

“Android based Meter Reader”

By

Kashif Manzoor (2011-NUST-SEECS-BE-SE-221)

Muhammad Unnus (2011-NUST-SEECS-BE-SE-240)

Usman Khan (2010-NUST-SEECS-BE-SE-273)



Project documentation submitted in partial fulfillment of the
requirements for the degree of

Bachelors of Engineering in Software Engineering (BESE)

NUST School of Electrical Engineering and Computer Science

National University of Sciences and Technology

Islamabad, Pakistan

(2015)

CERTIFICATE

It is certified that the contents and the form of this report titled “**Android based Meter Reader**” submitted by **Kashif Manzoor (2011-NUST-SEECs-BE-SE-221)**, **Muhammad Unnus (2011-NUST-SEECs-BE-SE-240)**, **Usman Khan (2010-NUST-SEECs-BE-SE-273)** have been found satisfactory for the requirement of degree.

Advisor : _____

(Dr. Khalid Latif)

Co-Advisor : _____

(Dr. Asad Anwar Butt)

DEDICATION

To Allah the Almighty

&

To our Parents and Faculty

ACKNOWLEDGEMENTS

We are deeply thankful to our Advisor and Co-Advisor, Dr. Khalid Latif, and Dr. Asad Anwar Butt for helping us throughout the course in accomplishing our final project. Their guidance, support and motivation enabled us in achieving the objectives of the project.

We are also thankful to our committee members Mr. Maajid Maqbool and Mr. Jaudat Mamoon for their valuable feedback. Apart from them we are also thankful to Mr. Iqbal Shah (Line Superintendent, WAPDA) for arranging our visits to meter reading offices, which helped us clarify the requirements for the project.

TABLE OF CONTENTS

INTRODUCTION	4
1.1 DOMAIN INTRODUCTION	4
1.2 PROBLEM BACKGROUND	6
1.3 SOLUTION METHODOLOGY	7
LITERATURE REVIEW	9
2.1 EXISTING SOLUTIONS	9
2.2 RESEARCH ARTICLE	10
2.3 VISIT TO METER READING OFFICES	10
2.4 DELIVERABLES	12
WEB APPLICATION	15
3.1 DATABASE DESIGN	15
3.2 WEB API	21
3.3 INTERFACE DESIGN	25
3.4 CHOOSING TECHNOLOGIES	33
3.5 FRAMEWORKS AND NODE.JS MODULES	34
3.6 FEATURE IMPLEMENTATIONS	41
MOBILE APPLICATION	56
4.1 CHOOSING TECHNOLOGY	56
4.2 INTERFACE DESIGN	56
4.3 DATABASE DESIGN	61
4.4 IMPLEMENTING SYNC FEATURE	63
4.5 IMPLEMENTING GOOGLE MAPS IN ANDROID APPLICATION	64
4.6 IMPLEMENTING OCR IN ANDROID APPLICATION	65
IMAGE PROCESSING	67
5.1 DATA SET PREPARATION	67
5.2 TESTING EXISTING SOLUTIONS	70
5.3 IMAGE PREPROCESSING	73

5.4 TESSERACT OCR	78
5.5 IMPLEMENTATION ON ANDROID	82
CONCLUSION	87
6.1 LEARNING OUTCOMES	87
6.2 LIMITATIONS	87
6.3 FUTURE WORK	87
6.4 COMMERCIALIZATION	88
6.5 IMPACT ON SOCIETY	88
REFERENCES	89

ABSTRACT

Reading meters for electricity and other utilities is a manual and error prone process. This project automates the process of meter reading and billing. This helps reduce billing errors caused due to manual readings and also guides meter readers about the locations of different customers.

The system consists of a web application and an android application. The website administrator can add customers and meter readers to the system database using the web application. He can assign meters from a particular location to meter readers for reading. The meter reader receives locations of meters in the android application. These locations are marked on a map. The meter reader goes to each location and captures an image of customer's meter using the android device. The captured image is processed to extract the meter reading. This is done using OpenCV and Tesseract (Image Processing Libraries). If the reading is recognized incorrectly, the meter reader is given the facility to manually enter the reading. This reading is sent along with the image to the web application server, where it is verified by the website administrator. After verification the customer's data is updated and his bill is generated, which is accessible through the web interface.

INTRODUCTION

Android based Meter Reader is a meter reading system that is designed to automate the tasks of meter reading and billing of customers. Its main purpose is to address and rectify the flaws of current paper-based reading system being used in Pakistan.

This report starts by introducing the project, its scope and the problem it is trying to address. Chapter 2 describes the Literature Review that was carried out for Design & Planning of the project. Chapter 3 describes the development methodology and results for Web Application, Chapter 4 for Mobile Application & Chapter 5 for Image Processing. Chapter 6 concludes the project, by describing learning outcomes of the project and its impact on society.

1.1 Domain Introduction

This project is related to the field of Automatic Meter Reading (AMR) in general, and On-Site AMR in particular. AMR involves the development of technologies to facilitate the process of reading different types of metering devices, and generating bills for customers based on the obtained readings.

AMR systems have evolved substantially over the past couple of years. They can be broadly divided into two categories.

On-Site AMR, which was used by earlier AMR systems. Meter reader had to visit each meter location, and use a handheld mobile device to take meter reading either using some automatic collection mechanism or through manual entry (Figure 1.1). Some meters were installed with specialized transmitters that could communicate with mobile devices to transmit the reading. These systems improved upon the old paper based methods (Figure 1.2). They provided accurate readings and better management of data,

which reduced the number of billing errors. They did not significantly decrease the labor costs, but they did decrease the amount of effort required by meter readers for the job.



Figure 1.1 On-Site AMR



Figure 1.2 Paper based meter reading

Remote AMR, is now being used in modern AMR systems. In these types of systems, specialized meters have to be installed, which have special transmitters for communicating wirelessly with the central server (Figure 1.3). In fact, these types of meter are now being called Smart Meters. They can communicate usage and diagnostic data to the central server in real time. This allows utility providers to generate highly accurate bills without performing any kind of estimations. It also helps in producing better usage statistics for customers and utility providers. The diagnostic data allows for better maintenance and more timely response to any kind of problems. There are no labor costs of meter readers in these systems. Though these type of systems provide a

lot of advantages, they also require quite a significant initial investment from utility providers.



Figure 1.3 Remote AMR

Due to the advantages of AMR, a lot of countries are now deploying different types of AMR systems at varying scales. Some countries are carrying out the deployment slowly, region by region, while others have already implemented the system throughout the country. The world's largest smart meter deployment was undertaken by Enel SpA, the dominant utility in Italy with more than 30 million customers. Between 2000 and 2005 Enel deployed smart meters to its entire customer base.

1.2 Problem Background

Pakistan is a third world country. It has been facing a serious energy crisis over the past couple of years. Moreover, consumers have been complaining against incorrect billing since long in the country. The government is taking some measures to solve these problems through installation of AMR systems across the country. However, this process is slow and unorganized. There is no clue as to when it will complete.

The current meter reading practice at DISCOs (distribution companies) is paper based. Meter reading of all the consumers is carried out on monthly basis. The meter reader visits each consumer, reads the meter and notes down the readings in readings book. These recorded readings are then input to the central system, so that the billing

process can proceed. This process is very time consuming and vulnerable to errors. Human intervention at multiple levels within the billing process leads to manipulation of data and results in incorrect billing. This increases the number of customer complaints, and delay in bill collection, and often results in losses to DISCOs.

Hence a smart metering solution is the need of the hour, which would help prevent under billing, over billing of customers. It should also make meter reading process more efficient and provide better management of data.

1.3 Solution Methodology

This main objective of this project is to automate the process of meter reading and billing. This helps reduce billing errors caused due to manual readings and also guides meter readers about the locations of different customers.

The system consists of a web application and an android application. The website administrator can add customers and meter readers to the system database using the web application. He can assign meters from a particular location to meter readers for reading. The meter reader receives locations of meters in the android application. These locations are marked on a map. The meter reader goes to each location and captures an image of customer's meter using the android device. The captured image is processed to extract the meter reading. This is done using OpenCV and Tesseract (Image Processing Libraries). If the reading is recognized incorrectly, the meter reader is given the facility to manually enter the reading. This reading is sent along with the image to the web application server, where it is verified by the website administrator. After verification the customer's data is updated and his bill is generated, which is accessible through the web interface.

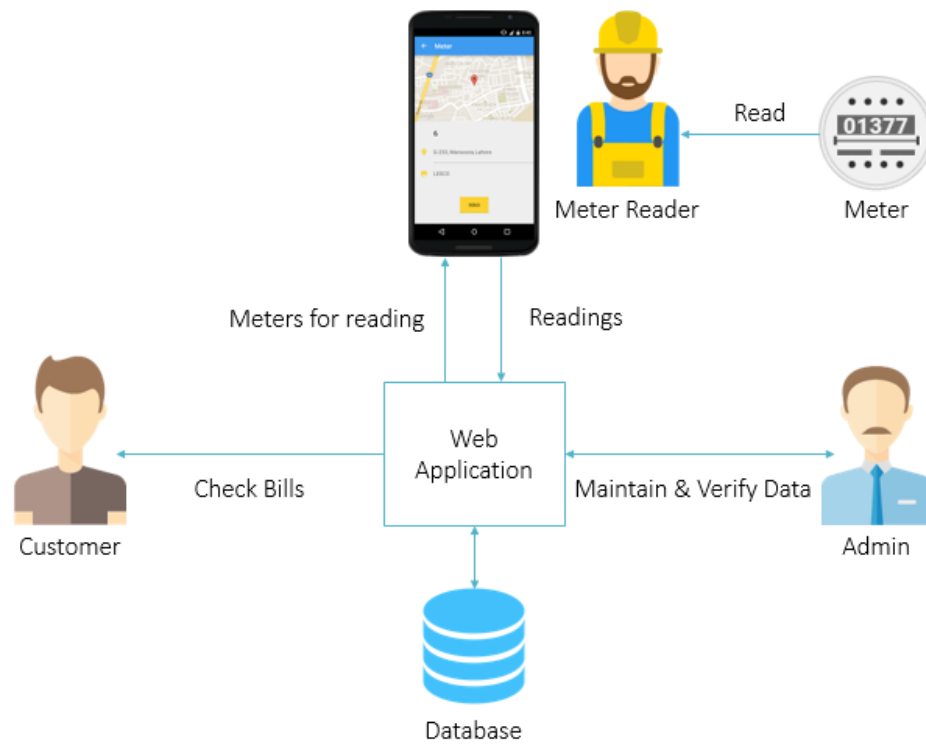


Figure 1.4 Flow of Data in the System

LITERATURE REVIEW

This chapter describes the literature review that was carried out before the development of the project. It was necessary because it helped clarify the requirements and features of the project. Moreover, it also contributed towards making the task of development easier.

The following sections describe the existing meter reading systems in use. A research article that was used as a guideline for this project is also described. The visits to Meter Reading offices are described. Moreover, the important deliverables for this project are identified in the end.

2.1 Existing Solutions

There have been reports of several Automatic Meter Reading (AMR) projects being carried out at different areas in Pakistan. The Kamalabad Subdivision of Islamabad Electric Supply Company, recently applied the Hand Held Unit (HHU) devices for meter reading. The method involves a meter reader taking a meter snap after manually entering the meter dial value. The project was carried out with the support of USAID Power Distribution Programme. The five-year USAID Power Distribution Programme was announced by Secretary of State Hillary Clinton in 2009 as one of the US efforts to support government of Pakistan to reform the power sector to mitigate the current energy crisis. ^[2]

The United States Agency for International Development (USAID) also installed more than 1,000 Automatic Meter Reading (AMR) devices on 85 grid stations of FESCO. These devices provide live load data information and perform load management. This aims to strengthen Pakistan's energy sector and train the staff of government-owned power distribution companies. ^[1]

Telenor Pakistan has also made some efforts for tackling the energy crisis. It has become the largest Automated Meter Reading (AMR) solutions provider in the country with an on-going deployment of 17,000 smart meters in five cities of Pakistan including Islamabad, Hyderabad, Multan, Lahore and Peshawar. This deployment will help reduce distribution losses, electricity theft and increase labor productivity by automating monitoring, meter reading and load management.

Despite all the efforts, a large part of the country still uses paper based meter reading system. This is mainly because adoption of new systems is slow and unorganized. Hence there is a strong need of an automatic meter reading system that can be adopted quickly and easily.

2.2 Research Article

The search for a quick and easy to implement AMR solution led to the discovery of a research article published in the International Journal of Computer Science and Mobile Computing (IJCSMC). The article was titled Android Based Meter Reading Using OCR. It suggested using an Android Application to take meter images and then extracting reading from the image using OCR. These readings would then be sent out to a web application server where customer data would be updated and bill would be generated. This would help reduce workload of the meter reader. The process of collecting the reading from meter, updating this reading to system and billing of customer would be made easy and accurate. This looked a very promising AMR solution, and it served as a guideline for this project. ^[6]

2.3 Visit to Meter Reading Offices

In order to identify the basic requirements of meter reading systems, a visit to meter reading offices was arranged. This visit also helped identify the flaws in existing systems. The main problem was with the management of data. Meter readers were recording readings on a readings book (Figure 2.1). These readings were then compiled and verified by commanding officers. The verified readings were sent to the data entry department where they were entered into the system database manually (Figure 2.2).

Therefore, it was decided to address this problem in the project. Some mechanism had to be devised that could automatically send and store the readings taken by the meter readers in the central database.

CONSUMER'S NAME AND ADDRESS										A.C. NO.			
3672/100/101										15375			
METER NO. 5582946										Meter Reading Cycle Date		Account No.	Tariff
S/Ph	3/Ph	MAKE	TYPE	M.F.	RANGE	AMPS	DATE INSTALLED	DATE REMOVED	REASONS FOR REMOVAL				
		HBM	28			1.4							
Financial Year 2013-14					Financial Year 2014-15								
Month	Reading		Advance	M/R's Initials/Total	Supdt. Initials	Month	Reading		Advance	M/R's Initials/Total	Supdt. Initial		
Jun	Peak					Jun	Peak						
	Off Peak						Off Peak						
May	Peak					May	Peak						
	Off Peak						Off Peak						
Apr	Peak					Apr	Peak						
	Off Peak						Off Peak						
Mar	Peak					Mar	Peak						
	Off Peak						Off Peak						
Feb	Peak					Feb	Peak						
	Off Peak						Off Peak						
Jan	Peak	1055	760	2		Jan	Peak						
	Off Peak						Off Peak						
Dec	Peak	9295	135	1	9502	Dec	Peak						
	Off Peak						Off Peak						
Nov	Peak	9160	1603			Nov	Peak						
	Off Peak		2556				Off Peak						
Oct	Peak	9005	890			Oct	Peak						
	Off Peak						Off Peak						
Sep	Peak	815				Sep	Peak						
	Off Peak						Off Peak						
Aug	Peak					Aug	Peak						
	Off Peak						Off Peak						
Jul	Peak					Jul	Peak						
	Off Peak						Off Peak						

Figure 2.1 Readings Book

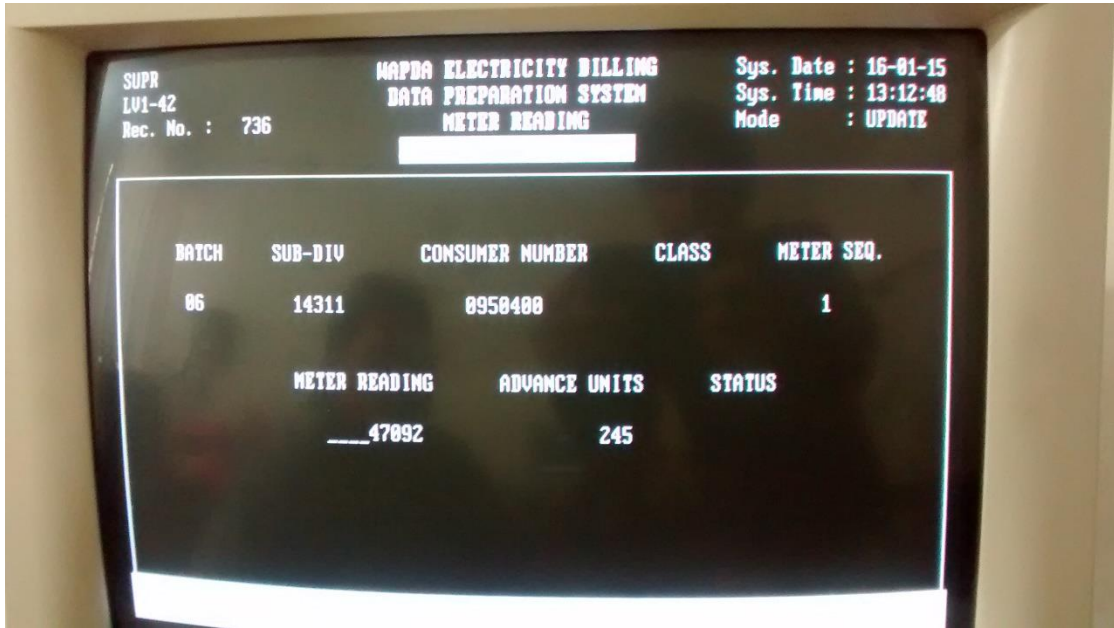


Figure 2.2 Data Entry System

2.4 Deliverables

The project is composed of three modules web application, mobile application and image processing. The deliverables for each of these modules are described below

Table 2.1: Web Application Deliverables

No.	Deliverable
1	Database Design
2	Web API Design
3	Interface Design
4	Implement Web API
5	Allow admin to add customers
6	Allow admin to add meters
7	Allow admin to add readers
8	Allow admin to assign meters to readers and unassign them

9	Send assigned meters to mobile application
10	Receive meter readings from mobile application
11	Associate reading to relevant meter
12	Allow admin to verify readings
13	Generate bill after reading verification
14	Display meter image along with bill
15	Allow customer to view his current and past billing information
16	Allow admin to update prices for bill calculation
17	Allow admin to create new admins
18	Perform admin authentication

Table 2.2: Mobile Application Deliverables

No.	Deliverable
1	Database Design
2	Interface Design
3	Receive a list of meters to read from web application
4	Keep track of the meters currently read by the meter reader
5	Display meter location in Google Map
6	Capture the image using camera and provide crop functionality
7	Integrate OCR solution
8	Perform OCR on cropped image to extract reading
9	Provide a manual entry fallback mechanism if OCR doesn't work
10	Save the readings locally in mobile app along with reading GPS location

11	Allow reader to retake a saved reading
12	Send saved readings along with image to web application
13	Perform reader authentication

Table 2.3: Image Processing Deliverables

No.	Deliverable
1	Create a dataset of meter images
2	Research about existing image processing libraries
3	Perform preprocessing on images
4	Test different OCR solutions and select the one with highest accuracy
5	Implement and test the OCR on preprocessed image

WEB APPLICATION

The web application laid out the foundation for the project. This is because the web application is responsible for the two most important tasks of the project.

Firstly, it communicates the meter information to the meter readers, without which the meter reading process cannot proceed. Secondly, it receives the meter readings performed by meter readers, verifies them and generates customer bills based on these readings.

This section describes the web application development methodology and implementation details for the project. It describes the architecture and design of the web application. The various components of the web application. The tools, technologies and frameworks used for the development of the web application.

3.1 Database Design

The first step in the development of web application was database design. It helped identify the information that would be needed to implement the various features of the application. Particular importance was given to the database design, since this project is database centric. That is it needs to access the database for most of its operations.

Efforts were made to keep the design as scalable as possible, since this project is intended to be scaled to fulfil the requirements of meter reading companies, each of which has its own diverse set of requirements. For this purpose, the most basic operations of meter reading companies were identified, and resources were created to fulfill the data needs of these operations.

Specific importance was given to introduce simplicity in the design of the database. This simplicity would later prove useful when scaling the system. Only the most relevant pieces of information was included in the database tables. Redundancy

was avoided as much as possible. The widely applicable principle of good software design, low coupling and high cohesion was followed while designing the database. This greatly simplified the operations of the system.

3.1.1 Identifying Tables

Keeping these valuable principles in mind, the main resources of the system were identified. These resources would actually form the tables of the database. The visits to meter reading and data entry offices helped greatly in this regard. It became quite clear what resources make up the core of the meter reading systems. The database design went through quite a lot of iterations that were mainly for refactoring and optimizations. In the end following tables formed our database

- customer - stores information about customers
- meter - stores information about meters
- reader - stores information about readers
- reader_meter - stores information about meters assigned to readers
- reading - stores information about readings
- price - stores information about price of units, tax etc.
- bill - stores information about customer bills

3.1.2 Identifying Fields & Constraints

The next step was identifying the fields for tables and their constraints. This required quite a lot of thought. In the beginning, only the most relevant fields could be identified. It was only after the features of the system became more clear, that fields for the tables were finalized. The following sections describe the fields for each table and rationale behind them. Primary keys are in **bold** and Foreign Keys are in *italic*. All fields have NOT NULL constraint unless stated otherwise. This is mainly because most of the fields are required for correct operation.

Customer Table

- **id** - unique identifier of customer

- name - displayed on customer bill
- address - the address where bills would be posted

Meter Table

- **id** - unique identifier of meter
- *customer_id* - identifies the customer to which meter belongs
- address - the address of meter, because it can be different from customer address. A customer can have meters registered in his name, at different locations.
- type - the type of meter either digital or mechanical. Helpful in running image processing algorithm customized according to type of meter.
- utility - the utility of meter either electricity or gas. Helpful in generating bills customized according to utility of meter.
- company - the company of meter (IESCO, PEPCO) to be displayed on bill
- precision - the position of decimal point of meter readings. Helpful for positioning decimal point in reading, if not recognized by image processing.
- lat - the latitude of meter's GPS location. Matched with latitude of reading to verify that meter reader actually visited the meter location to take reading.
- long - the longitude of meter's GPS location. Matched with longitude of reading to verify that meter reader actually visited the meter location to take reading.
- date - the connection date of meter to be displayed on bill
- last_reading - the last reading of the meter. Helpful when calculating units consumed for the bill. Also helps ensure that current reading is always greater than last reading.
- last_reading_date - the last reading date of meter. Helpful in sending meters assigned to meter readers on monthly basis.

Reader Table

- **id** - unique identifier of reader
- name - displayed when assigning meters to reader
- username - used by reader to login in android app. Helps in authentication.
- password - used by reader to login in android app. Helps in authentication.
- role - identifies the role of reader either reader or admin

Reader_Meter Table

- **id** - unique identifier of row
- *reader_id* - identifies the reader to which meter is assigned
- *meter_id* - identifies the meter that is assigned to reader
- start - the starting date of meter assignment. This helps keep record about which reader the meter was assigned to in a particular period.
- end - the ending date of meter assignment (DEFAULT NULL). Its value is set when meter assignment is revoked.

Reading Table

- **id** - unique identifier of reading
- *meter_id* - identifies the meter whose reading was taken
- *reader_id* - identifies the reader to who took the reading
- value - the value of the reading
- date - the date the reading was taken. This helps in setting last reading date of meters. It is also helpful in generating monthly bills.
- lat - the latitude of reading's GPS location. Matched with latitude of meter to verify that meter reader actually visited the meter location to take reading.
- long - the longitude of reading's GPS location. Matched with longitude of meter to verify that meter reader actually visited the meter location to take reading.
- status - the status of reading either VERIFIED or UNVERIFIED (DEFAULT). This helps show unverified readings to admin for verification.

Price Table

- **id** - unique identifier of price
- quantity - the quantity whose price is being recorded
- date - the date the price was specified. The latest price is used when generating current bills. The past prices remain saved for record.

Bill Table

- **id** - unique identifier of bill
- *meter_id* - the id of meter whose bill is being generated. Displayed on customer bill.
- *reading_id* - the id of reading that was used to generate bill. This id is used to retrieve the image of reading for display on customer bill.
- reading - the present reading of meter. Displayed on customer bill.
- units - the units consumed for the billing month. Displayed on customer bill.
- bill - the calculated bill for the month. Displayed on customer bill.
- date - the date the bill was generated. Used to display billing month on customer bill.

3.1.3 Identifying Relationships (ERD Diagram)

The next step was identifying the relationships between the database tables. These relationships have been indicated in the previous section. This section further describes them in detail.

An ERD diagram (Figure 3.1) was drawn to clarify the relationships between the resources of the project. The main advantage of this process was that, it helped identify the type of relationship between the resources. It could be a one-to-one, one-to-many or a many-to-many relationship. The type of relationship, dictated how the operation is implemented in the web application. The restrictions that should be applied to operation are also clarified.

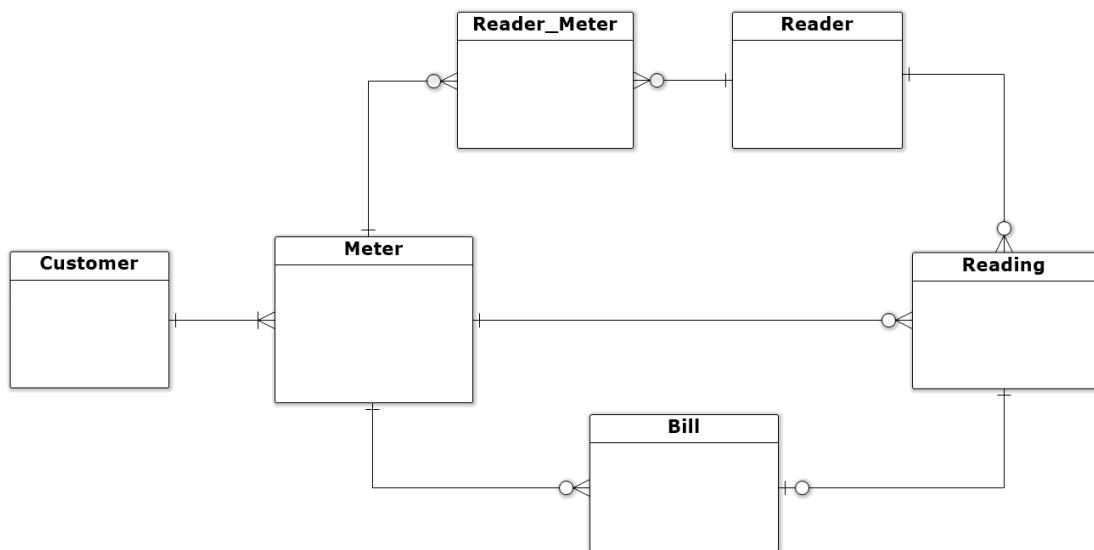


Figure 3.1 ERD Diagram

The following relationships were identified.

- A customer has one or many meters.
- A meter must be belong to customer and it belongs to only one customer.
- A meter is assigned to zero or one reader.
- A meter has zero or many readings.
- A meter has zero or many bills.
- A reader is assigned zero or many meters.
- A reader takes zero or many readings.
- A reading must belong to a meter and it belongs to only one meter.
- A reading must belong to a reader and it belongs to only one reader.
- A reading has zero or one bill.
- A bill must belong to a meter and it belongs to only one meter.
- A bill must be based on a reading and it is based on only one reading.

This section marks the end of the Database Design. All techniques and procedures that were used to design and develop the Database have been explained above.

3.2 Web API

After developing a comprehensive Database Design, the next task was designing the Web API of the application. For this purpose main resources of the system were identified. They were quite similar to the tables identified in the previous section. The operations that needed to be performed on these resources were identified. The data needs of these operations were fulfilled by exposing a Web API. The initial design of Web API only catered for the most important operations, it went through several iterations during the development of the web application to reach its final form. The RESTful architecture guidelines were followed to design the Web API. The advantage of using this architecture style was that it made the web application more scalable. Following paragraphs explain the procedures used for designing the Web API.

3.2.1 Identifying resources

- Admin
- Customer
- Meter
- Reader
- Reader_Meter
- Reading
- Price
- Bill

3.2.2 Identifying Resource Operations

Admin

- Add admin

Customer

- Add customer
- Get customer by id
- Get all customers

Meter

- Add meter
- Get meter by id
- Get all meters
- Update meter

Reader

- Add reader
- Get reader by id
- Get all readers

Reader_Meter

- Add reader_meter
- Get reader_meter by reader_id
- Get all reader_meter

Reading

- Add reading
- Get readings by meter_id
- Get all readings
- Update reading
- Delete reading

Price

- Add price
- Get price by quantity_name (unit, tax etc.)

Bill

- Add bill
- Get bills by meter_id

3.2.3 Implementation in Node.js

As explained later in the Section 3.4, the server-side technology used for web application implementation was Node.js. The following diagram shows the implementation of Web API in Node.js

```
var admin = require('../app/controllers/admin');
var customer = require('../app/controllers/customer');
var meter = require('../app/controllers/meter');
var reader = require('../app/controllers/reader');
var reading = require('../app/controllers/reading');
var reader_meter = require('../app/controllers/reader_meter');
var price = require('../app/controllers/price');
var bill = require('../app/controllers/bill');

module.exports = function (app, passport) {

  /**
   * Admin Routes
   */
  app.post('/admin', admin.addAdmin);

  /**
   * Customer Routes
   */
  app.post('/customer', customer.addCustomer);
  app.get('/customer/:customerId', customer.getCustomer);
  app.get('/customer', customer.getCustomers);

  /**
   * Meter Router
   */
  app.post('/meter', meter.addMeter);
  app.get('/meter/:meterId', meter.getMeter);
  app.get('/meter', meter.getMeters);
  app.put('/meter/:meterId', meter.updateMeter);
}
```

```

/**
 * Reader Router
 */
app.post('/reader', reader.addReader);
app.get('/reader/:readerId', reader.getReader);
app.get('/reader', reader.getReaders);

/**
 * Reader_Meter Routes
 */
app.post('/reader_meter', reader_meter.addReaderMeter);
app.get('/reader_meter/:readerId', reader_meter.getReaderMeter);
app.get('/reader_meter', reader_meter.getReaderMeters);

/**
 * Reading Routes
 */
app.post('/reading', reading.addReading);
app.get('/reading/:meterId', reading.getMeterReadings);
app.get('/reading', reading.getReadings);
app.put('/reading/:readingId', reading.updateReading);
app.delete('/reading/:readingId', reading.deleteReading);

/**
 * Price Routes
 */
app.post('/price', price.addPrice);
app.get('/price/:quantityName', price.getPrice);

/**
 * Bill Routes
 */
app.post('/bill', bill.addBill);
app.get('/bill/:meterId', bill.getMeterBills);
};

```

3.3 Interface Design

The next step that had to be performed to start the implementation of the web application was the Interface Design. A particular focus was given to this process because, the web application involves a lot of user interaction. The administrators need to access the system on daily basis, they need to perform certain operations in repetition. This ultimately requires the system to possess such properties as ease of use, simplicity. Efforts were made to make the user experience smooth and fluid. It was intended that the experience should be as less stressful as possible. The tasks were made as automated as possible, to reduce user workload and interaction. The main aim was to increase the productivity of user.

3.3.1 Choosing color scheme

The first decision that had to be made was the color scheme to be adopted. A simple color scheme was adopted with two complementary colors. These were the color blue and yellow.

The blue color was chosen because it possessed the following properties

- It creates a feeling of calmness, which will reduce stress on users.
- It promotes productivity, which will help increase work efficiency for users.
- It is considered a trustworthy and dependable color, hence employees will trust the system, and would be more willing to adopt it.

The yellow color was chosen because it possessed the following properties

- It stimulates the mental process and the nervous system, which will make tasks easier for users.
- It activates the memory, hence users would get used to the system quickly.
- It is the happiest color in the spectrum, which will make user experience enjoyable.

The text color was mostly black, and red color text was used to indicate errors.

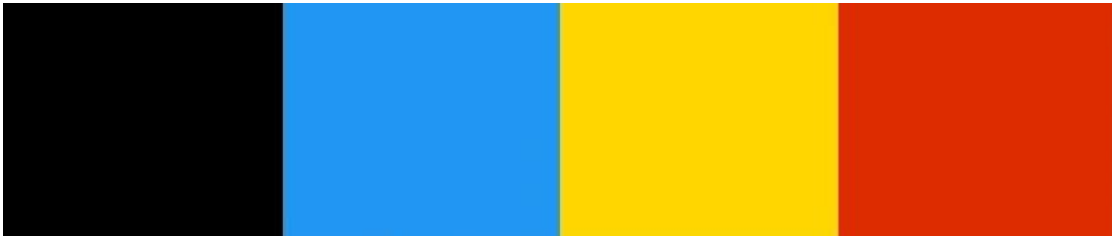


Figure 3.2 Color Scheme

3.3.2 Choosing a front-end framework

After choosing a suitable color scheme, the web app mockups had to be developed. After making some paper based mockups of the web app, it was decided to start developing the web interface of the most common features to speed up the development of the application. For this purpose a front-end framework had to be chosen.

There were several choices out there most famous of them being Twitter’s Bootstrap and the Foundation framework. Bootstrap was the initial choice. However, when it was decided that the Mobile App is going to be developed for Android, this decision was changed. A new design was introduced by Google for Android OS in 2014. It is called the Material Design. It is now being incorporated in all Google Products across different devices. This design is equally applicable for web, tablet and mobile applications. It gives a unique harmony to a product by introducing consistency and familiarity to the design across different devices. ^[3]

Now a front-end framework for the Material Design had to be discovered. There were several choices in this regard. A framework with wide compatibility and easy usage had to be chosen. A thorough search was carried out on GitHub for free Material Design frameworks. In the end we chose a framework called “Materielize”. It is quite new, but popular. It is implemented using basic CSS and Javascript technologies, which are quite easy to use. They can be used instantly by including the relevant files in HTML pages. Choosing this framework also helped set the typography for our application. The

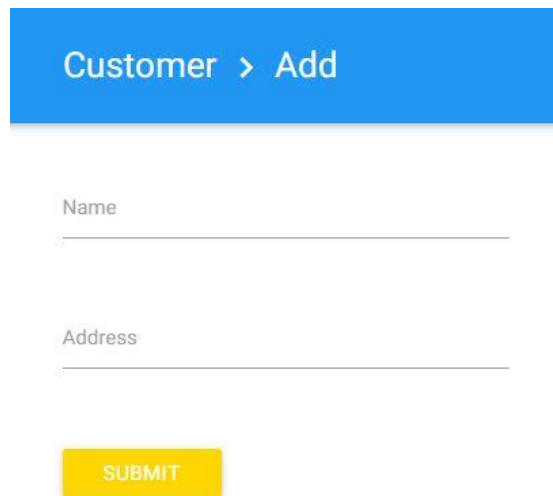
framework uses the Material Design typography for the text, which are a set of fonts by the name of “Roboto”.

3.3.3 Embracing the Material Design

Since Material Design was chosen for the interface design, efforts were made to ensure strict compliance with its main principles. For this purpose, the relevant sections of the Material Design Specification were read thoroughly to understand the essence of this design. The following paragraphs describes how some of the guidelines were used while designing the web application.

Choosing Button Style

The material design suggest using Raised Buttons (Figure 3.3) when layers are less, and it suggests using Ink Buttons (Figure 3.4) in dialogs to prevent too many layers of dimension.



Customer > Add

Name

Address

SUBMIT

Figure 3.3 Raised Button



Figure 3.4 Ink Button

Use Toasts for Feedback

Material Design suggests using toasts (Figure 3.5) for lightweight feedback about an operation. They automatically disappear after a timeout.

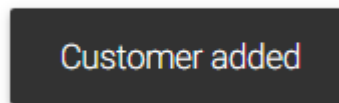


Figure 3.5 Toast

Use Dialogs for a Specific Task

Material Design suggest that dialogs (Figure 3.6) inform users about critical information, require users to make decisions, or encapsulate multiple tasks within a discrete process. The dialogs were used in the web application to allow admin to verify a particular reading in a data table.

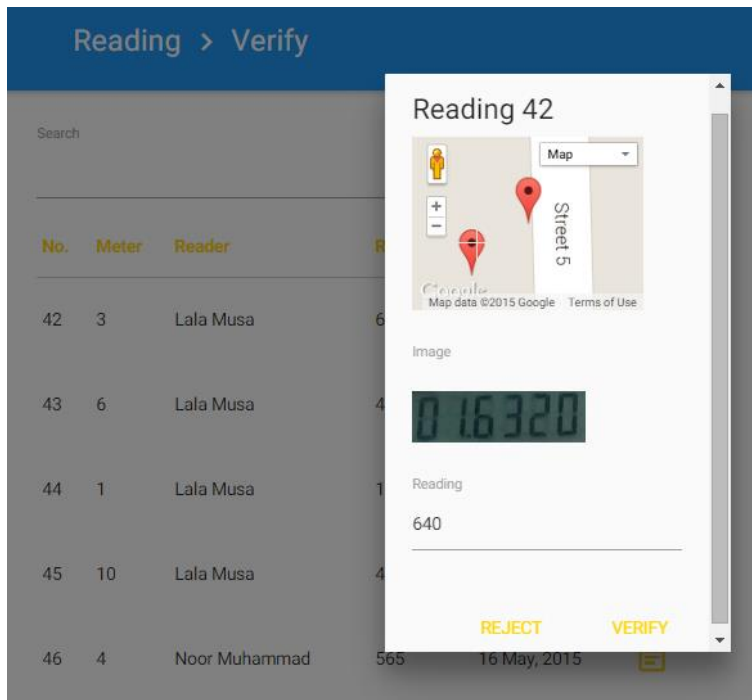


Figure 3.6 Dialog

Datatables for raw data

Material design says data tables (Figure 3.7) are used to present raw data sets, and usually appear in desktop enterprise products. They were used to allow admin to assign meters to readers.

No.	Address	<input type="checkbox"/>
1	G-157, Mansoor, Lahore	<input type="checkbox"/>
2	M-117, Jinnah Town, Sadiqabad	<input checked="" type="checkbox"/>
3	H-424, Bahria Town, Lahore	<input type="checkbox"/>

Figure 3.7 Data table

Tabs for grouped content

Material design states that tabs (Figure 3.8) make it easy to explore and switch between different views or functional aspects of an app or to browse categorized data sets. They were used to display bills for different meters belonging to a customer.

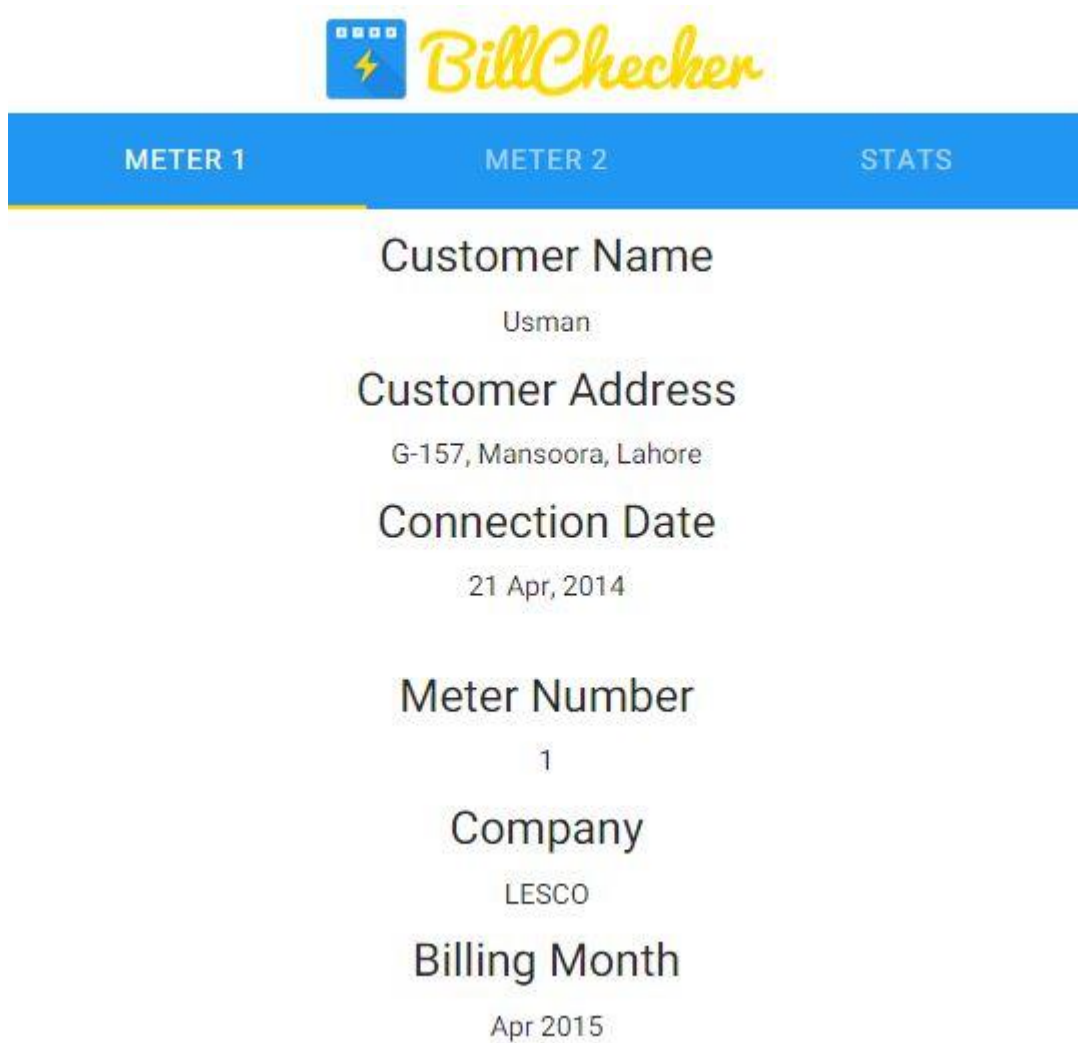


Figure 3.8 Tabs

Use Design Patterns

Some of the design patterns mentioned in the Material Design specification were also followed. These include Side Navigation Pattern (Figure 3.9) suitable for apps with large number of top-level views. The Error Handling Pattern (Figure 3.10) which states that when error occurs clearly communicate what is happening and describe how a user can resolve it. The Search Pattern (Figure 3.11) which states that search experience can be made significantly more gratifying by including some enhancements such as offering auto-completed search suggestions that match actual results in your application data.

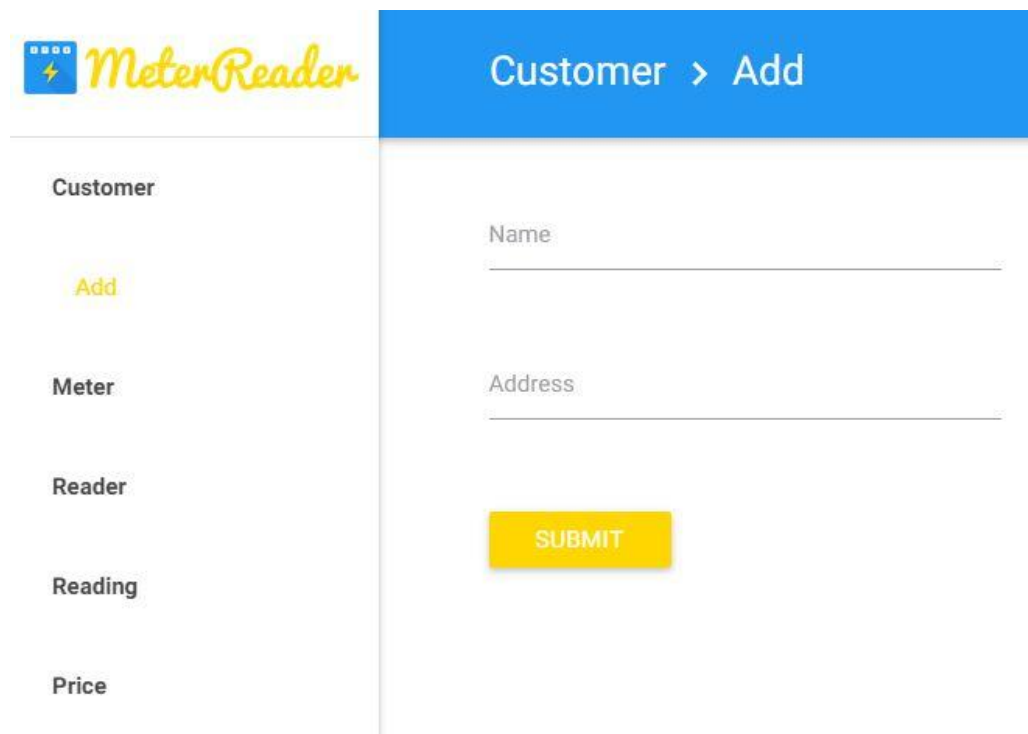


Figure 3.9 Side Navigation Pattern

Reader > Add

Name
Name is required

Username
lala_musa
Someone already has that username

Password
...
Password must be at least 6 characters

SUBMIT

Figure 3.10 Error Handling Pattern

Meter > Assign

Reader

a|

Abdur Rehman	<input type="checkbox"/>
Lala Musa	<input type="checkbox"/>
Noor Muhammad	<input type="checkbox"/>
Mustafa Kamal	<input type="checkbox"/>

1 G-157, Mansoor, Lanore

2 M-117, Jinnah Town, Sadiqabad

Figure 3.11 Search Pattern

3.3.4 Challenges

This section describes some of the challenges that were encountered when implementing the interface design in the web application.

The first challenge was adopting the “Materialize” framework to support the color scheme of the project. Sometimes these color schemes could be easily applied for example in case of text, buttons and navbar, only a class for the color had to be added for applying the color. However, in other cases the internal CSS properties of components had to be modified to adapt them to the color scheme of project. This had to be done for input field focus lines, date picker dialog, select input options and tabs. For this purpose a “custom.css” file was loaded after loading the materialize framework to override the properties of some components. The chrome element inspector tool proved useful for these scenarios. It helped change element properties in real-time and observe the results.

Another challenge was that the framework had no support for displaying errors in the fashion suggested by the Material Design. The only solution was to modify the CSS classes of form validation framework to achieve this.

Similarly the jquery autocomplete plugin also had a completely different design from the web application of the project. It was modified to replicate the design of select fields. This again required trial and error approach using the chrome inspector tool.

3.4 Choosing Technologies

The last step before starting the implementation of the web application was deciding which technologies to use. The previous section has already described the front-end framework “Materialize” used for developing the Web Application.

After that a database management system had to be selected. For this purpose, MySQL DBMS was chosen. This was because of its ease of setup, and its community version was free. It had wide compatibility, and good user support. Moreover, it was also selected due to past experience with this technology.

Now all that remained was to choose the Server-Side Technology. There were several choices JAVA, PHP, Node.js and Python. In the end it was decided to use Node.js for server side development. The main rationale behind using Node.js was its asynchronous nature. It helps prevent interface blocking, and increases performance. Moreover, past experience had proved it to be quite easy for development. Ease of development, meant more time to focus on optimizations and refactoring. ^[4]

3.5 Frameworks and Node.js Modules

The web application needed to make use of frameworks to implement some of its features. Even though one could implement them himself, but that would take time and a lot of thinking. There are already well built and well tested frameworks for almost about anything one requires to do in web applications. Therefore, instead of reinventing the wheel, it was decided to put these technologies to the use. The following sections state some of the complex operations of the web application that required the help of a framework to implement.

3.5.1 DOM Manipulation

DOM manipulation is one of the most common tasks for any web application. For this purpose jQuery was used. It is one of the most popular and powerful framework out there for DOM manipulation. Moreover, it also has an extensive list of functions that are used for Ajax Operations.

3.5.2 Form Validation

Most of the operations of the application incorporate some sort of forms. These forms had to be validated to notify users about invalid input. For this purpose a validation framework called “Parsley.js” was used. It was one of the most popular Javascript based validation framework on GitHub. It uses “data-parsley-” attributes on html elements to perform validations. For example the following code uses html5 minlength and required attributes to display errors using parsley

```
<input id="password" type="password" minlength="6"
  data-parsley-minlength-message="Password must be at least 6 characters"
  data-parsley-required-message="Password is required" required>
```

Figure 3.12 Password Validation

Custom Validators (Figure 3.13) had to be implemented for some special cases. For example checking if reader does not exist.

```
<input id="reader" type="text"
  data-parsley-known="known"
  data-parsley-trigger="change"
  data-parsley-required-message="Reader is required" required>

window.ParsleyValidator.addValidator('known', function (value, requirement) {
  if(findReader(value))
    return true;
  else
    return false;
}, 32).addMessage('en', 'known', 'Unknown reader');
```

Figure 3.13 Custom Validator

3.5.3 Autocomplete Input Fields

Some of the features of the web application required users to select a value from a large list of values. For example choosing the customer to whom the meter is being registered. This could be done using select input fields however, they would become too large as the number of customer increased. So, a better solution had to be provided. This led to the inclusion of autocomplete input fields in web application. They only displayed the values matching the user search, this greatly enhanced the user experience.

For this purpose a framework “jquery-autocomplete” was used. The reason for this choice was its ease of use and UI customizability. The following properties have to be defined for autocomplete

source: URL to the server or a local object

customLabel: The name of object's property which will be used as a label (displayed in autocomplete dropdown)

customValue: The name of object's property which will be used as a value (displayed in input field after a option is selected from dropdown)

The example of customer autoCompleter (Figure 3.14) is shown below

```
$('#customer').autocomplete({source: customerData, customLabel: "name", customValue: "name"});
```



Figure 3.14 Customer Autocompleter

3.5.4 Table Search and Pagination

Another complex task that had to be performed in the web application was table search and pagination. The tables contained a lot of data so they had to be displayed page by page, showing a specific number of rows at a time. Moreover user sometimes had to find a specific row, hence a search facility had to be provided for. For this purpose we had choices of two popular frameworks, DataTables.js and List.js. DataTables.js had not much support for Material Design UI. On the other hand List.js just performed the

search and pagination operations, it had nothing to do with UI. That's why it was the better framework for our web application.

The List.js was used to paginate the meters table. It also made the columns no. and address of meter table searchable. It required a class="list" attribute to be defined on <tbody> tag of the table. Moreover the id of the div that contained the search input and the table had to be provided to List constructor for initialization. An example is shown below

```
var options = {  
  valueNames: [ 'no', 'address' ],  
  page: 10,  
  plugins: [  
    ListPagination({})  
  ]  
};  
  
var meterList = new List('meters', options);
```



Figure 3.15 Table Search & Pagination

3.5.5 Charts for Customer Bills

The charts had to be displayed on Customer Bills to show monthly unit consumption of customers (Figure 3.16). For this purpose there were several choices such as Chartist.js, Morris.js, Google Charts. Again the one that was consistent with the

look and feel of our website was selected that is Google Charts. The Google Charts had been recently updated to match the Material Design specifications. The DataTable for chart had to be defined along with the properties of the chart. The rest was done automatically by the charts plugin as shown.

```
var data = new google.visualization.DataTable();
data.addColumn('string', 'Month');
data.addColumn('number', 'Units');

for(var i = 0; i < dates.length; i++)
    data.addRow([[dates[i], parseInt(units[i])]]);

var options = {
    chart: {
        title: 'Units Consumed',
        subtitle: dates[0] + ' to ' + dates[dates.length - 1]
    },
    colors: ["#ffd600"],
    width: 700,
    height: 500
};

var chart = new google.charts.Line(document.getElementById('chart' + index));

chart.draw(data, options);
```

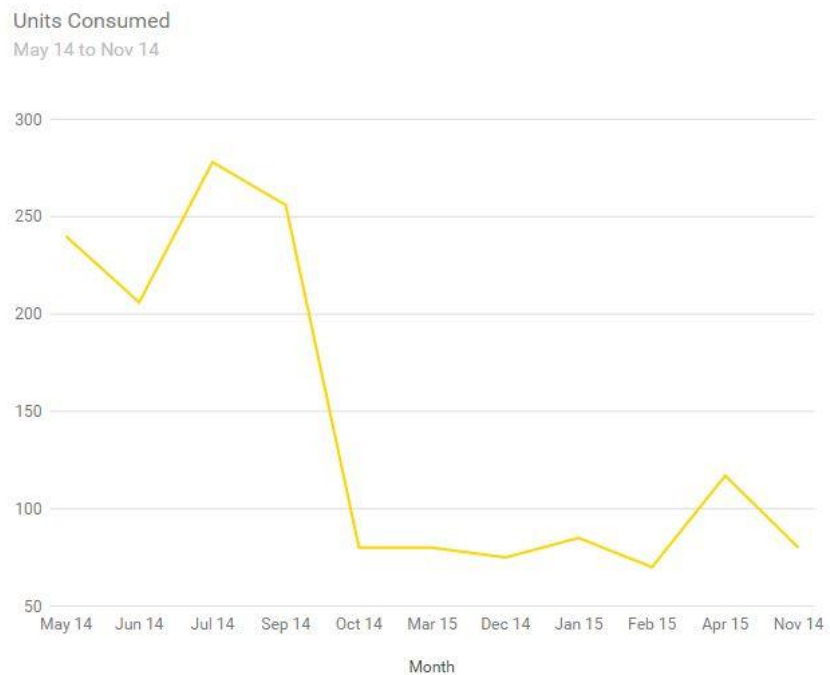


Figure 3.16 Google Charts

3.5.6 Server Side Templating

A server side templating engine was used for Node.js. It made the task of building up the pages very easy and efficient. All html pages were generated with complete data so they could be rendered instantaneously at the client side. No further requests had to be made by the client to server for fetching data. The name of templating engine used was Swig. It was chosen due to its easy syntax and filter support. It had constructs for if statements and for loops. It had filters for date formats and array manipulation. It could be used to create unique element ids. Figure 3.17 shows how Swig was used to construct past billing info table for customer billing html page.

```
<table id="table{{ loop.index0 }}" class="col s12 m12 l3">
  <thead>
    <tr>
      <th class="yellow-text text-accent-4">Month</th>
      <th class="yellow-text text-accent-4">Units</th>
      <th class="yellow-text text-accent-4">Bill</th>
    </tr>
  </thead>
  <tbody>
    {% for bill in meter.bills|reverse %}
    <tr>
      <td class="date">{{ bill.date|date('M y') }}</td>
      <td class="units">{{ bill.units }}</td>
      <td>{{ bill.bill }}</td>
    </tr>
    {% endfor %}
  </tbody>
</table>
```

Figure 3.17 Table Template

3.5.7 Server Side Async Handling

Another complex problem was encountered when building server Web API, that was the execution of sequential queries. The asynchronous nature of Node.js, allowed queries to be executed in a non-sequential manner. However, sometimes the result of one query was required to execute the next query. In these situations, queries had to be forced to execute in sequence. For this purpose, “async” module of Node.js was used. Its `async.series()` function enabled a set of callback functions to execute in sequence. For example bill calculation, required first to fetch the latest prices and then using them the bill was calculated, as shown below

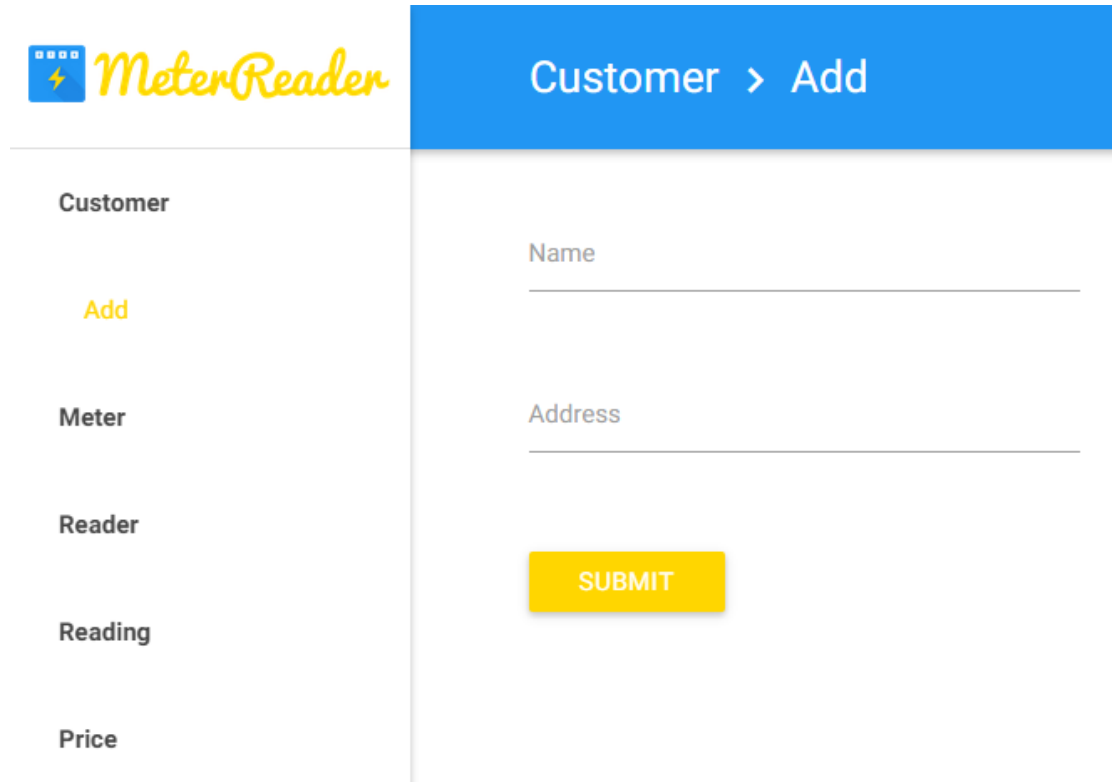
```
calculateBill:function(req,res){
  connection.getConnection(function (error, connection) {
    if (error)
      console.error(error);
    else {
      async.series([
        function(callback){
          connection.query('SELECT price FROM price WHERE quantity = ? ORDER BY date DESC', [ 'unit' ],
            function (error, data) {
              if (error)
                callback(error);
              else
                callback(null, data[0].price);
            });
        },
        function(callback){
          connection.query('SELECT price FROM price WHERE quantity = ? ORDER BY date DESC', [ 'tax' ],
            function (error, data) {
              if (error)
                callback(error);
              else
                callback(null, data[0].price);
            });
        }
      ],
      function(err, results){
        var unitPrice = results[0];
        var taxPrice = results[1];
        var units = req.body.value - req.body.last_reading;
        var bill = units * unitPrice + taxPrice;
        var insertData = {
          meter_id: req.body.meter_id,
          reading_id: req.params.readingId,
          reading: req.body.value,
          units: units,
          bill: bill
        };
        connection.query("INSERT INTO bill SET ?", insertData, function (error, data) {
          if (error)
            console.error(error);
          else
            resp.send({result: "success"});
        });
        connection.release();
      });
    }
  });
}
```

Figure 3.18 Using Async for Bill Calculation

3.6 Feature Implementations

The following sections briefly describe the high level features of the web application. Any unique features of the implementation is mentioned. The challenges encountered and how they were resolved is explained.

3.6.1 Allow admin to add customers



The screenshot shows the 'Add Customer' form in the MeterReader application. The application logo 'MeterReader' is visible in the top left corner. The page title is 'Customer > Add'. The form includes a sidebar with navigation options: Customer, Add (highlighted in yellow), Meter, Reader, Reading, and Price. The main form area contains two input fields: 'Name' and 'Address', both with horizontal lines below them. A yellow 'SUBMIT' button is located below the 'Address' field.

Figure 3.19 Add customer

The admin enters customer name and address, and presses submit. All fields are required. Errors shown if empty fields present. Toast for customer added shown on success. The customer is assigned a unique 'id' automatically using auto-increment id column. The implementation on server side involves a simple insert query to record customer details in database.

3.6.2 Allow admin to add meters

The screenshot shows the 'Meter > Add' page in the MeterReader application. On the left is a navigation menu with options: Customer, Meter, Add (highlighted), Assign, Reader, Reading, and Price. The main form contains the following fields:

- Customer: A text input field with a dropdown arrow.
- Address: A text input field.
- Type: A dropdown menu with the text 'Choose your option'.
- Utility: A dropdown menu with the text 'Choose your option'.
- Company: A dropdown menu with the text 'Choose your option'.
- Precision: A dropdown menu with the text 'Choose your option'.
- Search: A text input field.
- Latitude: A text input field.
- Longitude: A text input field.
- Date: A text input field.

Below the form is a Google Map showing a location in NUST. A red pin is placed on the map. At the bottom of the form is a yellow 'SUBMIT' button.

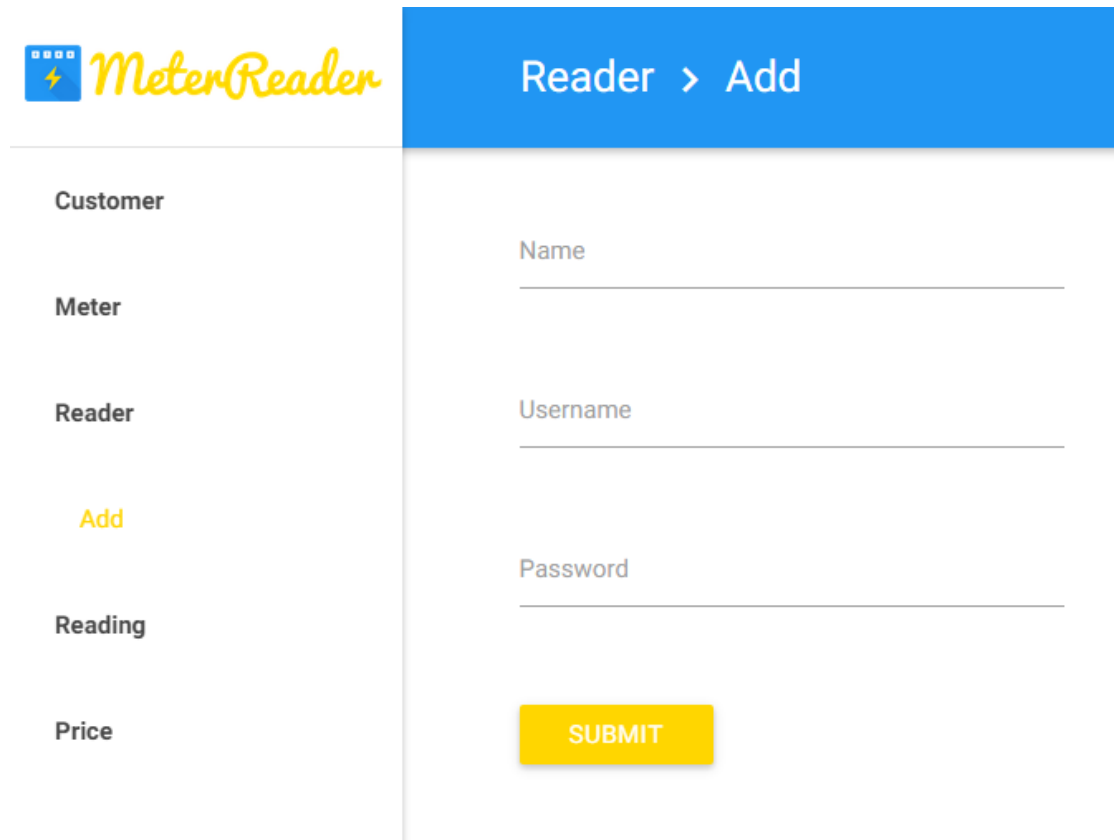
Figure 3.20 Add meter

The admin starts entering customer name, and is presented with autocomplete options. He chooses a customer from list. Upon choosing the address field defaults to customer address, but it can be changed if required. Admin selects other details about meter. The admin selects a location for meter. He enters a search query in search field. The map is centered to search location. The admin can then drag the map to change the location of meter. The latitude and longitudes are updated. The connection date of meter is selected from date picker. The information is submitted. Errors are shown if any field

missing. All fields are required. Toast for meter added shown on success. The meter is assigned a unique 'id' automatically using auto-increment id column. The implementation on server side involves a simple insert query to record meter details in database.

A custom parsley validator had to be implemented to check if the customer to whom meter is being registered, exists or not. An autocomplete plugin had to be used. The decision to use autocomplete was due to the fact that customers were far too many to be displayed in a simple select list. An autocomplete field was the necessary for better user experience. There were some problems with implementing the error UI for select fields. This was resolved by changing css properties of select fields using chrome inspector. After trial and error the relevant properties that needed to be changed were discovered, and CSS overrides for them were included in a "custom.css" file. Similarly the default UI for autocomplete plugin also had to be changed to match that of select fields. This was done to ensure consistency and coherence in user interface. This was again done by replicating properties of select fields to the properties of autocomplete fields.

3.6.3 Allow admin to add readers



The screenshot shows the MeterReader application interface. On the left is a sidebar menu with the following items: Customer, Meter, Reader, Add (highlighted in yellow), Reading, and Price. The top right of the page has a blue header with the text 'Reader > Add'. The main content area contains a form with three input fields: 'Name', 'Username', and 'Password'. Below these fields is a yellow 'SUBMIT' button.

Figure 3.21 Add reader

The admin enters reader's name, username, password and presses submit. All fields are required. Errors shown if empty fields present. Toast for reader added shown on success. The reader is assigned a unique 'id' automatically using auto-increment id column. The implementation on server side involves a simple insert query to record reader details in database.

A custom parsley validator had to be implemented to check if the username already exists.

3.6.4 Allow admin to assign meters

MeterReader Meter > Assign

Customer

Meter

Add

Assign

Reader

Reading

Price

Reader

Search

No.	Address	<input type="checkbox"/>
1	G-157, Mansoor, Lahore	<input type="checkbox"/>
2	M-117, Jinnah Town, Sadiqabad	<input type="checkbox"/>
3	H-424, Bahria Town, Lahore	<input type="checkbox"/>
4	M-223, Jinnah Town, Sadiqabad	<input type="checkbox"/>
5	F-1041, Satellite Town, Rawalpindi	<input type="checkbox"/>
6	G-233, Mansoor, Lahore	<input type="checkbox"/>
7	K-322, Muslim Town, Rawalpindi	<input type="checkbox"/>
8	K-322, Muslim Town, Rawalpindi	<input type="checkbox"/>
9	K-322, Muslim Town, Rawalpindi	<input type="checkbox"/>
10	P-181, Johar Town, Lahore	<input type="checkbox"/>

1 2

SUBMIT

Figure 3.22 Assign meter

The admin starts entering reader name, and is presented with autocomplete options. He chooses a reader from list. He then performs a search on the table to retrieve meters of a particular area. He assigns these meters to reader by selecting relevant checkboxes and pressing submit. These meters now become unavailable for assignment and their rows are removed from the table. A toast is shown, mentioning how many meters were assigned. The implementation on server side involves a check. This check is that when presenting the admin with meters for assignment, it only shows those meters which are not yet assigned to any meter reader. On submit the list of assigned meters are compiled in an array and sent to server. A simple insert query, inserts these assignment records in the database.

List.js proved very helpful in this feature implementation. The search, pagination and row removal were all performed using List.js. A custom validator was implemented to check whether the reader entered exists or not. JQuery was used to set the values of row checkboxes based on the value of main header checkbox.

3.6.5 Allow admin to verify readings

Customer

Meter

Reader

Reading

Verify

Price

Reading > Verify

Search

No.	Meter	Reader	Reading	Date	Verify
42	3	Lala Musa	640	08 May, 2015	
43	6	Lala Musa	482	09 May, 2015	
44	1	Lala Musa	1720	15 May, 2015	
45	10	Lala Musa	453.17	16 May, 2015	
46	4	Noor Muhammad	565	16 May, 2015	
48	2	Noor Muhammad	1134	20 May, 2015	
47	5	Mustafa Kamal	539	19 May, 2015	
49	11	Mustafa Kamal	258	21 May, 2015	
39	7	Abdur Rehman	274	08 May, 2015	
40	8	Abdur Rehman	249	08 May, 2015	

1 2

Figure 3.23 Verify reading

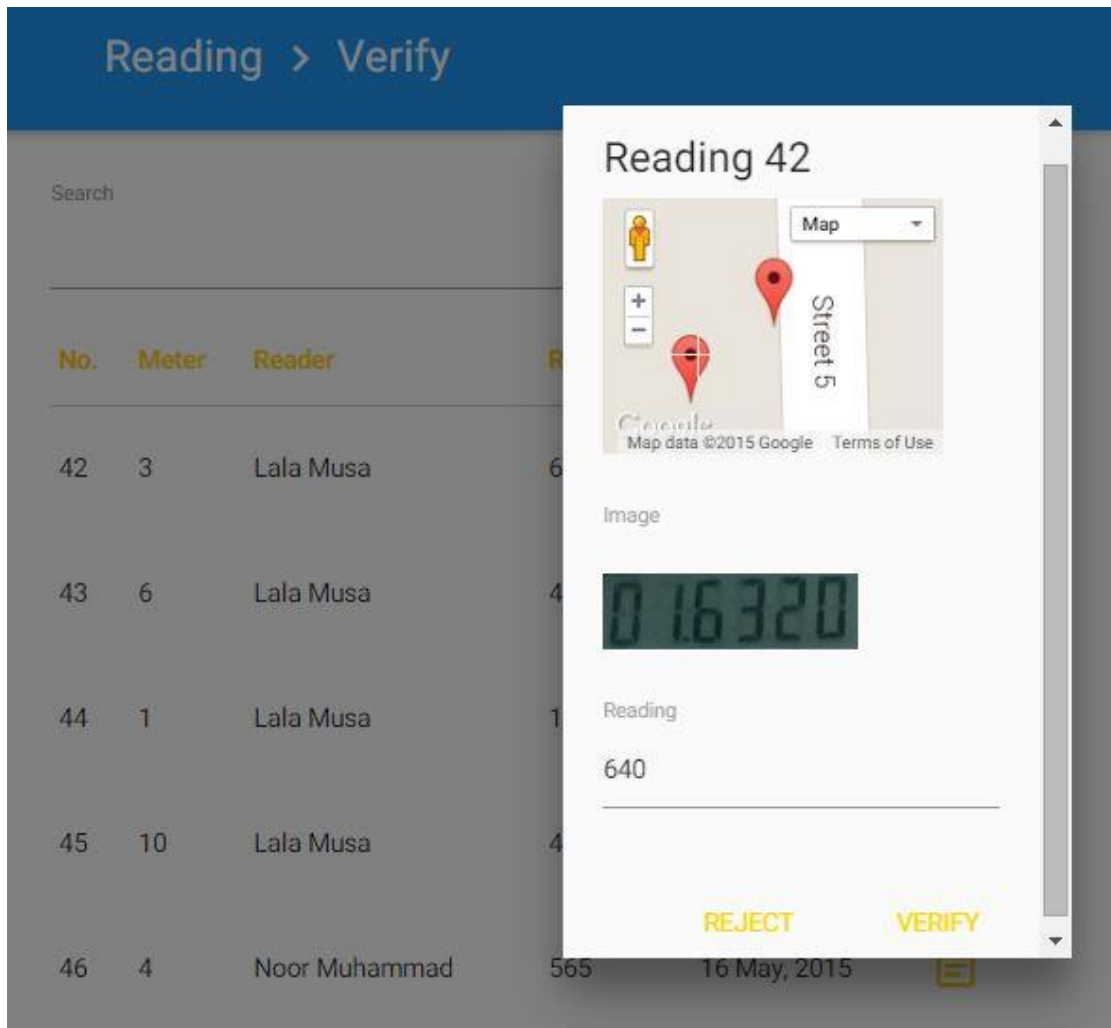


Figure 3.24 Verify Dialog

The admin either performs a search to find a specific reading, or he just clicks on verify button of any listed reading. A dialog box (Figure 3.24) opens up. The admin now starts verifying the reading. He compares the location of meter with the location of reading to check if the reader actually visited the meter location to take the reading. He matches the Image with the reading and changes the reading if it is incorrect. Here a check is performed to prevent admin from entering a reading smaller than last verified meter reading. After doing all this, the admin clicks the verify button to verify the reading. A tick mark replaces the verify icon, and a toast is shown saying reading verified. The admin could also have rejected the reading, this would then delete this

reading from record, and meter reader would have to take this reading again. In that case the verify icon would be replaced with a cross mark and a toast is shown saying reading rejected.

The server side implementation of this feature is quite complex. This is because a lot of operations have to be performed when a reading is verified. These include

- Update the reading value (if changed) and update its status to verified
- Update the last reading date field in meter table to current reading date and update the last reading field to current reading
- Retrieve the latest price of unit
- Retrieve the latest price of tax
- Use the current reading and last reading to calculate units consumed.
- Multiply units consumed with price of unit and add tax to the result
- Insert the calculated customer bill in the bills table

Figure 3.25 shows the server side code that is executed on reading verification. This is probably the most complex piece of code written for the server side.

The operations performed on reading rejection is quite simple. Only the current reading has to be deleted and no update has to be made to meter table because the reading has to be taken again.

Another strange problem was encountered while implementing this feature. The problem was that the map shown in verification dialog was not being centered to the meter location marker. It was always being positioned at the top left corner of the map. Every time the dialog opened, one had to drag the map to get to meter location. It was found out that a lot of other people had also faced this problem. It occurred when the map existed in a hidden div. The map could not calculate its dimensions and hence could not center itself on load. The solution was kind of a workaround. A `map.resize()` function had to be called whenever dialog was opened to center map correctly.

```

async.series([
  function(callback){
    connection.query('UPDATE reading SET ? WHERE id = ?',
      [{ value: req.body.value, status: req.body.status }, req.params.readingId],
      function (error, data) {
        if (error)
          callback(error);
        else {
          res.send({result : "success"});
          callback(null, null);
        }
      });
  },
  function(callback){
    connection.query('UPDATE meter SET ? WHERE id = ?',
      [{ last_reading: req.body.value, last_reading_date: req.body.last_reading_date }, req.body.meter_id],
      function (error, data) {
        if (error)
          callback(error);
        else
          callback(null, null);
      });
  },
  function(callback){
    connection.query('SELECT price FROM price WHERE quantity = ? ORDER BY date DESC', [ 'unit' ],
      function (error, data) {
        if (error)
          callback(error);
        else
          callback(null, data[0].price);
      });
  },
  function(callback){
    connection.query('SELECT price FROM price WHERE quantity = ? ORDER BY date DESC', [ 'tax' ],
      function (error, data) {
        if (error)
          callback(error);
        else
          callback(null, data[0].price);
      });
  }
],
function(err, results){
  var units = req.body.value - req.body.last_reading;
  var bill = units * results[2] + results[3];
  var insertData = {
    meter_id: req.body.meter_id,
    reading_id: req.params.readingId,
    reading: req.body.value,
    units: units,
    bill: bill
  };
  connection.query("INSERT INTO bill SET ?", insertData, function (error, data) {
    if (error)
      console.error(error);
    else
      console.log("Data Entered: " + JSON.stringify(data));
  });
  connection.release();
});

```

Figure 3.25 Reading Verification Code


3.6.6 Allow admin to update prices

The screenshot displays the 'MeterReader' application interface. On the left is a sidebar with navigation links: Customer, Meter, Reader, Reading, Price, and Update (highlighted in yellow). The main content area has a blue header with the breadcrumb 'Price > Update'. Below the header, there are three input fields: 'Quantity' (a dropdown menu with the text 'Choose your option'), 'Price' (a text input field), and a yellow 'SUBMIT' button.

Figure 3.26 Update price

The admin selects a quantity either price or tax, then enters the new price and presses submit. Toast for price updated shown on success. All fields are required. Errors are shown if empty fields present. The implementation on server side involves a simple insert query to record new price details in database. The price is recorded along with current date.

3.6.7 Allow customer to view current and past billing information




METER 1	METER 2	STATS
Customer Name Kashif	Customer Address M-223, Jinnah Town, Sadiqabad	Connection Date 12 Sep, 2014
Meter Number 2	Company PEPCO	Billing Month Apr 2015
Previous Reading 711	Present Reading 956	Units Consumed 245
Unit Price 10	Tax 0	Bill 2450
	Image 	

Figure 3.27 Customer's Current Bill

The customer enters the customer id and name at the bill checker page. The bill is shown only if both the customer id and name match a record. Bill for all the meters registered to customer are shown. The tab interface is used to separate the bill of each meter. A stats tab (Figure 3.28) is also shown, it contains customers past billing information (last 12 months only). A graph of units consumed for past 12 months is also shown. The min, max and avg units consumed is also displayed.

The image for current bills are accessed from img folder. The image exists in the "reading_23.jpg" format, here 23 is the reading id. The bill records have a reading id column which is used to retrieve the image.

A particular problem that was encountered while implementing this feature was displaying the graph. The graph was being shown in a hidden tab div. This prevented the graph from being displayed correctly, since it was unaware of its size. A workaround was used to solve this problem. First the stats tab was shown, so graph could be loaded, then after a small timeout (10 ms) the Meter 1 tab was selected programmatically. Since the customer expects to see the current bill, instead of stats. This enabled the graph to be displayed correctly.

METER 1

Min: 70 (Jan 15) Max: 245 (Apr 15) Avg: 136.57

Units Consumed
Oct 14 to Apr 15



Month	Units	Bill
Oct 14	160	1600
Nov 14	130	1300
Dec 14	110	1100
Jan 15	70	700
Feb 15	120	1200
Mar 15	121	1210
Apr 15	245	2450

METER 2

Min: 111 (Feb 15) Max: 177 (Apr 15) Avg: 144.67

Units Consumed
Feb 15 to Apr 15



Month	Units	Bill
Feb 15	111	1110
Mar 15	146	1460
Apr 15	177	1770

Figure 3.28 Customer's Past Bill

The server side code (Figure 3.30) was special in the sense that it used a special function of async module called the `async.map()` function. The `map` function iterates over an array one element at a time. It performs operations on the element. In this scenario, a `bill` property is added to each meter in the `meters` array. This code compiled all information about customer, his meters and the latest 13 bills of each meter (Figure

3.29). This information was used to render a view which was returned as HTML response page to customer.

```
▼ object {3}
  customer_name : Usama
  customer_address : P-181, Johar Town, Lahore
▼ meters [1]
  ▼ 0 {8}
    id : 10
    company : LESCO
    precision : 2
    lat : 31.458978
    long : 74.284651
    date : 2015-02-10T04:26:19.000Z
    last_reading : 317.54
▼ bills [2]
  ▼ 0 {5}
    id : 34
    reading : 317.54
    units : 173.78
    bill : 1737.8
    date : 2015-04-14T04:47:42.000Z
  ▼ 1 {5}
    id : 25
    reading : 143.76
    units : 143.76
    bill : 1437.6
    date : 2015-03-11T04:26:19.000Z
```

Figure 3.29 Customer Bill Data Structure

```

async.series([
  function(callback){
    connection.query('SELECT * FROM customer WHERE id = ?',
    [req.params.customerId],
    function (error, data) {
      if (error)
        callback(error);
      else {
        var customer_data = {customer_name: data[0].name,
                             customer_address: data[0].address};
        callback(null, customer_data);
      }
    }
  });
},
function(callback){
  connection.query('SELECT * FROM meter WHERE customer_id = ?',
  [req.params.customerId],
  function (error, data) {
    if (error)
      callback(error);
    else
      callback(null, data);
  });
}
],
function(error, results){
  async.map(results[1], function (meter, callback) {
    connection.query('select * FROM bill where meter_id = ? ORDER BY date DESC LIMIT 13'
    meter.id,
    function(error, bills) {
      meter.bills = bills;
      callback();
    });
  }, function(error) {
    if(error)
      console.log(error);
    else {
      var response = {customer_name: results[0].customer_name,
                     customer_address: results[0].customer_address,
                     meters: results[1]};
      res.render('layouts/billchecker', response);
    }
  });
  connection.release();
});

```

Figure 3.30 Bill Fetching Server Code

MOBILE APPLICATION

Mobile application plays a vital role in meter reading. It provides the interface through which meter readers interact with the system. It performs two important tasks. Firstly, it gives the meter readers a list of all meters, which he has to read. Secondly, it updates the web server with the reading taken by the meter reader.

This section describes the mobile application development methodology for the project. It describes the architecture and design of the mobile application. The various components of the mobile application. The tools, technologies and frameworks used for the development of the mobile application.

4.1 Choosing Technology

The first step was choosing technology for the development of mobile application. There were lot of choices that had to be made, such as Multi-platform vs. Native Development. Android Native Development was chosen because of its performance and support for OCR. Most of the OCR related related research is done in C languages and it was more feasible to integrate OCR with Android in comparison to Multi-Platform. Moreover performance gain in Android for OCR was better than Multi-Platform.

If we compare the Android to other major mobile application platforms like iOS or Windows Phone, Android is cheaper than iOS and it has more devices than Windows Phone.

4.2 Interface Design

Interface design first started with mockups. Adobe illustrator was used for designing the mockups. Google Material Design resources were used in designing the mockups. Material design was followed because now it is the standard for designing android applications. It helps bring uniformity to design across different

platforms. Following figures show the mockups of mobile application.

4.2.1 Login Screen

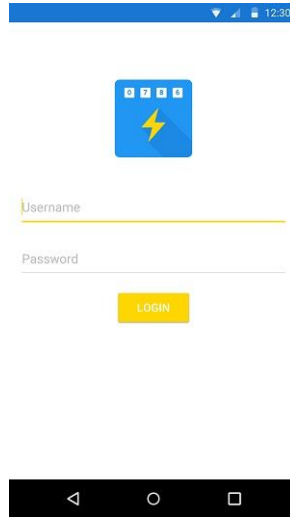


Figure 4.1 Login Screen

Each meter reader provides its username and password and login into the mobile application.

4.2.2 Home Screen



Figure 4.2 Home Screen

Home screen contains two tabs:

- Pending: Shows the list of pending meters that meter reader still has to read.
- Completed: Shows the list of meters that are already read by meter reader.

4.2.3 Meter Screen

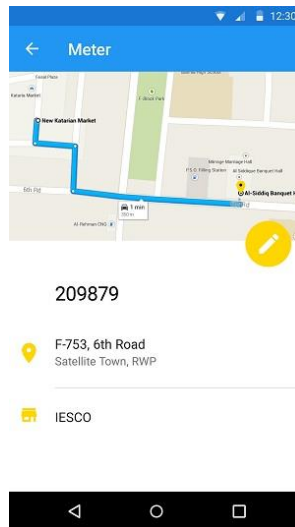


Figure 4.3 Meter Screen

Meter screen shows the location of meter using map, its meter number, its manufacturing company and additionally it also shows the address of location where meter is installed.

4.2.4 Reading Correct Screen

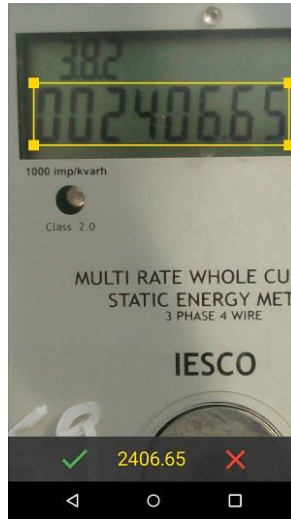


Figure 4.4 Reading Correct Screen

This screen is basically showing captured image of meter. At the bottom of the screen its showing reading of meter application has extracted from the image. On the left of that reading there is a green tick, as in this scenario reading is correct, so meter reader will go for the tick option.

4.2.5 Reading Incorrect Screen

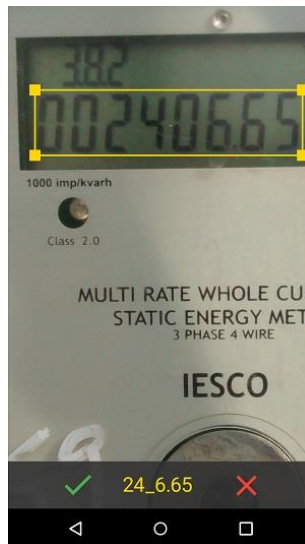


Figure 4.5 Reading Incorrect Screen

Again this screen is basically showing captured image of meter. At the bottom of the screen its showing reading that meter application has extracted from the image. On the right of that reading there is a red cross, as in this scenario reading is incorrect, so meter reader will go for the cross option.

4.2.6 Reading Correction Screen

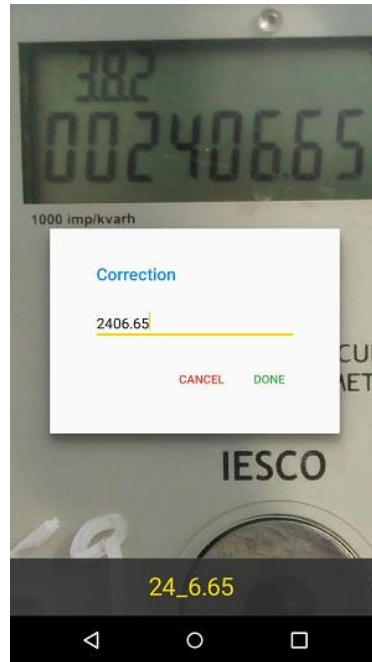


Figure 4.6 Reading Correction Screen

This screen shows a small widget for the correction of reading while displaying the image in background. This seemed the appropriate way to do it, reader doesn't have to move to other screen to see the exact reading. He can easily type the exact reading while seeing the actual reading from the image in background.

4.2.7 Reading Completed Screen



Figure 4.7 Completed Screen

This screen shows the meter readings that have been completed by the meter reader. The reader can select a reading from the list to get more information about the reading.

4.3 Database Design

The second step in development of mobile application was database design. This database will act as a local database for the mobile application, thus efforts were made to keep this database as simple as possible. This database should not take much space on the mobile, so that device may not run out of memory. This database had to store the most critical data of the whole system. As it saves the runtime data that will be created while taking the readings, it should be robust enough to save the critical data during an app crash. Following the minimalistic design principles only critical data will be saved in this database. For this purpose, the most basic operations that mobile application does while meter reading were identified, and resources were created to fulfill the data needs of these operations.

4.3.1 Identifying Tables

Keeping these valuable principles in mind, the main resources of the mobile application were identified. These resources would actually form the tables of the database. The database design went through quite a lot of iterations that were mainly for refactoring and optimizations. In the end following tables formed our database

- meter - stores information about meters
- reading - stores information about readings

4.3.2 Identifying Fields & Constraints

The next step was identifying the fields for tables and their constraints. This required quite a lot of thought. In the beginning, only the most relevant fields could be identified. It was only after the features of the mobile application became more clear, that fields for the tables were finalized. The following sections describe the fields for each table and rationale behind them. Primary keys are in **bold** and Foreign Keys are in *italic*. All fields have NOT NULL constraint unless stated otherwise. This is mainly because most of the fields are required for correct operation.

Meter Table

- **id** - unique identifier of meter
- address - the address of meter, because it can be different from customer address. A customer can have meters registered in his name, at different locations.
- utility - the utility of meter either electricity or gas. Helpful in generating bills customized according to utility of meter.
- company - the company of meter (IESCO, PEPCO) to be displayed on bill
- precision - the position of decimal point of meter readings. Helpful for positioning decimal point in reading, if not recognized by image processing.
- lat - the latitude of meter's GPS location. Matched with latitude of reading to verify that meter reader actually visited the meter location to take reading.

- long - the longitude of meter's GPS location. Matched with longitude of reading to verify that meter reader actually visited the meter location to take reading.

Reading Table

- **id** - unique identifier of reading
- lat - the latitude of meter's GPS location. Matched with latitude of reading to verify that meter reader actually visited the meter location to take reading.
- long - the longitude of meter's GPS location. Matched with longitude of reading to verify that meter reader actually visited the meter location to take reading.
- *meter_id* - id of the meter whose reading is been taken. It helps to determine which meter should be associated with this specific reading
- *reader_id* - id of the reader who has taken this reading. It helps to determine which reader has taken this specific reading
- image - image of the meter. This image is a proof of the reading. Actual reading's value should be same as being shown in this image.
- address - the address of the location where reading was taken. It should be equal to the address where meter was installed.

4.4 Implementing SYNC Feature

4.4.1 Introduction

SYNC Feature gets the newly assigned meters from the server and sends the data of read meters from the application. Basically when the reader presses the SYNC button in the application, it gets the new assigned meters to the reader by the admin and at the same time it update the server with the meters read by the meter reader.

This feature basically evolved from just uploading the new readings feature. Uploading the new read meters data was the common practice. Our unique feature was to get the new data, while uploading the data to server.

4.4.2 Challenge

There was a challenge that was faced while implementing this feature. When the SYNC button is pressed, two operations are performed. A GET request is sent to server to get new meters list that is to be read by the reader. A POST request is sent to server to upload the newly updated readings of meters.

Issue was that our get request was served before the post requests. Because POST request had more data to send than GET request gets from the server.

As the GET requests processes before the POST request, application will get the meters that are already read by the reader. Because POST hasn't been processed yet, server doesn't have the updated data and will send the same meters whenever the GET request come.

4.4.3 Solution

In order to solve this issue, whenever the SYNC button is pressed GET request wont be processed before the POST. Application makes sure when the POST request is done with uploading the new data to server; then GET request is sent to server.

4.5 Implementing Google Maps in Android Application

4.5.1 Introduction

Google Maps were used to basically help out reader to get the location where meter is applied. Google Maps API was used to implement Google maps into the application.

This feature not just navigates the reader to meter's location; but also acts as a check whether reader was actually at that location or not. Whenever reader gets to a location to read the meter. He presses read button, at that time application records reader's location co-ordinates and sends them along the reading and image of the meter.

4.5.2 Challenge

Actually Google Maps API default implementation is that application

periodically sends a request to the Google maps server to get co-ordinates of device's location. This consumed a lot of battery and some extra Internet bandwidth. Battery consumption is critical in this application, as reader has to use this phone for the whole day. Hence getting the co-ordinates periodically, is an operation, which uses GPS device, and it takes battery power.

4.5.3 Solution

In order to solve this issue, application was made to send to this request only once. Whenever the reader presses the read button this request was used as it callback function. It will get the co-ordinates only once, not periodically. While reader is taking the reading, application will get the co-ordinates from the server and send them as a proof. This solution actually improved the battery consumption of our mobile app.

4.6 Implementing OCR in Android Application

4.6.1 Introduction

OCR stands for optical character recognition. OCR is used to recognize characters in an image. So in our application it was used to extract reading form the image.

This feature was used to cut down the reader's hassle while he takes the reading. Application takes the image and extracts the reading out of it.

4.6.2 Challenge

OCR is a process that takes memory, processing and battery. Smartphone as compared to a computer lacks these resources. Moreover, application should not be taking much of these resources. Issue was to decide on which end OCR should be implemented server end or mobile application.

Sending the image to server, using OCR to extract the reading than sending that reading to server is a process that might save the battery life for mobile but it is making the most critical i.e. reading process on an internet connection. If server has to process

the image through OCR, application will be waiting for the server to respond. There are many issues that might halt the application, like what if Server goes down or Mobile Application loses Internet connection.

4.6.3 Solution

In order to minimize the internet and server dependency, OCR was implemented into the mobile application.

This solution is justified, since it takes very less of the resources. Considerable efforts were made to optimize the OCR onto mobile application.

Tesseract was used for OCR, this engine is known for its accuracy and the way it uses the resources. Thus implementing OCR with Tesseract improved battery consumption of our application.

IMAGE PROCESSING

The image processing was probably the most challenging part of this project. This was because there was a lack of prior knowledge or experience in this field.

The main objective of this portion was to automate the process of meter reading. It was meant to simplify the reading process for meter readers. It was there to remove errors from the reading process.

This section describes the image processing development methodology and implementation for the project. It describes the data set preparation for image processing. The different types of meters in the data set. The image processing techniques used to process the images and the rationale behind them. The tools, technologies and frameworks used for the development of the image processing solution. The challenges encountered while developing the image processing solution. The integration of image processing solution in the android application.

5.1 Data Set Preparation

The first step to start the development of an image processing solution was the preparation of a data set of meter images. Particular attention was given to collect meters of different types. Similar meters were avoided. This was done so that a comprehensive solution could be developed that could recognize readings form all types of meters.

It was discovered that meters were mainly of two categories, digital and mechanical. The mechanical meters were usually old, and most of the new meters were digital. Gas meters were of the mechanical type only, whereas Electricity meters were of both categories. There were several meter manufacturing companies such as Syed Bhais, PEL, KBK Electronic, Creative Electronics and MicroTech Industries. Figures 5.1 shows the different types of meters.

These meter images were then edited to crop out the display of meters. Since that is what would be fed into the image processing algorithm for processing. Figure 5.2 shows the cropped set of images.

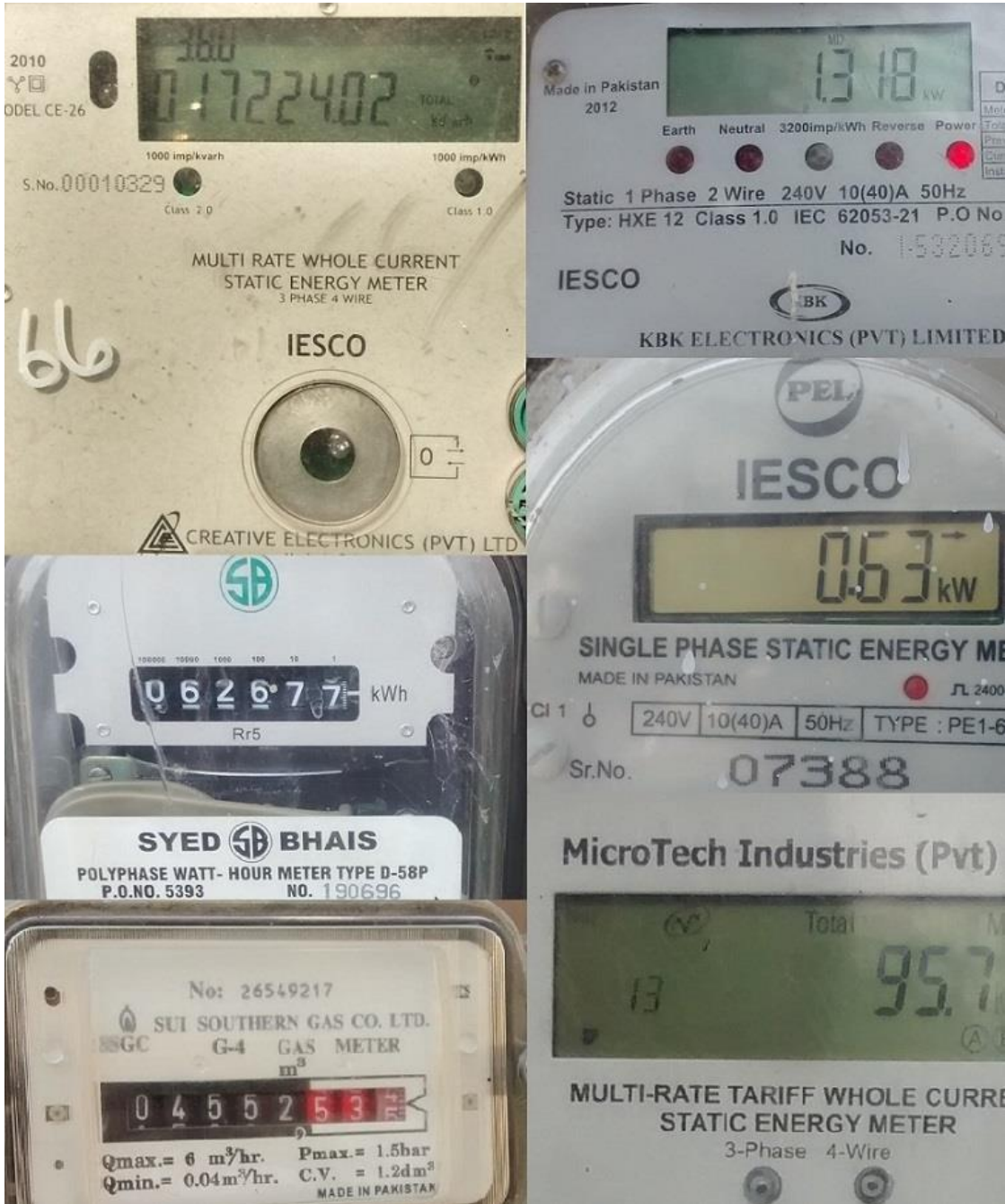


Figure 5.1 Different kinds of meter



Figure 5.2 Cropped Images

5.2 Testing Existing Solutions

The next step was searching for an existing solution that could fulfil the needs of this project. Digital Image Processing is a pretty advanced field, and a lot of work has already been done in this field. Therefore, instead of trying to create an Optical Character Recognition (OCR) solution from scratch, we searched for existing technologies and frameworks we could use to make our task easier.

5.2.1 Tesseract

The first framework to be discovered was Tesseract. It is widely regarded as the most accurate open source framework for OCR. It is now being sponsored by Google. The good thing about this framework is that it was recently ported to Android, and its API can be used to perform image recognition on Android. A sample Tesseract app by the name of “OCR Test” was available on Google Play Store. It performed real time image processing. It was used to perform tests on dataset of meter images but the results were not satisfactory (Figure 5.3).

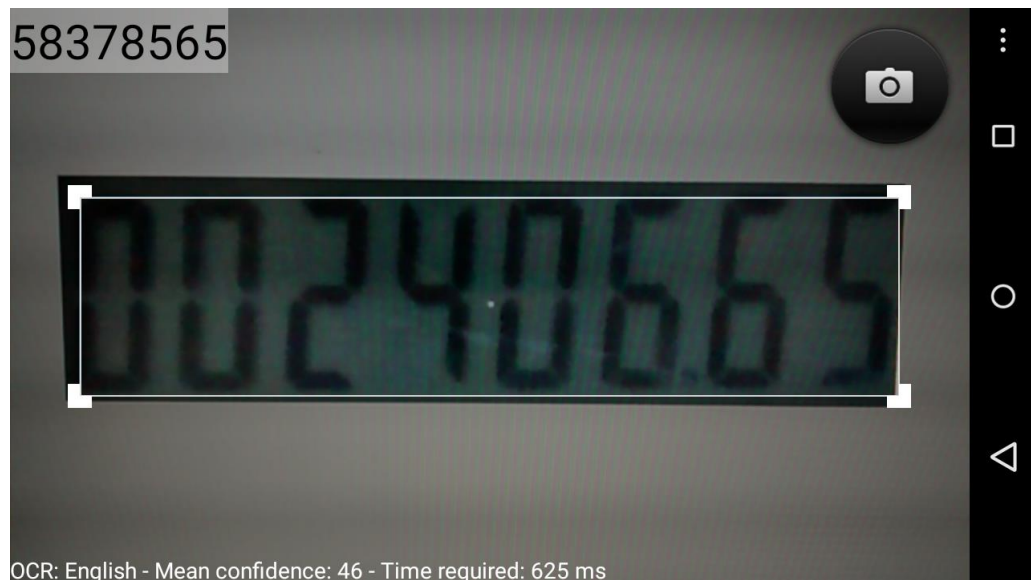


Figure 5.3 Tesseract OCR Test

It was later found out that the reason for unsatisfactory results was due to a limitation of Tesseract. It was originally designed to read clear connected fonts. The font being used on Meter Displays was 7-segment LCD font. The characters of this font were formed using 7 segments which were not connected to each other. That is why Tesseract did not perform well.

5.2.2 OpenCV

The next framework that was found was OpenCV. It is a library of programming functions mainly aimed at real time computer vision. It provides an extensive set of functions for image processing. Android is also among the supported platforms of OpenCV. Again a sample app was found on Google Play store for OpenCV OCR. The App is called “7-segment LCD Display Reader”. It is specifically designed to read 7-segment displays. This app is developed using OpenCV. The results of OCR for this app were quite good (Figure 5.4). However, they still had to be improved to meet the requirements of this project.

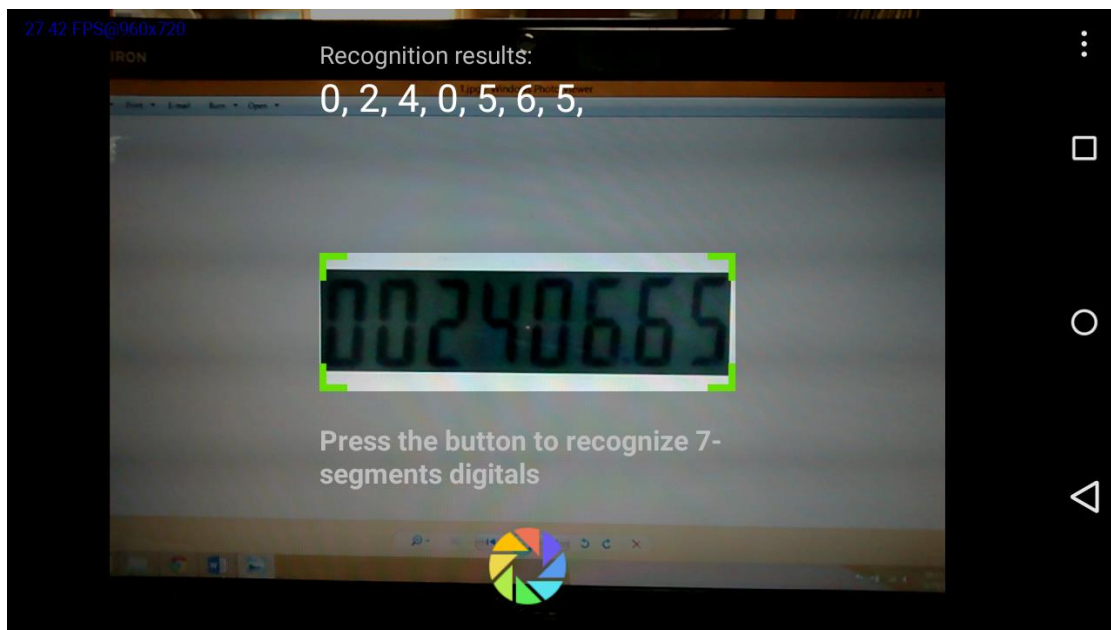


Figure 5.4 OpenCV OCR Test

5.2.3 SSOOCR Tool

Since the solution designed specifically for 7-segment OCR was performing better. Therefore another search was carried out to find open source solutions that were specifically designed for 7-segment OCR. This led to the discovery of Seven Segment Optical Character. It is specifically designed for recognizing text on 7-segment displays. However, officially it supports only the GNU Linux operating systems. Some tests were carried out on the meter images using this tool. The tool recognized very few readings correctly (Figure 5.5).^[7]



Figure 5.5 SSOOCR Test

5.2.4 Test Conclusion

After carrying out the above tests, it became clear that in order to develop a comprehensive and accurate OCR solution, a custom solution has to be implemented. This is mainly due to the diverse requirements of the project. Since OpenCV app gave the most accurate results among the three solutions that were tested. Hence, OpenCV was chosen for the development of the OCR solution of this project.

5.3 Image Preprocessing

During the course of this project it was discovered that preprocessing the images produced better results. This was first noticed when testing the OpenCV app. The OpenCV app first preprocessed the image and then fed it to its recognition algorithm. A Digital Image Processing course had to be taken to learn about various image processing techniques. This had to be done, so that appropriate choices could be made about which techniques to use.

Several techniques were used to preprocess the images. They are mentioned in the following paragraphs. Initially these image processing operations were performed on images using Matlab. The reason for using Matlab was its wonderful documentation. It was simple enough, so that a beginner could easily understand it. Later on, the code for Matlab was replicated in OpenCV, because there was no support for Matlab in Android, while OpenCV had quite good support. The experience gained from working in Matlab, had provided enough knowledge to implement the code in OpenCV. ^[5]

5.3.1 Binary Thresholding

The first step in preprocessing the images was binary thresholding. This technique was used to convert RGB images to BW images. In BW images each pixel was either black (0) or white (1). The software used for binary thresholding was initially an open source command line tool called ImageMagick. This was later replaced with Matlab due the extensive set of image processing functions available in Matlab. Figure 5.6 shows the results of binary thresholding operation on images using Matlab.

```
>> for c = 1:13
I = imread(strcat('C:\Users\Unnus\Desktop\Image Preprocessing\', num2str(c), '.jpg'));
I = rgb2gray(I);
bw = im2bw(I,0.35);
bw = imcomplement(bw);
imwrite(bw, strcat('C:\Users\Unnus\Desktop\Image Preprocessing\', num2str(c), '_bw.jpg'));
end
```



Figure 5.6 Binary Thresholding

As evident from the above figure, the binary thresholding operation is not performing very well on its own. It works well on some images while it is completely useless on others. A solutions needs to be developed that works equally well for all kinds of images.

Some other technique has to be applied to produce better results. Currently the images are firstly being converted to grayscale using “rgb2gray” function. Then “im2bw” function is used to convert them to binary images. The im2bw(I, level) converts the grayscale image I to a binary image. The output image replaces all pixels in the input image with luminance greater than level with the value 1 (white) and replaces all other pixels with the value 0 (black). Level is specified in the range [0, 1].

5.3.2 Smoothing Filters

The next technique that was used to preprocess images was the application of smoothing filters. These filters not only removed noise from the image, but also significantly improved the results of binary thresholding.

Two kinds of filters were used for the smoothing. The first one was the averaging filter. This filter adaptively thresholds each pixel based on the value of pixels in a surrounding window. If the current pixel is lighter than the average, then it is made white, otherwise it is made black. The second filter used was the median filter. For this filter each output pixel is assigned the median value in m-by-n neighborhood around the corresponding pixel in the input image.

```
>> for c = 1:13
I = imread(strcat('C:\Users\Unnus\Desktop\Image Preprocessing\', num2str(c), '.jpg'));
I = rgb2gray(I);
aI = imfilter(I, fspecial('average', 40), 'replicate');
mI = aI - I;
mI = medfilt2(mI, [5 5]);
bw = im2bw(mI, 0.02);
imwrite(bw, strcat('C:\Users\Unnus\Desktop\Image Preprocessing\', num2str(c), '_bw_smooth.jpg'));
end
```



Figure 5.7 Smoothing Filters

5.3.3 Removing Small Objects

Another technique that was used to improve the preprocessing of images was the removal of small objects from images. This was mainly done to remove the decimal points from images. The decimal points posed a problem later on in the project, when Tesseract was used to recognize the images. They were not being recognized by

Tesseract, moreover sometimes they effected the recognition of digits around them. Hence, it was required to remove them from the images. They were later on added to the final recognition result by using the stored precisions of meters in the meters table.

The “bwareaopen” function of Matlab was used for this purpose. The bwareaopen(bw, P) removes from a binary image all connected components (objects) that have fewer than P pixels. The results are shown in Figure 5.8.

```
>> for c = 1:13
I = imread(strcat('C:\Users\Unnus\Desktop\Image Preprocessing\', num2str(c), '.jpg'));
I = rgb2gray(I);
aI = imfilter(I, fspecial('average', 40), 'replicate');
mI = aI - I;
mI = medfilt2(mI, [5 5]);
bw = im2bw(mI, 0.02);
bw = bwareaopen(bw, 120);
imwrite(bw, strcat('C:\Users\Unnus\Desktop\Image Preprocessing\', num2str(c), '_objects_removed.jpg'));
end
```



Figure 5.8 Removing Small Objects

5.3.4 Morphological Closing

The last technique that was applied to images was the morphological closing operation. The main purpose of applying this technique was to close the gaps that

existed between the segments of the digits of 7-segment display. This had to be done, because Tesseract was unable to recognize the disconnected font of a 7-segment display.

For this purpose the `imclose` operation of matlab was used. The `imclose(I,SE)` performs morphological closing on the grayscale or binary image, and returns the closed image. The SE is a structuring element used to perform the closing operation. The SE used for meter images was a rectangular one. It was used because it matched most closely with shape and appearance of fonts used on meter displays. The results were quite good. The gaps between segments of digits were closed in almost all the images.

```
>> for c = 1:13
I = imread(strcat('C:\Users\Unnus\Desktop\Image Preprocessing\', num2str(c), '.jpg'));
I = rgb2gray(I);
aI = imfilter(I, fspecial('average', 40), 'replicate');
mI = aI - I;
mI = medfilt2(mI, [5 5]);
bw = im2bw(mI, 0.02);
bw = bwareaopen(bw, 120);
se = strel('rectangle', [9 3]);
closeI = imclose(bw, se);
imwrite(closeI, strcat('C:\Users\Unnus\Desktop\Image Preprocessing\', num2str(c), '_rectangle_closing.jpg'));
end
```



Figure 5.9 Morphological Closing

5.4 Tesseract OCR

The next phase in the development of image processing solution, was the recognition of characters from images. For this purpose it was decided to use Tesseract. Tesseract had very accurate character recognition capabilities, on top of that it had support for Android. Therefore, it was the logical choice. However, Tesseract was developed to recognize well connected, clear fonts on high contrast backgrounds. The LCD displays used on meters were low contrast, and the font used was disconnected 7-segment font. Therefore, the images had to be preprocessed so that they are suitable enough to be recognized by Tesseract. The preprocessing techniques that were used have already been described in the previous section. This section will now describe the Tesseract recognition procedure and results. ^[8]

5.4.1 Training Tesseract

The first step to start the recognition process, was the training of Tesseract. The main aim of the training process is to improve the recognition results of Tesseract. This procedure is used to train Tesseract to recognize fonts in a particular font. Even though this was not necessary, but it was done so that better results could be achieved. During training Tesseract learns the properties of the font and customizes itself to recognize the font. A square font was used to train Tesseract. Since after closing the characters in meter images, the 7-segment font becomes similar to a square font.

Two open source tools were used to train Tesseract to recognize square fonts. The first one was the jTessBoxEditor (Figure 5.10). This tool prepared a .tif training image, for tesseract to recognize. The second tool was the Seerak Tesseract Trainer (Figure 5.11). It used the image generated by jTessBoxEditor to train Tesseract. In the end it generated a font.traineddata file that could be used across platforms to work with Tesseract for character recognition.

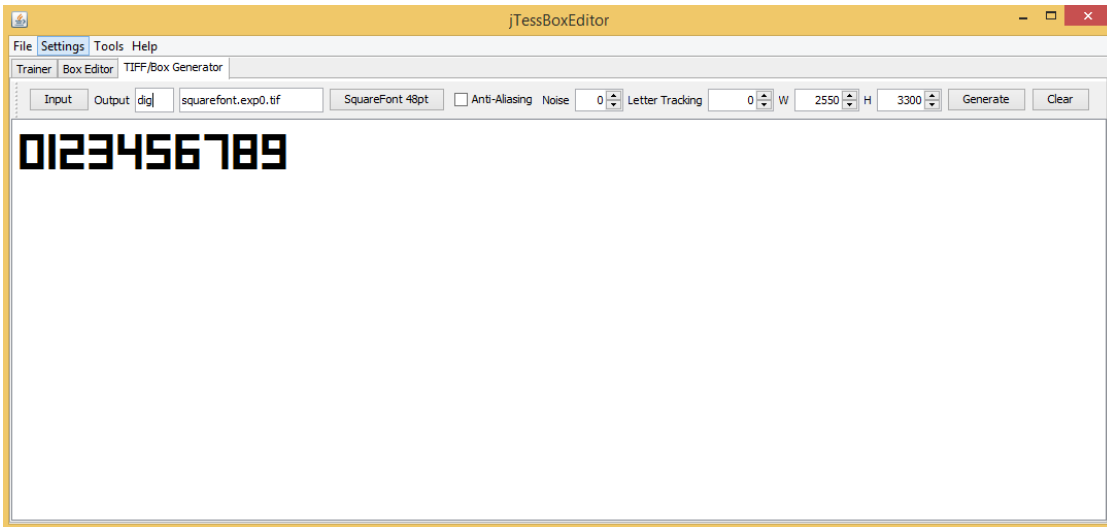


Figure 5.10 jTessBoxEditor

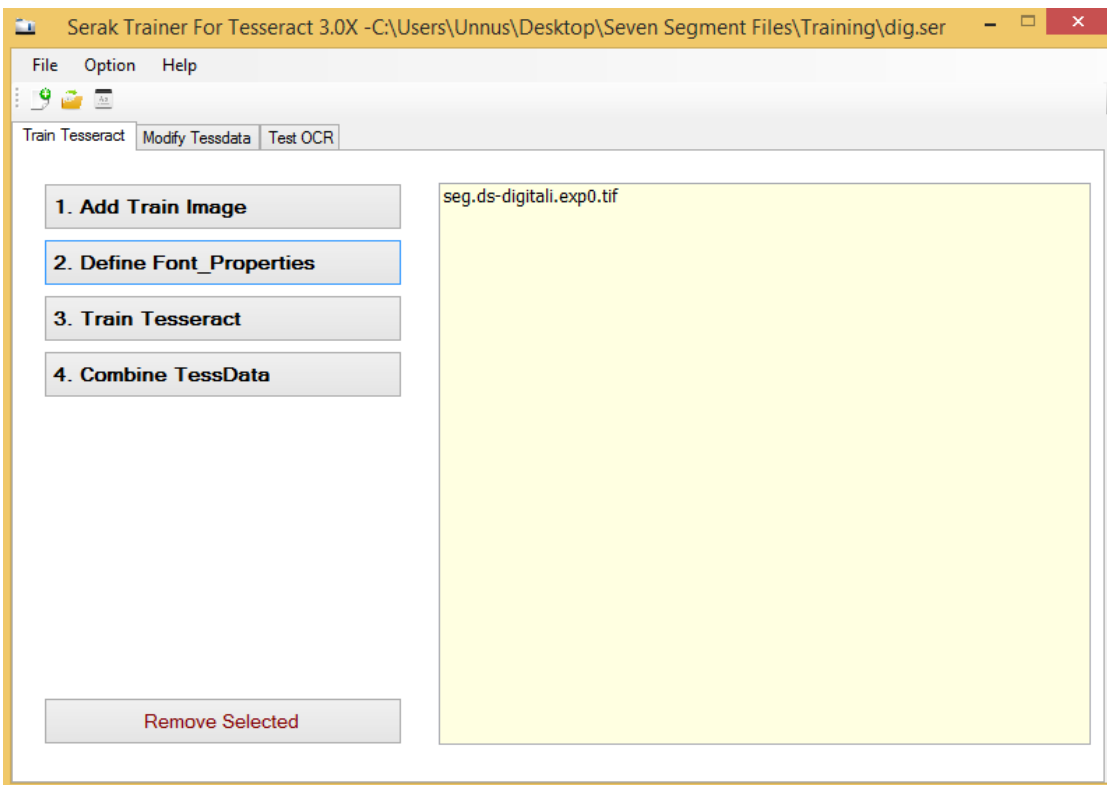


Figure 5.11 Seerak Tesseract Trainer

5.4.2 Recognizing Images

After training Tesseract with a square font, the next step was recognizing the meter images using Tesseract. This task was made easier by Seerak Tesseract Trainer, since it also provided support to run Tesseract on images using the generated dataset. All of this could be done using the GUI interface of Seerak Tesseract Trainer. Normally one would have to use command line to run Tesseract and perform recognition. The results of recognition are shown in the figures below. They were quite good. Out of 13 images, 11 of them were being recognized correctly.

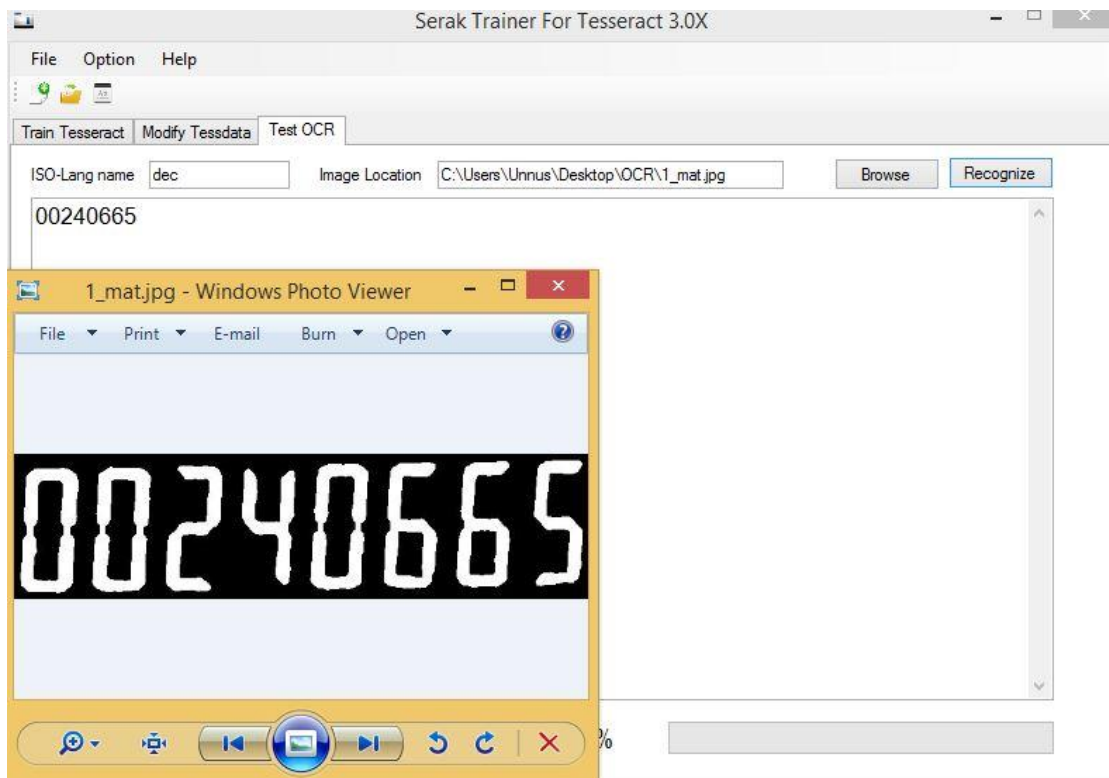


Figure 5.12 Recognition Result 1

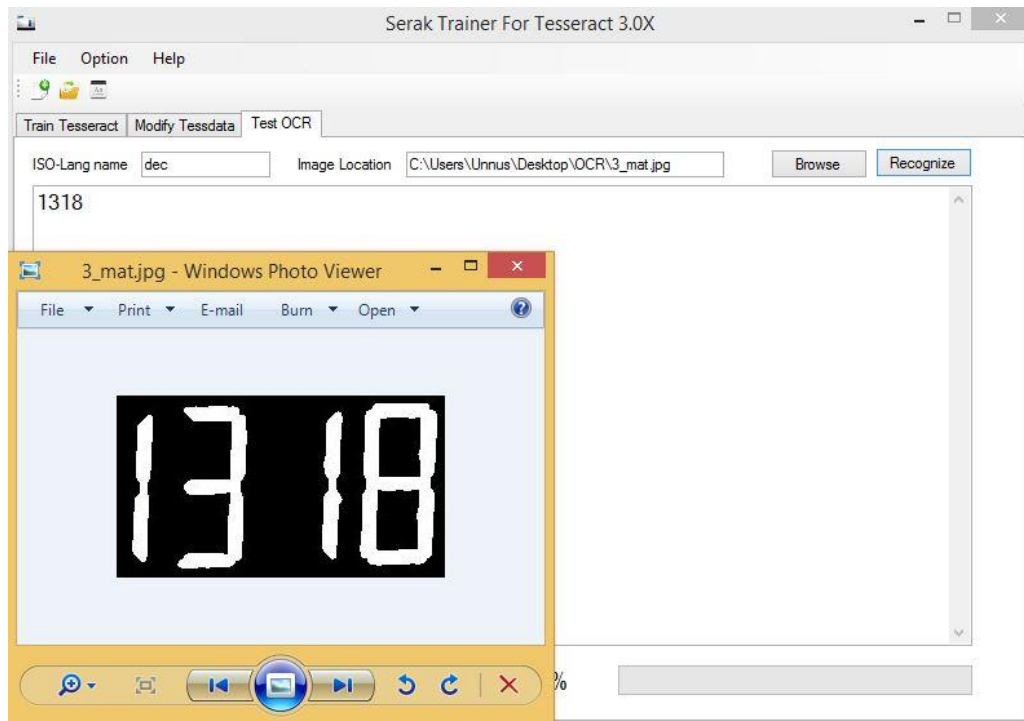


Figure 5.12 Recognition Result 2

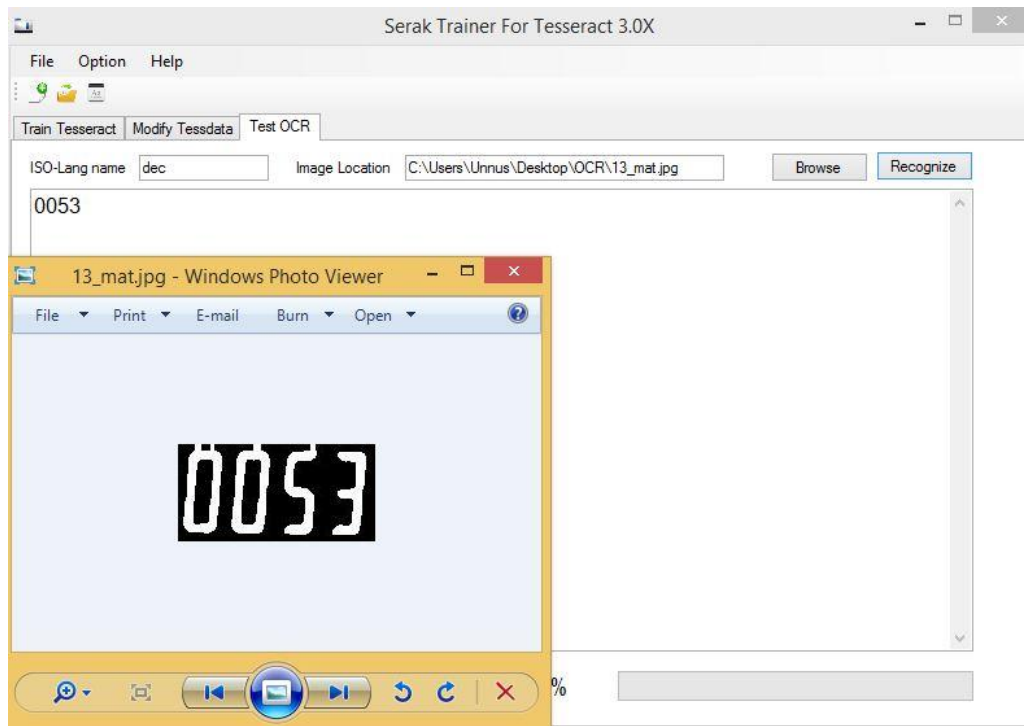


Figure 5.13 Recognition Result 3

5.5 Implementation on Android

The last step in the development of image processing solution was the implementation and integration of the solution in Android App. A choice had to be made about whether to implement the solution on server side or on the mobile side. It was decided to implement the solution on the mobile side. The reason for this choice is that it allowed the meter readers to make corrections to the reading at the spot if the image was not recognized correctly. They could also retry taking picture from a different angle to improve recognition. Even though, implementing the solution on Android had some performance drawbacks, but the benefits of this approach outweigh the drawbacks, that is why this approach was adapted.

5.5.1 OpenCV Preprocessing

The preprocessing solution was implemented in Matlab. Now this solution had to be replicated in OpenCV, because there was no support for Matlab in Android. For this purpose, functions similar to the ones used in the Matlab solution had to be found out. All functions that were used in Matlab solution were also available in OpenCV, except for one. The `bwareaopen()` operation that was used to remove small objects from the binary images. A custom function called `removeSmallObjects` was implemented using OpenCV to perform operations similar to the `bwareaopen()` function. The results for OpenCV solution are shown in Figure 5.14. They are almost exactly similar to the Matlab results. Figure 5.15 shows the code for OpenCV solution.



Figure 5.14 OpenCV Preprocessing

```

public class OCR
{
    public static void main( String[] args )
    {
        System.loadLibrary( Core.NATIVE_LIBRARY_NAME );

        for(int i = 1; i < 14; i++) {

            Mat I = Highgui.imread("C:\\Users\\Unnus\\Desktop"
                + "\\OCR\\" + i + ".jpg");
            Mat blurred = new Mat();
            Mat J = new Mat();
            Mat bw = new Mat();
            Mat closingElement = Imgproc.getStructuringElement(
                Imgproc.MORPH_RECT, new Size(3, 9));
            Mat closeI = new Mat();

            Imgproc.cvtColor(I, I, Imgproc.COLOR_RGB2GRAY);
            Imgproc.blur(I, blurred, new Size(40, 40));
            Core.subtract(blurred, I, J);
            Imgproc.medianBlur(J, J, 5);
            Imgproc.threshold(J, bw, 4, 255, Imgproc.THRESH_BINARY);
            OCR.removeSmallObjects(bw, 120);
            Imgproc.morphologyEx(bw, closeI, Imgproc.MORPH_CLOSE, closingElement);
            Highgui.imwrite("C:\\Users\\Unnus\\Desktop\\OCR\\"
                + i + "_cv.jpg", closeI);

        }

    }

    public static Mat removeSmallObjects(Mat image, double size) {
        List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
        Imgproc.findContours(image.clone(), contours, new Mat(),
            Imgproc.RETR_LIST, Imgproc.CHAIN_APPROX_SIMPLE);

        for(int i = 0; i < contours.size(); i++) {
            double area = Imgproc.contourArea(contours.get(i));
            if(area>0 && area <= size) {
                Imgproc.drawContours(image, contours, i, new Scalar(0, 0, 0), -1);
            }
        }

        return image;
    }

}

```

Figure 5.15 OpenCV Preprocessing Code

Here the averaging filter is applied using `Improc.blur()` function. The `removeSmallObjects()` function finds contours having area less than or equal to supplied size, and draws a black contour over it, to hide it.

5.5.2 Tesseract OCR on Android

The last task that needed to be done was the recognition of the preprocessed images using Tesseract on Android. For this, Tesseract code for Android was downloaded. This code had to be first compiled using Android NDK. The compiled code was then included in the Android App as a module dependency. After preprocessing the image using OpenCV, the Tesseract library was called to perform OCR on the preprocessed image. The result of the OCR was then displayed on screen. The code for calling Tesseract on the image is shown in Figure 5.16.

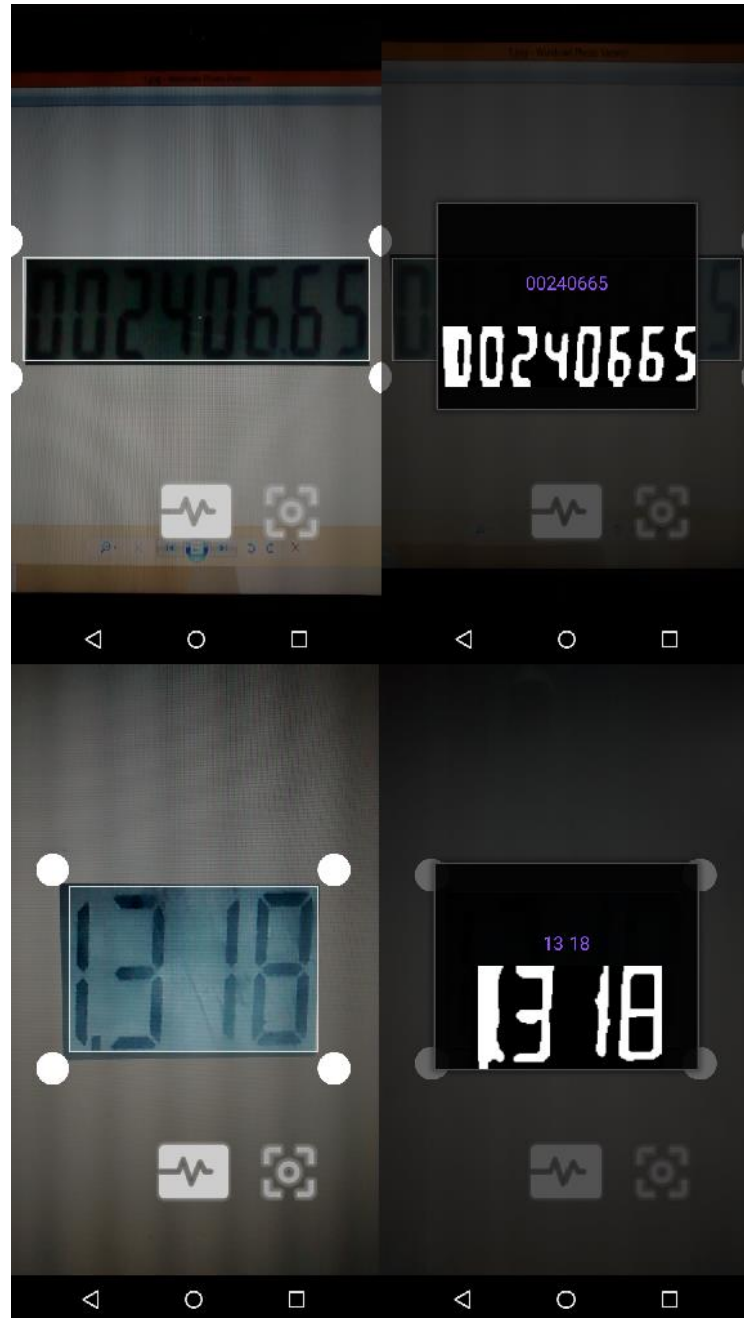
```
public String recognizeText(Bitmap bitmap) {
    Log.d(TAG, "Initialization of TessBaseApi");
    TessDataManager.initTessTrainedData(context);
    TessBaseAPI tessBaseAPI = new TessBaseAPI();
    String path = TessDataManager.getTesseractFolder();
    Log.d(TAG, "Tess folder: " + path);
    tessBaseAPI.setDebug(true);
    tessBaseAPI.init(path, "dig");
    tessBaseAPI.setVariable(TessBaseAPI.VAR_CHAR_WHITELIST,
        "0123456789");
    tessBaseAPI.setVariable(TessBaseAPI.VAR_CHAR_BLACKLIST,
        "!@#%$%^&*()_+=-qwertyuiop[]{}POIU" +
        "YTREWQasdASDfghFGHjklJKLl;L:'\"\\|~`xcvXCVbnmBNM,./<>?");
    tessBaseAPI.setPageSegMode(TessBaseAPI.OEM_TESSERACT_CUBE_COMBINED);
    Log.d(TAG, "Ended initialization of TessEngine");
    Log.d(TAG, "Running recognition on bitmap");
    tessBaseAPI.setImage(bitmap);
    String recognizedText = tessBaseAPI.getUTF8Text();
    Log.d(TAG, "Got data: " + recognizedText);
    tessBaseAPI.end();
    System.gc();
    return recognizedText;
}
```

Figure 5.16 Android Tesseract Code

The `init(path, "dig")` function initializes Tesseract to use "dig.traineddata" file for recognition. The path variable specifies the location of the file. The `setVariable()`

function is used to specify the characters that should be recognized by Tesseract. A bitmap image is provided to Tesseract for recognition using setImage() function. The getUTF8Text() function returns the results of recognition on the image.

The results of Tesseract OCR were quite good, 12 out of 13 images were recognized correctly. The results are shown in Figure 5.17.



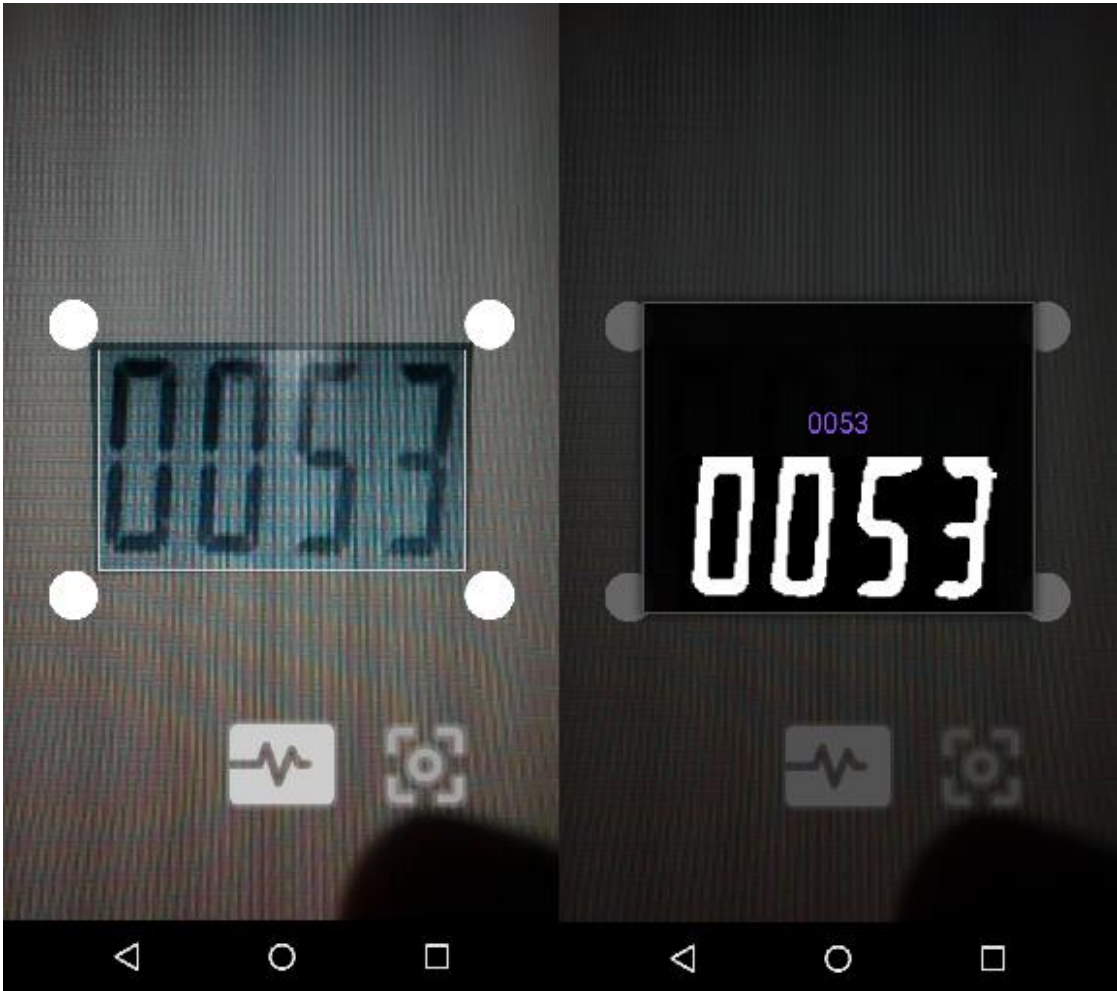


Figure 5.17 Android Tesseract Results

CONCLUSION

This project took a total of 5 months to complete. 1 month was spent on requirement gathering, 2 months on Project Planning and Design and 2 months were consumed in Project Development. The objectives of the project were achieved with satisfactory performance. The following sections discuss the learning outcomes of the project. The limitations of the project are described. The future work for the project is also identified. The chapter concludes with project's potential for commercialization and its impact on society.

6.1 Learning Outcomes

The development of this project has made it evident that software has great potential for making lives and work easier. The importance and advantages of good planning in software design has been realized. Moreover new and improved skills are acquired in the fields of image processing, web and mobile development.

6.2 Limitations

The system does have a few limitations at the moment. It is currently calculating bills using simple way of multiplying units consumed with unit price. However, actual bills are calculated using low tariff and high tariff values. The OCR currently works only for digital meters. The form fields are validated using client side validation. However, server side validation is also necessary in case JavaScript is turned off.

6.3 Future Work

The system can be improved in a number of ways. The auto display detection can be included in the image processing solution. The image recognition for mechanical meters can be added. The admins can be provided the ability to update customer, meter and reader data.

6.4 Commercialization

This project has significant potential for commercialization. It has been made to fulfill the most basic requirements of meter reading companies. It is built to be scalable, so that it can be adapted to fulfill the requirements of meter reading companies. First it can be introduced to private companies. Then once it has become robust enough it can be introduced at the national level.

6.5 Impact on Society

The meter reading system that involved human intervention at multiple points has now been reduced to just to two entities. The readers and the admins. The task of verifying and compiling paper based reading and entry of readings into the system, has been completely removed. This will definitely reduce workload and improve performance. The system would be more reliable and there would be less errors. Less errors would mean correct billing, which ultimately will save excessive or unrealized energy costs. This will contribute towards solving the energy crisis of Pakistan.

REFERENCES

[1] Automated meter readers at 85 grids,

<http://www.dawn.com/news/1126433>

[2] Mariana Baabar, US helping Pakistan to address incorrect billing complaints,

<http://www.thenews.com.pk/Todays-News-2-184392-US-helping-Pakistan-to-address-incorrect-billing-complaints>

[3] Material Design,

<http://www.google.com/design/spec/material-design/introduction.html>

[4] Node.js,

<https://nodejs.org/>

[5] OpenCV,

<http://opencv.org/>

[6] Rohit Dayama et al (2014), Android Based Meter Reading Using OCR, International Journal of Computer Science and Mobile Computing, Vol.3 (Issue.3): 536-539

[7] Seven Segment Optical Character Recognition,

<https://www.unix-ag.uni-kl.de/~auerswal/ssocr/>

[8] Tesseract OCR,

<https://code.google.com/p/tesseract-ocr/>