Bachelorthesis

Shun Ling Chin

Implementierung einer Datenerfassungs-App mit unterlagerter Datenbank

Shun Ling Chin

Implementierung einer Datenerfassungs-App mit unterlagerter Datenbank

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Wilfried Wöhlke
Zweitgutachter : Prof. Dr. Andreas Suhl

Abgegeben am 2. Februar 2017

Shun Ling Chin

Title of the Bachelor Thesis

Implementation of a Data Acquisition App with an Underlying Database

Keywords

Android, Application, Database, Data Transmission, FTP Server

Abstract

The aim of this bachelor thesis is to develop an Android application for managing the sales transaction using mobile devices. This mobile application allows user to add and edit sales transaction records, as well as to keep an overview of the transaction history. The records are first collected and being stored in a local database, it can be done without internet connection. The stored records can be uploaded to a server manually or at a scheduled time.

Shun Ling Chin

Thema der Bachelorthesis

Implementierung einer Datenerfassungs-App mit unterlagerter Datenbank

Stichworte

Android, Applikation, Datenbank, Datenübertragung, FTP Server

Kurzzusammenfassung

Das Ziel dieser Bachelorarbeit ist die Entwicklung einer Android Applikation zur Verwaltung von Verkaufstransaktion mit mobilen Geräten. Die mobile Applikation ermöglicht dem Benutzer neue Transaktionen zu generieren und existierende Transaktionen zu bearbeiten. Eine übersichtliche Aufbereitung der Transaktionsdaten gewährleistet einen stetigen Überblick für den Benutzer. Die Daten werden in einer lokalen Datenbank auf den mobilen Gerät eingetragen und können manuell oder nach geplanten Zeitintervallen auf einen Server übertragen werden.

Acknowledgements

First of all, I would like to express my greatest appreciation to Prof. Dr.-Ing Wilfried Wöhlke, my advisor. Not only that I want to thank him for suggesting this bachelor thesis and agreeing to supervise my work, but also for his helpful advices and valuable comments given to me throughout the whole process of completing this thesis. Besides that, I would also like to thank Prof. Dr. Andreas Suhl for agreeing to be my second examiner.

Other than that, this journey would not have been possible without the unflagging support from my parents, my boyfriend and my friends. To my parents, I would like to express my gratitude for their unconditional love and encouragement. Besides that, I feel extra thankful for my boyfriend, for his understanding as well as the emotional support. And last but not least, thanks for the moral support from all of my friends.

Contents

List of Tables

List of Figures

1. Introduction

”Mobile use is growing faster than all of Google’s internal prediction”, said Eric Schmidt, a former CEO of Google. Indeed, the world is undergoing a rapid technological revolution since Apple released their first iPhone in 2007 in order to reach the next goal in computing. The poster child of this technological boom is mobile devices. Mobile devices, having computational prowess that previously require much larger form factors, open up a whole new array of possibilities. The portability and affordability of mobile devices has brought computing to the masses at a scale the world had never witnessed before.

The fact that mobile devices are packaged with essential functionalities in today’s world, such as telephony and internet surfing, make them penetrate into the market quickly. For example, it is a logical choice for most consumers to pick a mobile device that can do telephony and much more over a conventional mobile phone. This high demand of mobile devices allow mass production of the components, which has cut down the cost of production tremendously. While the high-end devices offer many more features and demand a higher price tag, it is the low-end devices that are able to provide the ever-more affordable general-purpose computing.

Conventionally, business relies on dedicated solutions. One of the smartest example is the cash register that is only used for calculating and registering transaction at the point of sale. In comparison to the ubiquitousness of mobile devices as a whole, the demand for these single-purpose solutions is limited and thus could not be produced cost-effectively. With the extendibility of mobile devices as general-purpose computing units, new functionalities can be provided on existing devices in the form of mobile applications. Combined with the reusability as they can be easily repurposed in the future, the mobile devices present themselves as an attractive and better alternative to the conventional dedicated solutions.

The purpose of this thesis is to develop a mobile application that extends the functionality of a mobile device as a general purpose device to manage and track sales transactions. Each mobile devices with this mobile application, will be able to track transaction independently while synchronizing data between each other on demand. The mobile application allows the users to add, modify and store the latest sales transaction as well as the information of the users. Besides that, it should also be able to display overview page of the history transactions.

The rest of the thesis is structured as follows. Chapter 2 gives a general introduction on Android, Android Studio, and database. Chapter 3 focuses on the design and development of Graphic User Interface (GUI) as well as elaborates basic and special functions of the mobile application. Chapter 4 discusses the implementation of database for the collected data to store locally. Chapter 5 examines the data transmission from the database to server. Finally, Chapter 6 outlines the summary and overviews for this entire thesis.

2. Background

2.1. Android

Android system, which is now acquired by Google, is a software stack that consists of an operating system (OS) and a Software Development Kit (SDK) for mobile devices.

Android is an open system that is free to use by the developers to build any kinds of applications with the provided tools and Application Programming Interface (API) using Java programming language. In addition, Android, which is built on open source Linux community with about 300 software, hardware and carrier partners, has become the fastest-growing mobile OS.

Being the world's most popular mobile operating system, Android has hit 1.4 billion active Android users in more than 130 countries around the globe according to the statistic last updated by the Google CEO Sundar Pichai during the Nexus event in September 2015. As compared to May 2014, Android has 400 million new users. Besides that, from the last reported figures in May 2016, it stated that 65 billion apps had been downloaded from Google Play. Due to the obvious popularity of Android, the reported figures had shown that the market for Android mobile application is very huge. Hence, Android is the chosen system to develop the mobile application in this write up.

2.2. Android Studio

Android Studio is the official integrated development environment (IDE) for Android platform development. It has the fastest tools to create mobile applications on each type of Android device.

2.2.1. Project Structure

Table 2.1.: Purposes of each file in the project structure

File	Purposes
manifests	defines the components and characteristics of the application
java	contains the class definition for the activity
res	stands for resources
drawable	contains graphic files for the application
layout	configures layout behavior of an activity's User Interface (UI) components
menu	stores layout file for the menu bar of the application
mipmap	stores app or launcher icon file
values	holds values for text strings and other data type in the application
Gradle Scripts	compile and build the application

2.2.2. Component

The Android framework consists of different components to build an application.

Activity is a single display with UI that a user can have interaction with the application.

Intent is an abstract messaging object that use to ask for an action to be performed.

2.3. Database

A database, which is normally maintained by database management system (DBMS), is a comprehensive collection of data that is organized for easy access. DBMS is a program that allows user to update, store or retrieve information from database. Since database and DMBS are always closely bonded with each other, the term 'database' is generally applied to both of them.

One of the most common database models is relational database, which is invented by E.F.Codd in 1970 (Codd, 1970). Relational database displays data in tables with columns and rows. Furthermore, it is managed by Relational Database Management System (RDBMS) and uses Structured Query Language (SQL) for data manipulation. Due to the easy extendability and convenience, relational database is chosen as the database model for this application (see Chapter 4).

3. Functionality

As previously mentioned in Chapter 2, Android Studio is certainly the easiest and most convenient mechanism to create a user interface for a mobile application. Thus, Android Studio is used to develop this application. This chapter aims to provide an overview on how to design the GUI of application and to elaborate the basic functions and special functions of the application in details. Only part of the code are selected for discussion, complete code for the application can be viewed at Appendix A.

Before starting to program the application, there are a few requirements that have to be clearly identified as the application will be built according to these criteria. It is incredibly important to recognize all application requirements before programming begins. Main requirements of this application are briefly explained below.

Configurations of barcode scanner

The application has to be developed to configure with a barcode scanner in order to get the information of the customers from their customer card.

Choice of mobile devices

The application should be able to implement on tablet and smart phone.

Offline function

Even if the mobile devices has no network coverage, the application should be able to keep collecting data and the collected data will be synchronized with the server once the mobile device is connected back to the internet again. This is great for those who are using this mobile application in rural areas or in areas with low network coverage.

3.1. Basic Functions

First and foremost, an activity is necessary to be chosen as the launcher activity so that the application is able to run on a mobile device when the user clicks on the icon of the application. Without a launcher activity, the operating system will not know which activity to initiate when the application first launches.

The following XML code is under the manifest file *AndroidManifest.xml*. The launcher intent filter specifies that the ”MainActivity” (sale mode (see section 3.1.2)) is selected as the launcher activity.

```
<application
  android:allowBackup="true"
  android:icon="@mipmap/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/Theme.AppCompat.Light.NoActionBar" >
  <activity
    android:name=".MainActivity"
    android:launchMode="singleTop"
    android:screenOrientation="landscape" >
    <intent-filter
      android:label="@string/app_name" >
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER"
         />
    </intent-filter>
  </activity>
```

3.1.1. Login

Once the application is started, a dialog (see Figure 3.2) for login will pop out requesting user to enter their password in order to proceed. However, if the application is being used for the first time, the input field of register number will show ”Nicht Gesetzt” to indicate that the register number has not set up. In this case, technician will have to use master password to log into the application and set up a register number at Settings (see Section 3.2.6). User’s password can be created at activity Login Data (see Section 3.2.1). Subsequently, technician can log out by clicking on the option ”Abmelden” as seen in Figure 3.1 to allow user to log in using the newly created password.

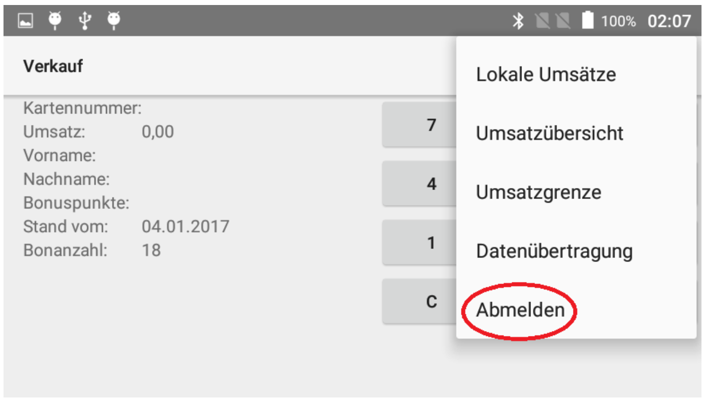


Figure 3.1.: Link to log out

Layout

Figure 3.2 shows the UI of login dialog.

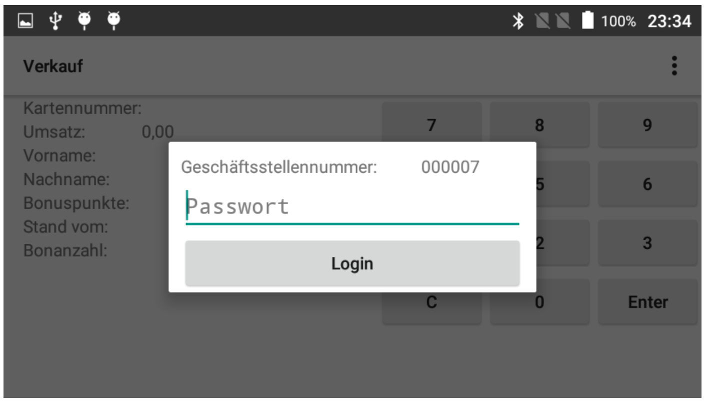


Figure 3.2.: Login dialog

The layout file *dialog_login.xml*, which is shown in the below code, defines the attributes of TextViews register number and its text field, EditText password as well as the Button login.

Linear layout is used to build the layout as shown in Figure 3.2). It is a view group that aligns all the views in either vertical or horizontal order. View group is the base class for layout that consists of views (also known as children) and a view is a basic building block which reacts to user input, for example Button and EditText.

The following codes describe the attributes of linear layout for login dialog. The dialog is specified in a vertical mode. Layout height is set as match parent, which means its height is same as the parent. The width of layout is set as 300 Density-independent Pixels (dp). Dp is an abstract unit that is based on the physical density of the screen. Besides that, the views inside of the linear layout arrange themselves 10dp from top, left and right using the statement *padding*. Padding refers to the inside of a view. The background for this login dialog is set as white colour.

```
LinearLayout
  android:orientation="vertical"
  android:layout_height="match_parent"
  android:paddingLeft="10dp"
  android:paddingRight="10dp"
  android:background="#fff"
  android:layout_width="300dp"
  android:paddingTop="10dp">
```

TextViews register number and its text field are enclosed in the relative layout. Relative layout allows the child views to position relatively to each other. Both of the height and width for relative layout are wrap content. By doing so, relative layout only displays big enough to enclose the child views in the dialog. Each of the TextViews are given their own ID through the statement *android:id*.

```
<RelativeLayout
   android:layout_width="wrap_content"
   android:layout_height="wrap_content">
   <TextView
     android:id="@+id/txtRegisterNrLabel"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Geschäftsstellennummer:">
 </TextView>

 <TextView
     android:id="@+id/txtRegisterNr"
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:gravity="center"
     android:layout_toRightOf="@id/txtRegisterNrLabel"
     android:hint="@string/not_set">
 </TextView>
</RelativeLayout>
```

EditText is a veneer for text entry. EditText for password has a height of wrap content and width is set as match parent since the length of user's password is unknown. The *android:hint* statement displays hint text for user when the input field is empty. The hint text is set as "Passwort". In addition, to indicate that EditText is used to key in password, it must have *inputType* set to *Textpassword*. In order to have the initial focus at the input field for password, the *requestFocus* tag is required for this EditText. Without doing this, the application will automatically focus on the first button.

```
<EditText
android:id="@+id/txtPassword"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:inputType="textPassword"
android:hint="Passwort">
<requestFocus></requestFocus>
</EditText>
```

Button is a widget that can be clicked in order to perform an action. It is used here to allow user to log into the application. Its height is wrap content whereas the width is match parent, which is as wide as the dialog.

```
<Button
  android:id="@+id/btnLogin"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="Login">
</Button>
```

Implementation

As the login dialog is part of the sale mode (see section 3.1.2), the complete java code for the implementation is under *MainActivity.java*.

The method *setCancelable()* is implemented so that user cannot dismiss dialog when accidentally clicks out of it or the back button of mobile device. Besides that, since a title is not needed for this login dialog, *requestWindowFeature()* method is used to remove the title bar of the dialog.

```
login.setCancelable (false);
login.requestWindowFeature(Window.FEATURE_NO_TITLE);
```

To react to the click on the Login button, the method *btnLogin.setOnClickListener()* is called. After that, the user's login credentials will be verified. At the end of the process, a toast message will display to indicate whether user has successfully logged in or the login process is failed due to different reasons. A toast message is a small popup that gives simple response about an operation and will automatically disappear after awhile.

The application is being used for the first time

If the entered master password is correct, toast message is shown to inform user that master login is successful and user will be directed to sale mode page.

```
if (Utility.MASTERPWD.equals(passwordString)) {
   loggedIn = true;
   masterMode = true;
   Toast.makeText(MainActivity.this,
      "Master Login Erfolg", Toast.LENGTH_LONG).show();
   login.dismiss();
   init();
}
```

Toast message will display to notify user that the register number has not set up, if user tries to log in without using the master password.

```
else if (registerNrLength == 0) {
   Toast.makeText(MainActivity.this,
      "Die Geschäftsstellen-Nummer ist noch nicht gesetzt.",
          Toast.LENGTH_LONG).show();
}
```

Also, when user leaves the password input field empty and clicks on the login button, toast message will pop up requesting user to enter the password.

```
else if (passwordLength == 0) {
   Toast.makeText(MainActivity.this,
      "Bitte geben Sie das Passwort ein.",
          Toast.LENGTH_LONG).show();
}
```

Register number has been set up

When the application could not retrieve any stored password, toast message is shown to notify user that no password is being registered. User has to set up the password at login data activity (see section 3.2.1).

```
else {
   String storedPassword =
      Utility.retrievePassword(MainActivity.this);
   if (storedPassword == null) {
    Toast.makeText(MainActivity.this,
    "Noch keine Passwort ist gesetzt", Toast.LENGTH_LONG).show();
  }
```

If the entered password is same as the stored password, toast message will display to show that the login is success.

```
else if (storedPassword.equals(passwordString)) {
  loggedIn = true;
  Toast.makeText(MainActivity.this,
    "Login Erfolg", Toast.LENGTH_LONG).show();
  login.dismiss();
  init();
  }
```

Toast message pops up to indicate that the entered password is incorrect and user has to re-enter the password in order to proceed.

```
else {
  Toast.makeText(MainActivity.this,
    "Falsches Passwort. Bitte versuchen Sie nochmal.",
       Toast.LENGTH_LONG).show();
  }
```

Shared preferences is used to store and retrieve the password. It is a method that is used to refer to file that consists of the key-values pair and give simple methods to write and read them. The implementation of this method can be found under *Utility.java*. The best way is to use the default shared preferences as it simplifies things by allowing the same preferences to access throughout the whole application without needing to specify the file name.

To save user's password to a shared preferences file, *SharedPreference.Editor* has to be used. Key and value are transfered to the preferences editor by using the method *putLong()* and method *commit()* is called to save them. In this case, *SETTING_PASSWORD* is used as the key to save the password.

```
public static boolean storePreference(Context context, String
   key, long value) {
  SharedPreferences preferences =
     PreferenceManager.getDefaultSharedPreferences(context);
  SharedPreferences.Editor editor = preferences.edit();
  return editor.putLong(key, value).commit();
}
```

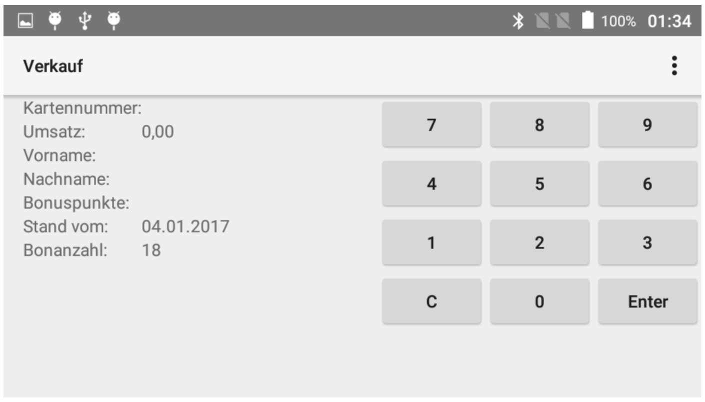
```
public static boolean storePassword(Context context, String
    password) {
  return storePreference(context, SETTING_PASSWORD, password);
}
```

The following method is used to retrieve the saved password. By providing the key that used to store the password, *getLong()* is called to get the saved password. If the preference file does not exist, it returns to default value, which is in this case null. Here, it means that no password can be retrieved as the user's password has not set up. In addition, if the Class Cast Exception (inappropriate of changing a class from one type to another) happens, it will also return the value to default value. For example, if the saved password is numeric type float, but here the demanded value is type string. Therefore, Class Cast Exception will occur.

```
public static String retrievePreference(Context context, String
    key, String defValue) {
  SharedPreferences preferences =
      PreferenceManager.getDefaultSharedPreferences(context);
  try {
    return preferences.getString(key, defValue);
  } catch (ClassCastException e) {
    Log.d("[Utility]", "Retrieving Preference string '" + key +
        "': " + e.getMessage());
    return defValue;
  }
}

public static String retrievePassword(Context context) {
  return retrievePreference(context, SETTING_PASSWORD, null);
}
```

3.1.2. Sale mode

After the user has logged in successfully, the main screen displays the activity sale mode (see Figure 3.3). In addition to showing the latest update date and the total number of receipt, sale mode allows the user to retrieve a customer's information by scanning the customer card and input new sale transaction details. This app bar of this screen also acts as the gateway to other functions of the application.


Verkauf

Kartennummer:
Umsatz: 0,00
Vorname:
Nachname:
Bonuspunkte:
Stand vom: 04.01.2017
Bonanzahl: 18

Figure 3.3.: Sale Mode activity

Layout

Figure 3.3 shows the layout of the sale mode. The app bar at the top of the screen displays the title and an overflow menu icon on the right. Overflow menu enables user to switch to the other activities of the application. On the left side of sale mode, it is the details panel which consists of TextViews that show various transaction information. At the right side of the screen, a custom keypad is implemented in place of the default input method to better integrate with the sale mode. While the default input method is more dynamic, it applies to a specific text view. For example, while the default input method also has the pseudo Cancel and OK buttons that could be used for clearing the input from the barcode scanner or inserting data in the database, they might be shown in a way that emphasize volume input field only. In comparison, having the custom keypad shown at all time alongside with the whole view of the sales details at the left side, provides a better user experience, as the Cancel and Enter button affect multiple input fields at once. The complete layout file for this activity is under *activity_main.xml*. Selected parts of the layout code are discussed below.

The following codes describe the attribute of linear layout in sale mode. The orientation of sale mode is specified in vertical order and it is given an identity (ID) name as main layout. Both of the height and width are set as match parent, so that it could utilize all the space of the screen. In order to get the focus from keypad, the statement *focusable* is set to true. Also, the statement *focusableInTouchMode* is stated true to let the view gains focus when the mobile device is in touch mode.

```
LinearLayout
   android:layout_height="match_parent"
   android:layout_width="match_parent"
   android:orientation="vertical"
   android:focusable="true"
   android:focusableInTouchMode="true"
```

```
  android:id="@+id/main_layout">
```

Codes below define the app bar at the top of the activity. The app bar is implemented using Android support library's Toolbar class to ensure that its behavior is consistent on all recent devices. It is assigned with a special ID so that it can be accessed later in the code. Width of app bar is set to match parent so that it will span the width of the screen. The height of this app bar is the same as the height of Android default app bar. The ”?” symbolizes an operator that used to access system configuration in Android. For this app bar, it is customized to have a light colour theme.

```
<android.support.v7.widget.Toolbar
  android:id="@+id/my_toolbar"
  android:layout_width="match_parent"
  android:layout_height="?attr/actionBarSize"
  android:background="?attr/colorPrimary"
  android:elevation="4dp"
  android:theme="@style/ThemeOverlay.AppCompat.ActionBar"
  app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>
```

Relative layout is used to enable the child views to position themselves relative to each other. By default, the first view in a relative layout will start from the top left of the layout. To specify each and every view, there are various layout parameters available. Here, the layout parameter is used to customize an extra space of 16dp from the left. The width and height of relative layout are both match parent.

```
<RelativeLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:layout_marginLeft="16dp ">
```

TextViews that are used for labeling and its corresponding input field are situated at the left side of the layout. In this situation, EditText is not used as input field because the input entry is only allowed via scanning the barcode with scanner or through the built-in keypad. If EditText is applied here, user will be able to key in the input with mobile device's keypad. All the TextViews for labeling have an attribute of wrap content for their width and height. The text to appear on the screen is also being specified. As for the TextViews that are used as input field, both the width and height are set as match parent since the length of the input text is unknown. Each of the TextViews are also given an ID. TextView ”Kartennummerlabel” is used as the reference for the other views to arrange their position by using the layout parameters.

Text View for labeling

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Kartennummer:"
  android:id="@+id/KartennummerLabel"/>

<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Umsatz:"
  android:id="@+id/UmsatzLabel"
  android:layout_below="@id/KartennummerLabel"/>
```

Text View for input field

```
<TextView
  android:layout\_width="match\_parent"
  android:layout\_height="match\_parent"
  android:id="@+id/Kartennummer"
  android:layout\_toRightOf="@id/KartennummerLabel"
  android:layout\_toLeftOf="@id/button7"
  android:layout\_alignBottom="@id/KartennummerLabel"/>

<TextView
  android:layout\_width="match\_parent"
  android:layout\_height="match\_parent"
  android:id="@+id/Umsatz"
  android:layout\_toRightOf="@id/KartennummerLabel"
  android:layout\_toLeftOf="@id/button7"
  android:layout\_alignBottom="@id/UmsatzLabel"
  android:layout\_below="@id/Kartennummer"/>
```

The attributes of button are defined below. First and foremost, the *focusable* in button is set as false so that the button will not be focused when it is being clicked but only executes the *onClickListener*. This method will be further discussed in the implementation section. The width and height for all the buttons are wrap content. Also, they are all given a name as ID and text to display on the screen. The statement *layout_alignParentRight* is used for button 3, 6, 9 and Enter so that the whole keypad is aligned to the right corner of the layout.

```
<Button
  android:focusable="false"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="1"
  android:id="@+id/button1"
  android:layout_toLeftOf="@id/button2"
  android:layout_below="@id/button4"/>

<Button
  android:focusable="false"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Enter"
  android:id="@+id/buttonEnter"
  android:layout_alignParentRight="true"
  android:layout_below="@id/button3"/>
```

Implementation

A flow diagram (see Figure 3.4) is created for the implementation of sale mode activity, as it involves several different tasks that the application has to perform. Each part of the implementation will be explained along the java code below.

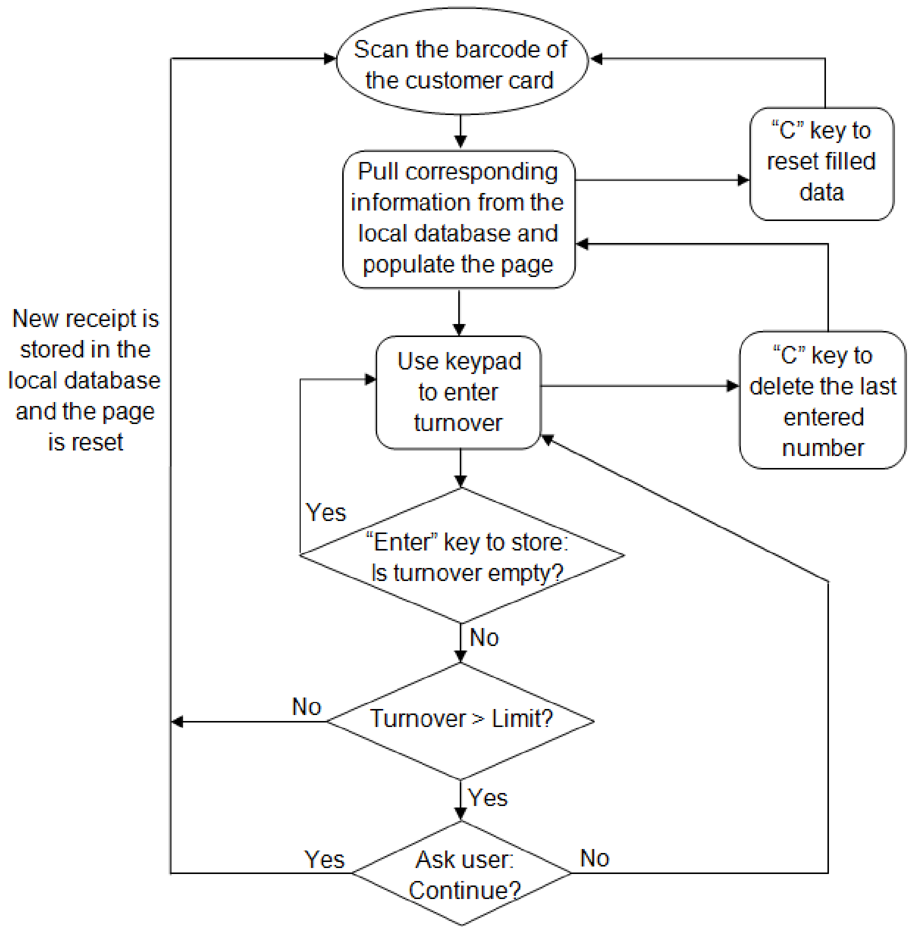

Scan the barcode of the customer card
Pull corresponding information from the local database and populate the page
"C" key to reset filled data
Use keypad to enter turnover
"C" key to delete the last entered number
New receipt is stored in the local database and the page is reset
Yes
"Enter" key to store: Is turnover empty?
No
No
Turnover > Limit?
Yes
Yes
Ask user: Continue?
No

Figure 3.4.: Flow diagram for the implementation of sale mode

The code for implementing the sale mode activity is under *MainActivity.java*.

Firstly, *onCreate()* method is executed when sale mode activity is created. It is then followed by calling *setContentView()* to open the UI of this activity.

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
```

The following shows that the toolbar view is being searched in the main activity layout. Toolbar is set as app bar for this activity window in order to be compatible with the old Android version before app bar was introduced.

```
Toolbar myToolbar = (Toolbar)findViewById(R.id.my_toolbar);
setSupportActionBar(myToolbar);
```

The input field for the TextView ”Kartennummer” and ”Umsatz” are initialized. The value ”0,00” is displayed on the ”Umsatz” input field .

```
kartennummerLabel = (TextView) findViewById(R.id.Kartennummer);
umsatzLabel = (TextView) findViewById(R.id.Umsatz);
umsatzLabel.setText("0,00");
```

Once the user has successfully entered sale mode, the date that the bonus file is created and the number of receipts stored in local database are shown on the screen. Implementation of database will be discussed in Chapter 4. *Shared preferences* is used to retrieve the date (see *Utility.java*). The date that shows on the screen is displayed in the following format.

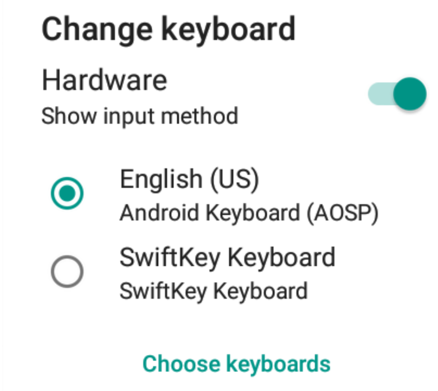
```
private static final String DATE\_FORMAT = "dd.MM.y";
```

PREFERENCE_EFFECTIVE_DATE is used as the key to retrieve the date. If there is no bonus file in the database, the displayed date will always be the current date. Otherwise, the application will show the date of bonus point calculation.

```
public static String retrieveEffectiveDate(Context context) {
   String effectiveDate = retrievePreference(context,
      PREFERENCE\_EFFECTIVE\_DATE, null);
   if (effectiveDate != null) {
     return effectiveDate;
}

public static String retrieveEffectiveDate(Context context) {
   return retrievePreference(context, PREFERENCE_EFFECTIVE_DATE,
      currentDate());
}
```

After that, the mobile device is connected to barcode scanner via Bluetooth. When the barcode scanner is connected for the first time, a toast message will pop up and require the user to enable the ”Hardware - Show Input Method” (see Figure 3.5). Enabling the device setting ”Hardware - Show Input Method” is necessary to keep the onscreen keyboard enabled even if a hardware input device (e.g. barcode scanner) is connected.

```
if (getResources().getConfiguration().hardKeyboardHidden ==
    Configuration.HARDKEYBOARDHIDDEN_NO) {
   ((InputMethodManager)getSystemService
   (Context.INPUT_METHOD_SERVICE)).showInputMethodPicker();
   Toast.makeText(this,
   "Ein Barcode-Scanner ist angeschlossen und die
      Bildschirmtastatur ist damit standardmäßig deaktiviert.
      Bitte aktivieren Sie die Bildschirmtastatur.",
   Toast.LENGTH_LONG).show();
}
```


Change keyboard

Hardware
Show input method [✓ on]

[✓ English (US)
Android Keyboard (AOSP)]

SwiftKey Keyboard
SwiftKey Keyboard

Figure 3.5.: Setting for Hardware - Show Input Method

Scanning the barcode on the customer card will trigger the method *onKey()*, thus, the data of the customer (card number, first name, last name and bonus point) can be pulled from the local database and populate the page. A toast message is shown to indicate that the barcode scan has completed.

```
@Override
public boolean onKey(View v, int keyCode, KeyEvent event) {
   if (event.getAction() != KeyEvent.ACTION_UP) {
      return false;
   }
```

```
  if (keyCode == KeyEvent.KEYCODE_ENTER) {
    kartennummerLabel.setText(scanSequence);

    new AsyncTask<String, Void, DatabaseHelper.Kunde>() {

      @Override
      protected DatabaseHelper.Kunde doInBackground(String...
         params) {
        return db.getCustomerWithNumber(params[0]);
      }

      @Override
      protected void onPostExecute(DatabaseHelper.Kunde result) {

        currentKundeID = result._id;
        vorname.setText(result.vorname);
        nachname.setText(result.nachname);
        bonuspunkte.setText(result.bonuspunkte);
      }
    }.execute(new String(scanSequence));

    scanSequence = "";
    Toast.makeText(MainActivity.this,
    "Barcode scan completed.", Toast.LENGTH\_LONG).show();
    return true;
  }
```

However, despite that the scan is succeeded but the card number of the customer cannot be found on the database, no result will be displayed on the screen. Instead, toast message will pop out to inform user that the card number is not recognised.

```
if (result == null) {
  currentKundeID = -1;
  kartennummerLabel.setText("");
  vorname.setText("");
  nachname.setText("");
  bonuspunkte.setText("");
  Toast.makeText(MainActivity.this,
  "Die Kartennummer '" + scanSequence
  + "' ist nicht bekannt.", Toast.LENGTH_LONG).show();
  return;
}
```

Functionality is added to all the buttons (keypad) and a listener is assigned to each and every one of them. When user clicks on the button, the listener of the button is triggered and thus onClick() method is called to display the entered value or to perform their respective action.

Furthermore, user can reset the filled data by clicking on the Cancel button. If there is a value at turnover, it has to be removed first only then the filled data can be cleared. As the decimal comma for the turnover value is fixed in place, entering a number from the numpad push the exisitng number to the left. It is the same when user clicks on the Cancel button, the last entered digits will be deleted and the digits in front will move one digit backward. For example, the entered value is 2,39. When user presses the Cancel button once, it becomes 0,23. Press the Cancel button for a second time, it goes to 0,02. And when user clicks the Cancel button for the third time, the stated value returns to default value 0,00.

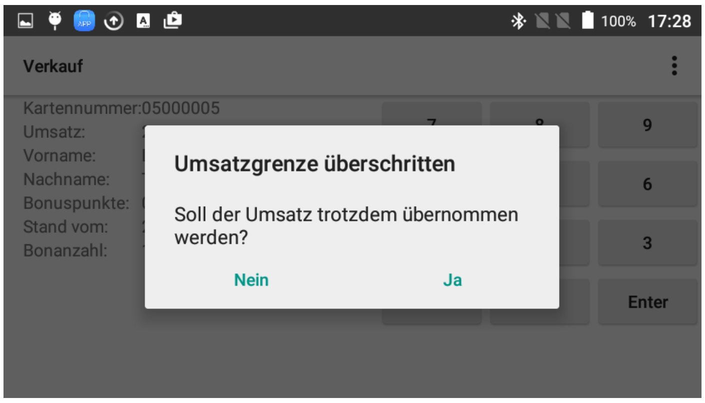
```
cancelButton.setOnClickListener(new View.OnClickListener() {
public void onClick(View v) {
switch (digitTracker) {
   case 0:
   currentKundeID = -1;
   kartennummerLabel.setText("");
   vorname.setText("");
   nachname.setText("");
   bonuspunkte.setText("");
   return;

   case 1:
   umsatzLabel.setText("0,00");
   break;

   case 2:
   CharSequence text = umsatzLabel.getText();
   umsatzLabel.setText("0,0" + text.charAt(text.length() - 2));
   break;

   case 3:
   String[] values = umsatzLabel.getText().toString().split(",");
   umsatzLabel.setText("0," + values[0] + values[1].charAt(0));
   break;

   default:
   values = umsatzLabel.getText().toString().split(",");
   umsatzLabel.setText(values[0].substring(0, values[0].length() -
      1) + ',' + values[0].charAt(values[0].length() - 1) +
      values[1].charAt(0));
   }
   --digitTracker;
```

```
    }
});

}
```

A digit tracker is applied to make sure that if the last digit displayed on the turnover input field is zero, the number in the input field will remain unchanged when button 0 is pressed. Unless the last digit is not a zero, only then number zero will be inserted into the input field.

```
final Button btn0 = (Button) findViewById(R.id.button0);
btn0.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        if (digitTracker == 0) {
            return;
        }
        insertValue('0');
    }
});
```

The Enter button is used to save the transaction. After all the information has been filled in, user has to click on Enter button in order to store the data. Before saving, it is checked whether there is a value at turnover input field. Also, if the turnover that user keys in exceeds the preset turnover limit, an alert dialog (See Figure 3.6) will pop out to ask user if the save process should continue despite that it has overlimit. Otherwise, if entered turnover did not exceed the limit, new receipt will be stored into the database and the page will reset.


Umsatzgrenze überschritten

Soll der Umsatz trotzdem übernommen werden?

Kartennummer:05000005
Umsatz:
Vorname:
Nachname:
Bonuspunkte:
Stand vom:
Bonanzahl:

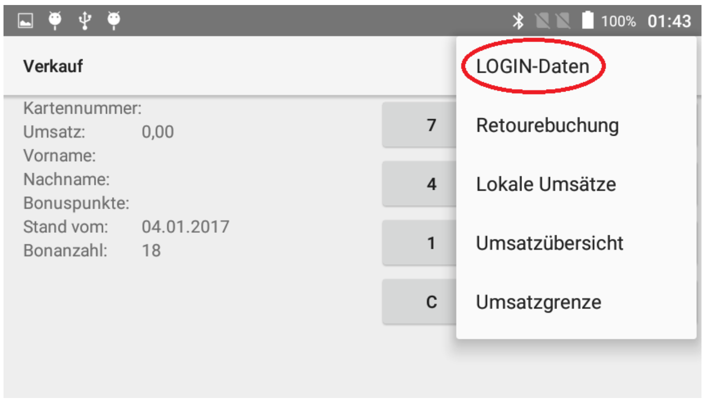
Figure 3.6.: Warning when the entered turnover exceeds preset turnover limit

```
if (limit != 0 && umsatzValue > limit) {
    AlertDialog.Builder builder = new
        AlertDialog.Builder(MainActivity.this);
    builder.setTitle("Umsatzgrenze überschritten")
    .setMessage("Soll der Umsatz trotzdem übernommen werden?")
    .setPositiveButton(R.string.yes, new
       DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int id) {
        storeReceipt(btnEnter, values);
      }
    })
    .setNegativeButton(R.string.no, null)
    .show();
} else {
    storeReceipt(btnEnter, values);
}
```

3.2. Special Functions

3.2.1. Login Data

There are a few special functions that are embedded into this application in order to create an application with better functionalities for the user. One of the special functions is to allow user to alter the login data. User can use this function to set up the password or change a new password. The register number is however not modifiable by normal users and can only be changed by the technician who logged in using a master password. To enable this function, user has to click on the option ”LOGIN-Daten” at overflow menu in sale mode (see Figure 3.7)



Kartennummer:
Umsatz: 0,00
Vorname:
Nachname:
Bonuspunkte:
Stand vom: 04.01.2017
Bonanzahl: 18

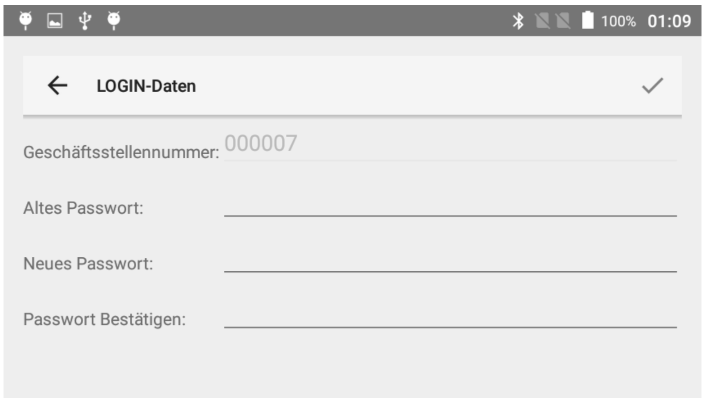
Figure 3.7.: Link to Login Data activity

For every activity that is created in this application, it has to be defined in the manifest file. The following shows that the login data activity is being defined in the *AndroidManifest.xml*.

```
<activity
  android:name=".LoginDataActivity"
  android:label="@string/title_activity_login_data"
  android:parentActivityName=".MainActivity" >
  <meta-data
    android:name="android.support.PARENT_ACTIVITY"
    android:value="com.haw_hamburg.scanner.MainActivity" />
</activity>
```

Layout

Figure 7 demonstrates the layout for login data. The XML codes below describes the properties of app bar and the rest of the UI for login data. The full content for this layout file can be seen at *activity_login_data.xml*.



Geschäftsstellennummer: 000007

Altes Passwort:

Neues Passwort:

Passwort Bestätigen:

Figure 3.8.: Login Data activity

First section of the code defines the layout of app bar. It has the exact same properties as the app bar in sale mode (see section 3.1.2). The only difference is the ID name. The rest of the layout in login data are enclosed in relative layout that enables the child views to position themselves relative to each other. The width and height of relative layout are set as match parent and wrap content respectively.

```
RelativeLayout<
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
```

The elements in relative layout consist of TextViews for the wording and EditTexts for text entry. Each of the elements is specified with an ID for reference. The width and height for the TextViews are all set as wrap content. On the other hand, height of all the EditTexts is set as wrap content while the width is set as match parent in order to provide enough space for register number and passwords. TextView register number is the reference for all the EditTexts to locate their positions in relative layout, whereas for the rest of the TextViews (except for TextView register number), they align themselves according to their corresponding EditText. For the input field of register number, it displays the hint text ”Nicht Gesetzt” to indicate that register number has not set up. This register number input field is also not editable. The modification of this input field can only be done by technician at Settings (see section 3.2.6).

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Geschäftsstellen-Nummer:"
  android:id="@+id/RegisterNrLabel"
  android:layout_marginTop="20dp"/>

<EditText
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/RegisterNr"
  android:layout_toRightOf="@id/RegisterNrLabel"
  android:layout_alignBottom="@id/RegisterNrLabel"
  android:layout_marginBottom=
  "@dimen/settings_dialog_baseline_margin_inverted"
  android:hint="@string/not_set"
  android:editable="false"
  android:focusable="false">
</EditText>
```

Besides that, there is a tick at the top right corner of the login data layout, which acts as save function. The tick is declared at *menu_login_data.xml*. The menu resource file for the tick is shown below.

```
<item android:id="@+id/action_save"
  android:icon="@drawable/ic_done_black_48dp"
  android:title="@string/action_save"
app:showAsAction="ifRoom"/>
```

Implementation

The implementation code for this activity can be viewed at *LoginDataActivity.java*.

For every activity in the application that uses an app bar, this activity has to extend *AppCompatActivity*. After the login data activity has successfully initiated, the *onCreate()* method opens the interface of the activity. Here the *onCreate()* method has to override the method of its superclass, so that the compiler is able to generate an error message if there is any problems during the debug process.

```
public class LoginDataActivity extends AppCompatActivity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_login_data);
```

The app bar is initialized and given a title as ”LOGIN-Daten”. The method *setDisplayHomeAsUpEnabled()* is used to create a back icon at the left hand side of the app bar in order to allow user returns to sale mode.

```
Toolbar toolbar = (Toolbar)findViewById(R.id.login_data_toolbar);
setSupportActionBar(toolbar);
ActionBar actionBar = getSupportActionBar();
   if (actionBar != null) {
      actionBar.setTitle(R.string.title_activity_login_data);
      actionBar.setDisplayHomeAsUpEnabled(true);
   }
}
```

The following shows how this activity is being carried out. Despite that the register number has not set up, user’s password can be created by using master password as old password and then key in the desired password at new password input field (for security reason, new password has to be entered twice).

Otherwise, the register number is displayed on the screen automatically, if the register number has been set up by technician. In order to change the password, user is required to key in their old password and new password in the corresponding input field and then clicks on the tick at the top right corner to save the data. It will be verified to see if all the text fields are filled in and make sure that all the input are entered correctly. If there is no error detected, a toast message will display on the screen showing that the password has successfully changed and the text field will return to empty string.

```
if (oldPasswordLength > 0) {
   if (oldPassword.equals(Utility.MASTERPWD)
   || oldPassword.equals(Utility.retrievePassword(this))) {
      String newPassword = txtNewPassword.getText().toString();
```

```
    int newPasswordLength = newPassword.length();
    if (newPasswordLength > 0) {
      if (!newPassword.equals(Utility.MASTERPWD)) {
        String confirmPassword =
           txtConfirmPassword.getText().toString();
        if (newPassword.equals(confirmPassword)) {
          Utility.storePassword(this, newPassword);
          Toast.makeText(this,
          "Das Password ist erfolgreich
             ge`color{mauve}\"a`ndert.",
             Toast.LENGTH_SHORT).show();
          txtOldPassword.setText("");
          txtNewPassword.setText("");
          txtConfirmPassword.setText("");
          }
        }
      }
  }
```

However, there are a few conditions that will cause the process of changing password failed:

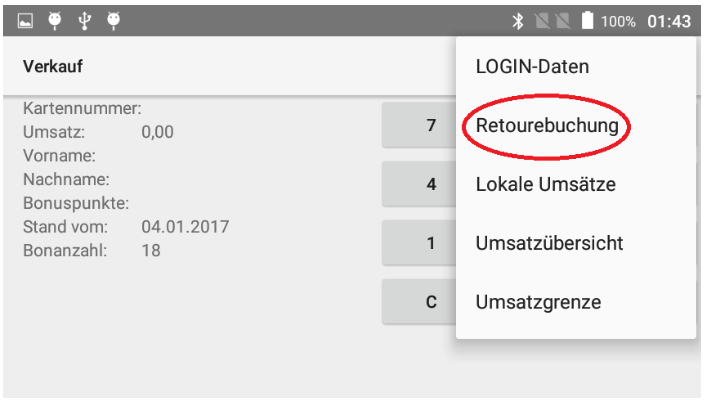
1. The entered new password does not match with the confirm password or vice versa, or one of the text fields is empty
2. Master password is entered as the new password
3. New password input field is empty
4. The entered old password is not the same as the user's old password
5. Old password input field is empty

Error that user made is displayed by a toast message that pops up after validation. User has to re-enter the correct information in order to change the password.

```
else {
    Toast.makeText(this,
      "Die neue Passwörter stimmen miteinander nicht überein.",
         Toast.LENGTH_SHORT).show();
}

else {
    Toast.makeText(this,
       "Dieses neue Passwort ist nicht zulässig. Bitte versuchen
          Sie eine andere Kombination.",
          Toast.LENGTH_SHORT).show();
}

else {
    Toast.makeText(this,
       "Bitte geben Sie ein neues Passwort ein.",
          Toast.LENGTH_SHORT).show();
}

else {
    Toast.makeText(this,
       "Das eingegebene Passwort stimmt mit dem aktuellen
          Password nicht überein.", Toast.LENGTH_SHORT).show();
}

else {
    Toast.makeText(this,
       "Bitte geben Sie das aktuelle Passwort ein.",
          Toast.LENGTH_SHORT).show();
}
```

3.2.2. Return mode

The function of return mode is to allow user to input new return transaction details. Choosing the option "Retourebuchung" (see Figure 3.9) at overflow menu directs user to the return mode.



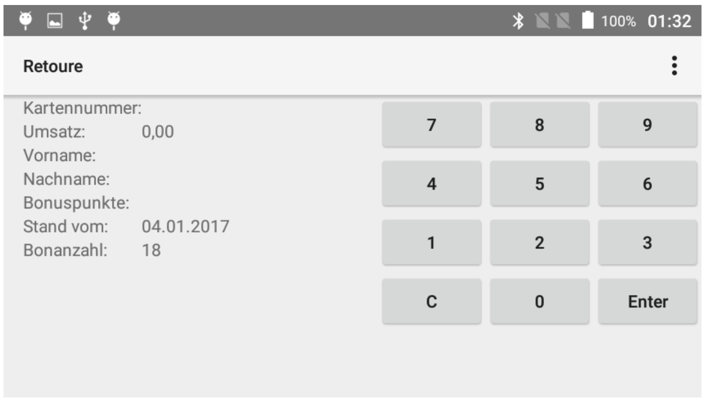
Verkauf

Kartennummer:
Umsatz: 0,00
Vorname:
Nachname:
Bonuspunkte:
Stand vom: 04.01.2017
Bonanzahl: 18

Figure 3.9.: Link to Return Mode

Layout

This activity shares the exact same layout as sale mode except for the title name to indicate that it is a return mode. Figure 3.10 shows the layout of return mode.



Kartennummer:
Umsatz: 0,00
Vorname:
Nachname:
Bonuspunkte:
Stand vom: 04.01.2017
Bonanzahl: 18

Figure 3.10.: Return Mode activity

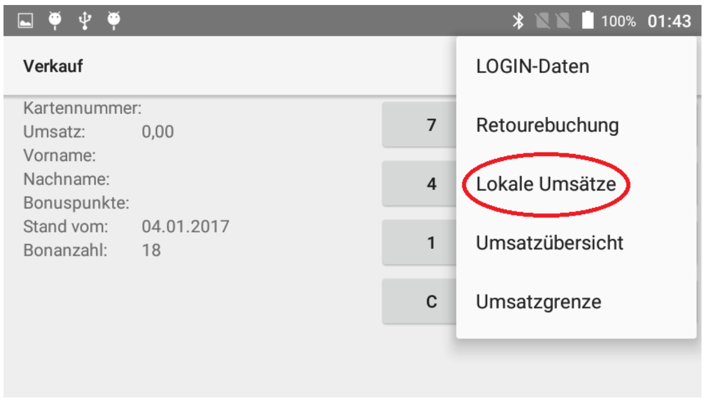
Implementation

In addition, return mode also has the same process flow as sale mode (see flow diagram at Figure 3.4). The application will switch back to sale mode if the return transaction is processed successfully.

```
if (mode == EntryMode.RETURN) {
  mode = EntryMode.SALE;
  entryModeMenuItem.setTitle(R.string.action_return_mode);
  setTitle(R.string.sale_mode);
}
    Toast.makeText(MainActivity.this,
      "Der Bon ist erfolgreich gespeichert.",
         Toast.LENGTH_LONG).show();
}
```

3.2.3. Edit (yet to be synchronized) turnover

By choosing the option ”Lokale Umsätze” at the overflow menu (see Figure 3.11) , user comes to the page where the turnover value can be edited. The turnover value can be modified using keypad. The rest of the information (card number and information of customer) are not editable. Also, a receipt cannot be deleted through this page. Nonetheless, this is only available for the sales data that are still stored locally and yet to be transfered to the server. The number of receipts remains the same at the end of the process.

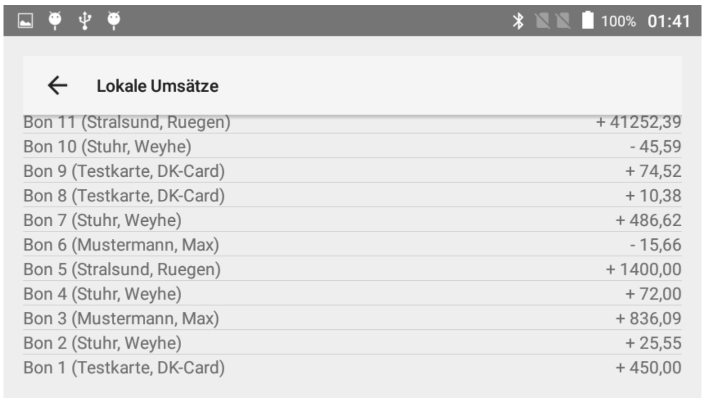


Kartennummer:	
Umsatz:	0,00
Vorname:	
Nachname:	
Bonuspunkte:	
Stand vom:	04.01.2017
Bonanzahl:	18

Figure 3.11.: Link to Edit (yet to be synchronized) Turnover activity

Layout

Figure 3.12 shows the layout for activity edit turnover.



Bon 11 (Stralsund, Ruegen)	+ 41252,39
Bon 10 (Stuhr, Weyhe)	- 45,59
Bon 9 (Testkarte, DK-Card)	+ 74,52
Bon 8 (Testkarte, DK-Card)	+ 10,38
Bon 7 (Stuhr, Weyhe)	+ 486,62
Bon 6 (Mustermann, Max)	- 15,66
Bon 5 (Stralsund, Ruegen)	+ 1400,00
Bon 4 (Stuhr, Weyhe)	+ 72,00
Bon 3 (Mustermann, Max)	+ 836,09
Bon 2 (Stuhr, Weyhe)	+ 25,55
Bon 1 (Testkarte, DK-Card)	+ 450,00

Figure 3.12.: Edit (yet to be synchronized) Turnover activity

The layout in Figure 3.12 contains a back icon, a Toolbar and a ListView. ListView is used to arrange similarly structured items and present them in a vertical scrollable list. The properties of ListView is shown below. Full codes that define the properties of views in this activity can be seen at *activity_local_values.xml*.

```
<ListView
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/local_values_listview"
  android:layout_below="@id/local_values_toolbar">
</ListView>
```

Two TextView objects are used to create each row in the list. One is for the label of receipt, which includes customer's name, and another one is to show the turnover values. TextView for turnover value is aligned to the right. The following codes that describe the attributes of the two TextViews is under *row_local_values.xml*.

```
<TextView
  android:id="@+id/bonLabel"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content" >
</TextView>

<TextView
  android:id="@+id/umsatzLabel"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:gravity="right" >
</TextView>
```

Implementation

The complete codes that define a ListAdapter can be viewed at *LocalValueListAdapter.java*. A ListAdapter is used for creating the list of data as seen in Figure 3.12. This adapter acts as a link between data source and AdapterView to retrieve data from database and convert the array list of items into views to display on the list. When the method *getView()* is called, *convertView* is null when this page starts up for the very first time because no view is created before this. The items from *row_local_values.xml* is inflated and will fill up the page. *ViewHolder* is applied here to get the views of the row. The row contains of the ID of receipt, name of customer and the turnover. The latest receipt is displayed at the top while the oldest receipt is at the bottom of the list. As the list is scrolled up, the view that is offscreen, is not deleted, instead, convertView recycles the view. The method *getTag()* is called to get instance of *ViewHolder* and assigned new data to the view that was previously cached. This view replaces the old view with new data and will be pushed up from the bottom of the list.

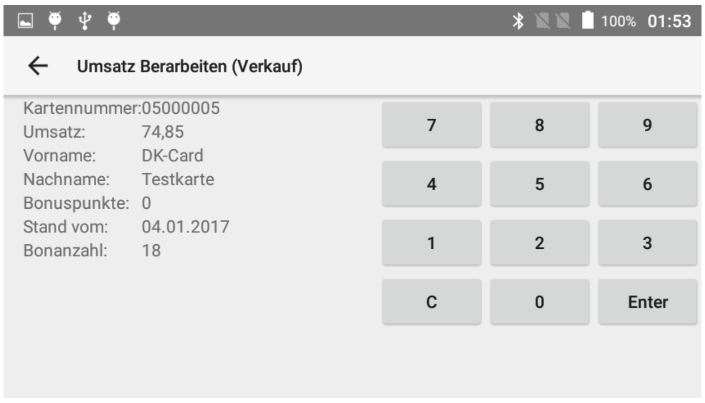
```
public View getView(int position, View convertView, ViewGroup
   parent) {
  if (convertView == null) {
        LayoutInflater inflater = context.getLayoutInflater();
        convertView = inflater.inflate(R.layout.row_local_values,
           null);

        ViewHolder holder = new ViewHolder();
        holder.bonLabel = (TextView)
           convertView.findViewById(R.id.bonLabel);
        holder.umsatzLabel = (TextView)
           convertView.findViewById(R.id.umsatzLabel);
        convertView.setTag(holder);
    }

    ViewHolder holder = (ViewHolder) convertView.getTag();
    DatabaseHelper.Bon bon = bons.get(position);
    holder.bonLabel.setText("Bon " + bon._id + " (" +
       bon.kunde.nachname + ", " + bon.kunde.vorname + ")");
    holder.umsatzLabel.setText((bon.retoure ? "- " : "+ ") +
       Utility.formatUmsatzString(bon.umsatz));
    return convertView;
}
```

The java code *LocalValuesActivity* describes the execution of Local Value activity. Each of the row is clickable. User can click on the receipt that its turnover value has to be edited. Once user has clicked on the receipt, it will trigger the *OnItemClickListener* of *ListView* and the method *OnItemClick()* is called. Intent is implemented to move user from this activity to Local Value Editor activty (see Figure 3.13) and the required data will be pulled from the database. Method *putExtra()* is used to send the data to Local Value Editor activty.

```
listView.setOnItemClickListener(new
    AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int
       position, long id) {
        Intent intent = new Intent(LocalValuesActivity.this,
           LocalValueEditorActivity.class);
        Bundle b = new Bundle();
        DatabaseHelper.Bon bon = (DatabaseHelper.Bon)
           adapter.getItem(position);
        b.putInt(ScannerContract.Bon._ID, bon._id);
        b.putLong(ScannerContract.Bon.COLUMN_NAME_ZEIT, bon.zeit);
        b.putLong(ScannerContract.Bon.COLUMN_NAME_UMSATZ, bon.umsatz);
        b.putBoolean(ScannerContract.Bon.COLUMN_NAME_RETOURE,
           bon.retoure);
```

```
      b.putString(ScannerContract.Kunde.COLUMN_NAME_KARTENNUMMER,
         bon.kunde.kartennummer);
      b.putString(ScannerContract.Kunde.COLUMN_NAME_VORNAME,
         bon.kunde.vorname);
      b.putString(ScannerContract.Kunde.COLUMN_NAME_NACHNAME,
         bon.kunde.nachname);
      b.putString(ScannerContract.Kunde.COLUMN_NAME_BONUSPUNKTE,
         bon.kunde.bonuspunkte);
      b.putInt(LOCAL_VALUE_POSITION, position);
      intent.putExtras(b);
      startActivityForResult(intent, REQUEST_CODE);
   }
});
```

Subsequently, user is directed to Local Value Editor activity (*LocalValueEditorActivity.java*) as seen in Figure 3.13 so that turnover value can be edited. The page is filled with the data pulled from database. Before entering new turnover value, old turnover value has to be first deleted by using Cancel button. After finish editing the turnover value, user can click on Enter button to save it. Turnover limit is checked before updating the new data in the database.



Kartennummer:05000005
Umsatz: 74,85
Vorname: DK-Card
Nachname: Testkarte
Bonuspunkte: 0
Stand vom: 04.01.2017
Bonanzahl: 18

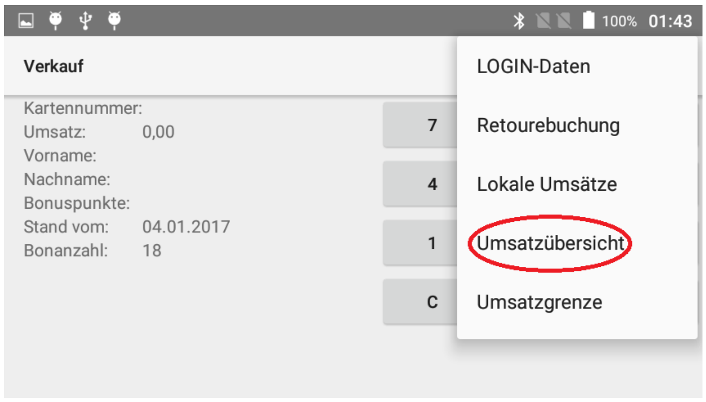
Figure 3.13.: Local Value Editor activity

After user has edited the turnover value, the method *onActivtyResult()* in *LocalValuesActivity.java* is called to receive the data and update the edited turnover value in the database. Before updating the new data, it will be checked if the received request code is the same as the sent request code, and whether the result code is matched, as well as making sure that it is not an empty intent and to check if *LOCAL_VALUE_POSITION* was passed in the intent.

```
protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {
  super.onActivityResult(requestCode, resultCode, data);
  if (requestCode == REQUEST_CODE && resultCode ==
     Activity.RESULT_OK
  && data != null && data.hasExtra(LOCAL_VALUE_POSITION)) {
    int position = data.getIntExtra(LOCAL_VALUE_POSITION, -1);
    if (position == -1) {
      return;
    }
    adapter.updateUmsatz(position,
       data.getLongExtra(ScannerContract.Bon.COLUMN_NAME_UMSATZ,
       0));
  }
}
```

3.2.4. Monthly Turnovers and Returns

This function presents an overview page for the monthly turnovers and returns. It can be viewed and deleted. The page will not be deleted automatically when the data are transfered to server, instead, it can only be deleted when user does it explicitly. The monthly amount of turnovers and returns as well as the total amount are shown. In addition, the last reset date and current date are also displayed on the page. User can reach this activity by clicking on the option ”Umsatzübersicht” at overflow menu (see Figure 3.14).


100% 01:43

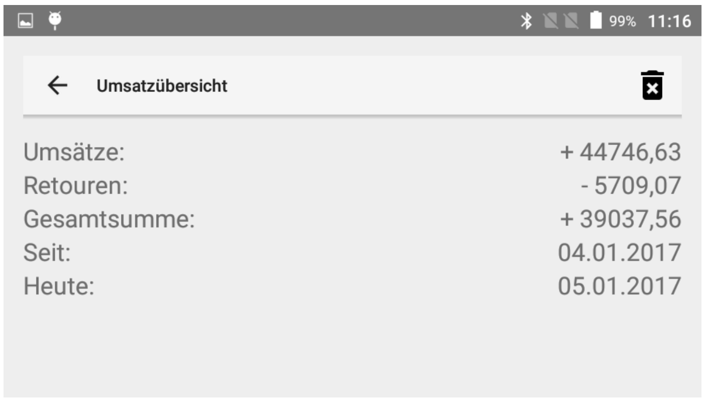
Verkauf

Kartennummer:
Umsatz: 0,00
Vorname:
Nachname:
Bonuspunkte:
Stand vom: 04.01.2017
Bonanzahl: 18

Figure 3.14.: Link to Monthly Turnover and Returns activity

Layout

Figure 3.15 shows the UI for this activty.



Umsätze: + 44746,63
Retouren: - 5709,07
Gesamtsumme: + 39037,56
Seit: 04.01.2017
Heute: 05.01.2017

Figure 3.15.: Monthly Turnover and Returns activity

The full content of this layout file can be seen at *activity_overview.xml*. The following codes describe the attributes of the layout in this activity. All the TextViews are enclosed in the relative layout. The width of this relative layout is set to be as big as the parent while the height is wrap content to just enough to include the content.

```
<RelativeLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
```

TextViews for label are located at the left of the screen and TextViews for input field is at the right. All of the TextViews are given an ID. TextViews for label are both wrap content for width and layout. As for input field, the width of the TextViews is set as match parent and the height as wrap content. Both of them have a text size of 20sp. Sp, which stands for Scale-independent Pixels is an unit for font size scaling. This statement *android:gravity="right"* ensures that the displayed text are aligned to the right.

TextView for label

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Umsätze:"
  android:id="@+id/UmsatzLabel"
  android:textSize="20sp"
  android:layout_marginTop="20dp"/>
```

TextView for input field

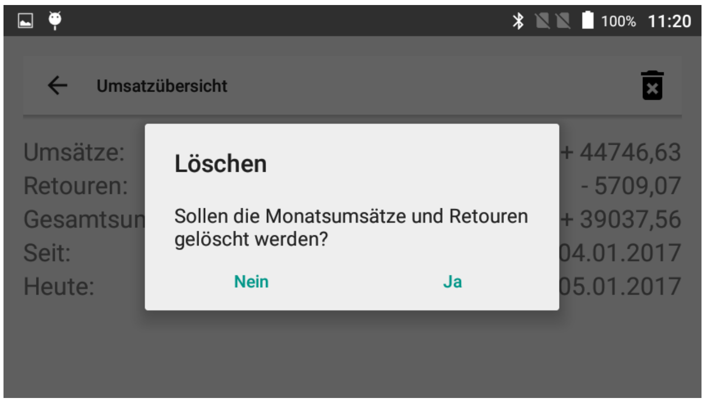
```
<TextView
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/Umsatz"
  android:textSize="@dimen/overview_text_size"
  android:gravity="right"
  android:layout_toRightOf="@id/RetoureLabel"
  android:layout_alignBottom="@id/UmsatzLabel"/>
```

Implementation

The codes to implement this activity can be seen at *OverviewActivity.java*. All the data will be displayed on the page once activity is created. The following statements indicate how the data are being retrieved and shown on the screen. Turnover and return values as well as the dates are retrieved from shared preferences (see the codes at *Utility.java*). *OVERVIEW_TURNOVER*, *OVERVIEW_RETURN* and *OVERVIEW_TIMESTAMP* are used as the key to store and retrieve their respective value in preference. In addition, The total value is calculated by subtracting return value from turnover value. There will be a plus or minus sign in front the total value to specify whether it is a positive amount or negative amount.

```
  long turnoverValue = Utility.retrieveOverviewTurnover(this);
  long returnValue = Utility.retrieveOverviewReturn(this);
  long totalValue = turnoverValue - returnValue;
  umsatz.setText("+ " +
     Utility.formatUmsatzString(turnoverValue));
  retoure.setText("- " + Utility.formatUmsatzString(returnValue));
  summe.setText((totalValue < 0 ? "- " : "+ ") +
     Utility.formatUmsatzString(Math.abs(totalValue)));
  seit.setText(Utility.formatTimestamp
  (Utility.retrieveOverviewTimestamp(this)));
  heute.setText(Utility.currentDate());
```

This overview page can be reset by clicking on the delete icon at the top right corner. An alert dialog as seen in Figure 3.16 will pop out before deleting the page to confirm the deletion.



Sollen die Monatsumsätze und Retouren gelöscht werden?

Figure 3.16.: Alert Dialog before reset

All the data will be reset after the page has successfully deleted. Shared preferences is used to remove the data in the overview page. Method *resetOverview()* is called to execute this action.

```
public static boolean resetOverview(Context context) {
  SharedPreferences preferences =
     PreferenceManager.getDefaultSharedPreferences(context);
  return preferences.edit().remove(OVERVIEW_RETURN)
  .remove(OVERVIEW_TURNOVER)
  .remove(OVERVIEW_TIMESTAMP).commit();
}
```

3.2.5. Turnover limit

In this activity, a preset turnover limit can be set, is displayed or can be changed by user. In order to be able to carry out this function, user has to choose the option ”Umsatzgrenze” (see Figure 3.17) at the overflow menu in sale mode activity. After that, user is linked to the turnover limit activity as shown in Figure 3.18.

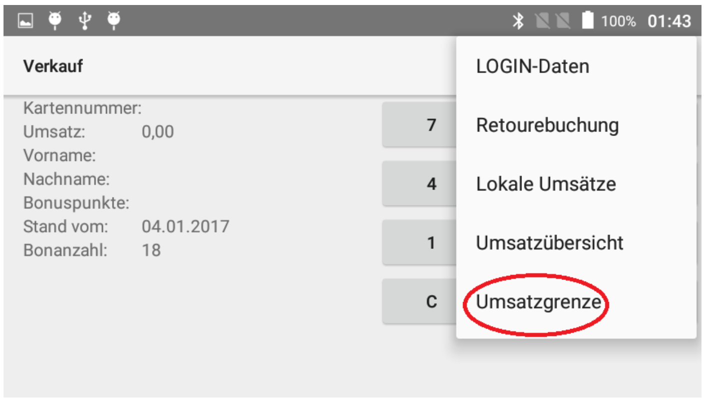


Figure 3.17.: Link to Turnover Limit activtity

Layout

Full code for the turnover limit activity layout file can be viewed at *activity_turnover_limit.xml*. Figure 3.18 shows the layout for Turnover Limit activity.

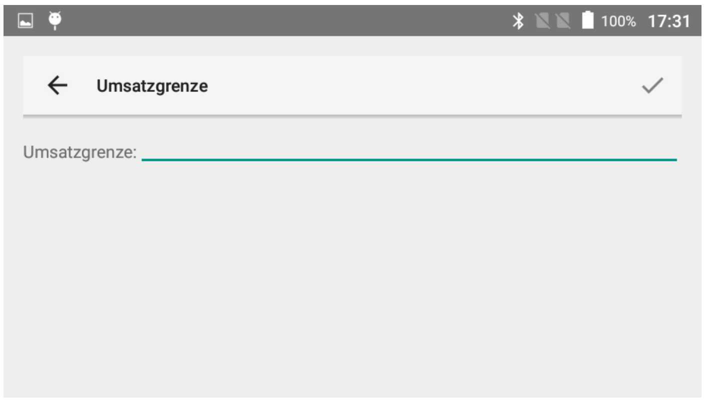


Figure 3.18.: Turnover Limit activity

The following codes describe the attributes of the app bar which located at the top of the layout. It shares the same properties as the app bar in the login data activity (see Section 3.2.1) except for the given ID.

```
<android.support.v7.widget.Toolbar
   android:id="@+id/turnover_limit_toolbar"
   android:layout_width="match_parent"
   android:layout_height="?attr/actionBarSize"
   android:background="?attr/colorPrimary"
   android:elevation="4dp"
   android:theme="@style/ThemeOverlay.AppCompat.ActionBar"
```

```
  app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>
```

The rest of the layout in turnover limit activity is very simple. It is made up of a TextView and a EditText that are wrapped in a relative layout. The width of relative layout is set to match parent and the height is set to wrap content.

```
<RelativeLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
```

The width and the height of TextView are both set to wrap content and also given a unique ID ”TurnoverLimitLabel”. The ”Umsatzgrenze:” text is displayed on the screen.

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Umsatzgrenze:"
  android:id="@+id/TurnoverLimitLabel"
  android:layout_marginTop="@dimen/settings_dialog_top_margin"/>
```

For the EditText, its width and height is match parent and wrap content respectively. ”TurnoverLimit” is labeled as the ID. The position of EditText is placed next to the TextView. Furthermore, only number with no more than 6 digits are allowed for the input.

```
<EditText
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/TurnoverLimit"
  android:layout_toRightOf="@id/TurnoverLimitLabel"
  android:layout_alignBottom="@id/TurnoverLimitLabel"
  android:layout_marginBottom=
  "@dimen/settings_dialog_baseline_margin_inverted"
  android:inputType="number"
  android:singleLine="true"
  android:maxLength="6">
</EditText>
```

Under the XML file *menu_turnover_limit.xml*, the properties of tick which is situated at the top right corner of the turnover limit layout is defined. This tick represents the function save.

Implementation

The following codes are under the java file *TurnoverLimitActivity.java*. Firstly, the application will check whether there is any text in the text field. When the application finds out that there is no text being entered, a toast message will display and request the user to define a new limit again. On the other hand, if the application has detected that the text field has input, it will attempt to convert the input string to numeric type long. At the same time, the application will try to catch the Number Format Exception. Number Format Exception is an exception that will happen when the application tries to convert a string to a numeric value, for example an integer, float or long, but the string does not have a proper format. Hence, Number Format Exception will be thrown, which means that the entered text is not a number or cannot be parsed. However, if the string managed to convert to numeric type long, the turnover limit will be stored and it means that the process of setting the turnover limit has succeeded. A toast message will then pop out to show that the turnover limit has successfully modified. Nonetheless, if the key in number is not able to be stored, a toast message is shown to indicate that this process has failed and user has to retry again.

```
private void save() {
  Editable text = txtTurnoverLimit.getText();
  if (text.length() > 0) {
    try {
      long limit = Long.parseLong(text.toString());
      if (!Utility.storeTurnoverLimit(this, limit)) {
        Toast.makeText(this,
        "Aktualisierung unerfolgreich. Bitte versuchen Sie
           erneut.", Toast.LENGTH_SHORT).show();
        return;
      }
      txtTurnoverLimit.setText(String.valueOf(limit));
      Toast.makeText(this,
      "Die Umsatzgrenze ist erfolgreich ge"andert.",
         Toast.LENGTH_SHORT).show();
      return;
    } catch (NumberFormatException e) {
      Log.d("Turnover Limit Activity", "Failed to parse: " +
         e.getMessage());
    }
  }
  Toast.makeText(this,
  "Bitte definieren Sie eine neue Umsatzgrenze.",
     Toast.LENGTH_SHORT).show();
}
```

The following shows the utility activity java code *Utility.java* for storing and retrieving of the turnover limit. Shared preferences is used to store and retrieve the defined turnover limit with key-values pair. In this case, *PREFERENCE_TURNOVER_LIMIT* is the key to save the value limit.

```
public static boolean storePreference(Context context, String
    key, long value) {
  SharedPreferences preferences =
     PreferenceManager.getDefaultSharedPreferences(context);
  SharedPreferences.Editor editor = preferences.edit();
  return editor.putLong(key, value).commit();
}

public static boolean storeTurnoverLimit(Context context, long
    limit) {
  return storePreference(context, PREFERENCE_TURNOVER_LIMIT,
     limit);
}
}
```

The following method is used to fetch the saved value. By providing the key that used to store the value, *getLong()* is called to get the saved value. If the preference file does not exist, it returns to default value, which is in this case zero. Here, it means that if the application is used for the first time, no limit is saved, so the turnover limit appears to be zero. In addition, if the Class Cast Exception (inappropriate of changing a class from one type to another) happens, it will also return the value to default value. For example, the demanded numeric type is long, however, the saved value is numeric type integer or float.

```
public static long retrievePreference(Context context, String
    key, long defValue) {
SharedPreferences preferences =
    PreferenceManager.getDefaultSharedPreferences(context);
try {
  return preferences.getLong(key, defValue);
} catch (ClassCastException e) {
  Log.d("[Utility]", "Retrieving Preference long '" + key + "': "
     + e.getMessage());
  return defValue;
}
}

public static long retrieveTurnoverLimit(Context context) {
return retrievePreference(context, PREFERENCE_TURNOVER_LIMIT, 0);
}
```

3.2.6. Settings

When the application is being used for the first time, this function allows technician to configure the general settings for the application. It will only appear on the overflow menu if the application is logged in using the master password, because user is not allowed to do any adjustments in settings. Other than setting up device number and register number, technician also can enter details for FTP server as well as the FTP user and password. Furthermore, technician can choose the types of data transmission and set up an alarm to transmit data. By choosing the option 'Einstellungen' (see Figure 3.19), it directs technician to Settings activity.

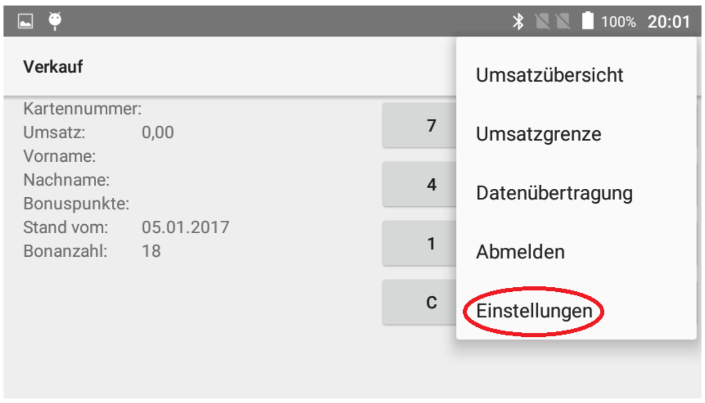
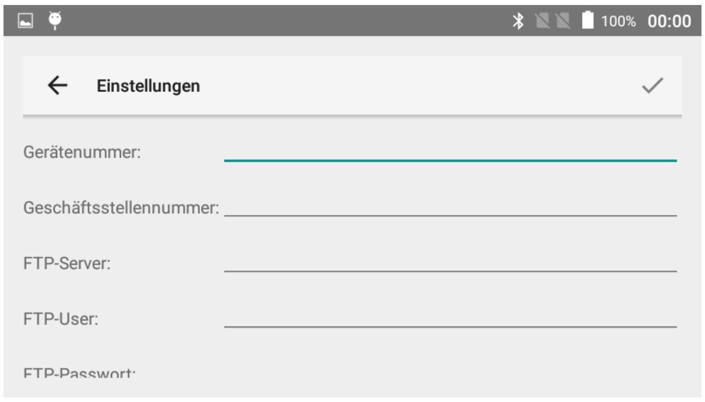

Verkauf
Kartennummer:
Umsatz: 0,00
Vorname:
Nachname:
Bonuspunkte:
Stand vom: 05.01.2017
Bonanzahl: 18

Figure 3.19.: Link to Settings activity

Layout

Figure 3.20 and 3.21 illustrate the layout for Settings. Complete layout file for this activity can be found at *activity_settings.xml*. As all the views in this activity could not fit into a page, ScrollView is used to ensure that all the contents enclosed in it are scrollable. The width and height of ScrollView are set to be as big as the parent.

```
<ScrollView
android:layout_width="match_parent"
android:layout_height="match_parent">
</ScrollView>
```


100% 00:00

Einstellungen

Gerätenummer:

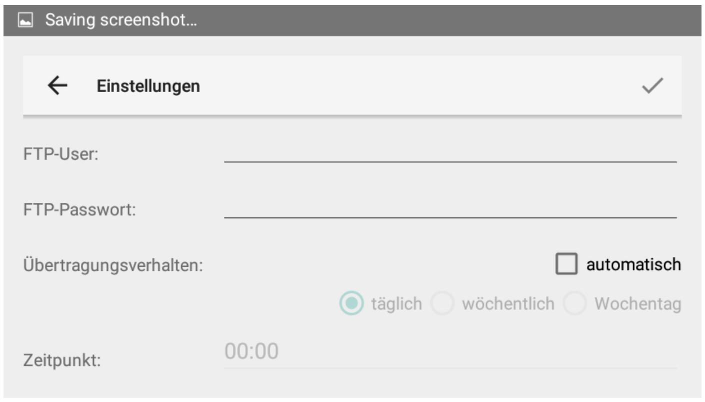
Geschäftsstellennummer:

FTP-Server:

FTP-User:

FTP-Passwort:

Figure 3.20.: Settings activity (a)


Saving screenshot...

Einstellungen

FTP-User:

FTP-Passwort:

Übertragungsverhalten: ☐ automatisch

[✓ täglich] wöchentlich Wochentag

Zeitpunkt: 00:00

Figure 3.21.: Settings activity (b)

TextViews are used for the all the labels to display their text. The width and height are wrap content for TextView to just enough to wrap its content.

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Gerätenummer:"
  android:id="@+id/DeviceNrLabel"
  android:layout_alignBottom="@+id/DeviceNr"
  android:layout_marginBottom="6dp"/>
```

For the input fields, EditText is applied to allow technician to key in the input, except for the type of data transmission. Besides that, only numbers are allowed in the device number and register number input field. Other than that, only 8 digits can be entered for device number. Whereas for register number, it can only accept maximum 6 digits.

```
<EditText
```

```
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@id/DeviceNr"
  android:layout_toRightOf="@id/RegisterNrLabel"
  android:inputType="number"
  android:singleLine="true"
  android:maxLength="8"/>
```

CheckBox is used to choose the type of data transmission. By selecting the CheckBox, it means that the data will be transfered to server automatically in certain period of time. On a contrary, unchecking the CheckBox, transfering the data has to be done by the user manually.

```
<CheckBox
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="automatisch"
  android:id="@+id/TransmissionBehaviorAuto"
  android:layout_alignParentRight="true"
  android:layout_alignBottom="@id/TransmissionBehaviorHeight"/>
```

RadioGroup is used to build the radio buttons. The radio buttons can only be selected when the above Checkbox is ticked. Also, only one radio button could be checked for each time. The orientation of the three radio buttons are displayed in a horizontal order.

```
<RadioGroup
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/TransmissionBehaviorFreq"
  android:orientation="horizontal"
  android:gravity="right"
  android:layout_below="@id/TransmissionBehaviorHeight">
  <RadioButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="täglich"
    android:id="@+id/radioDaily"/>
```

Implementation

The implementation codes for this activity can be seen at *SettingsActivity.java*. The data in Settings are all being stored and can be retrieved using shared preference in *Utility.java*.

There are two options to transmit the collected data to the server, manually or automatically at a certain time. By checking the CheckBox, the data will be sent to the server automatically either daily, weekly or weekday according to the alarm that has been set by the technician. Technician can set the time to transmit the data at ”Zeitpunkt”. However, without checking the CheckBox, it means that the data will be transfered manually and therefore, no adjustments can be made at radio buttons as well as input for time.

A dialog as shown in Figure 3.22 will pop out to allow the technician to set the time for the alarm. This dialog is a fragment that belongs to part of the Settings activity's UI. Method *onCreateDialog()* is called to draw the layout of this fragment in the activity. The time can be retrieved via the method *getarguments()* if it has been set before.

```
public static class TimePickerFragment extends DialogFragment
implements TimePickerDialog.OnTimeSetListener {

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    Bundle arguments = getArguments();
    int hourOfDay = arguments.getInt(HOUR_OF_DAY);
    int minute = arguments.getInt(MINUTE);

    return new TimePickerDialog(getActivity(), 3, this,
       hourOfDay, minute, true);
  }
```

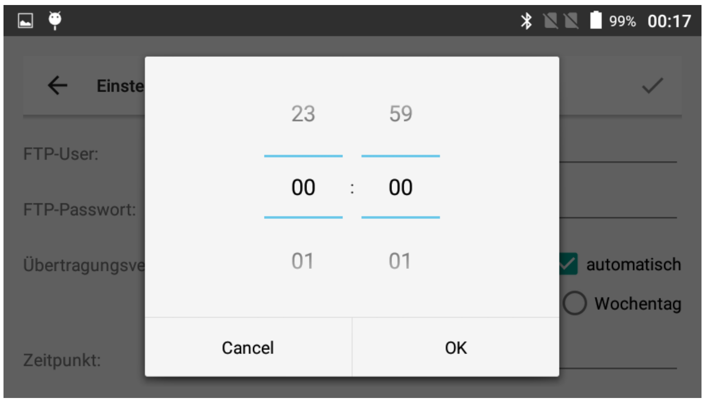


Figure 3.22.: Time Picker Dialog

As technician has completed the set up and clicks on the tick at the top right corner, method *save()* is called to store the entered data. Before saving, each of the input field is checked to make sure that there is an input. Otherwise, the saving process will fail and a toast message will appear to remind technician to fill in the empty input field.

```
private void save() {
  Editable deviceNrText = deviceNr.getText();
  int deviceNrLength = deviceNrText.length();
  if (deviceNrLength == 0) {
    Toast.makeText(this,
    "Bitte geben Sie die Gerätenummer ein.",
       Toast.LENGTH_LONG).show();
    return;
  }
```

Once the data has successfully saved, a toast message is shown to indicate that Settings has been updated.

```
  Toast.makeText(this,
  "Die Einstellungen sind erfolgreich aktualisiert.",
     Toast.LENGTH_LONG).show();
```

If the data is set to transfer automatically at a scheduled time, an alarm will be set up to execute the action. An alarm is scheduled to start from the next day and will repeat daily if the transmission frequency is set at daily. As for the weekly data transmission, the alarm will be starting from next week after the settings is updated and it repeats every week. On the other hand, the alarm will start from the next 5 week days and will repeat weekly for the weekday data transmission.

```
if (transmissionAuto) {
  switch (transmissionFreq) {
    case Utility.TRANSMISSION_FREQ_DAILY: {
      Calendar calendar = Calendar.getInstance();
      calendar.setTimeInMillis(System.currentTimeMillis());
      calendar.set(Calendar.MILLISECOND, 0);
      calendar.set(Calendar.SECOND, 0);
      calendar.set(Calendar.MINUTE, minute);
      calendar.set(Calendar.HOUR_OF_DAY, hour);
      calendar.add(Calendar.DATE, 1);
      manager.setRepeating(AlarmManager.RTC_WAKEUP,
         calendar.getTimeInMillis(), DAY_IN_MILLISECONDS,
         pendingIntent);
    }
    break;
```

In order to transfer the data automatically, a Broadcast Receiver is needed. Broadcast Receiver receives broadcast message or intent and will execute the appropriate action. The implementing class Transmission Cue Receiver extends the BroadcastReceiver class. To upload the collected data from the database to server, the method *onReceive()* is called when the Broadcast Receiver receives an intent from the alarm. The following code can be seen at *TransmissionCueReceiver.java*.

```
public class TransmissionCueReceiver extends BroadcastReceiver {

  @Override
  public void onReceive(final Context context, Intent intent) {
    new AsyncTask<Void, Void, Void>() {

      @Override
      protected Void doInBackground(Void... params) {
        Utility.synchronize(context,
           DatabaseHelper.getInstance(context));
        return null;
      }
    }.execute();
  }
}
```

4. Database

This chapter focuses on creation and implementation of the application's database by using the Android built in SQLite database.

4.1. Data Model

To organise the data of this application in an efficient way, entity-relationship diagram is drawn to conceptualize the data into diagrams. The entity relationship model in Figure 4.1 shows that entity customer (Kunde) and entity receipt (Bon) are connected by a *"hat"* relationship. It also depicts one to many relationship. In the case of this application, it means that a customer could have more than one receipt stored in the database.

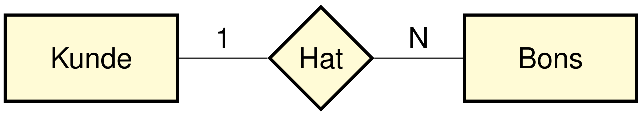


Figure 4.1.: Customer (Kunde) and Receipts (Bons) as Separate Entities (Entity-Relationship Diagram)

A table is created for each entity with the row represents the records and column represents attributes of the entity. The diagrams in Figure 4.2 and 4.3 shows the entities customer and receipt with their respective attributes. In addition, both the entities have a primary key *_id* as identifier for the table records. This identifier is an integer, which will increase as the records increases. Furthermore, *kunde_id* is defined as foreign key in entity Receipt.

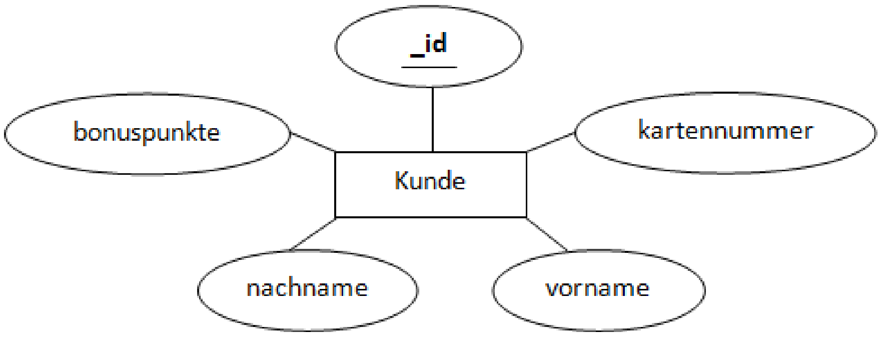


Figure 4.2.: Entity Customer (Kunde) with attributes

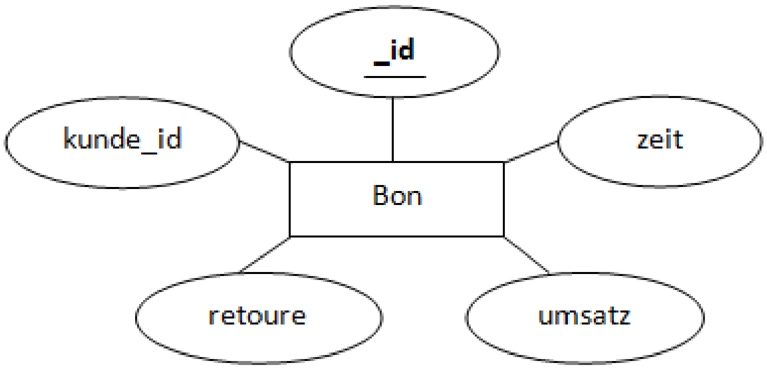


Figure 4.3.: Entity Receipt (Bon) with attributes

Contract class is used to build a database in Android Studio, as it consists of constants that define name for tables and columns. The following codes from *ScannerContract.java* shows how the database for this application is designed.

Firstly, to create table for customer, *BaseColumn* interface is implemented to define the name of the ID column. Table name and column name are defined in the following statement.

```
public static class Kunde implements BaseColumns {
  public static final String TABLE_NAME = "kunde";
  public static final String COLUMN_NAME_KARTENNUMMER =
     "kartennummer";
  public static final String COLUMN_NAME_VORNAME = "vorname";
  public static final String COLUMN_NAME_NACHNAME = "nachname";
  public static final String COLUMN_NAME_BONUSPUNKTE =
      "bonuspunkte";
  public static final String[] ALL_COLUMNS = { _ID,
      COLUMN_NAME_KARTENNUMMER,
  COLUMN_NAME_VORNAME, COLUMN_NAME_NACHNAME,
      COLUMN_NAME_BONUSPUNKTE };
```

Subsequently, table for customer is created by using the following command.

```
 public static final String CREATE_TABLE =
  "CREATE TABLE " + TABLE_NAME + "(" +
  _ID + " INTEGER PRIMARY KEY," +
  COLUMN_NAME_KARTENNUMMER + " TEXT," +
  COLUMN_NAME_VORNAME + " TEXT," +
  COLUMN_NAME_NACHNAME + " TEXT," +
  COLUMN_NAME_BONUSPUNKTE + " TEXT)";
```

In order to store information into database, *insert()* method is called and *ContentValues* is applied to insert or update values into the table.

```
public static long insert(SQLiteDatabase db, String kartennummer,
   String vorname, String nachname, String bonuspunkte) {
  ContentValues values = new ContentValues();
  values.put(COLUMN_NAME_KARTENNUMMER, kartennummer);
  values.put(COLUMN_NAME_VORNAME, vorname);
  values.put(COLUMN_NAME_NACHNAME, nachname);
  values.put(COLUMN_NAME_BONUSPUNKTE, bonuspunkte != null ?
      bonuspunkte : "0");
  return db.insert(TABLE_NAME, null, values);
}
```

The table name and column name for table receipt are defined in the following code.

```
public static class Bon implements BaseColumns {
  public static final String TABLE_NAME = "bon";
  public static final String COLUMN_NAME_ZEIT = "zeit";
  public static final String COLUMN_NAME_UMSATZ = "umsatz";
  public static final String COLUMN_NAME_RETOURE = "retoure";
  public static final String COLUMN_NAME_KUNDE_ID = "kunde_id";
  public static final String[] ALL_COLUMNS = { _ID,
      COLUMN_NAME_ZEIT,
     COLUMN_NAME_UMSATZ, COLUMN_NAME_RETOURE, COLUMN_NAME_KUNDE_ID
         };
```

Table for receipt is created as shown below. The keyword AUTOINCREMENT is used to ensure that the *_ID* of the new row that is inserted after the collected data is transfered to server is at least one larger than the largest ID that has ever before been inserted in that same table. If no data has been inserted into the database before, *_ID* will start at 1. By doing these, it allows user to know how many receipts have been collected in total, including those that have been transfered to the server. Besides that, the foreign key *kunde_id* is defined through the foreign key constraint. In the foreign key reference, a link is created between table customer and table receipt.

```
public static final String CREATE_TABLE =
"CREATE TABLE " + TABLE_NAME + "(" +
_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
COLUMN_NAME_ZEIT + " INTEGER," +
COLUMN_NAME_UMSATZ + " INTEGER," +
COLUMN_NAME_RETOURE + " INTEGER," +
COLUMN_NAME_KUNDE_ID + " INTEGER, FOREIGN KEY (" +
   COLUMN_NAME_KUNDE_ID + ") REFERENCES " + Kunde.TABLE_NAME +
   "(" + Kunde._ID + "))";
```

Implemention

After the database has been built up, different methods are used for the retrieval, storage or modification of data for different activities in the application. The methods for implementation of database are located at *DatabaseHelper.java*.

The first method is used to display the total number of receipts that have been saved in database in sale mode. By calling method *getReadableDatabase()*, the database is opened. The method *query()* from *Cursor* is used so that it will return a result set with a cursor pointing to the table. Subsequently, the cursor will start counting number of rows in the table. After the counting has completed, the opened database will be closed. The total number of rows in table receipt is equivalent to the total number of receipts.

```
public int getReceiptTotal() {
  SQLiteDatabase db = getReadableDatabase();
  Cursor c = db.query(ScannerContract.Bon.TABLE_NAME, new
     String[] { ScannerContract.Bon._ID },
  null, null, null, null, null);
  int count = c.getCount();
  c.close();
  return count;
}
```

The following method is applied to clear all the receipts that are stored in the database after the data has been transfered to the server.

```
public void deleteAllReceipts() {
  SQLiteDatabase db = getWritableDatabase();
  db.delete(ScannerContract.Bon.TABLE_NAME, null, null);
}
```

In Edit (yet to be synchronized) Turnover activity, all the receipts that is stored in the database are displayed in a list. *SQLiteQueryBuilder* is used to join the table receipt and table customer. The method *setTables()* enables the tables to join. Table receipt *LEFT OUTER JOIN* table customer means that it fetches all data from table receipt with matching data from table customer. The list in Edit turnover activity is arranged according to the receipt's ID.

```
public List<Bon> getAllReceipts(boolean ascending) {
  SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
  builder.setTables(ScannerContract.Bon.TABLE_NAME + " LEFT OUTER
     JOIN "
  + ScannerContract.Kunde.TABLE_NAME + " ON "
  + ScannerContract.Bon.COLUMN_NAME_KUNDE_ID + " = "
  + ScannerContract.Kunde.TABLE_NAME + "." +
     ScannerContract.Kunde._ID);
  String orderBy = ScannerContract.Bon.TABLE_NAME + "." +
     ScannerContract.Bon._ID
  + (ascending ? " ASC" : " DESC");
  SQLiteDatabase db = getReadableDatabase();
  Cursor c = builder.query(db, null, null, null, null, null,
     orderBy);
```

The method below is used to pull the details of customer and fill them in the corresponding text fields in sale mode when the card number is received after the scan of barcode scanner. The *query()* method is called to find the specific card number in the table for customer in order to fetch the specific information from database. However, if there is no cursor or the cursor did not move to the first row of the results, it will return null and no result will be displayed on sale mode.

```
public Kunde getCustomerWithNumber(String kartennummer) {
  SQLiteDatabase db = getReadableDatabase();
  Cursor c = db.query(ScannerContract.Kunde.TABLE_NAME,
     ScannerContract.Kunde.ALL_COLUMNS,
  ScannerContract.Kunde.COLUMN_NAME_KARTENNUMMER + " = ?",
  new String[] { kartennummer }, null, null, null);
  if (c == null) {
    return null;
  }
  if (!c.moveToFirst()) {
      c.close();
      return null;
  }
}
```

The method *addReceipt()* is used to insert the data of new receipt into database. By using method *getWritableDatabase()*, database is opened and ready for reading and writing. It will return method *insert()* in order to insert the data into database.

```
public boolean addReceipt(long zeit, long umsatz, boolean
     retoure, long kunde_id) {
    SQLiteDatabase db = getWritableDatabase();
    return ScannerContract.Bon.insert(db, zeit, umsatz, retoure,
         kunde_id) != -1;
}
```

The following method is applied when the turnover value is edited at Local Values Editor activity and trying to update the new turnover value into the database. The method *getWritableDatabase()* is called to allow the storage of this new data. Only one row of data is updated for each time. Also, the number of rows remain unchanged for this update.

```
public boolean updateReceipt(int _id, long umsatz) {
    SQLiteDatabase db = getWritableDatabase();
    int rowsUpdated = ScannerContract.Bon.updateUmsatz(db, _id,
         umsatz);
    if (rowsUpdated > 1) {
        Log.d("DatabaseHelper", "Multiple rows affected when updating
             Bon: _id = " + _id + ".");
        return false;
    }
    return rowsUpdated == 1;
}
```

In addition, new customers can be added or the information of customers can be edited using the following method. The database is opened to get ready for writing.

```
public boolean updateOrAddCustomer(String kartennummer, String
     vorname, String nachname, String bonuspunkte) {
    SQLiteDatabase db = getWritableDatabase();
    return ScannerContract.Kunde.updateOrInsert(db, kartennummer,
         vorname, nachname, bonuspunkte) == 1;
}
```

5. File transfer

The aim of this chapter is to give an introduction on the type of files that will be transfered to the server. Besides that, the implementation of uploading and downloading files between database and server will also be discussed. The implementation code for this chapter can be found under *Utility.java*.

5.1. Internal Description of Data

5.1.1. Record String

The following table shows the construction of a record string. All the receipts that are stored in the database will be converted into record string according to the syntax as shown in Table 5.1.

Table 5.1.: Construction of a Record String

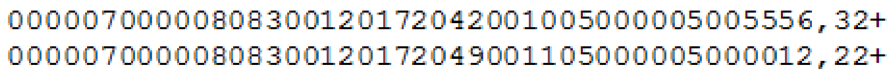
Position	Description	Syntax
1	Register Number	6 digits, numeric, (xxxxxx), required
7	Device Number	8 digits, numeric, (xxxxxxxx), required
15	Receipt Date	8 digits, numeric, (DDMMYYYY), required
23	Receipt Time	4 digits, numeric, (HHMM), required
27	Receipt Number	4 digits, numeric, (xxxx), required
31	Card Number	7 digits, numeric, (xxxxxxx), required
38	EAN-8 Checksum Digit	1 digit, numeric, (x), required
39	Turnover	9 digits, numeric, (VVVVVV,NN), required
48	Identifier	1 digit, defined: (+) Sale, (-) Return, required

The codes below shows how the structure of a record string is being constructed. The numerical values are aligned to the right and the left are filled with zeros. Turnover values are converted as 6 digits before the decimal point, a decimal point and 2 digits after the decimal point in the string. For example, for a turnover values of 16,78, it should be displayed as

000016,78 in the record string. The combination of register number, device number, receipt date, receipt time and receipt number must be unique.

```
private static final String SERVER_DATE_FORMAT_STRING =
    "ddMMyHHmm";

public static String formatServerUmsatzString(String registerNr,
    String deviceNr, long zeit, long bon_id, String kartennummer,
    long umsatz, boolean retoure) {
   return String.format("%s%s%s%04d%s%06d,%02d%s", registerNr,
       deviceNr, SERVER_DATE_FORMAT.format(new Date(zeit)), bon_id
       % 10000, kartennummer, umsatz / 100, umsatz % 100, retoure ?
       "-" : "+");
}
```

A record string should be displayed in the structure as shown in the Figure 5.1.



```
000007000008083001201720420010050000005005556,32+
000007000008083001201720490011050000005000012,22+
```

Figure 5.1.: Examples of Record String

5.1.2. Record File

A record file contains all the record strings that are needed to be transfered to the server. Figure 5.2 shows the set up of the name of the record file. The file name of record file should end with MDE.

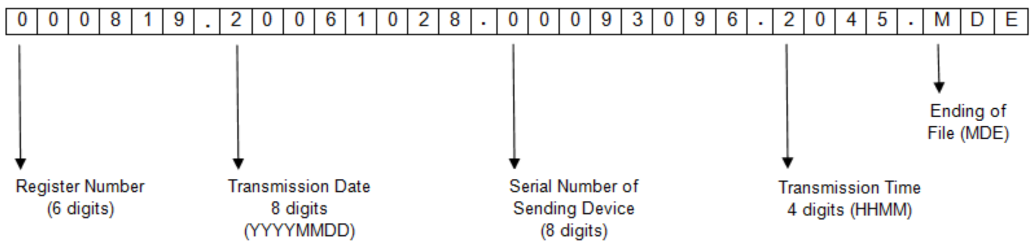

0 0 0 8 1 9 . 2 0 0 6 1 0 2 8 . 0 0 0 9 3 0 9 6 . 2 0 4 5 . M D E
Ending of File (MDE)
Register Number (6 digits)
Transmission Date 8 digits (YYYYMMDD)
Serial Number of Sending Device (8 digits)
Transmission Time 4 digits (HHMM)

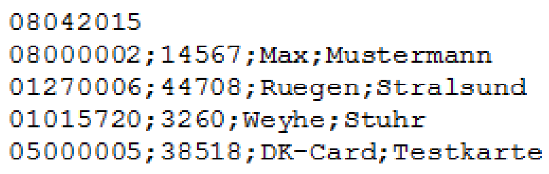
Figure 5.2.: Record File

The following codes describe the construction of the file name for record file according to Figure 5.2.

```
private static final String SERVER_FILE_DATE_FORMAT_STRING =
   "yMMdd";
private static final String SERVER_FILE_TIME_FORMAT_STRING =
   "HHmm";


public static String formatServerUmsatzFileNameString(String
   registerNr, Date date, String deviceNr) {
  return String.format("%s.%s.%s.%s.MDE", registerNr,
     SERVER_FILE_DATE_FORMAT.format(date),
  deviceNr, SERVER_FILE_TIME_FORMAT.format(date));
}
```

5.1.3. Bonus File

Figure 5.3 shows the contents of a bonus file. A bonus file is generated from the information in FTP server. It consists of not only the date of bonus point calculation as well as the first and last name of the customers, but also the current total bonus point of the customers. The first line displays the date of bonus point calculation in the form of DDMMYYYY. From the second line onwards, the details of the customers are displayedin the form of: card number (7 digits); total bonus point; first name; last name.



```
08042015
08000002;14567;Max;Mustermann
01270006;44708;Ruegen;Stralsund
01015720;3260;Weyhe;Stuhr
05000005;38518;DK-Card;Testkarte
```

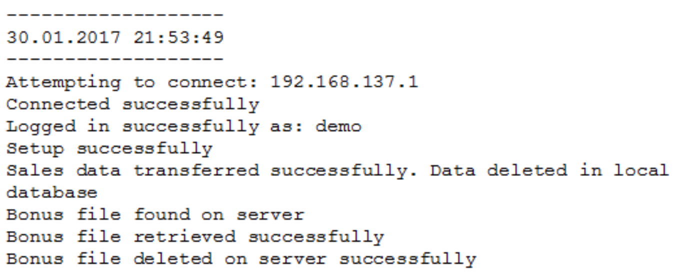
Figure 5.3.: Bonus File

5.1.4. Log File

All the process involved during the file transfer are documented in a log file. Figure 5.4 displays the contents of a log file.

The codes below shows the set up of the name of log file. It is the combination of register number and device number with the file extension *.log*.

```
String logName = registerNr + deviceNr + ".log";
```



```
-------------------
30.01.2017 21:53:49
-------------------
Attempting to connect: 192.168.137.1
Connected successfully
Logged in successfully as: demo
Setup successfully
Sales data transferred successfully. Data deleted in local
database
Bonus file found on server
Bonus file retrieved successfully
Bonus file deleted on server successfully
```

Figure 5.4.: Log File

5.2. Data Transmission

A server is a computer or a device which provides resources or services for other computers or devices, known as clients. One client can use several servers and a server can perform services for several clients. The server receives and gives responds to the requests send by the clients. File Transfer Protocol (FTP) is a commonly used protocol for transfering files between a server and client on a network.

In this bachelor thesis, as a basic test, an open source software FileZilla is selected as the program of choice to set up a laptop as FTP Server. Firstly, the laptop creates an adhoc network so that the mobile device can connect to the laptop. Now that both the devices are on the same network, the mobile device can access to the FTP Server. One important thing to note is to ensure that the port and the program are not blocked by the firewall. Subsequently, user accounts and user groups can be created on the FTP Server with FileZilla. Now, the mobile device which acts as FTP Client, can connect to the server with the created user account for the synchronization of data. The FTP Server has a flat hierarchy and is structured without subdirectories. Therefore, all files that are being transfered to FTP Server will be stored in the root of the home directory of the server.

In order to transfer the data, user can click on the option ”Datenübertragung” at the overflow menu (see Figure 5.5).

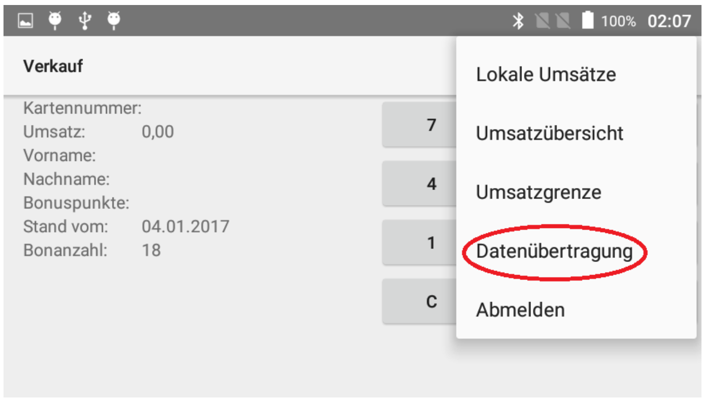


Figure 5.5.: Link to File Transfer

After user has clicked on the option, a dialog as seen in Figure 5.6 will pop out to confirm that user wants to transfer the data from database to server.

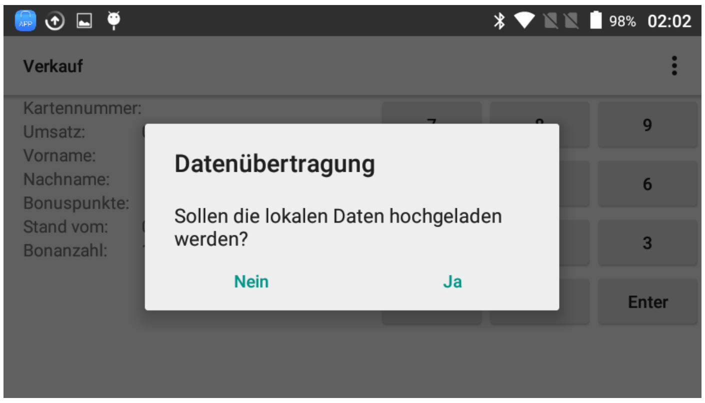


Figure 5.6.: Dialog for File Transfer

When data transmission starts, the method *synchronize()* is called to synchronize the local data with the server. First and foremost, the register number is being retrieved from the preference file to make sure that it has been set up. The same process happens to device number, FTP Server, FTP User and FTP Password.

```
public static String synchronize(Context context, DatabaseHelper
    db) {
   String registerNr = retrieveRegisterNumber(context);
   if (registerNr == null || registerNr.length() == 0) {
      return "Die Geschäftsstellennummer ist nicht gesetzt.";
   }
```

The date and time are recorded as the header of the data transmission in the log.

```
StringBuilder logBuilder = new StringBuilder();
logBuilder.append("--------------------");
logBuilder.append("\n");
logBuilder.append(LOG_DATE_FORMAT.format(date));
logBuilder.append("\n");
logBuilder.append("--------------------");
logBuilder.append("\n");
```

After that, record strings are constructed and combined as record file content by using *String-Builder*. The method *append()* of the *StringBuilder* is utilized here for better string concatenation performance. Lastly, the contents of record file are converted into a single string and the file name is constructed for record file.

```
List<DatabaseHelper.Bon> bons = db.getAllReceipts(true);
if (!bons.isEmpty()) {
  uploadUmsatz = true;
  StringBuilder sb = new StringBuilder();
  for (DatabaseHelper.Bon bon : bons) {
    sb.append(formatServerUmsatzString(registerNr, deviceNr,
       bon.zeit, bon._id,
    bon.kunde.kartennummer, bon.umsatz, bon.retoure));
    sb.append("\n");
  }
  fileContent = sb.toString();
  fileName = formatServerUmsatzFileNameString(registerNr, date,
     deviceNr);
}
```

However, if there are no receipts stored in the database, these steps will be skipped.

```
else {
  uploadUmsatz = false;
  error = "Keine Bons vorhanden";
}
```

Once the record file content is generated, the FTP Client, which is the mobile device, will attempt to connect to the FTP Server. The FTP Client is implemented using the open source Apache Commons Net library. After it has successfully connected, the FTP Client can log in using enabled username and password. Subsequently, the setup is successful, when FTP Client is set to passive mode and the file type of FTP Client is set to binary file type. All of these will be protocolled into the log file.

```
try {
  logBuilder.append("Attempting to connect: ");
  logBuilder.append(server);
  logBuilder.append("\n");
  client.connect(server);
  logBuilder.append("Connected successfully");
  logBuilder.append("\n");
  if (client.login(user, password)) {
    logBuilder.append("Logged in successfully as: ");
    logBuilder.append(user);
    logBuilder.append("\n");
    client.enterLocalPassiveMode();
    client.setFileType(FTP.BINARY_FILE_TYPE);
    logBuilder.append("Setup successfully");
    logBuilder.append("\n");
```

Now that the FTP Client is connected to the FTP Server, record file is generated and will be uploaded to the server by using *InputStream*. If the record file is successfully stored on the server, data in the local database will be deleted and it will be protocolled into log file. Otherwise, if the record file could not be uploaded, the failure will be recorded into the log file.

```
if (uploadUmsatz) {
  InputStream is = new
     ByteArrayInputStream(fileContent.getBytes("UTF-8"));
  boolean success = client.storeFile("/" + fileName, is);
  is.close();
  if (success) {
    db.deleteAllReceipts();
    logBuilder.append("Sales data transferred successfully. Data
       deleted in local database");
    logBuilder.append("\n");
  } else {
    error = "Die Umsatzdatei kann nicht hochgeladen werden.";
    logBuilder.append("Failed to transfer sales data");
    logBuilder.append("\n");
  }
}
```

Subsequently, a command is sent by the mobile device to server in order to request for the download of bonus file. If the command did not send successfully, it means that the query for bonus file is failed. The result will be recorded into log file. However, if the server has successfully received the command and there is a bonus file exists on the server, it will be protocolled into log file to indicate that the bonus file is found on the server. Also, the bonus file will be downloaded by FTP Client by using the *OutputStream* and stored in a temporary file.

```
String[] files = client.listNames(BONUS_FILE_PATHNAME);
if (files == null) {
  logBuilder.append("Failed to query bonus file");
  logBuilder.append("\n");
} else if (files.length == 1) {
  logBuilder.append("Bonus file found on server");
  logBuilder.append("\n");
  File tempFile = File.createTempFile("bonusdatei-" +
     System.currentTimeMillis(),
  null, context.getCacheDir());
  OutputStream os = new FileOutputStream(tempFile);
  boolean success = client.retrieveFile(BONUS_FILE_PATHNAME, os);
  os.close();
 }
```

As the bonus file is successfully downloaded from the server, the temporary file is read by the *FileReader* and *BufferedReader*. The temporary is being checked to make sure that the date of bonus point calculation is at the first line of the file and it is in the proper format. Then, it will continue to check on the following lines that contains the information of customers. It should be split in four parts for each line. At the end, if they are all in proper format, the database is updated with the downloaded bonus file. On the other hand, if the data of customer is not in appropriate format, it will be recorded into log file to indicate that the format of the bonus file content is invalid and database will not be updated.

```
if (success) {
  FileReader fr = new FileReader(tempFile);
  BufferedReader br = new BufferedReader(fr);
  String line = br.readLine();
  if (line != null) {
    if (line.length() == 8) {
      storeEffectiveDate(context, line.substring(0, 2) + "." +
         line.substring(2, 4) + "." + line.substring(4, 8));
      while ((line = br.readLine()) != null) {
        String[] data = line.split(";");
        if (data.length != 4) {
          logBuilder.append("Invalid format of the bonus file
             content");
          logBuilder.append("\n");
          break;
        }
        db.updateOrAddCustomer(data[0], data[2], data[3],
           data[1]);
      }
```

The bonus file on the server will be deleted after it has been transfered successfully to the local database. In the log file, it is protocolled that the bonus file is being downloaded. In addition, as the bonus file is deleted, it will also be documented into the log file to show that bonus file on server is deleted successfully.

```
tempFile.delete();
logBuilder.append("Bonus file retrieved successfully");
logBuilder.append("\n");
if (client.deleteFile(BONUS_FILE_PATHNAME)) {
  logBuilder.append("Bonus file deleted on server successfully");
  logBuilder.append("\n");
```

Last but not least, all the file transfer activities of data transmission that have been documented in the log file will be sent to the server from mobile device in the end.

Firstly, it is checked whether the local log file exists. If there is a log file in the mobile device, it will be appended to the server. After the log file is successfully transfered, it will be deleted in the mobile device. However, if the transfer failed, the result will be recorded into the log file to indicate that the existing log file did not transfer successfully.

```
if (contains) {
  FileInputStream fis = context.openFileInput(logName);
  success = client.appendFile(logName, fis);
  if (success) {
    context.deleteFile(logName);
  } else {
    logBuilder.append("Failed to upload existing local transfer
       log");
    logBuilder.append("\n");
  }
}
```

On the other hand, if there is no transmission log file found on the mobile device or the transfer of log file has succeeded, the log of current data transmission will be uploaded on the server.

```
if (!contains || success) {
  byte[] logData = logBuilder.toString().getBytes("UTF-8");
  InputStream is = new ByteArrayInputStream(logData);
  success = client.appendFile(logName, is);
  if (!success) {
    logBuilder.append("Failed to upload current transfer log");
    logBuilder.append("\n");
  }
  is.close();
}
```

If this upload operation fails, the current log, including this failure, will be written to the log file locally, appending to it if already exists.

```
if (!success) {
  byte[] logData = logBuilder.toString().getBytes("UTF-8");
  FileOutputStream fos = context.openFileOutput(logName,
     Context.MODE_APPEND);
  fos.write(logData);
  fos.close();
}
```

6. Conclusion

In this bachelor thesis, an Android application is created to manage and track the sales transactions, so that it is able to extend the functionality of mobile device (tablet or smartphone) as a general purpose device. Besides that, a barcode scanner is connected to the mobile device in order to scan the customer's card and retrieve the information of customer from the database. Different functionalities are implemented in the application for allowing user to add, edit and store the latest sales transactions. Furthermore, the application is developed to give user an overview of the transaction history. A database is also created to save the data of sale transactions locally. As a basic field test, a laptop is configured as the FTP server for the application to connect to. The data are exchanged as files between the local database and the server using this setup.

In conclusion, the main purpose of this bachelor thesis is achieved. The outcome of this work provides a general framework for managing sales transactions. The application can be easily tuned and further developed to fit the requirements of actual use.

Bibliography

[Android a] *Broadcast Receiver*. Android Developers. – URL https://developer.android.com/reference/android/content/BroadcastReceiver.html. – Accessed: January 30, 2017

[Android b] *ClassCastException*. Android Developers. – URL https://developer.android.com/reference/java/lang/ClassCastException.html. – Accessed: November 28, 2016

[Android c] *Intents and Intent Filters*. Android Developers. – URL https://developer.android.com/guide/components/intents-filters.html. – Accessed: November 16, 2016

[Android d] *Linear Layout*. Android Developers. – URL https://developer.android.com/guide/topics/ui/layout/linear.html. – Accessed: October 03, 2016

[Android e] *List View*. Android Developers. – URL https://developer.android.com/guide/topics/ui/layout/listview.html. – Accessed: December 10, 2016

[Android f] *NumberFormatException*. Android Developers. – URL https://developer.android.com/reference/java/lang/NumberFormatException.html. – Accessed: November 25, 2016

[Android g] *Saving Key-Value Sets*. Android Developers. – URL https://developer.android.com/training/basics/data-storage/shared-preferences.html. – Accessed: December 30, 2016

[Android h] *Setting Up the App Bar*. Android Developers. – URL https://developer.android.com/training/appbar/setting-up.html. – Accessed: December 11, 2016

[Android i] *Simple Date Format*. Android Developers. – URL https://developer.android.com/reference/java/text/SimpleDateFormat.html. – Accessed: Oktober 08, 2016

[Android j] *String Builder*. Android Developers. – URL https://developer.android.com/reference/java/lang/StringBuilder.html. – Accessed: January 30, 2017

[Apache 2017] *Apache Commons Net*. Apache Commons. 2017. – URL https://commons.apache.org/proper/commons-net/. – Accessed: January 30, 2017

[Beal 2016] BEAL, Vangie: *DBMS - database management system*. Webopedia. 2016. – URL http://www.webopedia.com/TERM/D/database_management_system_DBMS.html. – Accessed: September 13, 2016

[Codd 1970] CODD, E.F: *A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, Pg.377-387*. 1970. – URL http://dl.acm.org/citation.cfm?doid=362384.362685. – Accessed: August 6, 2016

[Codepath 2016] *Using an ArrayAdapter with ListView*. Codepath. 2016. – URL https://guides.codepath.com/android/Using-an-ArrayAdapter-with-ListView. – Accessed: December 10, 2016

[Google 2016a] *Material Icon*. Google. 2016. – URL https://material.io/icons/. – Accessed: November 20, 2016

[Google 2016b] *Unit and Measurements*. Google. 2016. – URL https://material.io/guidelines/layout/units-measurements.html#. – Accessed: October 03, 2016

[Rouse 2016a] ROUSE, Margaret: *database*. TechTarget. 2016. – URL http://searchsqlserver.techtarget.com/definition/database. – Accessed: September 13, 2016

[Rouse 2016b] ROUSE, Margaret: *server*. TechTarget. 2016. – URL http://whatis.techtarget.com/definition/server. – Accessed: January 11, 2017

[SQLite] *SQLite Autoincrement*. SQLite. – URL http://sqlite.org/autoinc.html. – Accessed: January, 2017

[Suhl] SUHL, A.: *Spezifikation für Datenerfassungs-App für Tablet oder Handy*. Wöhlke EDV-Beratung GmbH. – Accessed: January 05, 2017

[Tutorials a] *Android Activities*. Tutorials Point. – URL https://www.tutorialspoint.com/android/android_acitivities.htm. – Accessed: November 16, 2016

[Tutorials b] *Android List View*. Tutorials Point. – URL https://www.tutorialspoint.com/android/android_list_view.htm. – Accessed: December 10, 2016

[Wassermann 2011] TODD, Wassermann: *Google CEO: Mobile Growing Faster Than All Our Predictions*. Mashable. 2011. – URL http://mashable.com/2011/02/28/schmidt-mobile-growth/#fv_.g9XO2Sqc. – Accessed: July 18, 2016

[Wikipedia] *Server (Computing)*. Wikipedia. – URL https://en.wikipedia.org/wiki/Server_(computing). – Accessed: January 13, 2017

[Zilla 2017] *FileZilla the free FTP solution*. FileZilla. 2017. – URL https://filezilla-project.org/. – Accessed: January 13, 2017

A. CD-ROM

Appendix A :

- Bachelor Thesis (File Name: thesis.pdf)
- Android Studio Project (Folder Name: Scanner)

The Appendix A of this bachelor thesis is on a CD-ROM and is available from Prof. Dr.-Ing. Wilfried Wöhlke and Prof. Dr. Andreas Suhl.

Nomenclature

API	Application Programming Interface
DBMS	Database Management System
dp	Density-independent Pixel
FTP	File Transfer Protocol
GUI	Graphic User Interface
ID	Identity
IDE	Integrated Development Environment
OS	Operating System
RDBMS	Relational Database Management System
SDK	Software Development Kit
sp	Scale-independent Pixels
SQL	Structured Query Language
UI	User Interface

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 2. Februar 2017	
Ort, Datum	Unterschrift