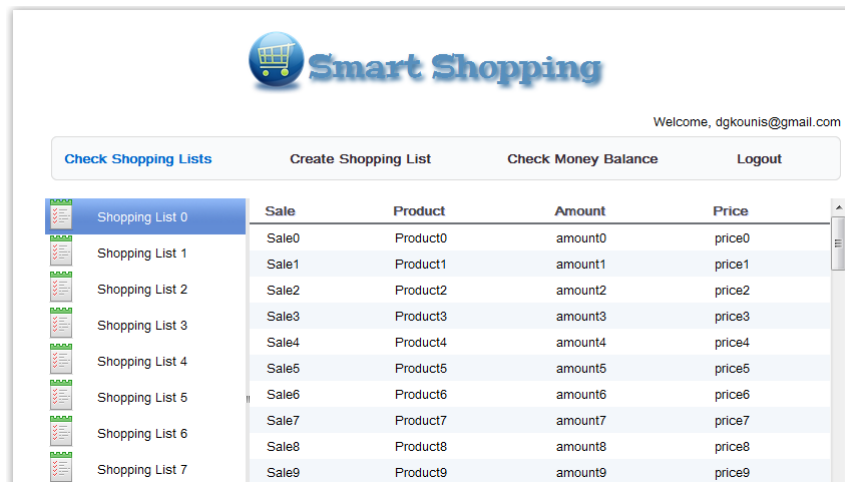


Figure 6.4: Check Shopping Lists Tab

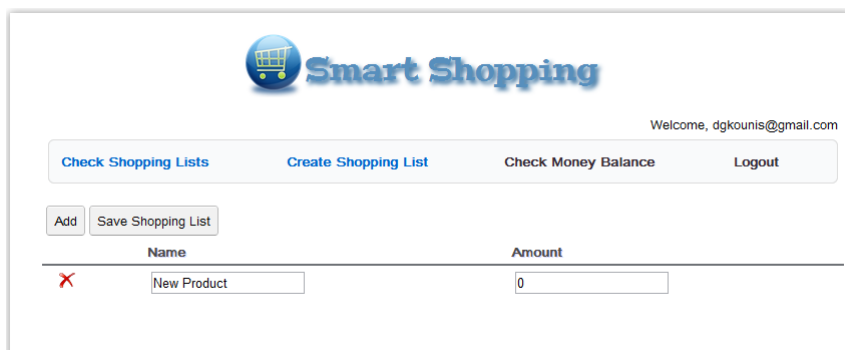


Figure 6.5: Create Shopping List Tab – Add Product

product to his new shopping list and to save this new list respectively. Each time the user presses the Add button, a new entry appears on the list letting the user add the name and the amount of the product (Figure 6.5). If the user changes his mind about a product he has added to the list, he can hit on the red marker on the left of this entry. This entry is then deleted from the list. When the user has finished making a shopping list, he can save this list to his set of lists by just pressing the corresponding button. A dialog then appears (Figure 6.6) asking the user to type in the name of the shopping list he just created. This dialog doesn't disappear from the screen unless the user presses the Cancel button resulting in returning to the list he just created or the OK button resulting in saving the list. If the user chooses the OK button, the dialog disappears and the web page is refreshed so the user can check his new list among his other shopping lists.

Furthermore, the user can keep track of the money he has spent shopping. The Check Money Balance tab of the tab menu of the home page displays the

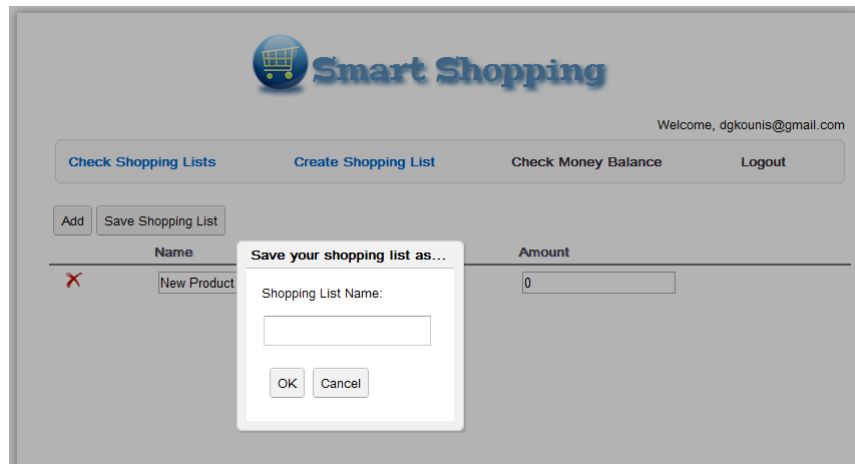


Figure 6.6: Create Shopping List Tab – Save Shopping List Dialog

amount of money spent and the date from which the amount started counting (Figure 6.7).



Figure 6.7: Check Money Balance Tab

To enable all the discussed functionalities of the website, we use the GWT-RPC mechanism³. The client (i.e. the user) by pushing a tab from the menu of the website invokes methods that are implemented on the server-side. In other words, the client makes RPC calls. In the server-side, many of the methods implemented access the MySQL database so as the user to see data about his shopping lists, and the money he has spent. Also, when the user creates a new list, the client-side sends the data regarding this list to the server using GWT-RPC and then the server-side stores this data to the remote database. GWT-RPC works in a similar way with the RequestFactory that we explained in the previous chapter.

³<https://developers.google.com/web-toolkit/doc/1.6/DevGuideServerCommunication>

6.2 OpenID

OpenID is an open standard that allows user authentication to be done in a simple way both for the user and the application that embeds this feature^{4,5}. The user has to have an account with an OpenID identity provider, such as a Google account, a Facebook account, a Yahoo account etc. In the case of the web interface that we develop, we only use Google as OpenID identity provider. The web interface constitutes a supplementary feature for the Android application. Considering that everyone that uses an application in an Android-based mobile phone can be expected to have a Google account, we think that everyone who wants to gain access to the web interface of our Android application can use his Google account. A user authenticates himself without having the need of creating a new account to access to the application. The authentication process is done via OpenID so our application doesn't have to store any usernames and passwords or verify if a user possesses a valid username-password value pair.

To enable OpenID for the authentication process of our web application, we use an OpenID library⁶. As we develop in Java, the OpenID4Java library⁷ is used.

A problem exists about integrating OpenID in a GWT-based web application. In a normal servlet development, redirections can happen in the server-side. But when using the GWT-RPC mechanism, we cannot make any redirections because the RPC calls in GWT are asynchronous. A solution to this is to make a redirection from the client-side by passing the redirection URL to the client. Thus, we make a GWT RPC call by invoking a method that implements the OpenID authentication process in the server-side. This method returns the URL that the browser will be redirected to so the user can authenticate himself. More details about the OpenID authentication process using Google as OpenID identity provider can be found on a Google relevant website⁸. In our web application, the gmail address of the user is requested from Google to help us identify the user and provide differentiated services.

⁴<http://openid.net/>

⁵<http://en.wikipedia.org/wiki/OpenID>

⁶<http://wiki.openid.net/w/page/12995176/Libraries>

⁷<http://code.google.com/p/openid4java/>

⁸<https://developers.google.com/accounts/docs/OpenID#AuthProcess>

Future Work and Conclusion

This semester project gives the opportunity to implement many more interesting ideas. We thought of some possible extension features that could be added to the next version of the project:

- Multiple foreign languages support for the mobile application and web interface
- Support for additional shops
- Extension of the website with all additional features of the Android application as browsing of reduction offers
- Sharing Shopping Lists with friends
- Dictionary-based corrections
- Price comparison
- Product suggestions based on purchases of friends in a shared group

In conclusion, we believe that this project has reached its goal. It is an efficient and user-friendly tool for the everyday shopper. Although, there are many smart-shopping-like applications on the Android market, none of them offers an OCR of receipts neither can compose a shopping list instead of the user. This makes the Smart Shopping project unique. During the implementation we faced many different challenges but the approach we took even added some non planned features. We first did not take into account that the receipt recognition could be so error-prone but this lead to the collaborative word correction which we find to be a very interesting feature. We learned many things while working on the project and especially while working in a team.

Building Tesseract

To build Tesseract on Mac OS X and Linux you need the following libraries:

- libleptonica-dev
- libpng12-div
- libjpeg62-div
- libtiff4-div
- zlib1g-div

Next step is to download the source package `tesseract-3.01.tar.gz` from the download page ¹. The build process is as follows:

```
./autogen.sh
./configure
make
sudo make install
sudo ldconfig
```

For more details check with the official web page ².

For Windows there is an installer which includes English language data.

If you want to run Tesseract from the command line the command is as follows:

```
tesseract image outputbasename [-l lang] [configs]
```

¹<http://code.google.com/p/tesseract-ocr/downloads/list>

²<http://code.google.com/p/tesseract-ocr/wiki/ReadMe>

To use the different language sets one should first download them from the official web site. Tesseract is not installed together with the language data.

Training Tesseract

The procedure is as follows. Create a text file with a text which is typically used for the training. It can be found in the training package on the web site and with the command:

```
convert -density 300 -depth 4 lang.font-name.exp0.pdf lang.font-name.exp0.tif
```

convert it into an image file. Here the lang should be replaced by the short form of the language that Tesseract uses. E.g. for English we put eng. The font-name is the the name of the font we train for and expN is the sequence number of the image for this exact font. Once the image files are present we move to the second step.

The training of Tesseract consist of first running Tesseract to detect automatically the letters in the images adding their x, y coordinates: `tesseract lang.font-name.exp0.tif lang.font-name.exp0 batch.nochop makebox`

After this first pass a box file with the name lang.font-name.exp0 is created for every image named lang.font-name.exp0.tif. In this box file there need to be a single character on every line and all letters from the image need to be in the box file in the correct sequence. On a second run, we manually need to check if the box file is correct and if not edit it according to the these rules. If there is a missing letter we need to add it to the box file on a new line or if two or more letters are recognized together we need to separate them and fill in the coordinates. This process is very time consuming and since this correction needs to be done manually it is probable that some letter can be missed or the coordinates are left incorrect.

After we are finished with the box files we feed them back into Tesseract with the following command:

```
tesseract eng.font-name.exp0.tif eng.font-name.box nobatch box.train.stderr
```

The next procedure is to detect all characters contained in all box files. This is done by:

```
unicharset_extractor *.box
```

Next, we create a `font_properties` file. This file needs to contain an entry for every font we train and set the following characteristics: `fontname` `<italic>` `<bold>` `<fixed>` `<serif>` `<fraktur>` to true or false.

e.g. `eng.verdana.box 0 0 0 0`

The last steps of the training are to create the clustering data and combine the files created so far:

```
mftraining -F font_properties -U unicharset -O lang.unicharset *.tr cntraining *.tr
```

```
combine_tessdata lang.
```

This last command creates the final language file used for the OCR.

Web Application Development

To be able to develop a GWT web application using the Eclipse IDE, we have to download and install the Google Plugin for Eclipse¹. We have to make sure that the GWT SDK and Google App Engine SDK are installed. To be able to run our web project, please go to the properties of the project and at the App Engine and Web Toolkit suboption of the Google option, disable the Use Google App Engine and Google Web Toolkit option respectively and then press OK. We do that only if these options are already enabled. Then, even if we hadn't enabled these options in the first place, enable them making sure that the App Engine and GWT SDKs are loaded to the options mentioned above. We can verify that by choosing the Configure SDKs suboption after we have chosen each of the Google App Engine and Web Toolkit options. This configuration is mandatory so as all necessary libraries for our project are loaded automatically. Furthermore, we have to do one last configuration. At the Java Compiler/Annotation Processing/Factory Path option within the properties of the project, please edit the path to requestfactory-apt.jar as `server_project_path/lib/requestfactory-apt.jar`. This is necessary to be able to use RequestFactory, i.e. to enable client-server communication.

¹developers.google.com/eclipse/docs/download

Web Application Deployment

In order to deploy the web application in a web server, there are some steps that we have to follow. At first, we have to put all the libraries used (e.g. .jar files) by the web application project into the `war/WEB-INF/lib` directory of it. Then, we can compile the application by choosing the GWT compile option of Eclipse. It would be useful to clean the project before attempting to compile it. We can do that by choosing Project and then Clean in Eclipse. By cleaning the project, we can verify that no error exists in the project that would prevent us from compiling it.

We have to make sure that before compiling (GWT compile) the web application project, we have set the `SERVER_URL` constant in the `Constants.java` file of the `com.smart.shopping` package in the shared folder of the project to the URL from which we can access to the web application. For example, if we deploy the web application in a local Apache Tomcat server, then we can access it by typing `http://localhost:8080/SmartShoppingServer` in our web browser. In this case, it would be better to use `http://IP` of the computer that hosts the local server:8080/SmartShoppingServer as the `SERVER_URL` constant. This is because, the OpenID provider uses this URL address to redirect the browser after the user has successfully signed in.

A point to pay attention when pressing GWT compile is the Entry Point Modules field on the GWT compile window. The two modules of our application, the `HomeModule` and the `LoginModule` should be loaded. If an error occurs, it would be useful to remove and add these two modules again.

As part of our semester project we had to deploy our web application in a web server owned by the Distributed Computing Group of ETH Zurich. As a result, one can have access to our web application by using the URL: `http://smart-shopping.ethz.ch/SmartShoppingServer`.

After having successfully compiled the project, we have to compress (zip) all the contents of the `war` directory of the project and then rename this zipped file to `SmartShoppingServer.war`. To deploy the web application to an Apache

Tomcat server, like the one the Distributed Computing Group assigned us, we have just to put the .war file into the /webapps folder of the server.