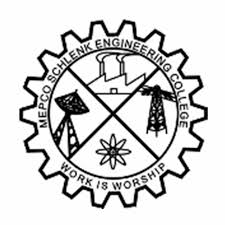
**Mini Project on Electric Bill Management using Database Connectivity**

**Department of Computer Science and Engineering**

Submitted by:

N .Guru Manikandan (14BCS068).

P. Bharathiraja (14BCS066).

V. Manish Kumar (14BSCS070).

G. Prabhu (14BCS072).

Guided By:

Mr. N. Umakanth. AP (CSE).

**Content**

**INTRODUCTION………………………………………………………………………… 3**

PROBLEM SCHEMA……………………………………………………………………………….

OBJECTIVE………………………………………………………………………………………….

**CHAPTER 1 – DATA DEFINITION LANGUAGE……………………………………. 5**

CONSTRAINTS……………………………………………………………………………………..

TABLE CREATION………………………………………………………………………………..

TABLE DELETION………………………………………………………………………………..

ALTERING TABLE………………………………………………………………………………..

**CHAPTER 2 – DATA MANIPULATION LANGUAGE……………………………… 8**

INSERT QUERY …………………………………………………………………………………..

DELETE QUERY…………………………………………………………………………………..

UPDATE QUERY………………………………………………………………………………….

**CHAPTER 3 – DATA RETRIVAL…………………………………………………… 10**

SELECT QUERY…………………………………………………………………………………..

SET OPERATION……………………………………………………………………………….....

JOIN OPERATION………………………………………………………………………………...

VIEWS………………………………………………………………………………………………..

**CHAPTER 4 – TRIGGERS …………………………………………………………… 15**

BENEFITS OF TRIGGERS………………………………………………………………………....

TRIGGER CREATION……………………………………………………………………………….

TRIGGER DELETION……………………………………………………………………………….

ENABLING/DISABLING TRIGGERS………………………………………………………….......

**CHAPTER – 5 FUNCTIONS AND PROCEDURES………………………………… 17**

FUNCTIONS…………………………………………………………………………………………..

PROCEDURES………………………………………………………………………………………..

PACKAGES…………………………………………………………………………………………....

**CHAPTER – 6 JAVA/OPEN DATABASE CONECTIVITY………………………… 20**

JDBC ARCHITECTURE……………………………………………………………………………..

JDBC COMPONENTS……………………………………………………………………………….

DRIVER MANAGER…………………………………………………………………………….

DRIVER…………………………………………………………………………………………...

CONNECTIONS………………………………………………………………………………….

STATEMENT …………………………………………………………………………………….

PREPARED STATEMENT……………………………………………………………………..

CALLABLE SATEMENT………………………………………………………………………..

RESULT SET……………………………………………………………………………………..

**PROJECT OUTCOME…………………………………………………………………..36**

**INTRODUCTION**

**Problem Schema:**

Electric Board of Tamil Nadu planned to maintain the database for Electric Bill Management. Employee of Electric Board will enter the number of units consumed by the consumer in database, which then used to calculate the bill amount. History of consumer bill details also retained in the database.

So the Electric Board has designed the following schema:

1. Consumer ( con\_id, con\_name, gender, eb\_type, deposit, city, mobile, email )
2. Employee ( emp\_id, emp\_name, gender, mobile, email, salary )
3. Bill ( bill\_id, con\_id, year, month, amount, status, pay\_date )
4. Reading ( con\_id, year, month, employee, units )
5. History ( con\_id, bill\_id, year, month, units, amount, paydate )

**Objective:**

Our Objective is to implement the give schema using the Database Management System Concepts and to manage the Electric Billing System.

1. To use the Database Concept to store the data in form table and to process these data to get the information.
2. To create the Graphical User Interface (GUI) to give the user a mean to access their information.
3. To create the database connectivity to the GUI using java/ASP.NET

**CHAPTER – 1 DATA DEFINITION LANGUAGE**

The Data-definition language provides commands for defining relation schemas, deleting relations, and modifying relation schemas.

**Constraints:**

There are different types of constraints they are helpful in using the table in an efficient way. The constraints which we are using are as follows:

1. **Not Null** – Attribute must contain some value.
2. **Unique** – Attribute must contain a unique value i.e. no data should be repeated.
3. **Primary key** – Combination of Not Null and Unique, it uniquely identifies the tuple in the relation table.
4. **Foreign key**- Its will check the values of the attributes for any tuple in the relation to the key attribute of another relation.
5. **Check** – Check constraint will allow the user to check the input with some set of values or some patterns.

**Table Creation:**

Table consist of number of rows (tuples) and columns (attributes) which is used to store the data. The Syntax for creating the table in SQL

**Syntax:**

CREATE TABLE <table\_name> (

V1 D1,

V2 D2,

… … );

Where Vi is Variable name, Di is Type of the variable.

**Example:**

CREATE TABLE Consumer (

con\_id CHAR(4)

CONSTRAINT con\_cid\_pk PRIMARY KEY

CONSTRAINT con\_cid\_ck CHECK ( con\_id LIKE 'C%' ),

con\_name VARCHAR2(15)

CONSTRAINT con\_cname\_nn NOT NULL,

gender CHAR(1)

CONSTRAINT con\_gen\_nn NOT NULL

CONSTRAINT con\_gen\_ck CHECK ( gender IN ('M','F') ),

eb\_type CHAR(1)

CONSTRAINT con\_type\_nn NOT NULL

CONSTRAINT con\_type\_ck CHECK (eb\_type IN ('D','C')),

deposit NUMBER

CONSTRAINT con\_dep\_nn NOT NULL,

city VARCHAR2(10),

mobile NUMBER(12)

CONSTRAINT con\_mob\_nn NOT NULL,

email VARCHAR2 (20) );

This query results in creation of new table

Name Null Type

--------------- -------------- ------------

CON\_ID NOT NULL CHAR(4)

CON\_NAME NOT NULL VARCHAR2(15)

GENDER NOT NULL CHAR (1)

EB\_TYPE NOT NULL CHAR (1)

DEPOSIT NOT NULL NUMBER

CITY VARCHAR2 (10)

MOBILE NOT NULL NUMBER (12)

EMAIL VARCHAR2 (30)

**Table Deletion:**

To delete the relation form the database **DROP** query is used. The Syntax for the deleting the table is as follows

**Syntax:**

DROP TABLE <table\_name>;

**Example:**

DROP TABEL CONSUMER;

This query will remove the table named CONSUMER from the database.

**Altering table:**

Once after the creation of the relation we can change the relation schema using **ALTER** command. The syntax for altering the table is as follows:

**Syntax:**

ALTER TABLE <table\_name> ADD/DROP/MODIFY <column name> <Type & Constraint>

1. ADD is used to add the new column in the relation schema or to add the constraint for the column.
2. MODIFY is used to change the data type of already existing column.
3. DROP is used to delete the column or to drop the constraint form the table.

**Example:**

ALTER TABLE BILL ADD STATUS VARCHAR2 (10);

ALTER TABLE BILL MODIFY STATUS NOT NULL;

ALTER TABLE BILLS DROP COLUMN STATUS;

**CHAPTER – 2 DATA MANIPULATION LANGUAGE**

The Data-manipulation language provides the ability to query information from the database and to insert tuples into delete tuples from, and modify tuples in the database.

**Insert Query:**

**Insert** query is used to insert the tuples into the table. The Syntax for the Insert query is as follows:

**Syntax:**

INSERT INTO <table\_name> VALUES (attirb1, attirb2, attrib3);

**Examples:**

INSERT INTO CONSUMER VALUES ('&con\_id','&con\_name','&gender','&type',deposit,'&city',mobile,'&mail');

**Delete Query:**

**Delete** query is used to delete the tuple from the table. The Syntax for the Delete query is as follows:

**Syntax:**

DELETE FROM <table\_name>

WHERE <condition>;

**Example:**

DELETE FROM CONSUMER

WHERE CON\_ID = 'C102';

The consumer with the id C102 will be deleted from the database

**Update Query:**

**Update** query is used to update the already existing tuple attribute values. The syntax for update is as follows:

**Syntax:**

UPDATE <teble\_name>

SET <attribute\_name>=<value>

WHERE <condition>;

**Example:**

UPDATE CONSUMER

SET DEPOSITE = 12000

WHERE CON\_ID = ‘C104’;

**CHAPTER – 3 DATA RETIRVAL**

**Select Query:**

**Select** query is used to retrieve the data from the relation schema. Selection can also be done based on some condition. The Syntax for the Select Query is as follows:

**Syntax:**

SELECT <column names> FROM <table\_name>

WHERE <condition>;

**Example:**

SELECT EMP\_ID, EMP\_NAME FROM EMPLOYEE;

EMP\_ ID EMP\_NAME

----------- ---------------

E101 Prabhu

E102 Kanaga

E103 Ramesh

E104 Ramu

SELECT CON\_NAME FROM CONSUMER

WHERE CON\_NAME LIKE '\_\_\_\_\_\_';

This will return the Consumer name who has six letter name.

CON\_NAME

---------------

Sheela

Simran

Sankar

**Set Operations:**

**Set** operators are used to join the results of two (or more) SELECT statements. The SET operators available in Oracle 11g are as follows:

* **UNION** – Gives the **distinct** tuples from both the SELECT query.
* **UNION ALL** - Gives **all** the tuples from both the SELECT query this includes the duplicate entries also.
* **INTERSECT** – Gives the tuple which are **common** in both the SELECT query.
* **MINUS** – This **removes** the **second query's results** from the output if they are also found in the **first query's results.**

**Points to Remember:**

* Same number of columns must be selected by all participating SELECT statements.
* Column names used in the display are taken from the first query.

The syntax for this set operation are as follows:

**Syntax:**

**Union:**

**<**SELECT STATEMENT>

**UNION**

<SELECT STATEMETN>

**Example:**

(SELECT CON\_NAME AS NAME FROM CONSUMER

WHERE GENDER = 'M')

**UNION**

(SELECT EMP\_NAME AS NAME FROM EMPLOYEE

WHERE GENDER = 'M');

NAME

---------------

Karthik

Kumar

Kumaran

Prabhu

Pradeep

**Intersect:**

**<**SELECT STATEMENT>

**INTERSECT**

<SELECT STATEMETN>

**Example:**

(SELECT CON\_NAME AS NAME FROM CONSUMER)

**INTERSECT**

(SELECT EMP\_NAME AS NAME FROM EMPLOYEE);

NAME

---------------

Prabhu

Ramu

**MINUS:**

**<**SELECT STATEMENT>

**MINUS**

<SELECT STATEMETN>

**Example:**

(SELECT CON\_NAME AS NAME FROM CONSUMER)

**MINUS**

(SELECT EMP\_NAME AS NAME FROM EMPLOYEE);

NAME

---------------

Anu

Karthik

Kumar

Kumaran

Pradeep

Sankar

**Join Operation:**

A Query that selects data from more than one table. A Join is characterized by multiple tables in the FROM clause. Oracle pairs the rows from these tables using the condition specified in the WHERE clause and returns the resulting rows. This condition is called the join condition and usually compares columns of all the joined tables.

The JOIN operations are:

* [**INNER JOIN operation**](http://docs.oracle.com/javadb/10.6.2.1/ref/rrefsqlj35034.html#rrefsqlj35034) **-** Specifies a join between two tables with an explicit join clause.
* [**LEFT OUTER JOIN operation**](http://docs.oracle.com/javadb/10.6.2.1/ref/rrefsqlj18922.html#rrefsqlj18922) - Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the first table.
* [**RIGHT OUTER JOIN operation**](http://docs.oracle.com/javadb/10.6.2.1/ref/rrefsqlj57522.html#rrefsqlj57522) - Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the second table.
* [**CROSS JOIN operation**](http://docs.oracle.com/javadb/10.6.2.1/ref/rrefsqljcrossjoin.html#rrefsqljcrossjoin)- Specifies a join that produces the Cartesian product of two tables. It has no explicit join clause.
* [**NATURAL JOIN operation**](http://docs.oracle.com/javadb/10.6.2.1/ref/rrefsqljnaturaljoin.html#rrefsqljnaturaljoin) - Specifies an inner or outer join between two tables. It has no explicit join clause. Instead, one is created implicitly using the common columns from the two tables.

**Syntax:**

SELECT <attribute\_name> FROM

<table\_name1> INNER/LEFFT OUTTER/RIGHT OUTER JOIN

<table\_name2> ON <join Condition>

[WHERE <condition clause>];

**Example:**

SELECT CON\_ID, EMP\_NAME FROM

CONSUMER JOIN READING JOIN EMPLOYEE

ON EMPLOYEE.EMP\_ID = READING.EMPLOYEE

ON CONSUMER.CON\_ID = READING.CON\_ID

WHERE MONTH = 2 AND YEAR = 2016;

CON\_ID EMP\_NAME

---------- ---------------

C105 Ramesh

**Views:**

A **view** is a logical representation of another table or combination of tables. A view derives its data from the tables on which it is based. These tables are called base tables. Base tables might in turn be actual tables or might be views themselves. All operations performed on a view actually affect the base table of the view. You can use views in almost the same way as tables. You can query, update, insert into, and delete from views, just as you can standard tables. The Syntax to create the View is as follows

**Syntax:**

CREATE OR REPLACE VIEW <view\_name> AS

<SELECT QUERY>;

**Example:**

CREATE VIEW EMPLOYEEVIEW AS

SELECT EMP\_ID AS ID, EMP\_NAME AS NAME,GENDER,MOBILE,SALARY FROM EMPLOYEE;

The **WITH** **CHECK OPTION** creates the view with the constraint that INSERT and UPDATE statements issued against the view cannot result in rows that the view cannot select.

**Examples:**

CREATE OR REPLACE VIEW MALEVIEW AS

SELECT \* FROM EMPLOYEE

WHERE GENDER = 'M'

WITH CHECK OPTION;

This will allow insertion only when it satisfies the WHERE condition for the view.

**CHAPTER – 4 TRIGGERS**

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

* *A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).*
* *A database definition (DDL) statement (CREATE, ALTER, or DROP).*
* *A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).*

Triggers could be defined on the table, view, schema, or database with which the event is associated.

**Benefits of Triggers:**

*Triggers can be written for the following purposes:*

* *Generating some derived column values automatically*
* *Enforcing referential integrity*
* *Event logging and storing information on table access*
* *Auditing*
* *Synchronous replication of tables*
* *Imposing security authorizations*
* *Preventing invalid transactions*

**Creating Triggers:**

The Syntax for creating the Triggers is as follows:

**Syntax:**

*CREATE [OR REPLACE] TRIGGER trigger\_name*

{BEFORE | AFTER | INSTEAD OF}

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col\_name]

ON table\_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

**Examples:**

CREATE OR REPLACE TRIGGER LOGINCREATECONSUMER

AFTER INSERT

ON CONSUMER

FOR EACH ROW

BEGIN

INSERT INTO LOGIN VALUES (:NEW.CON\_ID,LOWER(:NEW.CON\_ID));

END;

*This will insert the new tuple in the LOGIN table when there is an INSERTION in CONSUMER table*

***Trigger Deletion:***

*To delete the trigger we have to use DROP command. The Syntax to drop the create trigger is as follows:*

***Syntax:***

*DROP TRIGGER <trigger\_name>;*

*Example:*

*DROP TRIGGER LOGENTRY;*

***Enabling/Disabling Triggers:***

*Disabling doesn’t mean that the trigger is deleted or removed it will only be disabled we can enable the trigger whenever we need.The general Syntax for enabling and disabling the trigger is as follows:*

***Syntax:***

*ALTER TRIGGER <TRIGGER\_NAME> ENABLE/DISABLE;*

***Example:***

*ALTER TRIGGER LOGINCREATECONSUMER DISABLE;*

**CHAPTER –5 FUNCTIONS AND PROCEDURES**

**Procedure:**

A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program. The Syntax for the Procedure creation is as follows:

**Syntax:**

*CREATE [OR REPLACE] PROCEDURE <procedure\_name>*

*[(parameter\_name [IN | OUT | IN OUT] type [, ...])]*

*{IS | AS}*

*BEGIN*

*< procedure\_body >*

*END procedure\_name;*

**Examples:**

CREATE OR REPLACE PROCEDURE PROCDISPLAYDETAIL

(ID IN CHAR, NAME OUT VARCHAR,MAIL OUT VARCHAR ) AS

BEGIN

SELECT EMP\_NAME, EMAIL INTO NAME, MAIL FROM EMPLOYEE

WHERE EMP\_ID = ID;

END;

**Functions:**

Functions are subprograms that return a single value, mainly used to compute and return a value. A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

**Syntax**

*CREATE [OR REPLACE] FUNCTION function\_name*

*[(parameter\_name [IN | OUT | IN OUT] type [, ...])]*

*RETURN return\_datatype*

*{IS | AS}*

*BEGIN*

*< function\_body >*

*END [function\_name];*

**Example:**

CREATE OR REPLACE FUNCTION TOTALCUSTOMERS

RETURN NUMBER IS

TOTAL NUMBER (2):= 0;

BEGIN

SELECT COUNT (\*) INTO TOTAL

FROM CUSTOMERS

RETURN TOTAL;

END;

**Packages:**

*PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.*

A package will have two mandatory parts:

* Package specification
* Package body or definition

**Package Specification:**

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms. Thes Syntax for creating the Package specification is as follows:

**Syntax:**

CREATE OR REPLACE PACKAGE <package\_name> AS

FUNCTION <function name1>(argumetns) RETURNS TYPE;

FUNCTION <function name2>(argumetns) RETURNS TYPE;

.

.

.

END <package\_name>;

**Example:**

CREATE OR REPLACE PACKAGE ELECTRICBILL AS

FUNCTION TOTCONS RETURN INTEGER;

FUNCTION TOTEMP RETURN INTEGER;

END ELECTRICBILL;

**Package Body:**

*The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.*

*The CREATE PACKAGE BODY Statement is used for creating the package body. The Syntax is as follows:*

***Syntax:***

CREATE OR REPLACE PACKAGE BODY <package\_name> AS

FUNCTION/PROCEDURE <NAME>(ARGUMENT)

[RETURN TYPE] AS

BEGIN

END <NAME>;

FUNCTION/PROCEDURE <NAME1>(ARGUMENT)

[RETURN TYPE] AS

BEGIN

END <NAME1>;

END <package\_name>;

***Example:***

*CREATE OR REPLACE PACKAGE BODY ELECTRICBILL AS*

*FUNCTION TOTCONS*

*RETURN INTEGER AS*

*TCONS INTEGER;*

*BEGIN*

*SELECT COUNT(\*) INTO TCONS FROM CONSUMER;*

*RETURN TCONS;*

*END;*

*FUNCTION TOTEMP*

*RETURN INTEGER AS*

*TEMP INTEGER;*

*BEGIN*

*SELECT COUNT(\*) INTO TEMP FROM CONSUMER;*

*RETURN TEMP;*

*END;*

*END ELECTRICBILL;*

**CHAPTER –6 JAVA/OPEN DATABASE CONNECTIVITY**

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

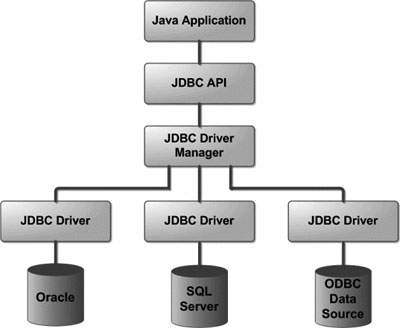
* Making a connection to a database.
* Creating SQL or MySQL statements.
* Executing SQL or MySQL queries in the database.
* Viewing & Modifying the resulting records.

**JDBC Architecture:**

*The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −*

* **JDBC API:** This provides the application-to-JDBC Manager connection.
* **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

**JDBC Components:**

**Driver Manager:**

This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain sub protocol under JDBC will be used to establish a database Connection.

**Driver:**

This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use Driver Manager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

***Connection:***

*This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.*

***Code:***

***public******static*** *Connection Connect()* ***throws*** *ClassNotFoundException, SQLException*

{

Class.forName("oracle.jdbc.driver.OracleDriver");

Connection connection = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","Guru","Guru");

**return** con;

}

This snippet will lode the Driver manager and get connection with the database.

**Statement:**

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

**Code:**

Statement stat = con.createStatement();

This will create a statement to execute query.

***PreparedStatement:***

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

***Code:***

*PreparedStatement pstat= con.prepareStatement("SELECT \* FROM EMPLOYEE WHERE EMP\_ID = ?");*

*This will allow the user to reuse the same statement by setting the different values for the parameter. To set the value in the parameter marked as ? we can use any of the following mthods.*

* *setString(STRING);*
* *setInt(Integer);*
* *setFloat(Float);*

*pstat.setString(1, “E101”);*

**Callable Statement:**

*To call the* ***stored procedures and functions****, CallableStatement interface is used.*

*We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.*

*Suppose if we need to get the name and email id of the Consumer or the Employee we can create a procedure to perform this task and we can use the callable statement to get the output.*

***Code:***

CallableStatement cstat=con.prepareCall(

"{call PROCDISPLAYDETAILCONSUMER(?,?,?)}");

cstat.registerOutParameter(2, java.sql.Types.**VARCHAR**);

cstat.registerOutParameter(3, java.sql.Types.**VARCHAR**);

*registerOutParameter is used to register the data type of the OUT parameters in the procedure so that the SQL queries will identify the data type and execute it.*

*To call the procedure execute() function is used.*

*cstat.execute();*

**ResultSet:**

The object of ResultSet maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row.

**Code:**

ResultSet rset = stat.executeQuery("SELECT \* FROM READING WHERE CON\_ID ='C101'");

To retirve the value from the ResultSet the following methods can be used.

* getString(attrib\_name);
* getInt(attrib\_name);
* getFloat(attrib\_name);

To move the cursour form one tuple to another next() method is used.

**while**(rset.next())

{

System.out.println(rset.getString(CON\_NAME));

}

**Execution of Staements:**

To execute the Statement we can use the following method. It will differ for different SQL statements.

* exectueUpdate() - It is used for create, drop, insert, update, delete etc.
* executeQuery() - executes the select query. It returns an instance of ResultSet.

**Code:**

**Inserting the date into the table:**

con=DbConnect.Connect();

stat=con.createStatement();

pstat = con.prepareStatement("INSERT INTO CONSUMER VALUES(?,?,?,?,?,?,?,?)");

ResultSet rset = stat.executeQuery("SELECT \* FROM CONSUMER WHERE CON\_ID='"+txtC.getText()+"'");

if(!rset.next())

{

pstat.setString(1,txtC.getText());

pstat.setString(2,textField.getText());

if(rdbtnMale.isSelected())

Gender = "M";

else

Gender = "F";

pstat.setString(3,Gender);

if(rdbtnNewRadioButton.isSelected())

Type = "D";

else

Type = "C";

pstat.setString(4,Type);

pstat.setString(5,textField\_2.getText());

pstat.setString(6,textField\_3.getText());

pstat.setString(7,textField\_4.getText());

pstat.setString(8,textField\_5.getText());

pstat.executeUpdate();

}

**Updating the date in the table:**

con=DbConnect.Connect();

pstat = con.prepareStatement("UPDATE EMPLOYEE SET EMP\_NAME = ?,GENDER = ?,MOBILE=?,EMAIL=?,SALARY = ?"

+ "WHERE EMP\_ID = ?");

pstat.setString(1, textField.getText());

if(rdbtnMale.isSelected())

Gender="M";

else

Gender = "F";

pstat.setString(2, Gender);

pstat.setInt(3, Integer.parseInt(textField\_1.getText()));

pstat.setString(4, textField\_2.getText());

pstat.setInt(5, Integer.parseInt(textField\_3.getText()));

pstat.setString(6, choice.getSelectedItem());

pstat.executeUpdate();

**Deleting the date from the table:**

con = DbConnect.Connect();

pstat = con.prepareStatement("DELETE FROM "+Table+" WHERE BILL\_ID =?");

String Id =JOptionPane.showInputDialog(panel, "Bill Id");

if(Id!=null)

{

if(Id.equals(""))

{

pstat.setInt(1,Integer.parseInt(Id));

pstat.executeUpdate();

}

}

**Retrieving the date from the table:**

rset = stat.executeQuery("SELECT CON\_ID FROM Bills WHERE CON\_ID ='"+id+"'");

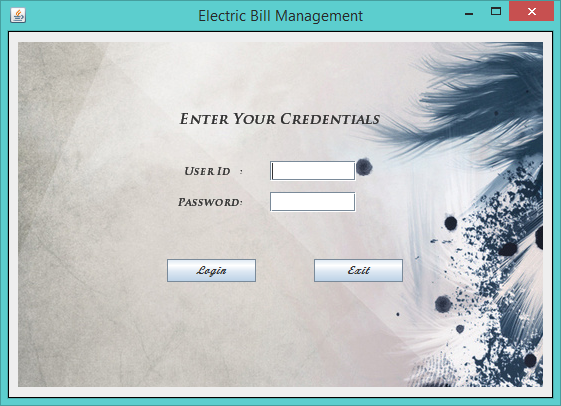
while(rset.next())

{

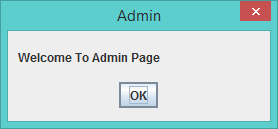
JOptionPane.showMessage(panel,rset.getString(CON\_ID));

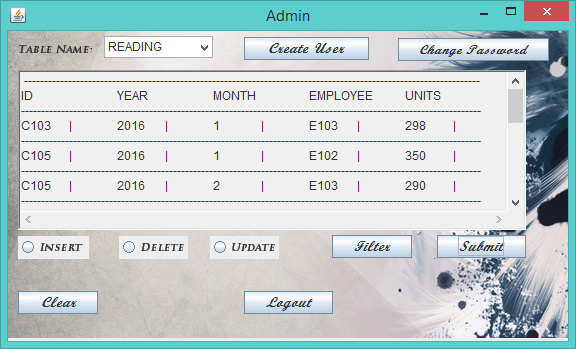
}

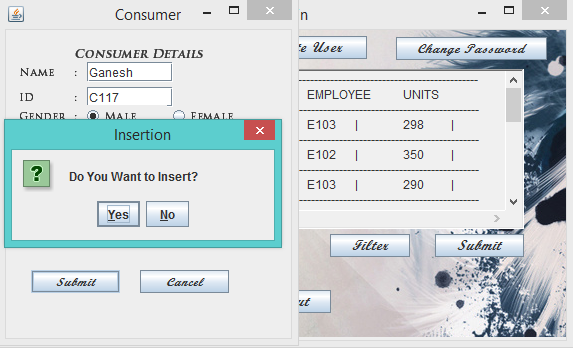
**Program Outcomes:**

**Login Page**

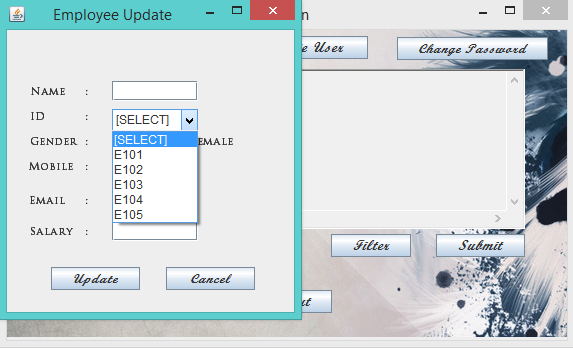
**Administrator Page:**

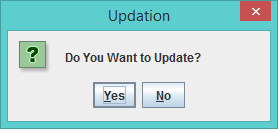




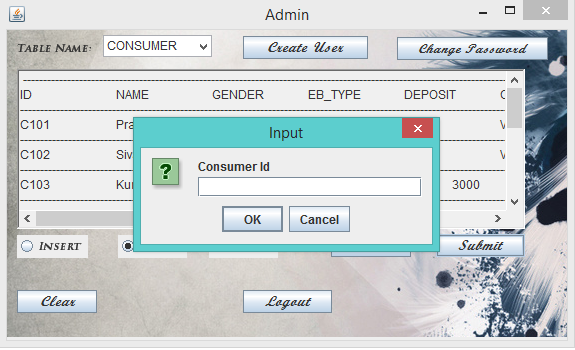


**Insertion:**

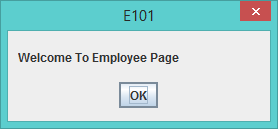
**Update:**

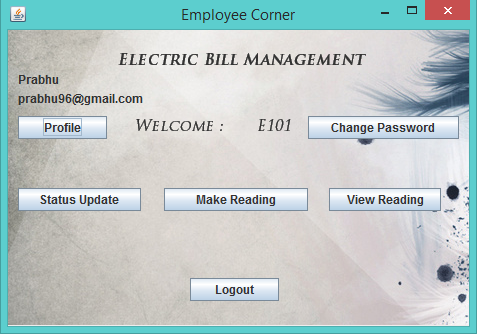


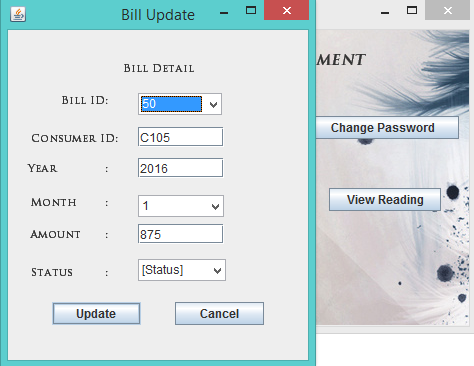
Yes will make the update in the database and No will cancel the operation.

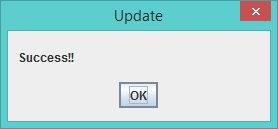
**Delete:**

**Employee Corner:**

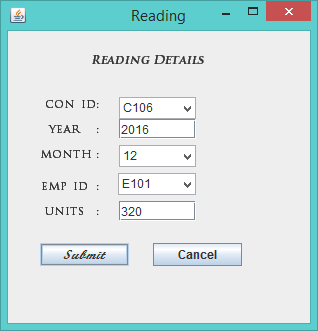


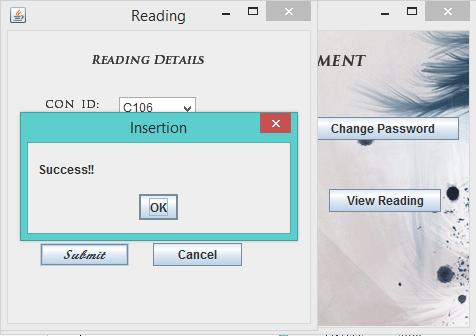


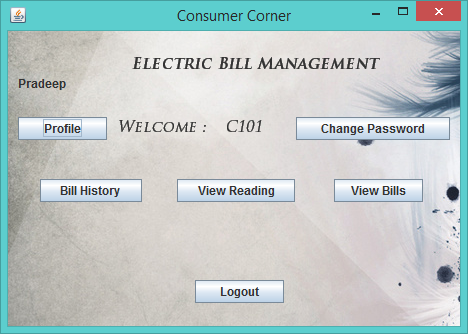
Status Update:

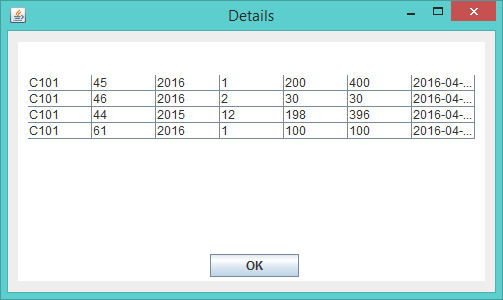


Make Reading:

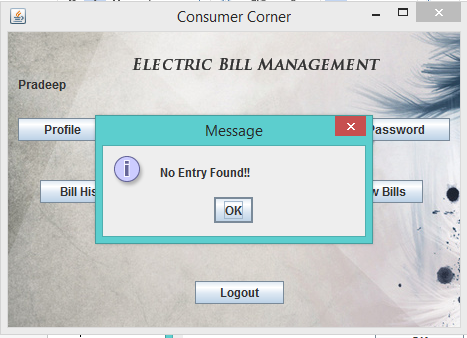




**Consumer Corner:**

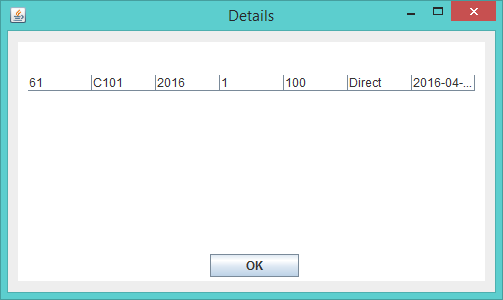
Bill History of the Consumer C101

View Reading:



If the reading is not yet taken for the consumer, it will show as No Entry Found.

View Bills:



This give the Bill History for the Consumer.