CA400
Technical Specification

Workload Measurement System for Nurses

Project Supervisor: Dr. David Sinclair

Ruth Leavey 17323886
Harley Martin 17401932

Date completed: 06/05/2021

Table of Contents

1. Introduction **3**
1.1. Overview 3
1.2. Glossary 3
1.3. Software Tools used 3

2. Motivation **4**

3. Design **5**
3.1. System Design 5
3.1.2 System Architecture flow diagrams 5
3.2. High level Design 6
3.2.1. Context Diagram 6
3.2.2. DFD 8
3.2.3. Class Diagram 11
3.2.4. Database Relationship Diagrams 11

4. Implementation **13**
4.1. Desktop Application 13
4.1.1. Desktop-database Connectivity 14
4.1.2. Login 15
4.1.3. Configuration 15
4.1.3.1. Work Model Configuration 15
4.1.3.2. Sub Model Configuration 21
4.1.3.2. Department & Ward - WorkModel and SubModel Assignment 24
4.1.4. Task Logging 27
4.1.5. Staff Information 30
4.2. Mobile Application 31
4.2.1. Mobile-database connectivity 31
4.2.2. Login/Logout 32
4.2.3. Task Logging 33
4.3. AWS Database 36
4.3.1. Tables and relationships 37
4.3.2. Views 45
5.3.3 Stored Procedures 49
4.3.4. Mock Data Insertion 52

5. Problems and Resolutions **57**
5.1. Covid 19 57
5.3 PC issues 58
5.4 Encryption 58
5.6 AWS permissions and bulk insert 59
5.7 Power BI 59
5.8 JFreeChart display 59

6. Results **59**

7. Future work **60**
7.1. Encryption 60
7.2. Defaults every shift. 60
7.3. Outlier tasks. 60
7.4. Bands 60

9. Research **61**
9.1. Requirement Research 61
9.2. Course Material Researched 64
9.3. Research References 65

1. Introduction

1.1. Overview

This technical specification is a technical description of the system we have designed for our Final Year Project in Computer Applications and Software Engineering. It outlines the motivation, research, design, implementation, problems and solutions of the project as well as results and future work.

We chose to develop a workload measurement system to be used by hospital nurse staff. The primary aim of the system is to record the work undertaken by nurses for each patient in a hospital and to output appropriate information reflecting the workload recorded. It's aim is to provide insight into the amount of work performed on each ward and to present management within the hospital the use of concrete information to help ensure they have the correct number of nurses on each ward to accommodate the work to be done.

The system requires nurses to record tasks completed for patients and include the intensity of the labour that was required. This information will be stored and calculated to give an overall workload score or weight. This workload score will be displayed to management in real-time to represent the current workload required and hence help to quickly identify if a ward is overwhelmed and if reallocation of nurses is needed. Additional data analysis is undertaken to identify any patterns and trends recurring and to create possible predictions for future ward workload.

1.2. Glossary

Pie Chart is used often throughout this TechnicalSpecification. It is used to identify a WorkModel or WorkSubModel, as their shape looks so similar to a pie chart. Pie Chart is never used to specify an actual pie chart.

1.3. Software Tools used

Java - The programming language used to write the desktop and mobile applications.

SQL - The programming language to create and update database tables.

Android Studio IDE - The environment the mobile application was coded in

Netbeans IDE - The environment the desktop application was coded in

Microsoft SQL Server - The type of RDMS (Relational database management system) used

AWS RDS - The type of Cloud database used

JDBC - API used to establish connection from desktop application to the database

jTDS - API used to establish connection between mobile application and database

JFree - The java library for creating desktop pie chart visualizations.

MPAndroidChart - The java library for creating mobile pie chart visualizations.

2. Motivation

With the COVID-19 pandemic currently affecting the lives of everybody worldwide in some shape or form, we were both very interested in making use of this opportunity to have our final year project contribute in some way to the fight against the virus. We wanted our project to be relevant to the current state of the world and to be useful in some way in helping to move forward.

The basis of this project originated from hearing Harley's aunt, who used to work as a nurse in a hospital, recall a terribly poor workload measurement system that was in place. She recalled that the system was imprecise and vague. Individual measures were not given to the individual jobs being done, which resulted in the implication that the same amount of work went into every job. Only a couple of the most common tasks were accounted for and they all had the same options for different "difficulty levels". This did not accurately reflect the reality of the work performed by nurses and therefore, it did not actually help management to schedule nurses according to which wards would need more/less work.

We liked the idea of designing and implementing a system to accurately measure the workload in the hospital wards and, through further conversation with other friends and family working in nursing and nurse management, it was agreed that a more thoughtfully designed system with accurate measurements would be highly appreciated in the healthcare industry.

It has been reported in recent years that hospitals in Ireland have a shortage of nurses so our healthcare system was already under strain before this pandemic. Thus, it is now even more important to ensure the nurses that are available are distributed among hospitals according to where the work is most needed. Our workload measurement system will help point out where in a given hospital the work is needed so that the staff rostering the nurses know where to put them.

It is well known that hospitals around Ireland are very frequently understaffed. This system will help to discover the amount of work needed. The hospitals can then schedule their own nurses and call upon nurses from nursing agencies when needed. Being able to assign values to the work being done will make it easy to measure the work. Once the hospitals can have an actual value for the work being done, they can assess whether they are equipped well enough to efficiently complete this work or whether they need more nurses and, most importantly, where they need these nurses.

3. Design

3.1. System Design

3.1.2 System Architecture flow diagrams

Figure 3.1.1.

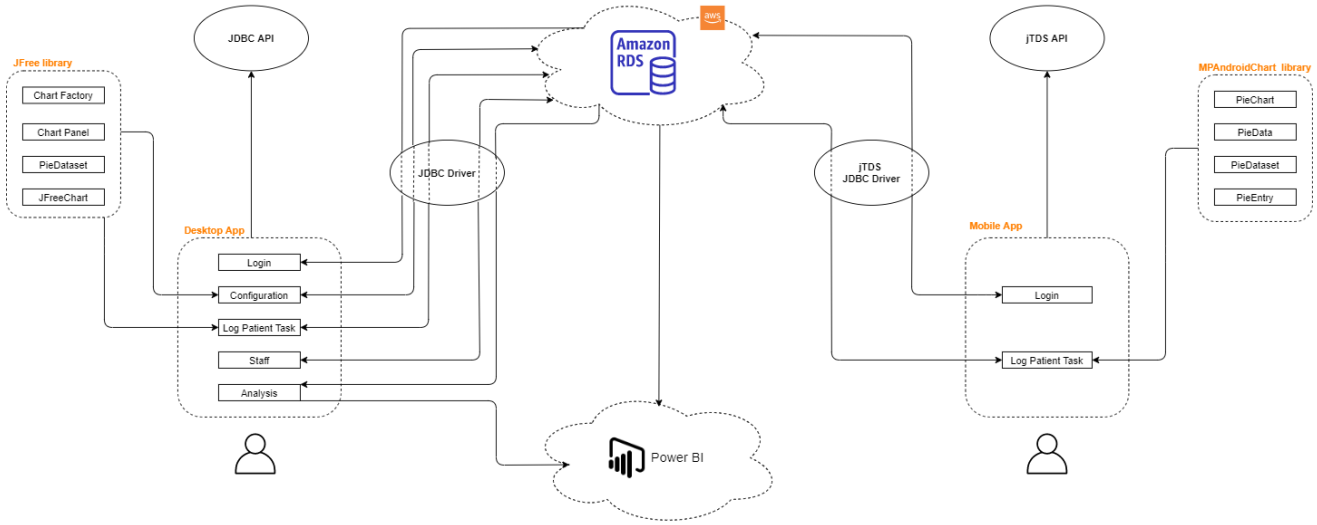
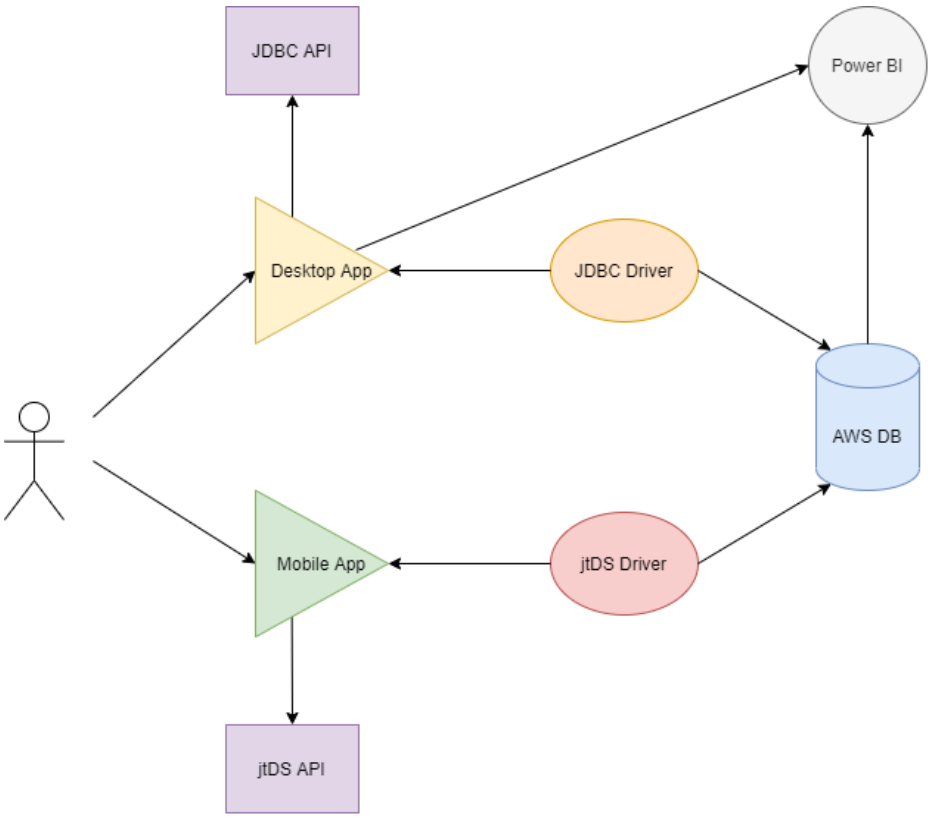


Figure 3.1.2.



3.2. High level Design

3.2.1. Context Diagram

Our system consists of two applications; a desktop and a mobile app. The desktop application returns two different types of accounts depending on the permission of the user. All admin and nurse managerial staff will have access to an account which allows for configuration, staff password updates, patient work log and analysis of data. All regular nurse staff will have access to another account which allows for patient work log only. The mobile account has only one type of account which allows patient work log and can be accessed by all possible users. To accommodate these aspects, we split our context diagram into three separate diagrams to allow for better understanding of our project. Figure 3.2.1.1. is a context diagram regarding the admin/manager/director type user and the desktop application. Figure 3.2.1.2. is a context diagram regarding the regular nurse user type and the desktop application. Figure 3.2.1.3. is a context diagram regarding all users and the mobile application.

Figure 3.2.1.1.

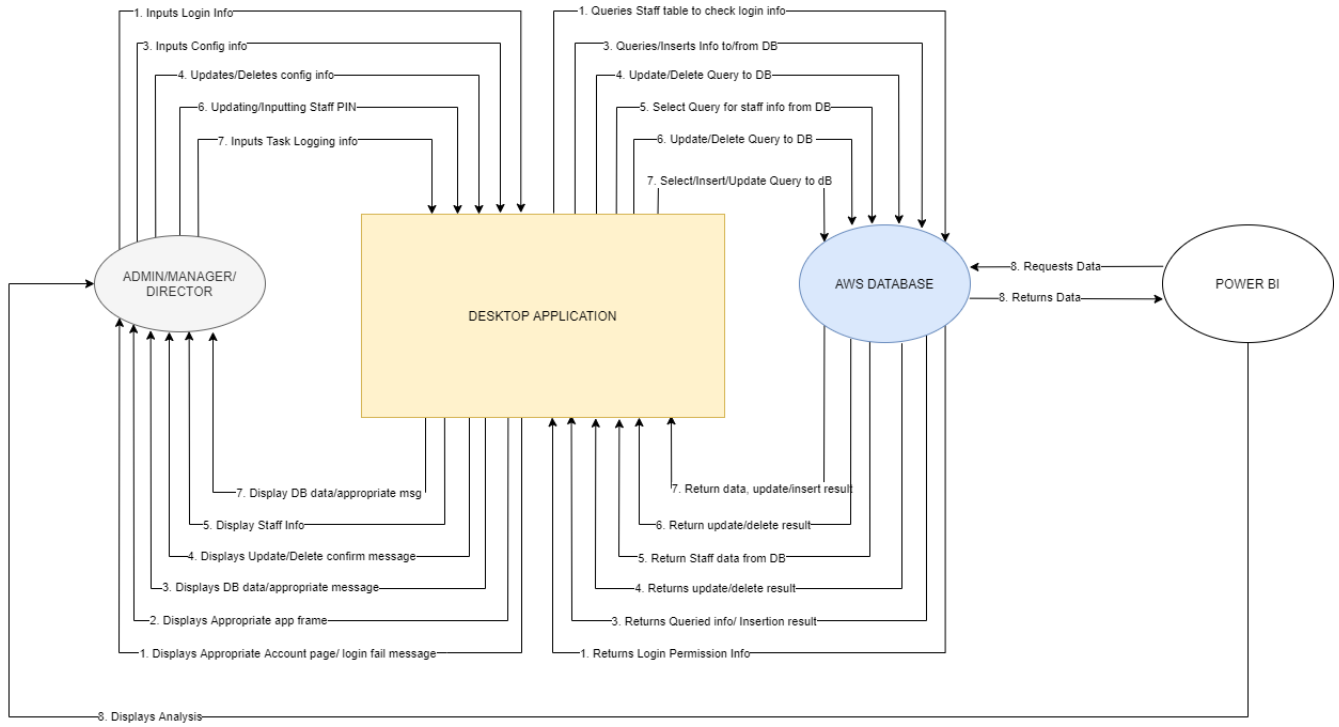


Figure 3.2.1.2.

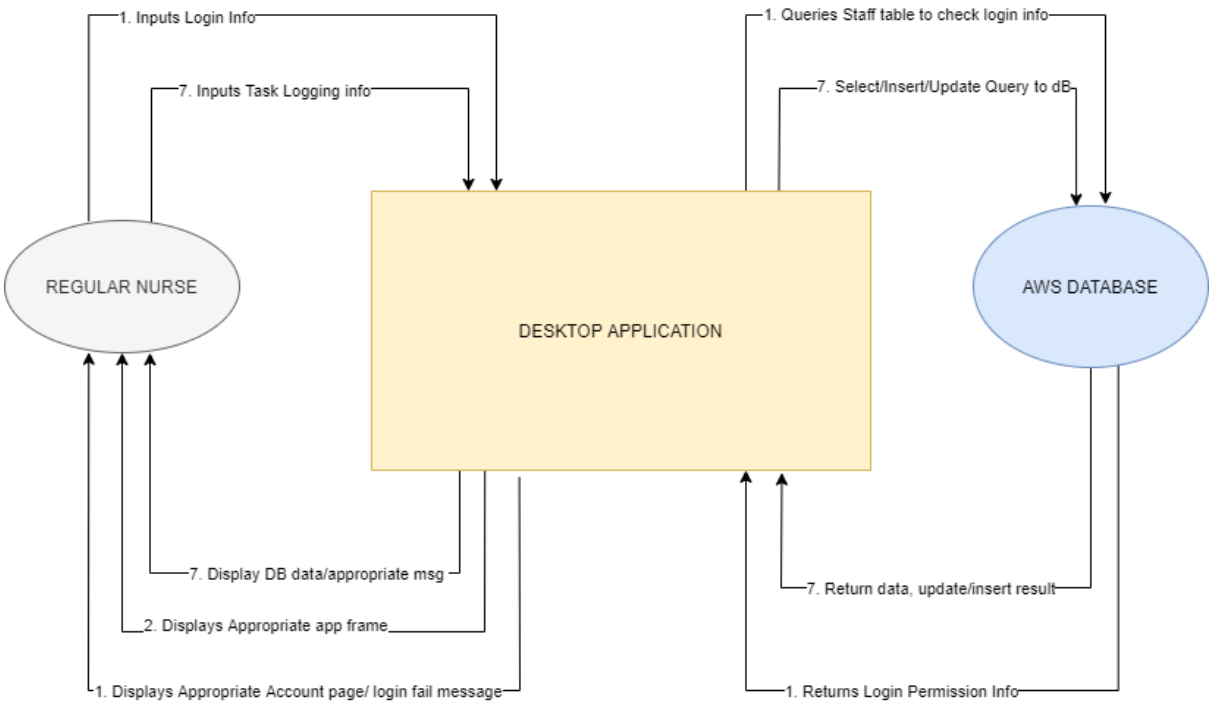
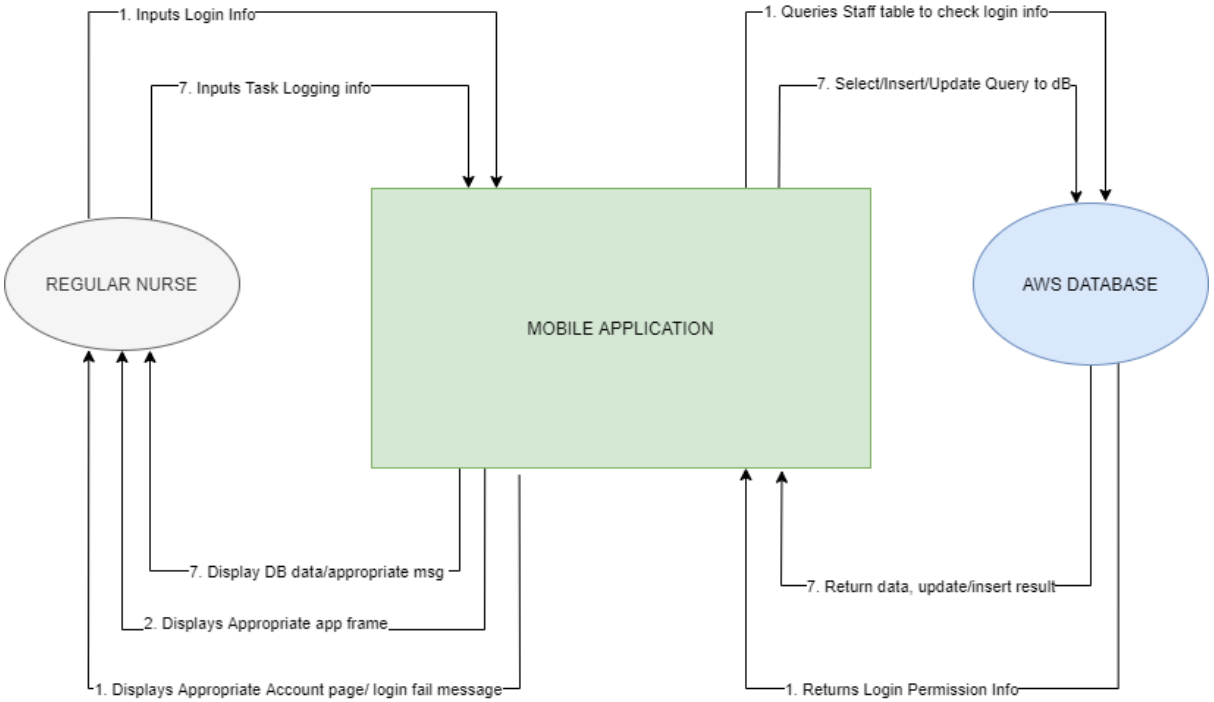


Figure 3.2.1.3.



3.2.2. DFD

Figure 3.2.2.1 represents a DFD for our project. The configuration process has been simplified in this diagram and can be seen in detail in Figure 3.2.2.2.

Figure 3.2.2.1. - System DFD

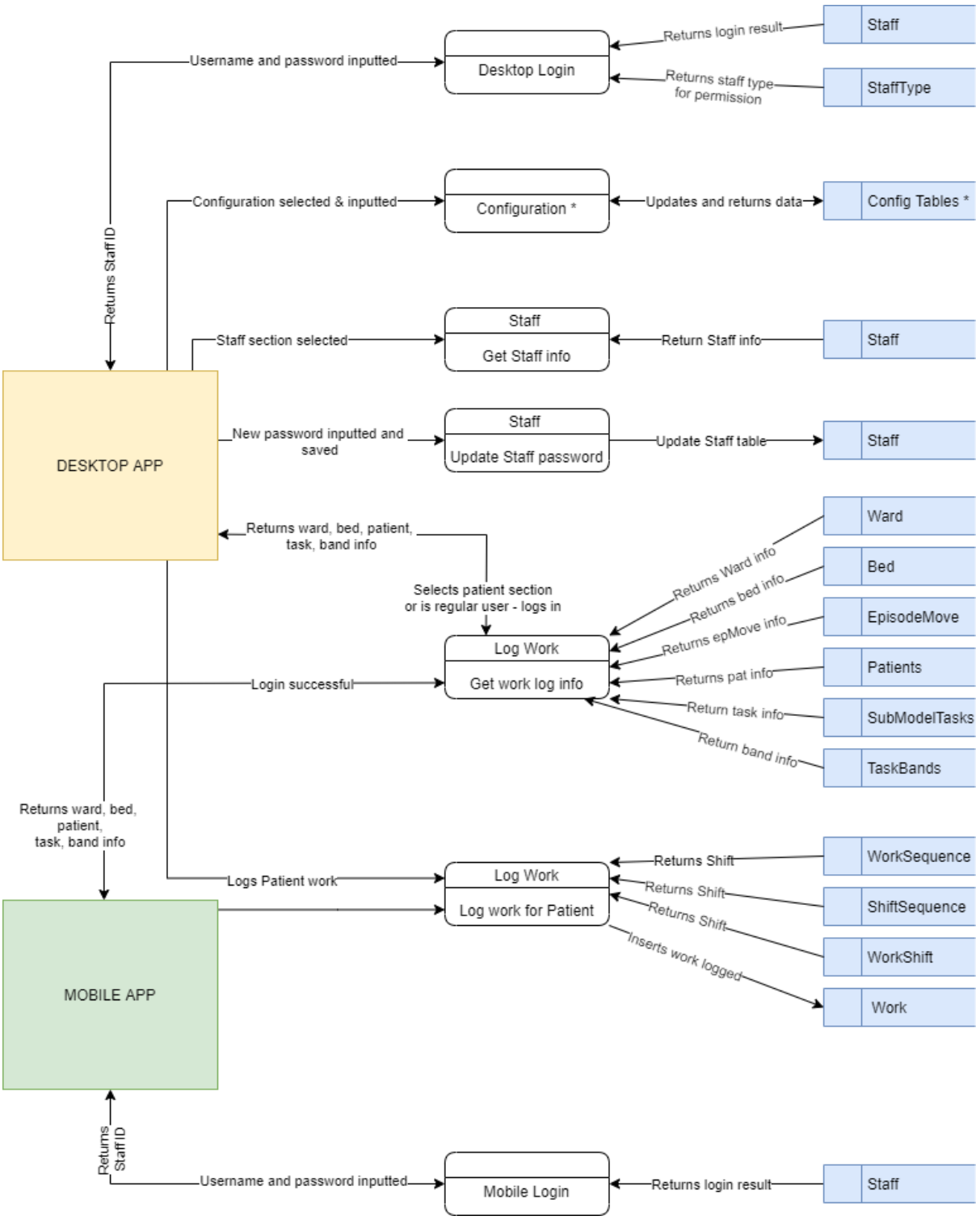
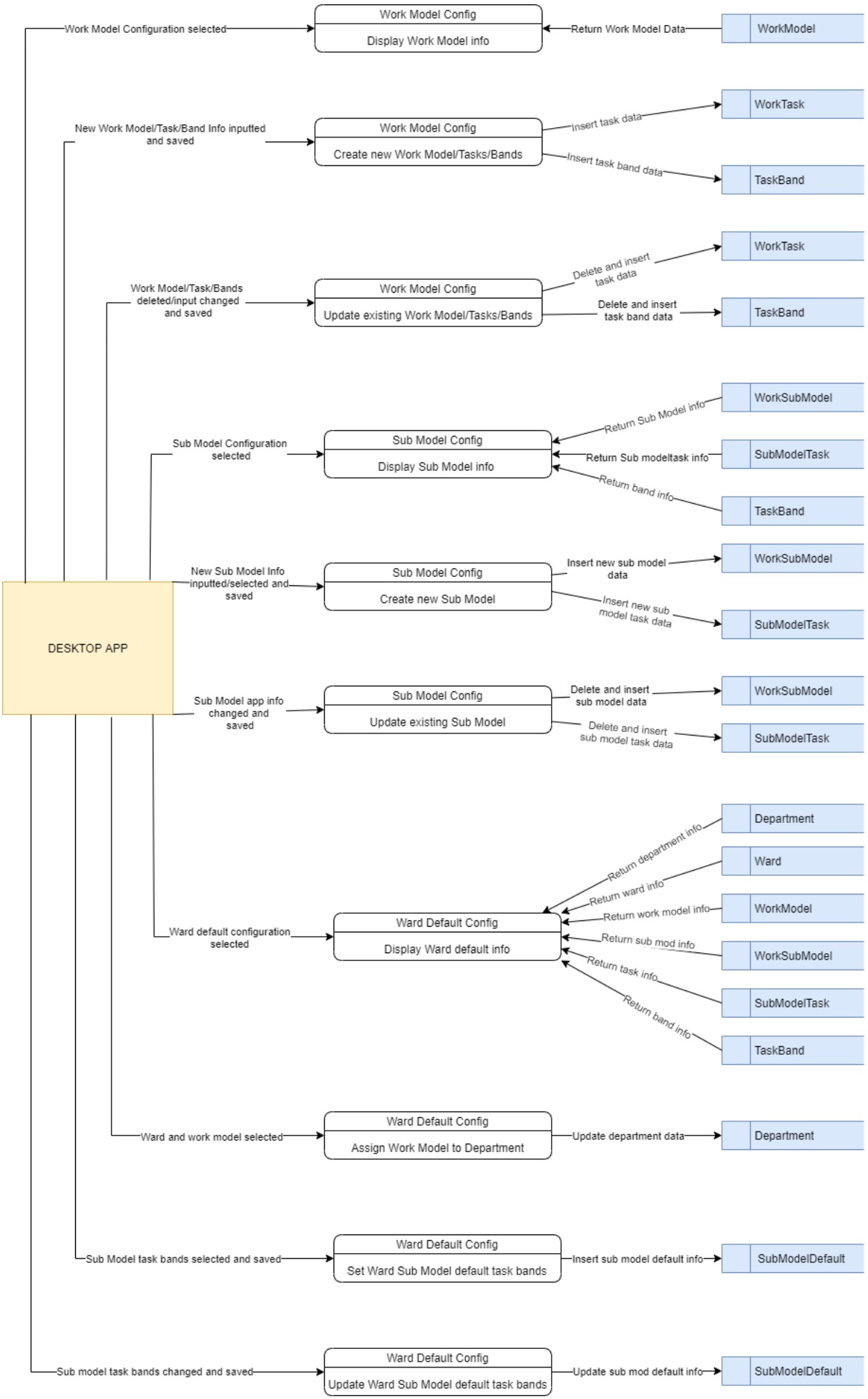


Figure 3.2.2.2. - DFD detailing configuration processes


DESKTOP APP
Work Model Configuration selected
Work Model Config
Display Work Model info
Return Work Model Data
WorkModel
New Work Model/Task/Band Info inputted and saved
Work Model Config
Create new Work Model/Tasks/Bands
Insert task data
WorkTask
Insert task band data
TaskBand
Work Model/Task/Bands deleted/input changed and saved
Work Model Config
Update existing Work Model/Tasks/Bands
Delete and insert task data
WorkTask
Delete and insert task band data
TaskBand
Sub Model Configuration selected
Sub Model Config
Display Sub Model info
Return Sub Model info
WorkSubModel
Return Sub modeltask info
SubModelTask
Return band info
TaskBand
New Sub Model Info inputted/selected and saved
Sub Model Config
Create new Sub Model
Insert new sub model data
WorkSubModel
Insert new sub model task data
SubModelTask
Sub Model app info changed and saved
Sub Model Config
Update existing Sub Model
Delete and insert sub model data
WorkSubModel
Delete and insert sub model task data
SubModelTask
Ward default configuration selected
Ward Default Config
Display Ward default info
Return department info
Department
Return ward info
Ward
Return work model info
WorkModel
Return sub mod info
WorkSubModel
Return task info
SubModelTask
Return band info
TaskBand
Ward and work model selected
Ward Default Config
Assign Work Model to Department
Update department data
Department
Sub Model task bands selected and saved
Ward Default Config
Set Ward Sub Model default task bands
Insert sub model default info
SubModelDefault
Sub model task bands changed and saved
Ward Default Config
Update Ward Sub Model default task bands
Update sub mod default info
SubModelDefault

3.2.3. Class Diagram

3.2.4. Database Relationship Diagrams

Fig 3.2.4.1 - Full database diagram

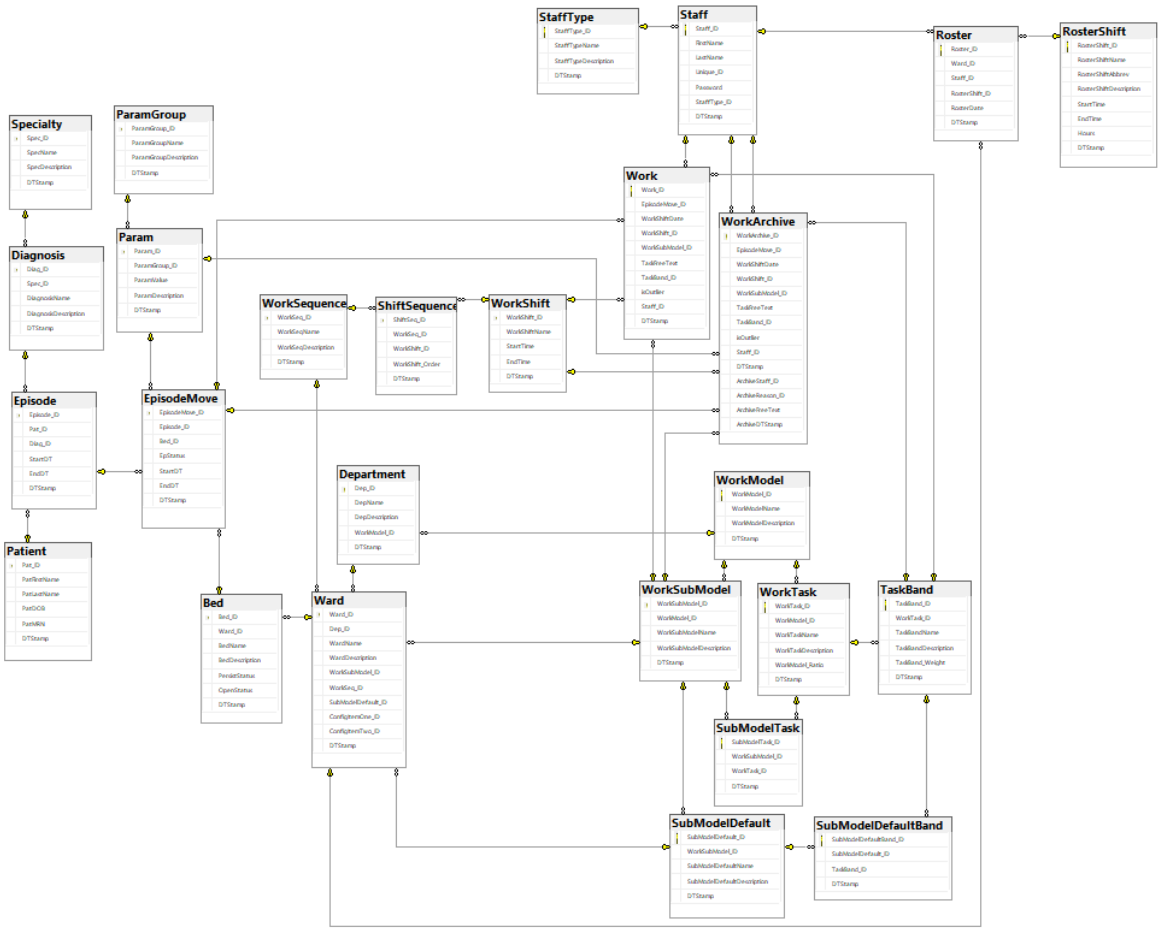


Figure 3.2.4.2 - tables created specifically for our application

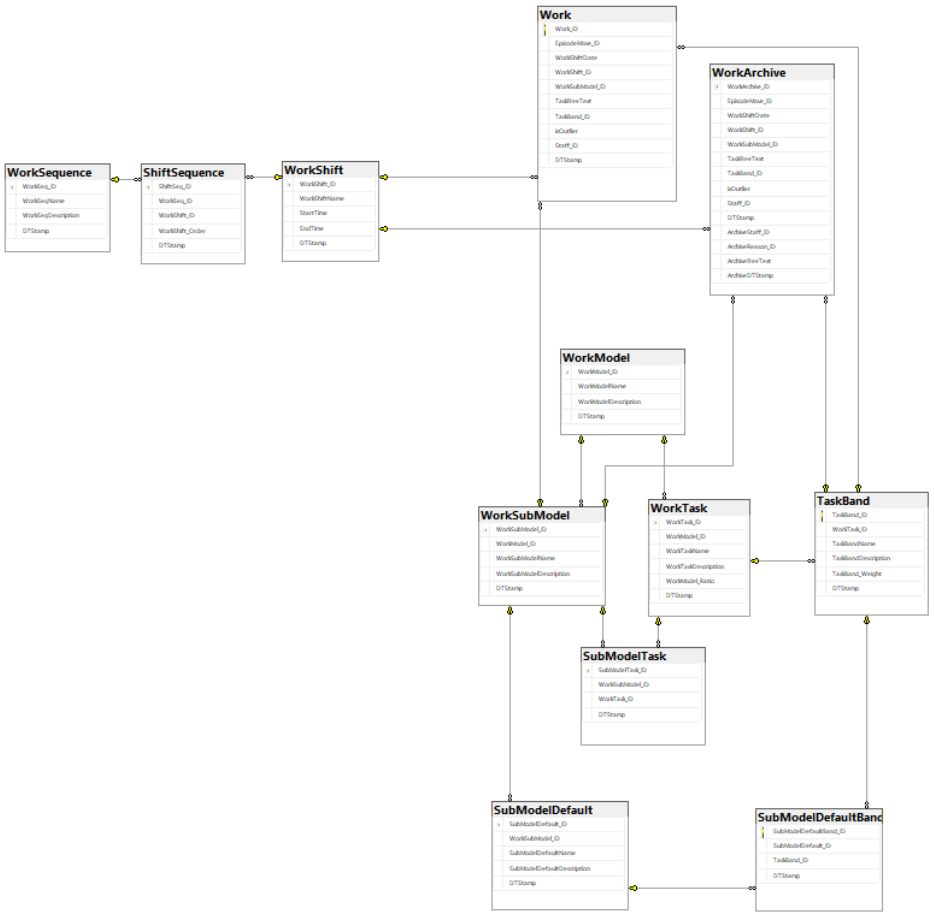
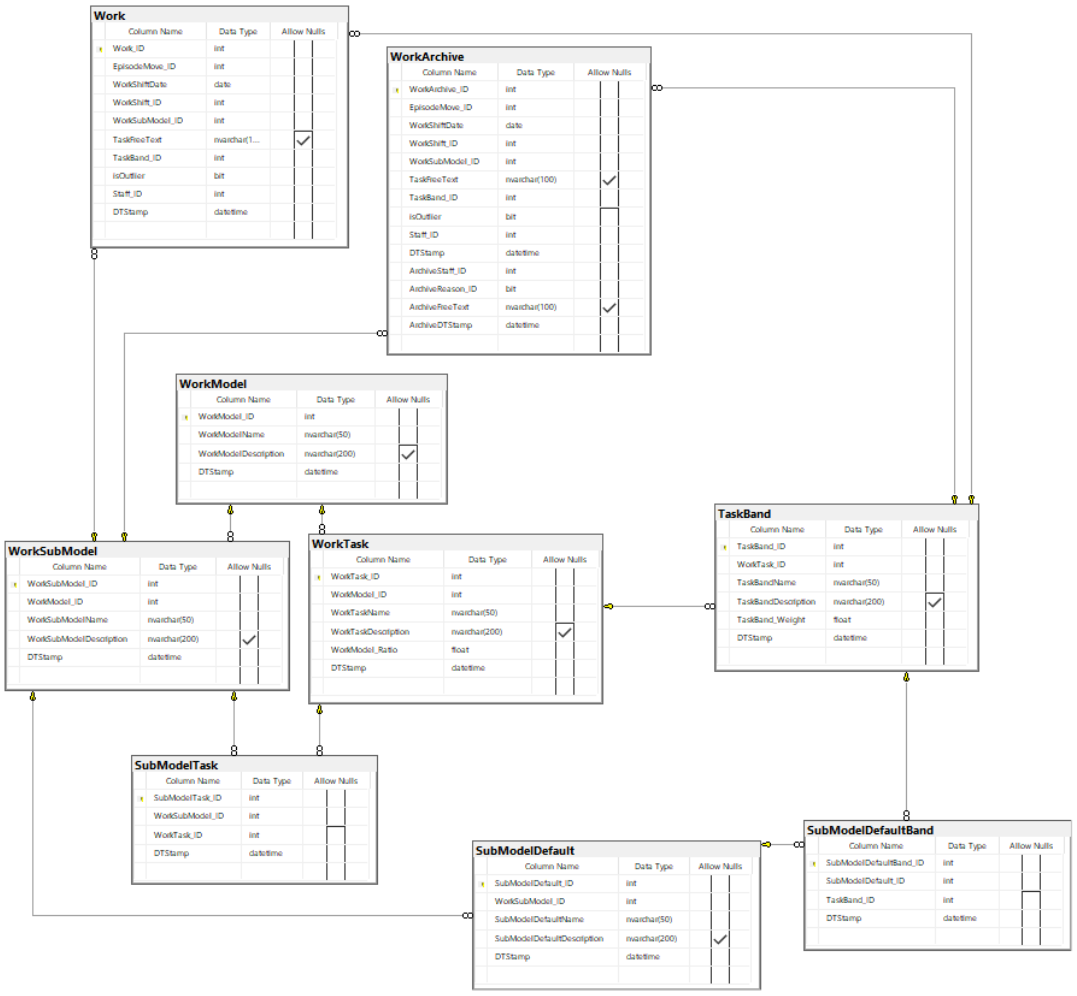


Figure 3.2.4.3 - core tables to our application



4. Implementation

4.1. Desktop Application

Our desktop application consists of 5 java files. HomePage.java is the main file that runs when the application is executed. The GUI for this file consists of a Java Swing Jframe which displays a login screen.

Upon executing this file, a connection is made to the database, and 2 lists are retrieved from the database; a list of staff IDs for staff who have admin user permission, and a list of staff IDs for staff who have regular user permissions. When a user enters a username and password and clicks the login button, the credentials are checked and the correct page is

displayed for the user, depending on their user permissions. For admin users, AccountHomePage.java is executed, which includes all features of our app. For regular users, NurseWorkLoggingAcc.java is executed, which only allows the user to log work for patients.

AccountHomePage.java has 4 sections. Instead of writing 4 java files, one for each section, all of the methods belonging to all 4 sections are written inside the one file, AccountHomePage.java. From a coding perspective, this is not ideal, but we chose to write our programme this way because breaking it up into 4 different files would cause our app to dispose of the current window and open a new window, every time a user clicked into a new section of the app. This makes the app look “blinky” and unappealing to the user. So we chose user satisfaction over a better coding practice to solve this dilemma.

4.1.1. Desktop-database Connectivity

This project incorporates a desktop application and an AWS RDS database. In order to generate a connection between both these components, JDBC was implemented. The JDBC driver was used in order to execute objects in the JDBC API. The objects of the JDBC API applied in our project were the Connection object, Statement/Prepared Statement/Callable Statement objects and the ResultSet object. The connection itself was created using the DriverManager class. First the JDBC driver was loaded.

HomePage.java - line 22

```
private String dbDriver = "com.microsoft.sqlserver.jdbc.SQLServerDriver";
```

We then created a String “dbURL” which represents the database URL of our system necessary for the connection.

HomePage - line 24

```
private static String dbURL= "jdbc:sqlserver://database-1.chx79b5ss8ae.us-east-1.rds.amazonaws.com:1433;"
        + "user=adminUsername;"
        + "password=adminPassword;"
        + "databaseName=WMS";
```

Class.forName() allows the initialization of the JDBC SQLServerDriver class. Following this, the connection is successfully created by using the getConnection method of the DriverManager class. It generates a connection with the first driver it comes to that can correctly connect to the database URL provided.

HomePage.java - line 241

```
Class.forName(dbDriver);
```

HomePage.java - line 247

```
Connection conn = DriverManager.getConnection(dbURL);
```

4.1.2. Login

In order to access the features that the desktop application provides, a user needs to first successfully login. The login page requests the user to enter a username and a password through java swing text field and JPasswordField. The application stores the text inputted by the user, the username as a String and the password as a character array.

HomePAge.java - line 279

```
String user = usernameField.getText();
char[] pass = passwordField.getPassword();
```

The isPasswordCorrect(), getPassword() and getStaffType() methods returns a boolean value as to whether or not the password and username entered are correct. getPassword() sends a SQL query to the database to the existing password corresponding to the username entered. This password is added to another character list and is compared to the one which holds the password entered using Arrays.equal(). If they, true boolean value is returned, otherwise false. The password array is filled with 0s after it is used for security purposes.

HomePage.java - line 347

```
protected static boolean isPasswordCorrect(Connection conn, String user, char[] input) {     // https://docs.oracle.com/javase/tutorial/uiswing
    boolean isCorrect = false;
    Integer staffType = getStaffType(conn, user);
    String password = getPassword(conn, user);

    char[] correctPassword = new char[password.length()];   // user entered pw is array so we create array of DB gathered pw to compare
    for (int i = 0; i < password.length(); i++) {
        correctPassword[i] = password.charAt(i);
    }
    isCorrect = Arrays.equals (input, correctPassword);     // bool isCorrect = are they identical
    Arrays.fill(correctPassword,'0');   // zero out password so java code not storing it in variable
    return isCorrect;
}
```

getStaffType() is used to return the staff type to the application and store it as a variable. If the password and username are both correct, the application checks the staff type and displays the appropriate account JFrame to the user.

HomePage.java - line 315

```
protected static Integer getStaffType(Connection conn, String user) {
    Integer staffType = 0;
    try {
        Statement selectStaffTypeStmt = conn.createStatement();     // initialize statement
        ResultSet rs = selectStaffTypeStmt.executeQuery(String.format("SELECT StaffType_ID FROM Staff WHERE Unique_ID = '%s'", user))
        while (rs.next()) {
            staffType = rs.getInt(1);
            System.out.println("staffType: " + staffType);
        }
    }
    catch (SQLException e) {
        System.out.println("SQL Exception caught: " + e.getMessage());
    }
    return staffType;
}
```

4.1.3. Configuration

Before the application can be successfully utilized, configuration must be undertaken by the system administrator. There are three stages in the configuration process; Work Model configuration, Sub Model configuration and Ward default configuration.

4.1.3.1. Work Model Configuration

Work Model configuration involves creating or updating a Work Model (see user guide for detailed explanation). In order to implement Work Model configuration, the user is first given the option to input a Work Model name into a java swing TextField. The app queries the database for a WorkModel_ID corresponding to the name entered and stores the result in the variable “id”. The returnId() method in DatabaseHelperSP returns 0 if the no id is returned indicating that the WorkModel name does not exist. If “id” does not equal 0, the name entered already exists in the database and the existing Work Model information is displayed to the user. Otherwise, the name entered does not already exist and a new Work Model is created, the database updated and the user notified accordingly. This is achieved through SQL queries through stored procedures as well as JOptionPane dialog box from java swing.

AccountHomePage.java - line 1620

```
String pieChartName = pieChartNameTF.getText();
if (pieChartName.equals("")) {
    JOptionPane.showMessageDialog(this, "Field cannot be blank. Please enter a Pie Chart name!");
}

else {
    taskTablePieNameLabel.setText(pieChartName);
    //Add current tasks from db table

    //returnID(name) is called to check if the name entered exists in the WorkModel table and returns it's WorkModel_ID is so, otherwise it
    Integer id = DatabaseHelperSP.returnID(HomePage.conn, pieChartName);
    System.out.println("work model id: " + id);
    //if id is not 0 that means returnID() retuned an existing WorkModel_ID associated to the pieChartName entered
    if (id != 0) {
        System.out.println(pieChartName + ": This model name exists in the table so we will be adding to it, not creating a new model.");
        displayExistingPieChartTasks(id);
    }
    else {
        //The pieChartName variable does not exist already and must be added to the WorkModel table
        System.out.println(pieChartName + ": This name does not exist so we will be creating a new model and adding new tasks to it.");
        prepareGUITableForNewModel(pieChartName);
    }
}
```

DatabaseHelperSP.java - line 22

```
// return WorkModel_ID for the argument name entered
public static int returnID(Connection conn, String name){
    CallableStatement cstmt = null;
    Integer returnid = 0;
    try {
        System.out.println("attempting method: returnID");

        cstmt = conn.prepareCall("{call ReturnWorkModelID(?)}");
        cstmt.setString(1, name);
        cstmt.executeQuery();
        ResultSet rs = cstmt.getResultSet();

        if(rs.next()){
            returnid = rs.getInt(1);
        }
        else {
            System.out.println("returnID: DOES NOT EXIST!!");
        }

    }
    catch (Exception e) {
        System.out.println("ERRORRRRR!!");
        System.out.println("LINE NUMBER: " + getLineNumber());
        System.out.println(e.getMessage());
    }
    return returnid;
}
```

Subsequent to this, the system allows tasks to be created for the Work Model specified in the previous step. This is achieved through the user inputting task information (task name, task ratio and task description) and selecting “Add”. This is achieved by a JTable and DefaultTableModel which has an addRow() method which stores the input in a table given the text fields are not empty and the task doesn’t already exist. Once the user has entered their chosen task information, they can choose to save changes in order to save this information to the system. Due to the nature of our application, the task ratios MUST sum to 100. The method getSumAllTasks() is called to find the total of the ratios inputted and returns the result to a variable called “ratioSum”. If “ratioSum” is greater than 100, the user is notified accordingly via JOptionPane and they will not be able to save their changes until they meet this requirement. The user can delete tasks before they save their changes. This is done by removing the task from the JTable using DefaultTableModel.removeRow() as well as adding any task name deleted to a java List called “deletedTasks”. Once ‘save changes’ is carried out successfully, any tasks in List “deletedTasks” are deleted from the database and all new task information is inserted into the database through SQL queries in stored procedures.

AccountHomePage.java - line 1505

```
private void addTaskButtonActionPerformed(java.awt.event.ActionEvent evt) {
    boolean taskExists = doesTaskExist();   //find out whether the task name entered already exists in this primary model
    taskTblModel = (DefaultTableModel)taskTable.getModel();
    if (pieChartNameTF.getText().equals("") || taskNameTF.getText().equals("") || taskWeightTF.getText().equals("")
            || taskDescriptionTF.getText().equals("")) {
        JOptionPane.showMessageDialog(this, "Fields cannot be blank. Please enter all data!");
    }
    else if (taskExists) {
        JOptionPane.showMessageDialog(this, "This task name already exists and cannot be added into the table!");
    }
    else {
        String data [] = {taskNameTF.getText(), taskWeightTF.getText(), taskDescriptionTF.getText()};
        taskTblModel = (DefaultTableModel)taskTable.getModel();
        taskTblModel.addRow(data);
        JOptionPane.showMessageDialog(this, "Make sure to select this task from the list to create task bands!! ");
        clearTaskNameTFTaskWeightTF();
    }
}
```

AccountHomePage.java - line 1875

```
private void deleteTaskButtonMouseClicked(java.awt.event.MouseEvent evt) {
    //**** IMPORTANT TO NOTE: We want to delete tasks from GUI tabel ONLY!!!
    //Deletions to the database are carried out when the user selects "save changes".
    taskTblModel = (DefaultTableModel)taskTable.getModel();
    String taskToDelete = JOptionPane.showInputDialog(this, "Please enter the task name to delete:");
    System.out.println("task to be deleted is: " + taskToDelete);

    deleteTaskThatUserEntered(taskToDelete);
}
```

AccountHomePage - line 1861

```
protected void deleteTaskThatUserEntered(String taskToDelete) {
    for (int i = 0; i < taskTblModel.getRowCount(); i++){
        String t = taskTblModel.getValueAt(i, 0).toString();
        if (taskToDelete.equals(t)) {
            taskTblModel.removeRow(i);
            deletedTasks.add(t);
            JOptionPane.showMessageDialog(this, taskToDelete + " was deleted");
        }
        else {
            System.out.println(taskToDelete + " was not deleted");
        }
    }
}
```

AccountHomePage - line 1925

```
private void saveTaskChangesButtonActionPerformed(java.awt.event.ActionEvent evt) {
    //ratioSum created to find sum of task weights in the Jtable
    taskTblModel = (DefaultTableModel)taskTable.getModel();
    float ratioSum = getSumAllTasks();

    //if the weights sum up to more than 100, pie chart cannot be made
    //return appropriate message to the user
    if (ratioSum != 100) {
        JOptionPane.showMessageDialog(this, "Your tasks need to sum up to 100 in order to be saved!");
    }   // this currently only tells user they need to = 100. doesnt actually MAKE them yet.

    else {
        //delete any previously deleted tasks from the table
        //note that the list called deletedTasks contains the task name of all deleted tasks from the JTable
        deleteTasksInDeletedList();
        saveTaskChanges();
    }
}
```

AccountHomePage - line 1905

```
protected void saveTaskChanges () {
    //Add to WorkTask table
    //need to know: WorkModel_ID, WorkTaskName, WorkTaskDescription, WorkModel_Ratio, DTStamp
    for (int i = 0; i  <taskTblModel.getRowCount(); i++) {
        String taskName = (String)taskTblModel.getValueAt(i, 0);
        Float taskWeight = Float.parseFloat(taskTblModel.getValueAt(i, 1).toString());
        String taskDescr = (String)taskTblModel.getValueAt(i, 2);

        addNewTask(taskName, taskWeight, taskDescr);
    }
    JOptionPane.showMessageDialog(this, "Your changes have been saved!");
}
```

AccountHomePage.java - line 1885

```
protected void addNewTask(String taskName, Float taskWeight, String taskDescription) {
    //check if each task name is already in the workTask db table by calling the checkTaskName method
    ResultSet rs = DatabaseHelperSP.checkWorkTaskName(HomePage.conn, taskName, workModelId);
        try {
            //if the worktask table does contain the name and workModel_id, the task does exist and must not be changed as it was never delet
            if (rs.next()) {
                System.out.println("Task name already exists and does not need to be changed");
            }
            else {
                //this task is new and can be inserted!
                System.out.println("Adding task !");
                DatabaseHelperSP.insertIntoTaskTable(HomePage.conn, workModelId, taskName, taskDescription, taskWeight);
                System.out.println("new task added");
            }
        }
        catch (SQLException e) {
            System.out.println(e.getMessage());
        }
}
```

AccountHomePage.java - line 1918

```
protected void deleteTasksInDeletedList() {
    for (int i = 0; i < deletedTasks.size(); i++) {
        //if deleted task name exists in db, delete from WorkTask table in db.
        String deletedTask = deletedTasks.get(i);
        //Note: the second arg is workModelId. This is to avoid deleting any tasks with the same name that exist in a different WorkModel.
        DatabaseHelperSP.deleteTask(HomePage.conn, deletedTask, workModelId);
    }
}
```

DatabaseHelperSP.java - line 271

```
//insert new task into the WorkTask table
public static void insertIntoTaskTable(Connection conn, Integer workModelId, String taskName, String taskDescr, Float workModelRatio){
    CallableStatement cstmt = null;
    try{
        System.out.println("attemptong method: insertIntoTaskTable");

        cstmt = conn.prepareCall("{call insertTask(?, ?, ?, ?)}");
        cstmt.setInt(1, workModelId);
        cstmt.setString(2, taskName);
        cstmt.setString(3, taskDescr);
        cstmt.setFloat(4, workModelRatio);
        int rowCount = cstmt.executeUpdate();
        if(rowCount>0){
            System.out.println("Row updated");
        }
    }catch(SQLException e) {
        System.out.println("insertIntoTaskTable: ERROR!");
        System.out.println("LINE NUMBER: " + getLineNumber());
        System.out.println(e.getMessage());
    }
}
```

DatabaseHelperSP.java - line 366

```
//if the taskName= name in the deletedTasks list exists in the WorkTask table, delete it from the WorkTask table
public static void deleteTask(Connection conn, String taskName, Integer id){
    CallableStatement cstmt = null;
    try {
        System.out.println("attempting method: deleteTask");
        cstmt = conn.prepareCall("{call DeleteTask(?, ?)}");
        cstmt.setString(1, taskName);
        cstmt.setInt(2, id);
        int rowCount = cstmt.executeUpdate();
        if(rowCount>0){
            System.out.println("Row updated. task deleted");
        }
        else {
            System.out.println("0 rows updated. task did not exist in DB so did not need to be deleted.");
        }

    }catch(SQLException e) {
        System.out.println("deleteTask: ERROR! A problem prevented deletion!");
        System.out.println("LINE NUMBER: " + getLineNumber());
        System.out.println(e.getMessage());
    }
}
```

If tasks have been successfully configured, the next step is to configure each task's bands. Similar to above, the user inputs the band information (task band name, description and weight) into TextFields and these are stored as variables in the app. Once the information is entered, an "Add" button adds it to a table displayed. This is achieved by a JTable and DefaultTableModel which has an addRow() method which stores the input in a table. Before the band can be added, an 'if' condition checks that the text fields are not empty, that the band weight is between 0 and 1 and that a band name doesn't already exist. The user is notified via JOptionPane message dialogs.

AccountHomePage.java - line 1720

```
private void addBandButtonActionPerformed(java.awt.event.ActionEvent evt) {
    //***** IMPORTANT TO NOTE: the bands are not added to the db here as there might be deletions.
    // Instead they are only added to the GUI table here and once they select the "save changes" button, insertions are made to the db.
    boolean bandExists = doesBandExist();   //find out whether the band name entered already exists
    bandTblModel = (DefaultTableModel)bandTable.getModel();

    if (bandNameTF.getText().equals("") || bandWeightTF.getText().equals("")) {
        JOptionPane.showMessageDialog(this, "Please enter all data!");
    }
    else if( Integer.parseInt(bandWeightTF.getText())>1 || Integer.parseInt(bandWeightTF.getText())<0) {
        JOptionPane.showMessageDialog(this, "Your band weight can only be between 0 and 1!");
    }
    else if (bandExists) {
        JOptionPane.showMessageDialog(this, "This band name already exists and cannot be created for the task!");
    }
    else {
        String data [] = {bandNameTF.getText(), bandWeightTF.getText()};

        bandTblModel = (DefaultTableModel)bandTable.getModel();
        bandTblModel.addRow(data);

        clearBandNameTFBandWeightTF();
    }
}
```

Once a user has added their task band information, the “Save Changes” button can be selected and task band information inputted is updated in the database. Before this, the app calls ensureNoBandsEqual() which returns a boolean as to whether no two band weights are equal in the table. If this is the case, deleteBandsInDeletedList() is called to remove any bands in the database which were deleted from the JTable. Following this, saveBandChanges() is called which adds all new task band information to the appropriate table in the database.

AccountHomePage.java - line 1817

```
private void saveChangesButtonActionPerformed(java.awt.event.ActionEvent evt) {
    bandTblModel = (DefaultTableModel)bandTable.getModel();
    //find the sum of all task weights present in the JTable
    float ratioSum = getSumAllBands();

    if (!ensureNoBandsEqual()){
        JOptionPane.showMessageDialog(this, "You cannot have two of the same band weights!");
    }
    else {
        deleteBandsInDeletedList();
        saveBandChanges();
    }
}
```

AccountHomePage.java - line 1909

```
protected void deleteBandsInDeletedList() {
    for (int i = 0; i < deletedBands.size(); i++) {
        //if band in DeletedBands list exists in db, delete from TaskBand table in DB
        String s = deletedBands.get(i);
        DatabaseHelperSP.deleteBand(HomePage.conn, s, workTskId);
    }
}
```

AccountHomePage.java - line 1897

```
protected void saveBandChanges() {
    bandTblModel = (DefaultTableModel)bandTable.getModel();
    //do all of the following for each band in bandTable
    for (int i = 0; i < bandTblModel.getRowCount(); i++) {
        String bandName = (String)bandTblModel.getValueAt(i, 0);
        Float bandWeight = Float.parseFloat(bandTblModel.getValueAt(i, 1).toString());

        addNewTaskBand(bandName, bandWeight); // works one band at a time
    }
    JOptionPane.showMessageDialog(this, "Your changes have been saved!");
}
```

AccountHomePage.java - line 1877

```
protected void addNewTaskBand(String bandName, Float bandWeight) {
    //checkTaskBand() method which takes the workTask_ID, name and weight and returns a result set of bands
    ResultSet rs = DatabaseHelperSP.checkTaskBandName(HomePage.conn, workTskId, bandName, bandWeight);
    try {
        //if this current taskband already exists in the TaskBand table then it does not need to be inserted again
        if (rs.next()) {
            System.out.println("Band exists. No changes needed!!");
        }
        else {
            //this current rs.next() taskband is new and must be inserted into the table
            System.out.println("Adding band!");
            DatabaseHelperSP.insertIntoTaskBandTable(HomePage.conn, workTskId, bandName, "description", bandWeight);
            // NEED TO ADD: need to allow users to enter descr
        }
    }
    catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
```

DatabaseHelperSP.java - line 389

```
//if the bandName= name in the deletedTasks list exists in the TaskBAnd table, delete it from the TaskBAnd table
public static void deleteBand(Connection conn, String bandName, Integer taskId){
    CallableStatement cstmt = null;
    try {
        System.out.println("attempting method: deleteBand");
        cstmt = conn.prepareCall("{call DeleteBand(?, ?)}");
        cstmt.setString(1, bandName);
        cstmt.setInt(2, taskId);
        int rowCount = cstmt.executeUpdate();
        if(rowCount>0){
            System.out.println("Row updated. band deleted: " + bandName);
        }
        else {
            System.out.println("0 rows updated. band did not exist in DB so did not need to be deleted.");
        }
    }
    catch(SQLException e) {
        System.out.println("deleteBand: ERROR! A problem prevented deletion!");
        System.out.println("LINE NUMBER: " + getLineNumber());
        System.out.println(e.getMessage());
    }
}
```

DatabaseHelperSP.java - line 271

```
//insert new band into the TaskBand table
public static void insertIntoTaskBandTable(Connection conn, Integer workTskId, String taskBandName, String taskBandDescription, Float workTskWeight){
    CallableStatement cstmt = null;
    try{
        System.out.print("attempting method: insertIntoTaskBandTable");

        cstmt = conn.prepareCall("{call insertTaskBand(?, ?, ?, ?)}");
        cstmt.setInt(1, workTskId);
        cstmt.setString(2, taskBandName);
        cstmt.setString(3, taskBandDescription);
        cstmt.setFloat(4, workTskWeight);
        int rowCount = cstmt.executeUpdate();
        if(rowCount>0){
            System.out.println("Row updated");
        }
    }catch(SQLException e){
        System.out.print("insertIntoTaskBandTable: Error: ");
        System.out.println("LINE NUMBER: " + getLineNumber());
        System.out.println(e.getMessage());
    }
}
```

4.1.3.2. Sub Model Configuration

Sub Model configuration involves creating or updating a Sub Model (see user guide for detailed explanation). In order to implement Sub Model configuration, the user must first select the Work Model from which they wish to derive the Sub Model from (from JComboBox). If no Work Models exist in the database, none have been successfully configured and the user will be notified accordingly. An itemStateChanged event listener is utilized and once the item state is changed to a Work Model name in the JComboBox. JComboBox.getSelectedItem() is stored as variable workModNameCombo representing the Work Model name selected. The name is used as a parameter in DatabaseHelperSP.returnId() to update the “workModelId” variable to the corresponding Work Model Name chosen. The method displayTasksForSelectedWorkModel() is called to display all Work Model configured tasks to the table.

AccountHomePage.java - line 1953

```
private void subModelComboBoxItemStateChanged(java.awt.event.ItemEvent evt) {
    wrkTaskTblModel.setRowCount(0);
    //this if statement ensures that the code within is called ONLY when the comboBox contains WorkModelNames in it
    //it solves an error which occurs when we call subModelComboBox.removeAllItems in wardPieSelectionActionPerformed(). When subModelComboBox.removeAllItems is called,
    //it technically changes item state as it removes them all. However we need to remove all otherwise we will have duplicate WorkModelNames in the ComboBox every time w
    if (subModelComboBox.getItemCount() > 1) {
        //this bool will be used to in enterSubModelName() to ensure that a WorkModel has been selected before a sub model name is entered.
        subModComboClicked = true;
        String workModNameCombo = subModelComboBox.getSelectedItem().toString();
        //update the WorkModel_ID to that of the selected WorkModelName
        workModelId = DatabaseHelperSP.returnID(HomePage.conn, workModNameCombo);
        //the below method is called so we can check if any WorkTask items exist in the WorkTask table associated with the WorkModel_ID
        displayTasksForSelectedWorkModel();
    }
}
```

AccountHomePage.java - line 1931

```
protected void displayTasksForSelectedWorkModel() {
    ResultSet rs = DatabaseHelperSP.checkWorkTask(HomePage.conn, workModelId);
        try {
            //use while as it is possible for a WorkModel object to have more than one task and we want to return all of them to the GUI table
            while (rs.next()) {
                System.out.println("The workModel name has tasks in the table!");
                //Get each tasks name and corresponding ratio
                String taskName = rs.getString("WorkTaskName");
                Float ratio = rs.getFloat("WorkModel_Ratio");
                String taskDesc = rs.getString("WorkTaskDescription");
                System.out.println("name: " + taskName + " ratio: " + ratio);
                //create the data list so that the task names and ratios can be added to the GUI table
                String data [] = {taskName, ratio.toString(), taskDesc};
                wrkTaskTblModel = (DefaultTableModel)subPrimTaskDisplayTable.getModel();
                wrkTaskTblModel.addRow(data);
            }
        }
        catch (SQLException e){
            System.out.println(e.getMessage());
        }
}
```

Once the user chooses the Work Model, they then input the name of the Sub Model which they wish to configure. If the name entered already exists in the database, the existing Sub Model information is displayed to the user. If the name entered does not already exist, a new Sub Model is created, the database updated and the user notified accordingly. This is achieved with SQL queries through stored procedures as well as java swing classes (e.g. JComboBox, JOptionPane, etc.). Tasks are displayed on a JTable and added using javax.swing.table.DefaultTableModel methods.

AccountHomePage.java - line 2074

```
protected void displayExistingSubModelTasks(Integer subId) {
    System.out.println(subId);
    //Create result set rs to check if the subModel_ID has any existing WorkTask_IDs associated with it from SubModelTask table
    ResultSet rs = DatabaseHelperSP.checkWorkSubTask(HomePage.conn, subId);
    try {
        while (rs.next()) {
            System.out.println("The workSubModel name has tasks in the table!");
            Integer taskId = rs.getInt("WorkTask_ID");
            //Now find all the WorkTask info associated with the current WorkTask_ID
            displaySubModelTaskAndWeight(taskId);
        }
    }
    catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
```

AccountHomePage.java - line 2054

```
protected void displaySubModelTaskAndWeight(Integer taskId) {
    ResultSet rs2 = DatabaseHelperSP.checkWorkTask(HomePage.conn, workModelId);
        try {
            while (rs2.next()) {
                if (taskId.equals(rs2.getInt("WorkTask_ID"))) {
                    String taskName = rs2.getString("WorkTaskName");
                    Float ratio = rs2.getFloat("WorkModel_Ratio");
                    System.out.println("name: " + taskName + " ratio: " + ratio);
                    //create the data list so that the task names and ratios can be added to the GUI table
                    String data[] = {taskName, ratio.toString()};
                    subTblModel = (DefaultTableModel)subModeljTable.getModel();
                    subTblModel.addRow(data);
                }
            }
        }
        catch (SQLException e) {
            System.out.println(e.getMessage());
        }
}
```

Subsequent to this, the system allows Work Model tasks to be chosen for the Sub Model specified in the previous step. This is achieved by adding a MouseClicked event to the table holding the Work Model task information. When a task is clicked, each piece of information in the row is stored as a String variable before being transferred to the Sub Model JTable displayed to the user. Our app uses javax.swing.table.DefaultTableModel methods such as getModel(), addRow(), getValueAt(), getSelectedRow() to do this.

Update task weights

Once tasks have been transferred to the Sub Model JTable and displayed to the user, the “Update Ratio” must be selected. This is to ensure that all task ratios sum to 100. Sub Models are never likely to include ALL the Work Model tasks and hence will need their task ratios updated accordingly. An actionPerformed event listener is utilized for the “Update Ratios” button and when clicked, we first find “sumSubTbl”, the total sum of all the task ratios in the Sub Model JTable by calling getSumAllSubModelTaskRatios(). After that, for each task, we assign a new ratio “newRatio” which is the original ratio divided by “sumSubTbl”, multiplied by 100. When this is done, we update the row in the JTable so that the updated ratio displays.

AccountHomePage.java - line 2186

```
private void subUpdateRatiosActionPerformed(java.awt.event.ActionEvent evt) {
    double sumSubTbl = getSumAllSubModelTaskRatios();
    //Iterate through the tasks chosen for the subModel in the subModel Jtable
    for (int i = 0; i < subTblModel.getRowCount(); i++) {
        //Notice we only look at row 0, ie (String)subTblModel.getValueAt(0,0) This is because once the task weight is reconfigured it is removed from th Jtable
        //and then a new row is added with it's reconfigured task ratio/weight
        //new rows are added to the bottom of the Jtable so the next task weight to be reconfured moves up to row position 0
        String taskName = (String)subTblModel.getValueAt(0, 0);
        double subWeight = Double.parseDouble(subTblModel.getValueAt(0, 1).toString());
        //this is the calculation for our new submodel pie chart, we divide the current weight by sumSubTbl(sum of weights calculated above)
        //and then multiply by 100 so that it acts as a percentage number
        String newRatio = String.format("%.2f", (subWeight/sumSubTbl)*100);
        String data2[] = {taskName, newRatio};
        subTblModel.addRow(data2);
        //as mentioned above original task row is deleted. When this happens the next task to update with become row(0)
        subTblModel.removeRow(0);
    }
}
```

Save Changes

Once a user is satisfied with their Sub Model tasks, the “Save Changes” button can be selected and the Sub Model task information is updated in the database. Before this, the app finds the sum of ratios and ensures that the “subrtioSum” does not exceed 100. If it does, the appropriate message dialog is displayed to the user to update ratios before they save. Otherwise, each Sub Model task is inserted into the database.

AccountHomePage.java - line 2223

```
private void subSaveChangesActionPerformed(java.awt.event.ActionEvent evt) {
    //ensure tasks add to 100
    float subrtioSum = getSumAllSubModelTaskRatios();
    // iterate through the jtable
    if (subrtioSum != 100) {
        JOptionPane.showMessageDialog(this, "Your sub model tasks add up to: " + subrtioSum + ". " + "Your tasks need to sum up to 100 in order to be saved! "
                + "Make sure to press the 'update task weights' button before you save your changes.");
    }
    else {

        for (int i = 0; i < subTblModel.getRowCount(); i++) { // for each task in submodel GUI table
            String subTskNme = (String)subTblModel.getValueAt(i, 0);    // harley needs help (i,0) what tey stand for
            //get worktask id
            //return workTaskID from workTask table given task name and workModel id
            Integer wrkTskId = DatabaseHelperSP.subreturnTskID(HomePage.conn, subTskNme, workModelId);

            saveSubModelChanges(wrkTskId);
        }
        JOptionPane.showMessageDialog(this, "Your changes have been saved!");
    }
}
```

AccountHomePage.java - line 2205

```
protected void saveSubModelChanges(Integer wrkTskId) {
    //check subModelTask table for the same subModel_ID and workTask_ID
    ResultSet rs = DatabaseHelperSP.checkSubWorkTaskName(HomePage.conn, subId, wrkTskId);
        try {
            if (rs.next()) {
                //if task name already exists in sub model, it does not need to be inserted
                System.out.println("Sub Task name already exists and does not need to be changed");
            }
            else {
                //insert into subModeltask table subId, workTaskID
                DatabaseHelperSP.insertIntoSubTaskTable(HomePage.conn, subId, wrkTskId);
            }
        }
        catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
```

DatabaseHelperSP.java - line 200

```
public static ResultSet checkSubWorkTaskName(Connection conn, Integer subId, Integer taskId){
    CallableStatement cstmt = null;
    ResultSet rs = null;
    try{
        System.out.println("attemptong method: checkSubWorkTaskName");

        cstmt = conn.prepareCall("{call CheckSubModelWorkTaskName(?, ?)}");
        cstmt.setInt(1, subId);
        cstmt.setInt(2, taskId);
        cstmt.executeQuery();

        rs = cstmt.getResultSet();

    }catch(SQLException e) {
        System.out.println("checkSubWorkTaskName: ERROR!");
        System.out.println("LINE NUMBER: " + getLineNumber());
        System.out.println(e.getMessage());
    }

    return rs;
}
```

4.1.3.2. Department & Ward - WorkModel and SubModel Assignment

The final area in configuration is department & ward WorkModel and SubModel assignment (See user manual for detailed description). A JComboBox displays the existing departments in the hospital, found by a SQL query to return a result set and each Department name in the result set added to the JComboBox using javax.swing.JComboBox.addItem() method. An itemStateChanged event is used to recognise when a department name has been chosen by the user. When this happens the JPanel which displays the WorkModel JTable is set to true and fillWorkModTable() method is called to populate the table with existing Work Model information. This is achieved by calling the SQL query in DatabaseHelper.returnWorkModelInfo() to return a result set and add data from that result set to the WorkModel JTable. The method fillWardComboBox() is also called which adds the Ward names to the JComboBox associated with the department chosen.

AccountHomePage.java - line 2336

```
private void deptComboBoxItemStateChanged(java.awt.event.ItemEvent evt) {
    deptTF.setText("");
    //if the count is more than one it means that departments exist in the db
    if (deptComboBox.getItemCount()>1){
        String deptName = deptComboBox.getSelectedItem().toString();
        deptTF.setText(deptName);
        int workModId = DatabaseHelper.returnDepWorkModelID(deptName);
        //if workModId is 0 it means that no work mod id exists in the db
        if(workModId!=0){deptWorkModelId = workModId;}
        //set the work model table visible to the user and fill it with work model names and descriptions
        workSubModDisplayJP.setVisible(true);
        deptWorkModJP.setVisible(true);
        wardSubModJP.setVisible(false);
        fillWorkModTable();

        clearComboBox(configWardComboBox, "Ward");
        fillWardComboBox(deptName);

        //workModelTF.setText(workModeNameExisting);
    }
}
```

AccountHomePage.java - line 2543

```
private void fillWorkModTable(){
    deptToWorkTblModel.setRowCount(0);
    ResultSet rs = DatabaseHelper.returnWorkModelInfo();
        try {
            while (rs.next()) {
                String workModName = rs.getString("WorkModelName");
                int id = rs.getInt("WorkModel_ID");
                String workModDesc = rs.getString("WorkModelDescription");
                String data [] = {workModName, Integer.toString(id), workModDesc};
                deptToWorkTblModel.addRow(data);
            }
        }
        catch (SQLException e){
            System.out.println(e.getMessage());
        }
}
```

The user is prompted by a text label to select the WorkModel they wish the department to be assigned to. A mouseClicked event is utilized on the deptWorkModelTable so that when a WorkModel is selected from the table the database is updated accordingly. The DatabaseHelper method updateDeptTable() involves the SQL query to do this.

AccountHomePage.java - line 2358

```
private void deptWorkModelTableMouseClicked(java.awt.event.MouseEvent evt) {
    // pop up message asking user if they are sure they want to add the work model to that dept
    String clickedWorkModName = (String)deptToWorkTblModel.getValueAt(deptWorkModelTable.getSelectedRow(), 0);
    String clickedWorkId = (String)deptToWorkTblModel.getValueAt(deptWorkModelTable.getSelectedRow(), 1);
    deptWorkModelId = Integer.parseInt(clickedWorkId);
    String deptName = deptTF.getText();
    //find id associated with department name
    //if yes selected update department table with
    if(JOptionPane.showConfirmDialog(this,"Save work model " + clickedWorkModName + " to department " + deptName + "?", "Save Work Model",
            JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE) == JOptionPane.YES_OPTION){
        //update table
        System.out.println("Update dept table");
        DatabaseHelper.updateDeptTable(deptWorkModelId, deptName);
        workModelTF.setText(clickedWorkModName);
        deptWorkModJP.setVisible(false);
        wardSubModJP.setVisible(false);

        fullWardComboJP.setVisible(true);
    }else{
        System.out.println("table exited and or no clicked");
    }

}
```

Once a WorkModel is chosen a ward is selected associated with the SubModel the user wishes to create default task bands for. An JComboBox.itemStateChanged() event as JTable.mouseClicked() event is used as well as further SQL queries in order to do this.

AccountHomePage.java - line 2321

```
private void configWardComboBoxItemStateChanged(java.awt.event.ItemEvent evt) {
        Display All Sub Model with the departments WorkModId
       String wardName = configWardComboBox.getItemAt(0);
       if(wardName != null && !wardName.equals("ward")){
       }
    if(configWardComboBox.getItemCount()>1){
        String wName = configWardComboBox.getSelectedItem().toString();
        wardTF.setText(wName);
        fillSubModTable();
        workSubModDisplayJP.setVisible(true);
        deptWorkModJP.setVisible(false);
        wardSubModJP.setVisible(true);
    }
}
```

AccountHomePage.java - line 2382

```
private void subModelTableMouseClicked(java.awt.event.MouseEvent evt) {
    //Lists made to store all task info, will be used when creating the pie chart for display
ibModelTaskIds is now global var so we can tell the user any task names they didnt set default bands for
      List taskNames = new ArrayList<String>();
    List taskRatios = new ArrayList<String>();
ibModelTaskIds is now a global variable as we want to make sure all task bands were set a default and need this list to do so later in saveChanges b
      List taskIds = new ArrayList<Integer>();

    //show up pie chart for tasks to be selected
    String subModName = (String)wardToSubTblModel.getValueAt(subModelTable.getSelectedRow(), 0);
    subModelTF.setText(subModName);
    subModIdForDefault = Integer.parseInt((String)wardToSubTblModel.getValueAt(subModelTable.getSelectedRow(), 2));
    System.out.println("Sub model id for " + subModName + " is: " + subModIdForDefault);

    ResultSet rs = DatabaseHelperWorkLog.returnPatientPieInfo(subModIdForDefault);
    //clear subModelTaskIds and subModelTaskNames list to ensure that only the submodel tasks are added associated to the clicked sub model
    subModelTaskIds.clear();
    subModelTaskNames.clear();
    storeTaskInfoInLists(rs, subModelTaskNames, taskRatios, subModelTaskIds);
    System.out.println("sub model task id list: " + subModelTaskIds);
    System.out.println("sub model task names: " + subModelTaskNames);
    displayPieAndRespondToClick(subModelTaskNames, taskRatios, subModelTaskIds, subModName);
}
```

The final step involves logging the default band for each task band in a SubModel. This is achieved by selecting tasks from a pie chart displayed to the user and bands from a JFrame. The implementation matches that of “5.1.4. Task Logging” below.

4.1.4. Task Logging

During task logging, a user logs a specific task for a patient on a ward in the hospital (See user manual for detailed explanation). A JComboBox is used to display a drop down list of the hospital wards and a JTable is used to display patient information once a ward is chosen. Both components work similarly to that of previous JComboBox and JTable methods mentioned previously; connecting to the database to find appropriate data through stored procedures containing appropriate SQL queries and incorporating itemStateChanged() and mouseClicked event listeners. During this, the appropriate SubModel_ID and Ward_ID are stored as variables associated with the ward selected from the dropdown.

NurseWorkLoggingAcc.java - line 307

```
subModId = DatabaseHelperWorkLog.returnWardSubModId(wardName);
wardId = DatabaseHelperWorkLog.returnWardId(wardName);
```

Once a patient is selected from the table, the app stores that patient's information as variables and calls a method from DatabaseHelperWorkLog.java called returnPatientPieInfo(subModId) which returns a result set of all task information relating to "subModId". The information in this result is then stored in Lists in the application. These Lists are used further when displaying the information to the user in a pie chart.

DatabaseHelperWorkLog.java - line 124

```
//return result set of all task information
//This info is needed for the pie chart display once a patient is selected
//uses view GetSubModelTaskRatio which contains the recalculated sub model ratios
public static ResultSet returnPatientPieInfo(int subModId){
    ResultSet rs = null;
    try{
        String SQL = "SELECT WorkModel_ID, WorkTask_ID, WorkTaskName, TaskSubModelRatio FROM vw_GetSubModelTaskRatio WHERE WorkSubModel_ID='" + subModId + "'";
        Statement s = conn.createStatement();
        rs = s.executeQuery(SQL);
    }catch(SQLException e){
        System.out.print("returnPatientPieInfo: The problem is:  ");
        System.out.println(e);
    }
    return rs;
}
```

Task Pie

As mentioned above, the JTable consisting of all patients currently on the chosen ward uses a mouseClicked event listener. When a patient is selected from the table, the JPanel which displays the pie chart is first cleared of anything previously added. The app then displays a pie chart to the user. Each slice of the pie chart represents a different Sub Model task and the size of each pie slice relates to each task's ratio. In order to display the pie chart, the java library JFree is used. This library allows the application to create a new DefaultPieDataset which consists of the label (taskName) and value (taskRatio) of each slice of the pie chart using org.jfree.data.general.DefaultPieDataset.

NurseWorkLoggingAcc.java - displayPieAndRespondToClick() - line 430

```
//these three lines are needed to clear the jpanel BEFORE display a new pie each time a new patient is clicked
pieChartDisplayJP.removeAll();
pieChartDisplayJP.revalidate();
pieChartDisplayJP.repaint();
//display pie
DefaultPieDataset pieDataset = new DefaultPieDataset();
fillPieDataSet(names, ratios, pieDataset);
bandTblMod = (DefaultTableModel)bandTable.getModel();
ChartPanel chartPanel = getPieChartPanel(fName, pieDataset);
pieChartDisplayJP.add(chartPanel, BorderLayout.CENTER);
```

The method fillPieDataset() populates the data set accordingly. The application also uses org.jfree.chart.ChartPanel as well as org.jfree.chart.ChartFactory to create and display the pie chart.

NurseWorkLoggingAcc.java - line 467

```
//we need to fill the pie dataset with the task names and their ratios
public void fillPieDataSet(List names, List ratios, DefaultPieDataset pieDataset){
    for(int i = 0; i<names.size(); i++){
            String tsk = (String)names.get(i);
            float rtio = (Float)ratios.get(i);
            pieDataset.setValue(tsk,  rtio);
        }
}
```

NurseWorkLoggingAcc.java - line 477

```
public ChartPanel getPieChartPanel(String name, DefaultPieDataset pieDataset){
    JFreeChart chart = ChartFactory.createPieChart(name, pieDataset, true, true, false);
    PiePlot p = (PiePlot)chart.getPlot();

    ChartPanel chartPanel = new ChartPanel(chart);
    return chartPanel;
}
```

The library also allows for a ChartMouseEvent and ChartMouseListener which makes the pie chart clickable. When a task in the pie chart is clicked, the method taskClicked(ent) returns the task name that was selected. The String variable “ent” represents the entity clicked and is found by calling ChartMouseEvent e.getEntity.toString(). The String stored in “ent” contains the task name between a left and a right bracket. Hence in the taskClicked() method, the characters in between the left and right bracket are added to a list and then concatenated back into a string before being returned and stored in the “clickedTaskName” variable. The corresponding ratio and id for the clicked task are found by calling returnClickedTaskRatio() and returnClickedTaskId(). Once we have the task ID, the app queries the database to return a result set of the band information associated with that task. The band information is added to the bandTable using ResultSet.next() and DefaultTableModel.addRow().

NurseWorkLoggingAcc.java - displayPieAndRespondToClick() - line 443

```
chartPanel.addChartMouseListener(new ChartMouseListener(){
    @Override
    public void chartMouseClicked(ChartMouseEvent e) {
        //get the taskName that was clicked
        String ent = e.getEntity().toString();
        //call taskClicked method to concatenate the entity which contains the task name
        clickedTaskName = taskClicked(ent);
        //check for the name and it's corresponding ratio from the lists
        clickedTaskRatio = returnClickedTaskRatio(clickedTaskName, names, ratios);
        clickedTaskId = returnClickedTaskId(clickedTaskName, names, ids);
        System.out.println(clickedTaskName);
        System.out.println("Task ratio for above task: " + clickedTaskRatio);
        //get the weight of the task clicked
        //we should have sub id
        int subId = subModId;
        //return bands into list -- changes made in dbhelper to use view
        ResultSet rs = DatabaseHelperWorkLog.returnTaskBands(clickedTaskId, wardId);
        addBandsToTbl(rs);
        displayBandFrame(bandFrame);
          clearBandTableAndExitWhenAsked();
    }
    public void chartMouseMoved(ChartMouseEvent e) {}
});
```

NurseWorkLoggingAcc.java - line 487

```
public String taskClicked(String entity){
    String taskName = "";
    int startIndex = entity.indexOf("(");
    int endIndex = entity.indexOf(")");
    ArrayList taskNameL = new ArrayList<String>();
    for(int i=startIndex + 1; i<endIndex; i++){
        taskNameL.add(entity.charAt(i));
    }
    for(int j = 0; j<taskNameL.size(); j++){
        taskName = taskName.concat(taskNameL.get(j).toString());
    }
    System.out.println(clickedTaskName);
    return taskName;
}
```

BandFrame

The method displayBandFrame() method uses JFrame methods setSize(), setLayout(), setTitle(), etc. as well as setVisible(true) which displays the frame containing the band table containing bands for the task selected.

NurseWorkLoggingAcc.java - line 542

```
public void displayBandFrame(JFrame bandFrame){
    bandFrame.setSize(300, 300);
    bandFrame.setLayout(new BorderLayout());
    bandFrame.setTitle("Bands");
    bandFrame.add(bandJP, BorderLayout.CENTER);
    bandJP.setVisible(true);
    bandFrame.setLocationRelativeTo(pieChartDisplayJP);
    bandFrame.setVisible(true);
    clearBandTableWhenExited();
}
```

Log

Another MouseListener, MouseClicked is used for the band table and when a band is selected insertIntoWorkTable() method is called from DatabaseHelperWorkLog.java which calls the SQL stored procedure which updates the database accordingly.

NurseWorkLoggingAcc.java - line 348

```java
private void bandTableMouseClicked(java.awt.event.MouseEvent evt) {
    //get band name selected
    String band = (String)bandTable.getValueAt(bandTable.getSelectedRow(), 0);
    //added more hidden cols to give the following info:
    Integer bandId = (Integer)bandTable.getValueAt(bandTable.getSelectedRow(), 1);
    Integer taskId = (Integer)bandTable.getValueAt(bandTable.getSelectedRow(), 2);

    if(confirmBandLog(band)){
        //we need to set the row count to 0 and dispose the window so
        //1. the band table clears and 2. the band table exits after use
        bandTblMod.setRowCount(0);
        bandFrame.dispose();
        bandFrame.setVisible(false);
        //we need the current shift of when the task is being logged
        workShiftId = getWorkShiftId();
        //have epMovId from patientTableClicked() above
        //subModId already have from jComboBoxItemChanged above
        //isOutlier DONT HAVE!!!
        Integer isOutlier = 0;
        Date workShiftDate = Date.valueOf(LocalDate.now());
        DatabaseHelperWorkLog.insertIntoWorkTable(patEpMovId, workShiftDate, workShiftId, subModId, taskId, isOutlier, bandId, staffId);
        //confirmation message to the user
        JOptionPane.showMessageDialog(pieChartDisplayJP, band + " has been logged for " + fName);
    }
}
```

DatabaseHelperWorkLog.java - line 195

```java
//insert the task, band logged for the patient in the ward
public static void insertIntoWorkTable(Integer epMovId, Date shiftDate, Integer shiftId, Integer workSubModId, Integer workTaskId,
        Integer isOutlier, Integer taskBandId, Integer staffId){
    try{
        CallableStatement myCall = conn.prepareCall("{call sp_InsertWork(?, ?, ?, ?, ?, ?, ?, ?)}");
        myCall.setInt(1, epMovId);
        myCall.setDate(2, shiftDate);
        myCall.setInt(3, shiftId);
        myCall.setInt(4, workSubModId);
        myCall.setInt(5, workTaskId);
        myCall.setInt(6, isOutlier);
        myCall.setInt(7, taskBandId);
        myCall.setInt(8, staffId);
          myCall.executeUpdate();
        myCall.execute();


    }catch(SQLException e){
        System.out.print("insertIntoWorkTable problem is :  ");
        System.out.println(e);
    }
}
```

4.1.5. Staff Information

A mouse listener is utilized in the app and when the “staffButton” is selected the appropriate JPanel is setVisible(true). This JPanel contains a table of staff information and provides the option to update the staff password. Data from the database is added to the table similarly to previous methods; returning a result set from the database via SQL query and adding to the table using DefaultTableModel.addRow(). An actionPerformed event is used to listen for when the “editAPasswordButton” is clicked and a JOption input dialog is returned to the user for input regarding the staff password they wish to change. The app performs if statements to check whether the password change can be successfully updated or not and if it can DatabaseHelperSP.updatePWForUser() is called. This contains appropriate SQL query to update the information in the database.

AccountHomePage.java - line 2464

```
private void changePasswordForUser(String user, char [] oldPw, char [] newPw) {
    if(user.isEmpty() || oldPw.length == 0) {
        JOptionPane.showMessageDialog(this, "Please enter a username and password.");
    }

    else if (HomePage.isPasswordCorrect(HomePage.conn, user, oldPw)) {   // if old pw matched user
        // then remove old pw and add new pw to DB
        DatabaseHelperSP.updatePwForUser(user, newPw);

    }
    else {
        JOptionPane.showMessageDialog(this, "Invalid username/password combination, try again");    // old pw doesnt match user
    }
}
```

DatabaseHelperSP.java - line 478

```
public static void updatePwForUser(String user, char [] newPw) {
    String newPass = "";
    for (int i = 0; i < newPw.length; i++) {
        newPass += newPw[i];
    }

    CallableStatement cstmt = null;
    try {
        System.out.println("attempting method: updatePwForUser");

        cstmt = HomePage.conn.prepareCall("{call UpdatePwForUser(?, ?)}");
        cstmt.setString(1, user);
        cstmt.setString(2, newPass);
        cstmt.execute();

    }
    catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
```

4.2. Mobile Application

4.2.1. Mobile-database connectivity

This project incorporates a mobile application and an AWS RDS database. In order to generate a connection between both these components, jTDS was used. jTDS is a type 4 JDBC Driver and supports all features of JDBC. The JDBC driver was used in order to execute objects such as the Connection object, Statement/Prepared Statement/Callable Statement objects and the ResultSet object. The connection itself was created using the DriverManager class. We created a String “connectionURL” which represents the database URL of our system necessary for the connection. Class.forName() allows the initialization of the JDBC SQLServerDriver class. Following this, the connection is successfully created by using the getConnection method of the DriverManager class. It generates a connection with the first driver it comes to that can correctly connect to the database URL provided.

ConnectionHelper.java - line 31

```
Class.forName("net.sourceforge.jtds.jdbc.Driver");
ConnectionURL = "jdbc:jtds:sqlserver://" + ip +";DatabaseName="+ db + ";user=" + userName+ ";password=" + password + ";";
connection = DriverManager.getConnection(ConnectionURL);
System.out.println("Connected!");
```

4.2.2. Login/Logout

Similar to the desktop app, the mobile app takes the input provided by the user by the username and password EditText elements in android studio. The method ensureNoEmptyFields() checks that the username and password EditText fields are not empty and displays an appropriate Toast message. The method checkForCorrectPassWord() calls the database via a SQL query and checks whether the password entered is the correct password associated with the username. If the username and password are correct an Intent to begin the MobileAccountHomePage.java activity and startActivity(intent) is then called to display the account page to the user.

MainActivity.java - line 31

```
public void clickListenerLoginButton(){
    Button loginButton = findViewById(R.id.loginButton);
    loginButton.setOnClickListener((v) → {
            EditText userET = findViewById(R.id.usernameEditText);
            EditText passwordET = findViewById(R.id.passwordEditText);

            String username = userET.getText().toString();
            char[] pass = (passwordET.getText().toString()).toCharArray();

            //check if username and password are empty
            ensureNoEmptyFields(username, pass);
            //check if password is correct
            checkForCorrectPassword(username, pass);
            //
            Arrays.fill(pass, val: '0');

    });
}
```

MainActivity.java - line 70

```
protected static boolean isPasswordCorrect(Connection conn, String user, char[] input) {     // https://do
    boolean isCorrect;
    String password = getPassword(conn, user);

    char[] correctPassword = new char[password.length()];   // user entered pw is array so we create array
    for (int i = 0; i < password.length(); i++) {
        correctPassword[i] = password.charAt(i);
    }
    isCorrect = Arrays.equals (input, correctPassword);     // bool isCorrect = are they identical
    Arrays.fill(correctPassword, val: '0');   // zero out password so java code not storing it in variable
    return isCorrect;
}
```

To allow the user to logout, a MenuInflater is used alongside the menu in the menu resource file which sets an item in the menu to “Logout”. An OnItemSelected listener is used to display the item “Logout” to the user when they select the menu in the Action bar of their mobile device. When the “logout” item is selected by the user, the app uses an AlertDialog to ask the user for confirmation that they wish to logout.

MobileAccountHomePage.java - line 66

```java
@Override
public boolean onCreateOptionsMenu(Menu menu){
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_menu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.logout) {
        logoutUser();
        return true;
    }

    return super.onOptionsItemSelected(item);
}
```

MobileAccountHomePage.java - line 82

```java
public void logoutUser(){
    new AlertDialog.Builder( context: this)
            .setMessage("Are you sure you want to logout?")
            .setCancelable(false)
            .setPositiveButton( text: "Yes", (dialog, id) -> {
                    staffId = 0;
                    Intent intent = new Intent( packageContext: MobileAccountHomePage.this, MainActivity.class);
                    startActivity(intent);
            })
            .setNegativeButton( text: "No",  listener: null)
            .show();
}
```

4.2.3. Task Logging

Spinner

The home page that a user sees when they successfully login is that of a Spinner (android dropdown). The method fillWardSpinner() is called in the onCreate() method and it connects and queries the database for any existing ward names. These ward names are stored in a list before being added to the spinner using ArrayAdapter.setDropDownViewResource, Spinner.setAdapter() methods.

An OnItemSelected listener is utilized for user interaction with the dropdown and if changed, DatabaseHelper.returnPatientInfoWard is called to return a result set of all patient information currently admitted on the ward selected. This result set is passed into the fillPatientTable() method. For each, addRowToTable() adds this information as a row in the table through adding views, TableRow, TableRow.LayoutParams, TextView, TextView.setText(), TableRow.addView().

MobileAccountHomePage.java - line 117

```java
@Override
public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
    //clear table
    int count = wardTbl.getChildCount();
    System.out.println("Count: " + count);
    //index out of bounds error sometimes happens -NEED TO FIX **** FIXED BY -1
    if(count>1){
        wardTbl.removeViews( start: 1,  count: count-1);
    }
    String wardName = parent.getItemAtPosition(position).toString();
    if(!wardName.equals("Select Ward")){
        subModelId = DatabaseHelper.returnWardSubModId(wardName);
        wardId = DatabaseHelper.returnWardId(wardName);
        ResultSet rs = DatabaseHelper.returnPatientInfoOnWard(wardId);
        fillPatientTable(rs);
    }
}
```

An OnClickListsner is created for the table so that when a user selects a row (Patient), information is stored as variables to be used for updating the database further down the line. It also calls goToPatientPieChartActivity() which creates a new intent and opens PieChartActivity.java which displays the tasks for the patient selection.

MobileAccountHomePage.java - line 182

```java
public void setPatientTableClickListener(TableRow wardRow){
    wardRow.setClickable(true);
    wardRow.setOnClickListener((v) -> {
            v.setBackgroundColor(R.color.bluePrimary);
            TableRow tr=(TableRow)v;
            int id = tr.getId();
            TextView t1 = (TextView)tr.getChildAt( index: 0);
            clickedPatFName = t1.getText().toString();
            TextView t2 = (TextView)tr.getChildAt( index: 1);
            clickedPatLName = t2.getText().toString();
            TextView t4 = (TextView)tr.getChildAt( index: 3);
            String epMovStr = t4.getText().toString();
            clickedPatEpMovId = Integer.parseInt(epMovStr);
            TextView t5 = (TextView)tr.getChildAt( index: 4);
            String bedIdStr = t5.getText().toString();
            int bedId = Integer.parseInt(bedIdStr);
            System.out.println("ROW CLICKED : " + clickedPatFName + " AND ID: " + id + " AND EPMOV: " + clickedPatEpMovId + "AND BEDID: " + bedId);
            goToPatientPieChartActivity();

    });
}
```

MobileAccountHomePage.java - line 213

```
public static void addViewToRow(TableRow row, String viewText, TableRow.LayoutParams p, Context context){
    TextView tv = new TextView(context);
    tv.setText(viewText);
    tv.setTextSize(16);
    tv.setTextColor(Color.BLACK);
    tv.setGravity(Gravity.CENTER);
    tv.setLayoutParams(p);
    row.addView(tv);
}
```

Pie

As mentioned before, JFree java library is used throughout the desktop application. This library relies on java swing packages and therefore cannot be compiled with android studio. Because of this, another library called MPAndroidChart was used to display the pie chart visualisation to the user. In the same way as the desktop, patient task information is returned from the database as a result set and stored in java lists. An array list is first created and filled with PieEntry containing each task and corresponding ratio added to the list via a for loop and ArrayList.add().

PieChartActivity.java - displayPie() - line 80

```
PieChart patientPie = findViewById(R.id.patientPie);
ArrayList<PieEntry> pieTasks = new ArrayList<>();
for(int i=0; i<tasks.size(); i++){
    pieTasks.add(new PieEntry((Float)ratios.get(i), (String)tasks.get(i)));
}
```

A PieDataSet stores the ArrayList of each PieEntry. Built-in methods are utilized such as PieDataSet.setColors, PieDataSet.setValueSize, etc. The PieDataset is then used to be stored as PieData and used by the PieChart variable which fills the pie chart with information using setData()

PieChartActivity.java - displayPie() - line 86

```
PieDataSet pieDataSet = new PieDataSet((pieTasks), label: "Tasks");
pieDataSet.setColors(ColorTemplate.COLORFUL_COLORS);
pieDataSet.setValueTextColor(Color.BLACK);
pieDataSet.setValueTextSize(16f);

PieData data = new PieData(pieDataSet);

patientPie.setData(data);
patientPie.getDescription().setEnabled(false);
patientPie.setDrawHoleEnabled(false);
```

An OnChartValueSelectedListener() is used with the pie chart displayed to the user. When selected the name and ratio of the selected slice are found through PieEntry.getLabel() and PieEntry.getValue() methods which return the task name and ratio and we store them as variables for future use. The application also changes to TaskBandActivity.java where the user can select the band for their chosen task. TaskBandActivity.java makes use of a

TableLayout similar to MobileAccountHomePage.java using addRowToBandTable() and MobileAccountHomePage.addViewToRow() before adding each TableRow to the table.

PieChartActivity.java - displayPie() - line 100

```
patientPie.setOnChartValueSelectedListener(new OnChartValueSelectedListener() {
    @Override
    public void onValueSelected(Entry e, Highlight h) {
        PieEntry pe = (PieEntry)e;
        clickedTaskName = pe.getLabel();
        clickedTaskRatio = pe.getValue();
        System.out.println("Clicked name: " + clickedTaskName);
        System.out.println("Clicked ratio : " + clickedTaskRatio);
        clickedTaskId = returnClickedTaskId(clickedTaskName, tasks, ids);
        System.out.println("Clicked task id : " + clickedTaskId);

        //go to band activity
        Intent intent = new Intent( packageContext: PieChartActivity.this, TaskBandActivity.class);
        startActivity(intent);
    }
```

An OnClickListener is set to each row and when selected the app connects and inserts into the database, the work logged for the patient. DatabaseHelper.insertIntoWorkTable() inserts the information stored related to the task being logged into the Work table in the database. Once this is done, the app sends the user back to the PieChartActivity.java and notifies them that the task was logged through a Toast message.

TaskBandActivity.java - line 136

```
private void logWork(Integer bandId, Integer taskId){
    //add work
    workShiftId = getWorkShiftId();
    //isOutlier DONT HAVE!!! *************************
    Integer isOutlier = 0;
    Date workShiftDate = Date.valueOf(LocalDate.now().toString());
    DatabaseHelper.insertIntoWorkTable(clickedPatEpMovId, workShiftDate, workShiftId, subModelId, taskId, isOutlier, bandId, staffId);

    //go back to pie activity*****************************************************
    Intent intent = new Intent( packageContext: TaskBandActivity.this, PieChartActivity.class);
    startActivity(intent);
    //toast not working properly only showing some text ***********************
    Toast toast = Toast.makeText(getApplicationContext(), text: clickedBandName + " was logged for "  + clickedTaskName + " for "
            + clickedPatFName, Toast.LENGTH_LONG);
    toast.show();
    System.out.println("Band Name clicked: " + clickedBandName);
    System.out.println("Task Name clicked: " + clickedTaskName);
    System.out.println("Patient Name: " + clickedPatFName);
}
```

4.3. AWS Database

We used SQL Server for our database. We hosted it in the cloud with Amazon Web Services RelstionalDatabse Service.*Go into more detail here.*
See user guide for a more human friendly explanation of the elements these tables represent.

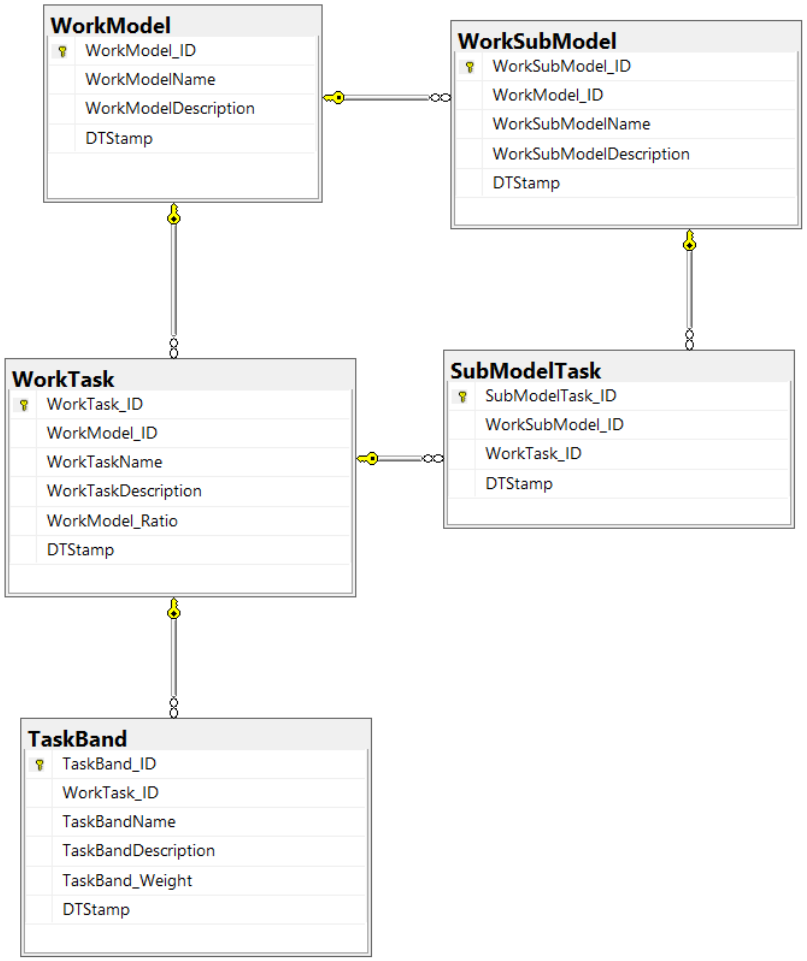
4.3.1. Tables and relationships

Our database contains 27 tables. We have created our application based on the assumption that it would integrate fully with the hospital's own system in order to get information about their departments, wards, patients, staff, etc. For our project, we have created a number of these tables that we felt we needed most to demonstrate the working of our application and populated them with mock data. The other tables are specific to our own application and how it works.

Below is a list of all the tables in our database. Tables 1-11 are specific to our application. Tables 10-27 are tables that we have created to replicate a hospital's own system to demonstrate how our application integrates with it.

1. WorkModel
2. WorkTask
3. TaskBand
4. WorkSubModel
5. SubModelTask
6. SubModelDefault
7. SubModelDefaultBand
8. Work
9. WorkArchive
10. Param
11. ParamGroup
12. Department
13. Ward
14. Bed
15. Specialty
16. Diagnosis
17. Patient
18. Episode
19. EpisodeMove
20. StaffType
21. Staff
22. RosterShift
23. Roster
24. RosterChange
25. WorkShift
26. WorkSequence
27. ShiftSequence

Below, with the help of diagrams, we will explain the tables at the core of our application.
WorkModel, WorkTask, TaskBand, WorkSubModel, SubModelTask:



WorkModel - Each record in this table identifies a WorkModel created by the user.
Primary key: WorkModel_ID (database generated identity value).
Foreign keys: null.
How data is entered: InsertWorkModelData stored procedure triggered by user.

WorkTask - Each record in this table is a WorkTask created by the user to be a unit of some existing WorkModel.
Primary key: WorkTask_ID (database generated identity value).
Foreign keys: WorkModel_ID references WorkModel_ID in WorkModel table.
How data is entered: InsertTask stored procedure triggered by user.

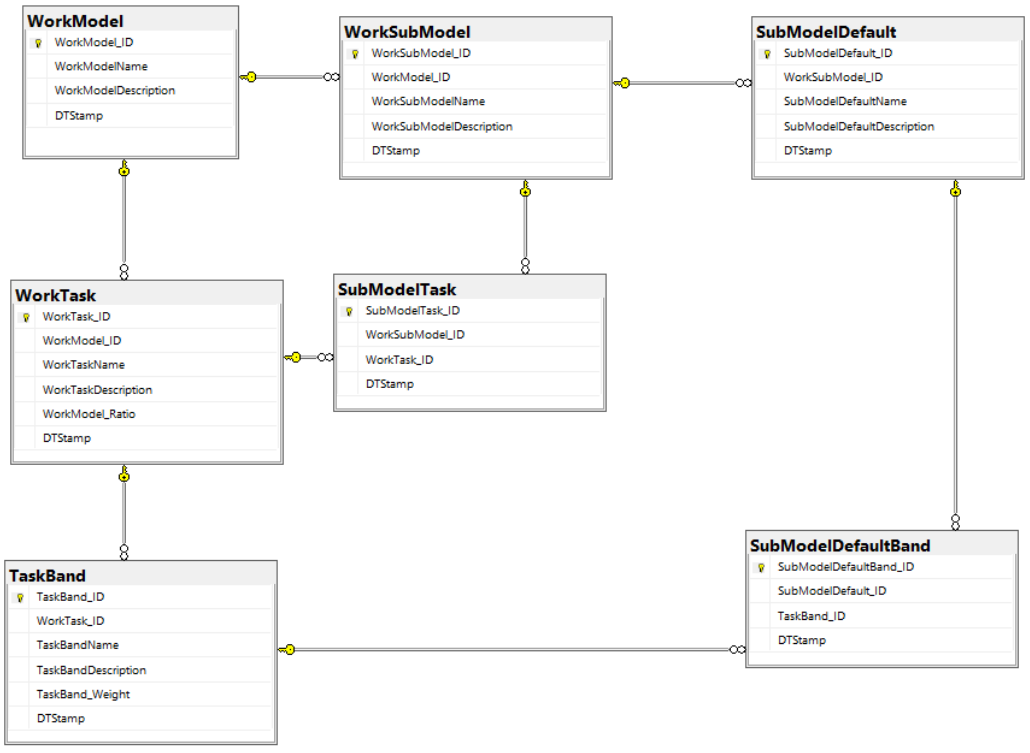
TaskBand - Each record in this table is a TaskBand created by the user to be a component of some existing WorkTask.
Primary key: TaskBand_ID (database generated identity value).
Foreign key: WorkTask_ID references WorkTask_ID in WorkTask table.

How data is entered: InsertTaskBand stored procedure triggered by user.

WorkSubModel - Each record in this table identifies a WorkSubModel created by the user to be a child model of some existing WorkModel.
Primary key: WorkSubModel_ID (database generated identity value).
Foreign keys: WokrModel_ID references WorkModel_ID in WorkModel table.
How data is entered: InsertSubModelData stored procedure triggered by user.

SubModelTask - Each record in this table is simply a link between a SubModel and a WorkTask.
Primary key: SubModelTask_ID (database generated identity value).
Foreign keys: WorkTask_ID references WorkTask_ID in WorkTask table.
How data is entered: InsertSubModelTask stored procedure triggered by user.

SubModelDefault and SubModelDefaultBands:

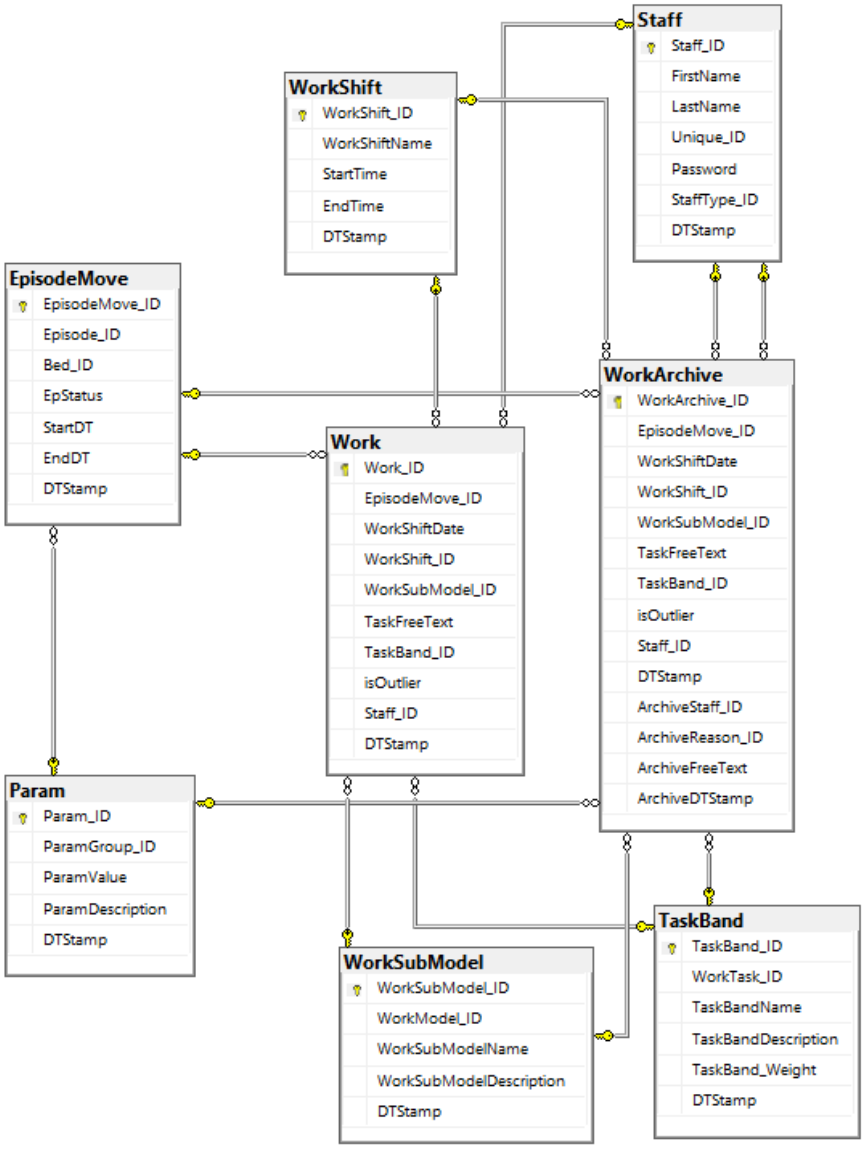


SubModelDefault - Each record in this table identifies a SubModelDefault model.

Primary key: SubModelDefault_ID (database generated identity value).
Foreign keys: WorkSubModel_ID references WorkSubModel_ID in WorkSubModel table.
How data is entered: SQL INSERT statement triggered by user (at first, we wrote our SQL statements in our java code before deciding to migrate all SQL functionality to stored procedures in the database, but we did not get time to move all of them over).

SubModelDefaultBand - Each record in this table is simply a link between a SubModelDefault and a TaskBand.
Primary key: SubModelDefaultBand_ID (database generated identity value).
Foreign keys: TaskBand_ID references TaskBand_ID in TaskBand table.
How data is entered: SQL INSERT statement triggered by user (at first, we wrote our SQL statements in our java code before deciding to migrate all SQL functionality to stored procedures in the database, but we did not get time to move all of them over).

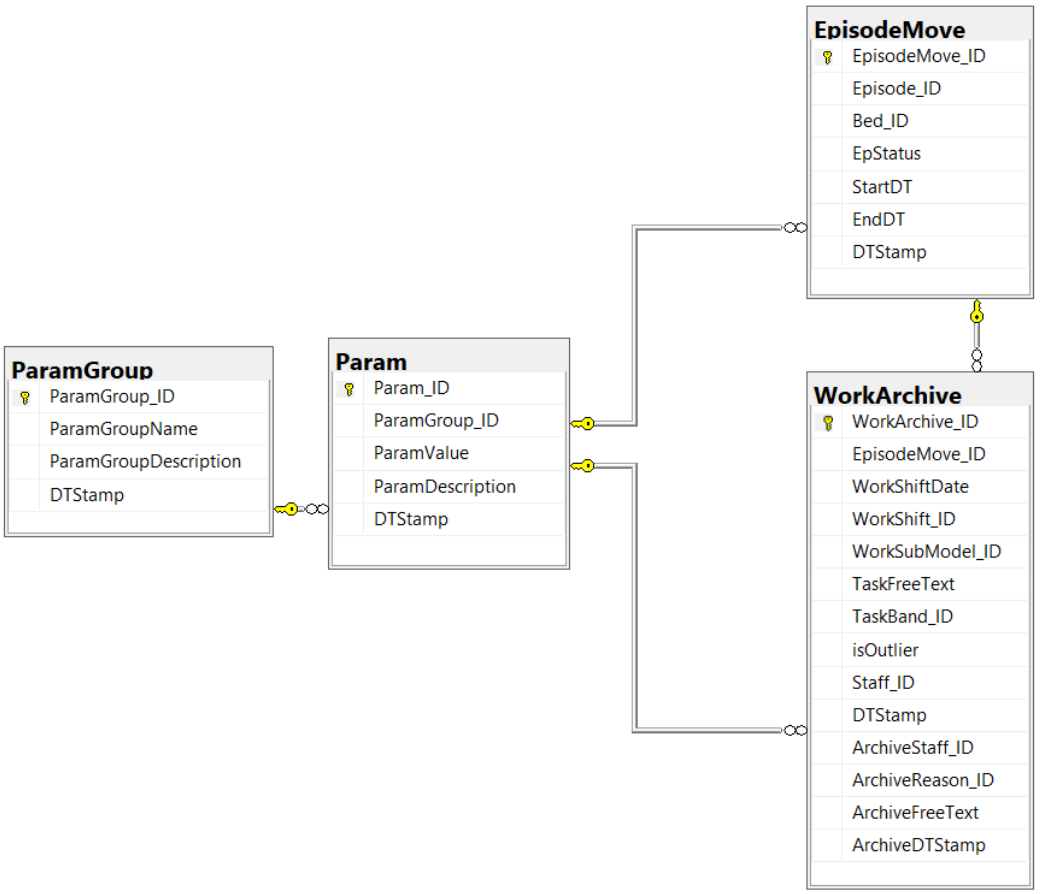
Work and WorkArchive:



Work - Each record in this table is a log of work (a selection of some band belonging to some task) for some patient for some work shift.
Primary key: Work_ID (database generated identity value).
Foreign keys: EpisodeMove_ID references EpisodeMove_ID in EpisodeMove table.
WorkShift_ID references WorkShift_ID in WorkShift table.
WorkSubModel_ID references WorkSubModel_ID in WorkSubModel table.
TaskBand_ID references TaskBand_ID in TaskBand table.
Staff_ID references Staff_ID in Staff table.
How data is entered: sp_InsertWork stored procedure triggered by user.

WorkArchive - Each record in this table is a log of work that was overwritten (a new work log was entered for the same patient in the same work shift.
Primary key: WorkArchive_ID (database generated identity value).
Foreign keys: EpisodeMove_ID references EpisodeMove_ID in EpisodeMove table.
WorkShift_ID references WorkShift_ID in WorkShift table.
WorkSubModel_ID references WorkSubModel_ID in WorkSubModel table.
TaskBand_ID references TaskBand_ID in TaskBand table.
Staff_ID references Staff_ID in Staff table.
ArchiveStaff_ID references Staff_ID in Staff table.
Archive Reason_ID references Param_ID in Param table.
How data is entered: sp_ArchiveWork stored procedure called by sp_InsertWork stored procedure triggered by user.

Param and ParamGroup:



The Param and ParamGroup tables were created to accommodate miscellaneous items in our database. They store parameters that are inserted into columns in other tables in our database. The ParamGroup table is essentially a list of variables, and the Param table is a list of possible values for each of those different variables. Each record in Param has a foreign key to ParamGroup to identify which variable it is a possible value for. These 2 tables come into play when inserting data into the EpisodeMove table, the WorkArchive table and the TaskBand table.

The *EpisodeMove* table has a column called EpStatus. This is an integer which identifies the status of a patient during an episode move. Below is a snippet of code which inserts records pertaining to the EpStatus variable into the ParamGroup table and the Param table.

```
41
42  --insert param groups
43
44  insert into dbo.ParamGroup
45      (ParamGroupName,ParamGroupDescription,DTStamp)
46  values
47      ('EpisodeStatus','something about the status eg. expected in, for transfer or for home',CURRENT_TIMESTAMP)
48  --return the ID for use in entering the Params
49  declare @EpStatus_ID as int
50  set @EpStatus_ID = (select p.ParamGroup_ID from dbo.ParamGroup p where p.ParamGroupName = 'EpisodeStatus')
51
52
53  --insert some params
54
55
56  insert into dbo.[Param]
57      (ParamGroup_ID,ParamValue,ParamDescription,DTStamp)
58  values
59      (@EpStatus_ID,'tci','patient is expected in but has not yet physically arrived',CURRENT_TIMESTAMP),
60      (@EpStatus_ID,'adm','patient is on the ward',CURRENT_TIMESTAMP),
61      (@EpStatus_ID,'trans','patient is expected as a transfer from another ward but has not yet physically arrived',CURRENT_TIMESTAMP),
62      (@EpStatus_ID,'disc','patient is discharged, but has not yet physically left the ward',CURRENT_TIMESTAMP)
63
64  declare @tci_ID as int
65  declare @adm_ID as int
66  declare @trans_ID as int
67  declare @disc_ID as int
68
69  set @tci_ID = (select p.Param_ID from dbo.[Param] p where p.ParamValue = 'tci')
70  set @adm_ID = (select p.Param_ID from dbo.[Param] p where p.ParamValue = 'adm')
71  set @trans_ID = (select p.Param_ID from dbo.[Param] p where p.ParamValue = 'trans')
72  set @disc_ID = (select p.Param_ID from dbo.[Param] p where p.ParamValue = 'disc')
73
```

A ParamGroup record (a variable) is created called “EpisodeStatus”. Four Param records (possible values for this variable) are created, each being a possible status of a patient’s Episode Move. Each of these four Params has a foreign key pointing to the record in ParamGroup which specifies that these values belong to the variable “EpisodeStatus”. When a patient’s Episode Move is created, the value in the EpStatus column refers to one of these 4 Params.
The 4 values indicate 4 possible scenarios:

- The patient is expected into the hospital and the staff are preparing work for them
- The patient is admitted to the hospital and residing in a bed in some ward
- The patient is in transit (being handed over from one ward to another)
- The patient is formally discharged from the hospital but still resides in the ward and may need some care until they physically leave

The *WorkArchive* table has a column called ArchiveReason_ID. This is an integer which identifies the reason that a work log was overwritten and pushed into the WorkArchive table. Below is a snippet of code which inserts records pertaining to the ArchiveReason_ID variable into the ParamGroup table and the Param table.

```
88
89  -- also some archive params
90
91  insert into dbo.ParamGroup
92      (ParamGroupName,ParamGroupDescription,DTStamp)
93  values
94      ('ArchiveReason','something about the archive reason, probably only two',CURRENT_TIMESTAMP)
95  --return the ID for use later
96  declare @ArchiveReason_ID as int
97  set @ArchiveReason_ID = (select p.ParamGroup_ID from dbo.ParamGroup p where p.ParamGroupName = 'ArchiveReason')
98
99
100 --and the reasons
101 insert into dbo.[Param]
102     (ParamGroup_ID,ParamValue,ParamDescription,DTStamp)
103 values
104     (@ArchiveReason_ID,'error','an error record gets archived and flagged as error',CURRENT_TIMESTAMP),
105     (@ArchiveReason_ID,'update','an update gets archived and replaced with latest',CURRENT_TIMESTAMP)
106
107 declare @error_ID as int
108 declare @update_ID as int
109 set @error_ID = (select p.Param_ID from dbo.[Param] p where p.ParamValue = 'error')
110 set @update_ID = (select p.Param_ID from dbo.[Param] p where p.ParamValue = 'update')
111
```

A ParamGroup record (a variable) is created called “ArchiveReason”. Two Param records (possible values for this variable) are created, each being a possible reason for archiving a work log. Each of these two Params has a foreign key pointing to the record in ParamGroup which specifies that these values belong to the variable “ArchiveReason”. When a work log is archived by a user inserting a new work log for the same patient during the same work shift, the value in the ArchiveReason_ID column refers to one of these 2 Params.
The 2 values indicate 2 possible scenarios:

- The work log was an error or mistake (the user logged the incorrect band so followed up by logging the correct band, thus archiving the incorrect log)
- The work log is being updated (a patient’s state has changed during a shift but work was already logged for this patient during the shift. In this case, the new updated work log is entered, and the old outdated worklog is archived)

The *TaskBand* table holds a number of bands for every task in the WorkTask table. Each WorkTask needs a NoWorkRequired band that users can select to indicate that they did not need to perform this task for this patient during this work shift. When users configure their system, they are not required to enter this band themselves; the system automatically enters it for them.

The InsertInsert.sql script (section 5.3.4) calls the stored procedure UpdateNoWorkRequiredValue (section 5.3.3) to insert a record into the ParamGroup table for a variable called NoWorkRequiredValue. The script calls the stored procedure to enter a record in the Param table which will hold a value (given as an input parameter) to be used as the value of the TaskBandWeight column in a TaskBand record for a NoWorkRequired band.

Normalisation:
Our database and its relationships are normalised to Boyce-Codd Normal Form. It meets the criteria for:
First Normal Form: single-valued attributes in each table, same-type attributes for each column in each table, unique column names in each table, order of records in each table does not matter

Second Normal Form: tables are in first normal form, there are no partial dependencies (all of our tables have single-column primary keys which makes partial dependency impossible)
Third Normal Form: tables are in second normal form, no transitive dependencies (for each table, no columns depend on any columns other than the primary key)
Boyce-Codd Normal Form: third normal form, for any dependency A -> B, A is a super key.

4.3.2. Views

In this section we will discuss some of the most useful and notable views in our database.

vw_GetWorkScores:
This view gives us the WorkScore of each patient in the hospital. The WorkScore is the figure that indicates how much work was required to care for a patient in some given work shift. The view returns the WorkScore for every patient on every work shift in the past. This means that for any work shift on any day, we can view the WorkScore of any patient who was in the hospital on that shift.

vw_GetWorkScores returns:

- The episode move
- The ward
- The date
- The work shift
- The sub model being used by the ward
- The amount of sub model work done (this is a figure between 0-100. If it is 100, it means the most amount of work that could possibly be done was done. This means every task required full work)
- The amount of outlier work done (this is a figure to indicate how much extra work was done for this patient. Extra work is work that is not usually done on this ward. For example, a surgical patient post-operation may need to be on a medical ward as their base ward because they have got diabetes, but they still need their surgical wound cared for, so the wound care would be an outlier task and would not be counted as regular patient WorkScore but as extra work) ((This outlier feature was added to the database but we did not get time to build it into the user interface))
- The amount of work model work done (this indicates how much work out of an entire WorkModel did this work log account for. This piece of information is not very useful as it is unlikely the user would want to compare one task in one ward with all tasks performed in the entire hospital, but we included it just in case we found some use for it in future).

The vw_GetWorkScores view has a nested view where it gets its information from; the nested view is vw_GetAllWorkWithWeights. vw_GetWorkScores takes EpisodeMove_ID, Ward_ID, WardName, WorkShiftDate, WorkShift_ID, WardSubModel_ID, and

SubModelWork directly from vw_GetAllWorkWithWeights and. It creates a new column SubModelWork from the sum of WeightWRTSubModel column from vw_GetAllWorkWithWeights.

vw_GetAllWorkWithWeights:
This view returns every work input that has been entered into the database. This means there is a row in this view for every time a nurse has selected a task for some patient and selected a band to indicate how much work was required to complete that task for that patient. Each row in the vw_GetAllWorkWithWeights view presents all the information about the work input. This view uses nested views vw_GetSubModelTaskRatio and vw_GetSubModelRatioTotal, which provide the information about each task's ratio that will allow us to compute:

- This task's ratio with respect to the entire WorkModel (to get the figure for how much work on average this task requires in comparison to other tasks around the hospital)
- This task's ratio with respect to the SubModel this ward is using (to get the figure for how much work this task requires in comparison to other tasks performed on this ward)
- The selected band's weight contribution to the WorkScore of this patient on this shift

This view also uses the nested view vw_GetAdjustedBandWeight. This nested view currently does not contribute anything to the calculations of the vw_GetAllWorkwithWeights. It was written to be used with the ideal GUI display of the models that we did not succeed in fully implementing. vw_GetAdjustedBandWeight is explained in detail below.

vw_GetAllWorkWithWeights returns:

- The date
- The workshift
- The start and end times of that shift
- The episode move (which we can use to find patient ID, name, diagnosis, etc.), the bed, the ward
- The SubModel that this ward was using at the time
- The task that was entered and its ratio with respect to the WorkModel is was derived from
- Whether this task was an outlier task or not (outlier tasks are tasks that are not inherently included in the ward sub model because they are not usually performed on patients in this ward, but they are logged as extra work for this patient on top of the work of all of the tasks in the ward sub model, in the case where a patient has some particular needs outside of those usually performed in this ward)
- The free text that was entered (if any) as a note with the work input
- The band that was entered
- The band's weight with respect to the task it belongs to
- The task's ratio with respect to the submodel
- The bands weight w respect to the submodel
- The ratio of the sum of this SubModel's tasks out of all of the tasks in its parent WorkModel

vw_GetAllWardsWithCurrentWorkShift:
Some wards will go through only 2 shifts in 24 hours, a day and a night shift, while some wards will go through perhaps 3 shifts (morning, midday, and night) in 24 hours or maybe

even 4, depending on the unique workflow and routine of each ward. This view returns every ward in the hospital and the work shift that is currently ongoing in each of the wards. This view is called in our application when a user triggers it by opening the page where they will log work for patients. When they select a ward and patient to enter work for, this view is called to find the current work shift that is ongoing in the ward, so that the application can include the work shift ID in the work record entered for whichever task and band the user logs for a patient. The view calls on a nested view vw_GetWardsWithSequencesAndShifts which calls on a nested view vw_GetSequencesAndShifts.

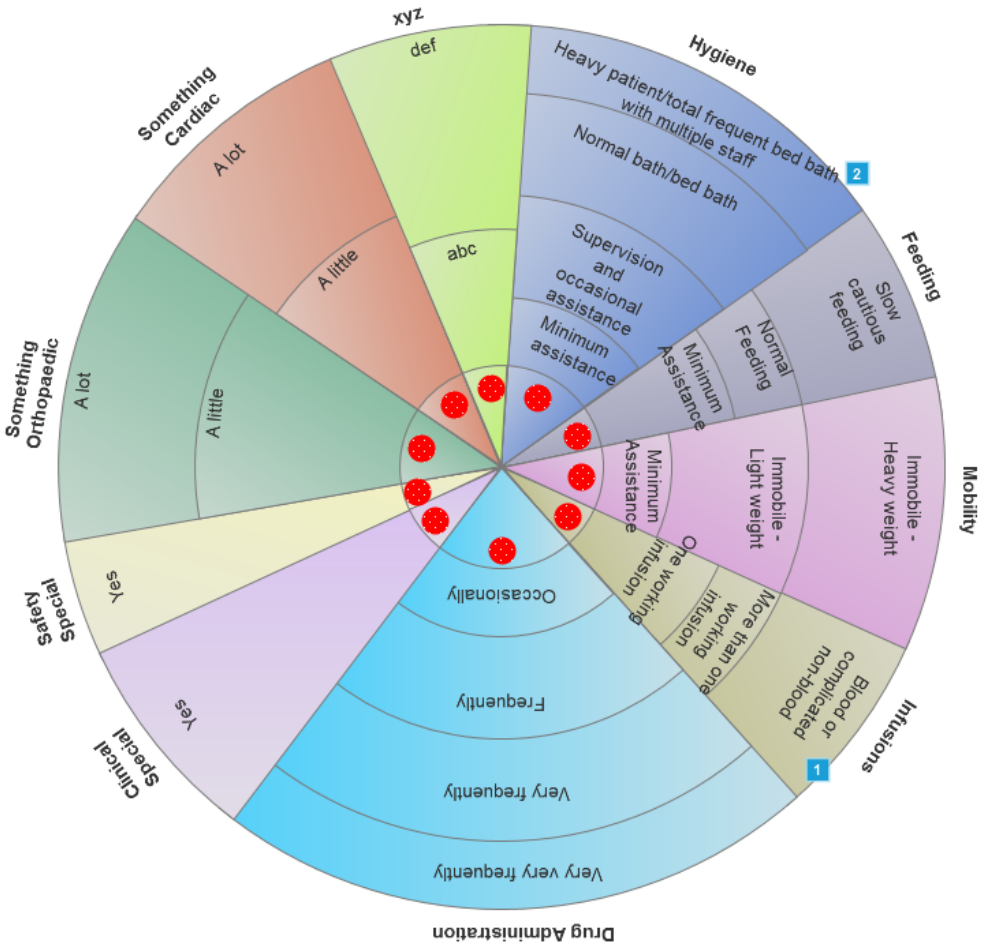
vw_GetSequencesAndShifts returns each WorkSequence that exists in the database (a work sequence is a set of shifts that run in sequence of each other which can be assigned to a word. An example is DayNight, it consists of a day shift and a night shift. Another example is MorningAfternoonNight, it consists of a morning shift, an afternoon shift and a night shift). It returns the WorkSequence, the names of the WorkShifts that contribute to this WorkSequence, and the order of each of these shifts in the WorkSequence (For DayNight, the day shift is order number 1, and the night shift is order number 2). This view also returns the start and end times of each of these WorkShifts, their duration (end time minus start time) and whether or not the WorkShift passes through midnight or not.

vw_GetWardsWithSequnecesAndShifts builds on vw_GetSequencesAndShift by including the names of the wards which use each WorkSequence.

Then, vw_GetAllWardsWithCurrentWorkShift uses vw_GetWardsWithSequnecesAndShifts and uses the current timestamp to calculate which of the WorkShifts in this wards WorkSequence is ongoing at this moment. For whatever ward is selected by a nurse to log work for some patient, this view finds the current WorkShift for the ward, and includes it in the work log that the nurse enters for the patient.

vw_GetWardsWithSubModelInclBands:
This view returns every ward, the SubModel the use, the WorkModel that is the parent of this SubModel, the tasks that exist in this SubModel with their ratios, and the bands that exist in each of those tasks with their weights. This view is called by our java application when a user's actions trigger it by opening the page where they will log work for patients. This view allows us to display the correct SubModel for the ward that the user selected, and to display the correct bands, when the user selects a task to log work for.
This view builds the band information onto the nested view vw_GetWardsWithSubModel which returns everything except the band information.

vw_GetAdjustedBandWeight:
This is a view that was created to be used with a feature of our app that did not end up being implemented into the user interface. We had originally hoped to use our graphics library in java to display the bands of each task on the models in the app. Our idea for what the models would look like is below:



To display this model, the size of each band on a task would correspond to the weight of this band in the database. The bands on the very inside of each slice (indicated above by red markings) are the No Work Required bands which, when selected, indicate that this task was not performed for this patient during this work shift. This means a score of 0 goes into the database for this work input. However, to display the bands while having their sizes correspond with their weight in the database, the inside band would not be displayed since its weight is 0. So, to accommodate for this, the No Work Required band for each task would be given some small weight, maybe 0.1, so that it would be able to take up a small space on the model in order for it to be clickable by the user. Then, when vw_GetAllWorkWithWeights is called, it calls upon vw_GetAdjustedBandWeights to adjust the weight of the No Work Required bands from whatever small weight they are currently assigned for display purposes to 0 so that they do not account for any weight in a patient's WorkScore.

4.3.3 Stored Procedures

Many of our stored procedures are simple SQL queries so do not need explanations. In this section we will discuss some of the most useful and most notable stored procedures in our database.

sp_InsertWork and sp_ArchiveWork:
This stored procedure is the one that is called every time a user logs work for some patient. It overwrites any work log that exists for the same patient in the same current work shift, by calling the sp_ArchiveWork stored procedure. The reasons for overwriting could be an error or an update. An error would be if an incorrect work log was inputted by mistake. An update would be if a patient's state changed during a shift and a previously logged work input for this patient in this shift is no longer accurate. The possible reasons for archiving a work record are stated in the Param table and the ArchiveReason_ID is a foreign key to the Param_ID of the reason.

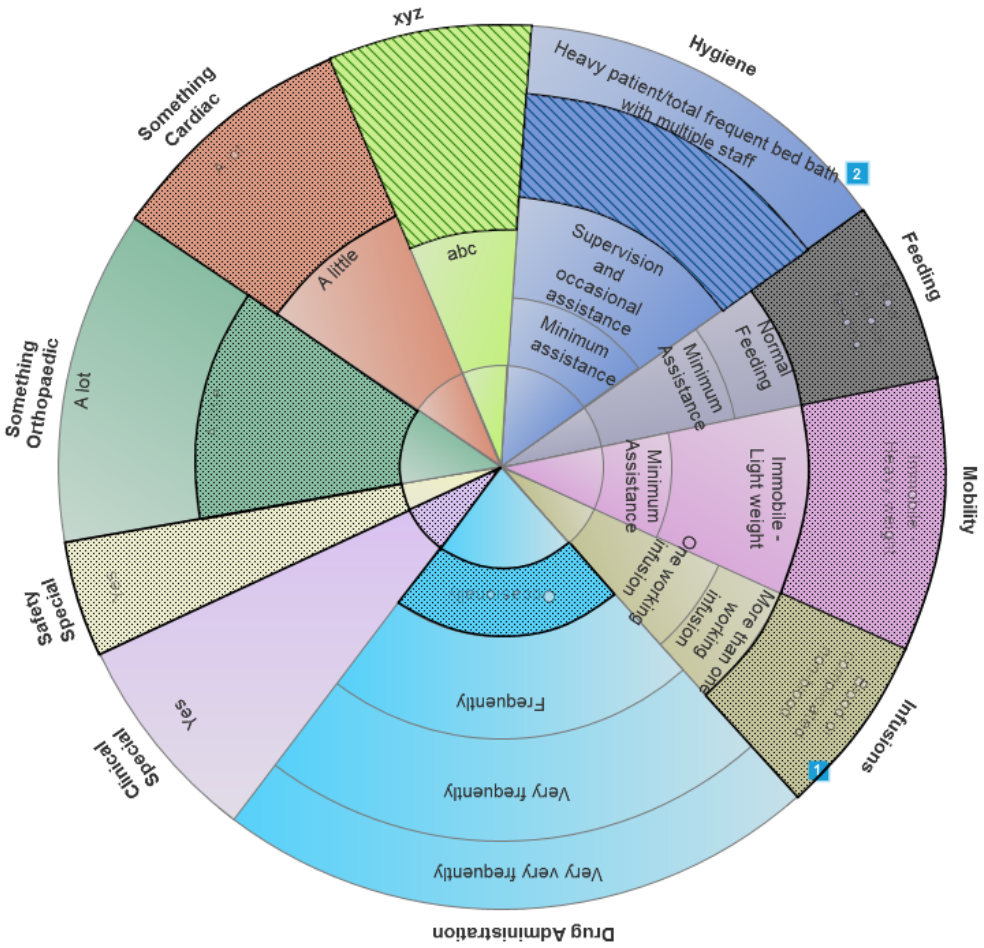
When sp_InsertWork is called, it first checks if an identical work log already exists. When this application is used in practice, a user may log work for a patient during some shift and another user may also intend to do the same thing and log the same work for the same patient in the same shift. There is no need to keep both records in the database, so if an identical (except for staff_ID of who is logging and timestamp) record exists for this patient during this shift, the stored procedure goes no further. sp_InsertWork also takes ArchiveReason_ID and ArchiveFreeText as input parameters. These will be null if this is the first work log for this patient on the current shift, and will be passed to the sp_ArchiveWork if this is an updated work log for this patient on the current shift.

If no identical record exists, the stored procedure checks if an non-identical record exists for this patient in this shift. If it does, then it was either an error and it being updated to the correct work log, or it was accurate at the time of logging but is now being updated because of a change of state of the patient. If a non-identical record exists, the stored procedure sp_ArchiveWork is called and moves the existing non-identical record into the WorkArchive table with a flag for either an error or an update. By default, with no additional input by the user, the application will flag the archived record as being updated, as this is the most likely reason for the archive. We have the functionality written into our database to allow for the user to inform the application that this is an error instead of the default update but we did not get time to study the graphics library enough to implement this in the user interface.

If no non-identical record exists for this patient during this workshift, then this is the first time any user has attempted to insert work for this patient during this shift. It will be inserted straight into the Work table.

sp_GetWardSubModel:
This stored procedure takes a Ward_ID as an input parameter, and returns the records from vw_GetWardsWithSubModelInclBands which relate to this ward. It is triggered by the user opening the page to log work for patients, and selecting the ward in which the patient resides.

sp_GetPatientCurrentAndPastShiftWork:
This stored procedure was written to accommodate for the functionality we intended the application GUI to have when a user is logging work for some patient. We did not get time to implement this in the user interface. We intended for the GUI when a user is logging work for a patient to look something similar to the following diagram:



The above diagram demonstrates our original intention for what a user sees when they select a patient and are about to log work for them for several tasks.

- If a task has already been logged for this patient during the current work shift, the band that was logged for that task will appear highlighted in some way; in this illustration we have used dots. The tasks that have already been logged for the current shift in this example are Something Orthopaedic, Safety Special, Clinical Special, Drug Administration, Infusions, Mobility, and feeding.
- If a task has not been logged for this patient during the current work shift, the most recent band that was logged in the previous shift will appear highlighted in some way that indicates to the user that this highlighted band represents a **previous** shift so the user knows they still must enter data for this task during the current shift; in this illustration, we have used diagonal lines. The tasks that have not been logged for the current shift in this example, are Hygiene, and xyz, the blue and green slices at the top of the model. We can see which bands were logged for these 2 tasks in the previous shift.

How the stored procedure sp_GetPatientCurrentAndPastShiftWork retrieves this information is detailed below:

- Takes Ward and EpisodeMoove as input parameters
- Finds the sub model that is assigned to the selected ward
- Finds the current WorkShiftDate (will usually be the current timestamp, except if the current shift is a shift that passes through midnight and the current time is past midnight, in which case the WorkShiftDate will be yesterday's date, as the WorkShiftDate is always the date that the current shift started on)
- Finds the current WorkShift_ID and the its WorkShiftOrder in order to find the WorkShift that came before it
- Searches the Work table for records pertaining to the current episode move, on the current work shift, on today's date, and returns them marked as "CURRENT WORK RECORDS"
- Searches the Work table for records pertaining to the current episode move, on the previous work shift (whether that was a previous shift today, or the final shift of yesterday), and returns them marked as "LATEST WORK RECORDS"

UpdateNoWorkRequiredValue:
This stored procedure was written to show how we would set the band sizes of our tasks if we were able to implement the GUI design we had originally intended (see *vw_GetAdjustedBandWeight* in Section 5.3.2).

This stored procedure takes an input parameter of a number to set as the NoWorkRequired value. First, it checks if there is a record for NoWorkRequired in the database, and if there is then it updates it to the new value. Otherwise, if there is no value set yet, it checks that the ParamGroup (variable) for NoWorkRequiredValue exists, and if it does then it adds the new value to the Param table. Otherwise, if the ParamGroup does not even exist yet, it creates a new record for it, and then creates a new record in the Param table for the new value.

4.3.4. Mock Data Insertion

For the purpose of demonstration, we wrote 3 SQL scripts which populate the tables in our database with mock data from a CSV file we created, called WorkModelUpload.csv in our project's DBwork folder, which is intended to replicate the results of the Time & Motion study that a hospital must carry out as a prerequisite to using our application.

bulkInsert.sql and InsertInsert.sql:
This script populates the core tables needed to use the application, WorkModel, WorkTask, TaskBand, WorkSubModel, SubModelTask, SubModelDefault, and SubModelDefaultBands. One of the scripts is called bulkInsert.slq which bulk inserts the information from WorkModelUpload.csv into a temporary table which will then be used to populate the WorkModel table, WorkTask table, and TaskBand table. We developed a lot of our project by running our SQL scripts on the databases residing on our own local PCs. We realized the bulkInsert file would cause problems when we would try to run it on our cloud hosted database. The bulk insert command uses an absolute file path. Ther server in the cloud hosting our database would, of course, not be able to find the file with this path, as it is an absolute path to the file on our local PCs. We tried some options and ended up manually inserting the records in a new file called InsertInsert.sql.

The InsertInsert file sets the value of the NoWorkRequired band to 0 using the stored procedure UpdateNoWorkRequiredValue (the explanation of this stored procedure explains why it is needed). Next the script creates a WorkModel named FirstWorkModel. A temporary table called #WorkModelUpload is created to store all of the information we manually inserted (the information which we copied over from WorkModelUpload.csv).

Next, a temporary table called #WorkTaskUpload is created which is almost identical to the WorkTask table except that it includes columns for the results of the Time & Motion study. The script then performs calculations on these columns to convert the average time nurses spend on this task into a WorkModelRatio for this task.

Now that the data has been prepared and is waiting in temporary tables to be inserted into permanent tables, the script creates a WorkModel record named FirstWorkModel. It then add records to the WorkTask table for each task in the temporary table #WorkTaskUpload. Next, it enters records in the TaskBand table for the different bands retrieved from the Time & Motion study. These entries currently reside in the temporary table #WorkModelUpload so they are copied from there into the TaskBand table. Then for every WorkTask that has been created, a band with the NoWorkRequiredValue is created.

At this point, the script has created a WorkModel and populated it with WorkTasks which each are populated with TaskBands. Next, 3 WorkSubModels are created from this parent WorkModel, and for each, a subset of its WorkTasks are assigned.

Next, for each of these 3 WorkSubModels, the script creates 2 Default Models. For each task in these models, a band is selected to be the default band for that task. The temporary tables created earlier are now dropped, and when the script ends, our database is populated with:

- 1 WorkModel
- 12 WorkTasks
- 3 WorkSubModels as children of the WorkModel, each containing a subset of the 12 WorkTasks
- 6 DefaultModels, 2 for each WorkSubModel, each containing a selected default band for every task of the WorkSubModel they were created for

populateTablesRough:
This script populates the tables in our database that we are working on the assumption of populating with data from the hospital's own existing IT system. It fills up tables like Staff, Department, Ward, Bed, WorkSequence, Specialty, Diagnosis, etc. It does not populate the Patient, Episode, or EpisodeMove tables as they are populated in our third script. This script also assigns the WorkSubModels we created in the InsertInsert.sql script to Wards in the hospital, and assigns DefaultModels to each ward also.

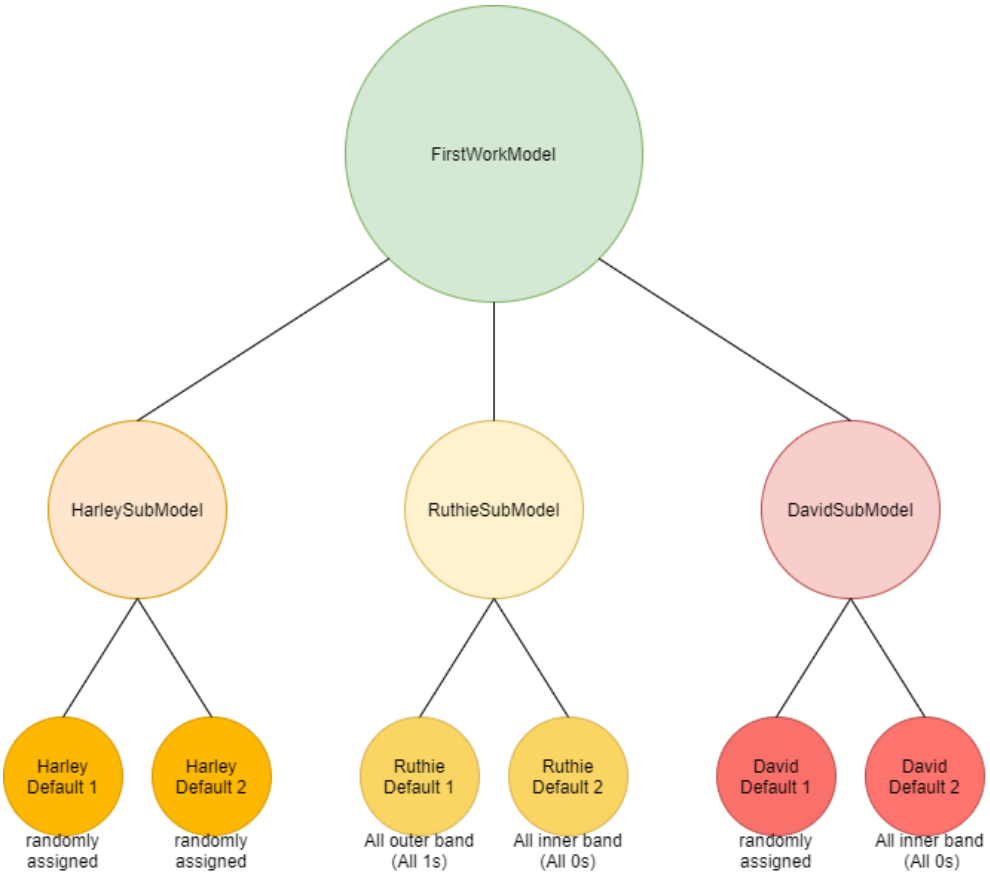
When this script ends, the main tables we need to be aware of are:

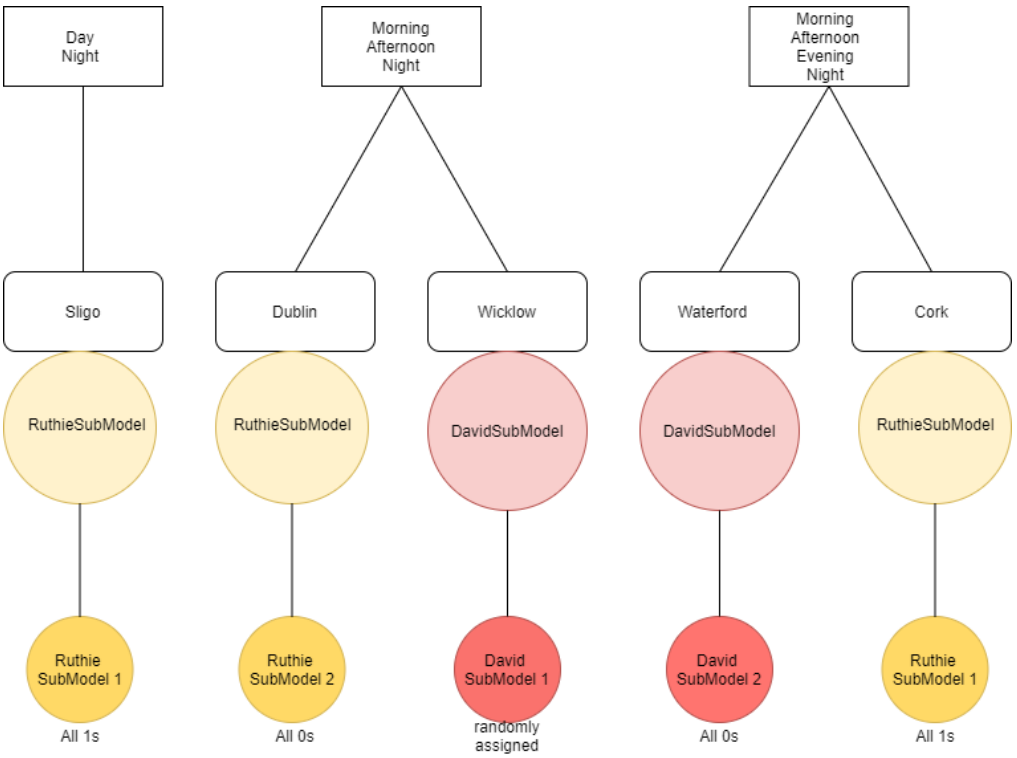
- 8 specialties
- 14 diagnoses
- A collection of work shifts, each a part of some work sequence

- 6 staff types, including one called dedicated to the system which is used to insert mock work logs in populatePatients.sql so that we can tell that a real user did not log those records
- Multiple staff members of each type
- A collection of RosterShifts (we intended to fill out a roster so that we could assign certain nurses to be working particular times on particular days but we dropped this to prioritize other features we were working on)
- 3 departments named East Wing, West Wing and south Wing
- 5 wards named Sligo (East Wing), Dublin & Wicklow (West Wing), Waterford & Cork (South Wing)

The last few lines in this script also assign each ward with a WorkSubModel, a Default Model, and a WorkSequence.

Therefore, our hospital set up at the end of this script is illustrated in the following diagrams:





populatePatients.sql (controlled variety in mock data):
This script populates the Patient, Episode, and EpisodeMove tables with patients over a course of 4 months, from the start of February 2021 to the end of May 2021. It creates a number of Patients, and gives them all an Episode and an EpisodeMove. It runs the stored procedure sp_InsertDefaultWorkForDateRange_DEV on each patient to log work for every patient during every shift on every ward. This inputs approximately 58,000 records into the WorkTable.

Below is the set up of the hospital database once the initial patients are added and their default work has been logged:

Ward	**No. of patients**	**Default Bands of WorkSubModel**
Sligo	8	RuthieSubModel1 (all 1s)
Dublin	8	RuthieSubModel2 (all 0s) (some overwritten shortly)
Wicklow	4	DavidSubModel1 (random choice)
Waterford	4	DavidSubModel2 (all 0s)
Cork	2	RuthieSubModel1 (all 1s)

- All 8 patients in the Sligo ward will have a workload score of 100, because the default band of each task logged for them is the band of weight 1.

- All 8 patients in the Dublin ward will have a workload score of 0, because the default band of each task logged for them is the NoWorkRequired band of weight 0.
- All 4 patients in the Wicklow ward will have a random workload score between 0-100 because the default band of each task logged for them is randomly chosen using the following SQL code (this default band is set in InsertInsert.sql):

```
954  set @defaultBandID = (select top(1) b.TaskBand_ID from dbo.TaskBand b where b.WorkTask_ID = @taskID order by newid())
```

- All 4 patients in the Waterford ward will have a workload score of 0, because the default ban do feach task logged for them is the NoWorkRequired band of weight 0.
- All 2 patients in the Cork ward will have a workload score of 100, because the default band of each task logged for them is the band of weight 1.

We intentionally configured these wards to have some similarities between them and some controlled variety, so that we can see this reflected in our graphs in the analysis section of our application.
Sligo, Dublin and Cork all use the same WorkSubModel: RuthieSubModel.

- Sligo and Dublin have the same number of patients but different Default Models.
- Sligo and Cork have different numbers of patients but the same Default Models.

Wicklow and Waterford use the same WorkSubModel: DavidSubModel.

- Wicklow and Waterford have the same number of patients but different Default Models; Wicklow's default model inserts random bands for every task, where Waterford's default value inserts NoWorkRequired for every task.

Once this has been done, the script then calls the stored procedure named sp_UpdateWorkForDateRangeDayOfWeek_DEV to add some further controlled variety on the Dublin patients. This updates the work logs for Dublin patients throughout the 4 month period so that:

- Mon-Thurs during the morning shift only, Staff member CNM1 logs the largest band (weight 1) for all tasks, meaning the workload score of a patient at these times is 100.
- Fri-Sun for all shifts of the day, Staff member StdN1 logs the smallest band besides NoWorkRequired. The WorkSubModel that Dublin uses has been configured so that all of its task's smallest bands besides 0 are of weight 0.2, meaning the workload score of a patient at these times is 20.

This additional controlled variety allowed us to make clearer graphs in our analysis section because every value was not the exact same, and we knew where to expect the differences.

5. Problems and Resolutions

5.1. Covid 19

Covid-19 had a massive effect on the approach to our final year project. DCU campus was closed for lectures, only set times were given to us for use of the computer labs and very limited seats were available in the library. Neither of us were rarely located in the same county and extreme Covid-19 government restrictions for a large portion of the year prevented us from seeing one another in person. This was a complete contrast to any large project we had previously done (E.g. third year project), where we were able to frequently meet up and work alongside one another. This did cause problems regarding efficient

communication as well as limited our capabilities at times as pair programming and face-to-face interaction were processes we consider ourselves to work at our best in. To try resolve these problems we frequently shared contact over social media and had many zoom calls during times where we were having issues. We also organised weekly zoom meetings with our project supervisor, Dr. David Sinclair, to share our progress and problems with.

5.2. Time Management and Prioritization

This is a final year project to be completed over the course of semester one and semester two. In our functional spec, we put together a schedule for the implementation and completion of each part of our project. However, due to required study for other modules being taken and problems we faced throughout our projects development, there were times where we did not meet the schedule deadlines we set ourselves. On occasion, we struggled to manage our time, especially when certain aspects of the project took longer than we expected due to unfamiliarity or unexpected issues faced. To try solve this problem we frequently weighed the aspects of the project which we deemed higher priority and made decisions at times to leave a lower priority feature if it was taking too long. We also updated our schedule to stay on track and maintain a structured approach to the development of our project.

Moving sql statements from java to database. Did not get time to move them all over.
Rewriting SPs as views.

5.3 PC issues

During project development, Ruth's laptop broke and was weeks in repair by the laptop provider before being deemed unrepairable and a replacement finally being sent to her. The time period for this was nearly the entire second semester with frequent attempts at repairing it over 8 weeks, the case being further investigated before a replacement was sent and took another two weeks to be delivered. This caused stress among the team and problems in our project development as Ruth's old laptop (which she was working with) slowed down completely as she continued to work on the project due to memory issues. This problem was resolved through our supervisor David Sinclair as well as Gary Conway and Sean Haran who set Ruth up with a lab computer which she could access remotely and gave her permission to download any necessary software for the project. This solution was used until Ruth finally received her replacement laptop.

5.4 Encryption of database

A real world implementation of our project would deal with sensitive personal data. Many of our database tables require encryption in order to keep personal information safe and secure. Our SQL Server database has a feature called AlwaysEncrypted which we attempted to use to encrypt our database. We ran into some problems and decided to drop the encryption of our database since it was not a core feature of what our application does.

We want to stress that we recognize how important encryption is in a setting such as ours, but after spending 3 weeks trying to encrypt the database and getting closer and closer to success but still not fully accomplishing it, we decided to leave encryption aside and focus on the key functionalities we designed our app to have. In this section of the technical

specification, we will explain how we went about attempting to encrypt our database and where we ran into problems.

We used SQL Server's Always Encrypted feature. There are two types of encryption available: deterministic and randomized. Deterministic always generates the same ciphertext for some given string. Randomized could encrypt the same string differently every time it is encrypted. Columns which will be queried or grouped need to be encrypted with the deterministic method. So, our primary key columns in each table would need to be encrypted with the deterministic method and our non-prime columns could be encrypted with the randomized method.

The columns are encrypted with a Column Encryption Key which would encrypt selected columns in our database. This CEK is stored in the database itself and the key is encrypted by a Column Master Key. So having access to the database itself is not enough to encrypt or decrypt data, one must have access to the CMK.

The CMK is stored externally outside the database to keep it secure. The CMK can be stored in Windows Certificate Store, Azure Key Vault, or a Key Store Provider. Our project would require the CMK to be stored within our java application in a Java Key Store Provider.

Inserts to the database could then be generated in plaintext, and the application would use the CMK to encrypt the column values before they enter the database. The database would therefore only ever see the encrypted data.

Queries to the database would also be generated in plaintext, and the application would use the CMK to encrypt the column values before the database matches them with its own encrypted values. Upon returning a resultset to the application, the information would be encrypted and the application would use the CMK to decrypt it to display back to the user.

We encountered three problems when trying to encrypt the database:

1. We could not figure out where to store the CMK. Our research and attempts seemed to imply that it had to be stored in a specific key store software such as Windows Key Store, Azure Key Vault, or some other Key Store Provider. However, if we stored our key in our Windows Certificate Stores, our apps users would of course not have access to the folders on *our local PCs*. So, we tried to utilize Java Key Store in our application, and store the key in a folder inside our project directory. The error messages we were receiving seemed to imply that the application could not find the key from the given path. It seemed to imply that the key must be stored in Windows Certificate Store rather than just a folder in our application.

2. We found ourselves in a deadlock situation when inserting NVARCHAR data into our database from our application. The AlwaysEncrypted feature of SQLServer requires data to be inserted in stored procedures. It would not work if SQL statements written in java tried to insert data. This was okay because we wanted to keep our SQL code in our database and for our java application to just call upon it.
   Our JDBC statements declare the data type of the input parameter we are sending to these stored procedures to insert data. For a java String, the default SQL data type is

VARCHAR. Our columns were of data type NVARCHAR so we had to explicitly declare in our JDBC statements that these input parameters were to be inserted as NVARCHAR, using the “N(?)” notation.

However, the stored procedures seemed to not accept the input parameters if “N(?)” was specified. We could not figure out why the stored procedures were not accepting the correct data type when we declared it to be the correct data type.

3. AWS RDS has an option to encrypt the database at the server side upon creation, but this option was not available to us on the student Free Tier package we had.

Possible resolutions we would look into if we have the time:

1. Further research into the Java Key Store library to figure out how it can store the CMK within the application.
   Looking into a user downloading a key upon installation of the application and it being stored in their Windows Certificate Store on *their local PC*.
2. Research further the issue of being required to use the “N(?)” notation, but also not being able to use it.
3. The only solution to the AWS problem would be to use the paid service so we have less limitations.

5.5 AWS permissions and bulk insert

For our insertion of mock data, we intended to use a bulk insert command for efficiency. This worked okay on our local PCs because we could give the absolute file path to the CSV file that contained the information we wanted to bulk insert. However, the AWS server hosting our database would not be able to use this absolute path to find the file because it would be searching its own storage.

To solve this problem, we attempted to use AWS S3 service to store the CSV file, and then bulk insert from S3 to RDS. We had to use the AWS IAM service to create IAM policies to allow this. The IAM policies are essentially just a statement of rules. We needed to create IAM policies to allow our S3 storage bucket and our RDS database to communicate. When we tried to create these policies, we were informed that our IAM user did not have the permissions to create IAM policies.

We could not figure out how to create these policies. We attempted to contact AWS Customer Support but this is not supported on the Free Tire version.

5.6 Power BI

Power BI was chosen as the analytics service to be used for the analysis section of our project with the intention of embedding the Power BI dashboard into the desktop application. However, it was discovered when developing this section that Power BI can only be embedded into a web page. Our resolution to this problem was to link the Power BI dashboard to the appropriate “Analysis” button on the application which opens the dashboard in a web page in the user’s browser. In order to successfully implement this, we

needed to create an embed code that would allow the application direct access to the project dashboard without the need to login to a POWER BI account. When we attempted to implement this, however we were met with an error message which told us that we did not have permission to do this and to contact our organisation admin (DCU ISS) to enable this feature for us. We emailed them and have not been granted permission since.

5.7 JFreeChart display

In the initial design on how the patient task pie chart would display (See User Manual for illustration), we intended for the task bands to exist within their corresponding task slice of the pie chart. During development, jFree and MPAndroidChart java libraries were used due to their built in methods to create pie charts with data as well as the onClick listeners both contained. However, by using these libraries there was no direct way to split a slice of the pie chart into sections (where the task's bands were to be placed). To resolve this issue, the bands were instead displayed on a new JFrame (in the desktop app) and a new layout resource file (mobile app) for the user to select once a task was selected.

6. Results

- Allows users to login.
- Grants users account permission depending on their staff type.
- Connects to an AWS cloud based database.
- Allows users to configure information necessary to the functionality of the app: WorkModel, SubModel, Tasks, TaskBands, Wards, Departments, SubModel defaults.
- Selects database data and displays the data on the application interface.
- Updates data in the database where changes have been inputted on the application interface.
- Deletes data in the database where deletions have been made by the user on the application interface.
- Displays visualization via pie chart of database data (tasks) to the user on the application interface.
- Allows users interaction with the pie chart visualization.
- Allows users to view staff information.
- Allows users to update staff passwords through the application and updates accordingly in the database.
- Implements stored procedures to hold SQL queries.
- Implements views to join data in the database for analysis and ease of retrieval from the application.
- Implements SQL calculations for application and analysis use.
- Analyses data collected from the database.
- Displays data collected for analysis as visualisation graphs for the users.
- Provides predictive analysis to the user of data displayed in analysis.
- Allows the user to filter the analysis graphs to display desired data.

7. Future work

7.1. Encryption

Future work on encryption would be necessary as we are aware this system deals with personal data which needs to be encrypted for security purposes. Encryption is discussed in section 5.4 within Problems and Resolutions.

7.2. Defaults every shift.

Our mock data inserts the default band values for every patient for every shift. This currently only happens when the mock data insertion scripts are run. We would continue to work on this aspect in the future so that the default band is automatically logged to the database at the end of a shift if the task itself holds no record of being logged that shift.

7.3. Outlier tasks.

The concept of outlier tasks are looked at for the scenario when a patient is located in one ward but may need tasks performed that are more commonly found on another ward. This would be an additional task that nurses log work for external to the WorkSubModel assigned to the patient's ward. This may happen due to an underlying condition or by needing to be transferred between wards because different procedures need to be carried out. We have created our database to accommodate outlier tasks but our application has yet to allow the user to log work for them and so that would be an area of future work. The vw_GetWorkScores view flags work logs if they are outlying tasks from the WorkSubModel in use.

7.4. Band representation on GUI

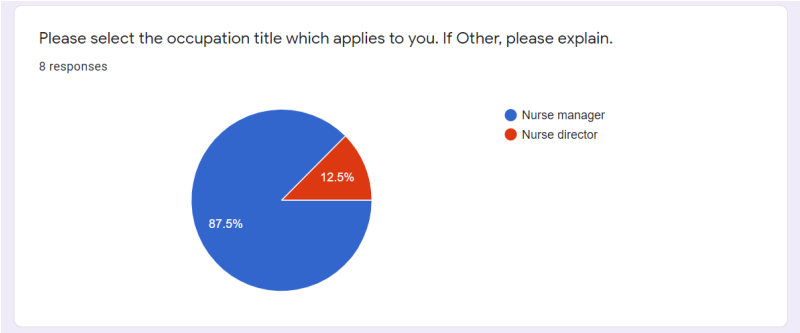
As mentioned in **5.8 JFreeChart display**, the initial design involved each task band being displayed on the model presented to the user. This aspect would be further explored and implemented through future work. This would also involve the application use of vw_GetAdjustedBandWeight mentioned in **4.3.2. Views.**

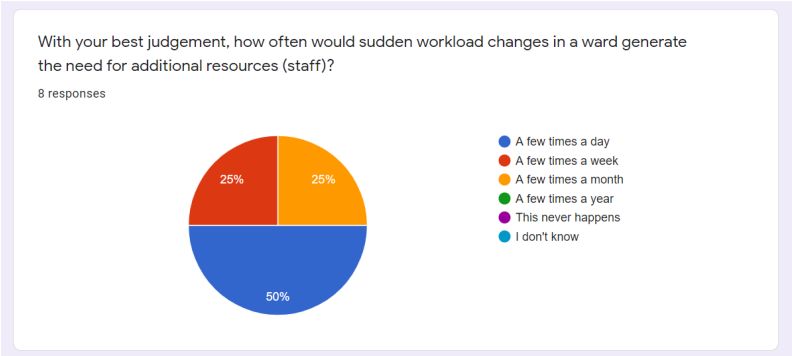
More application modifications we would implement in future work would be to allow for the most recently logged band for a patient's task to be highlighted. This would let nurses know which tasks have already been entered for the shift and also show what bands and tasks were entered in the previous shift, if they were not entered this shift. This would make use of our stored procedure in section **4.3.3 sp_GetPatientCurrentAndPastShiftWork.**

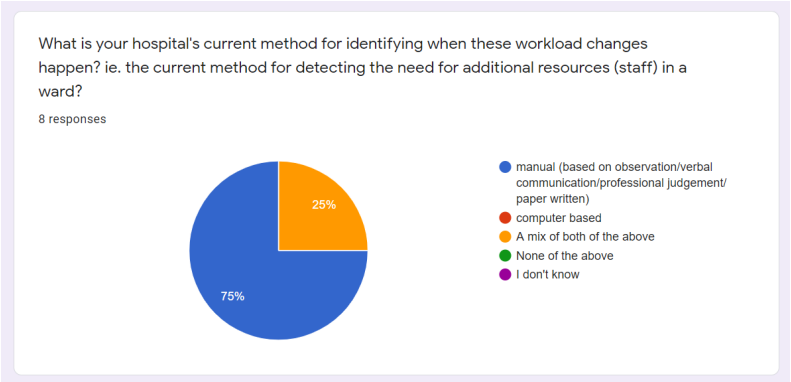
9. Research

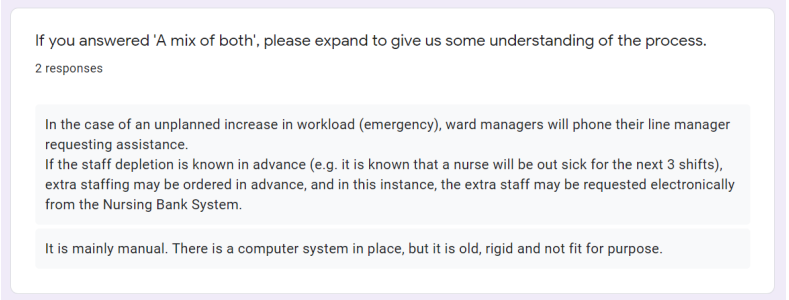
9.1. Requirement Research

As part of research for this project, a questionnaire was carried out with a collection of eight nursing staff working in an Irish Hospital. The nature of this questionnaire was completely anonymous and appropriate ethical approval was granted from DCU. Below are the results gathered from this questionnaire which helped us gauge a better understanding of expectations and requirements from currently working nurse professionals.



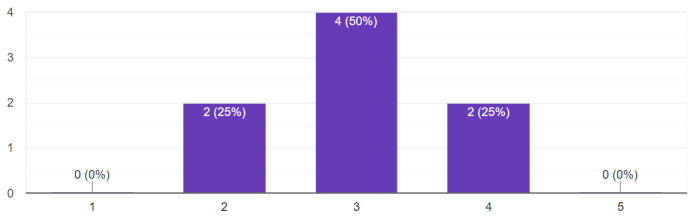






On a scale of 1 to 5, how satisfied are you with your hospital's current method?

8 responses



Please outline the strengths/advantages of your hospital's current method.

8 responses

There is clarity on what staff are needed and when.

we get additional resources by ringing or reporting verbally.

It maintains open channels of communication.

at times it can be let to us as ward managers to try and source extra resources especially if its last minute. we would ask our ward staff to assist in overtime if we are short, this can be quite time consuming on busy wards. at times our CNM3 and site manager will assist and try to source extra resources.

Hospitals rostering is primarily based on historic rostering with periodic changes based on patient dependency.

I believe you have to visually go to a ward to estimate the activity and respond efficiently

It has some value for generic wards

It appears to be working when I need help I ask and most of the time I get it

Please outline the weaknesses/disadvantages of your hospital's current method.

8 responses

Whilst the information is gathered and available, the demand for extra staff can often exceed the supply.

There is no set plan or policy in place to identify and provide additional resources when required.

Time delays in contacting appropriate people.

it is time consuming for ward manager to try and source extra resources when the wards are busy

This is anecdotal evidence with no record of decision making data. Not evidence based. Nil methodology for calculations of daily WTE

Time consuming

Too subjective and completely unsuitable for specialty areas. Too rigid.

short notice to senior managent for help can sometimes mean extra agency staff cannot be got

Please outline any features you would add to/remove from your hospital's current method.

8 responses

It would be useful to have all ward demands assessed by higher management on a dashboard so that extra staff could be sent to the busy wards, hence quicker intervention.

A permanent method to provide additional resources.

Add an online timeline of events/communications from each unit which can be audited and tracked.

extra resources needed to assist ward based staff and managers

All hospital systems need an easy to use, efficient electronic acuity/measurement system to determine the safest staffing and skillmix for a unit/ward/service.

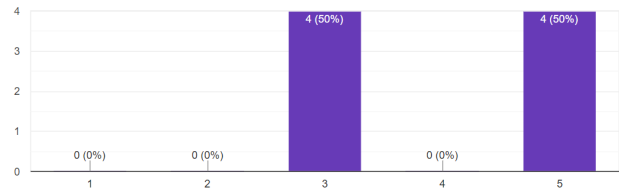
an electronic first stage before visiting the ward would be helpful

Ensure all nursing activities are included. Review and update the scoring method.

Delay in getting staff and getting them in to fill the deficits

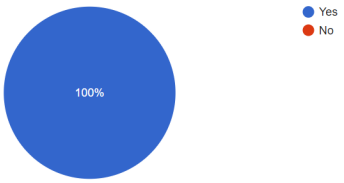
On a scale of 1 to 5, how helpful might you find our proposed system?

8 responses



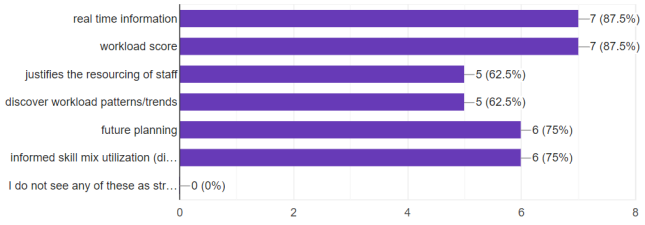
The main goal of our proposed system is to calculate a workload score to indicate how busy a ward is in real time. Would you find this workload score beneficial?

8 responses



What feature(s) of our proposed system would you consider to be strengths/advantages ? You may select more than 1.

8 responses



Please propose any other features you can think of that could further enhance the system we are creating.

8 responses

- Must be user friendly, quick tick-box options requiring minimal typing and free text sections.
- There should be an assigned extra nurse to collect the data.
- Inclusion of specialized areas.
- none
- An easy to use, easily accessible system that could be used and integrated with community nursing. A system which determines the skillmix i.e. HCAs for specials and interlinks with a robust rostering system
- I think this system would work well in combination with visibility to visualise the activity.
- Be able to project workload for following shift/day. Thank you
- This ward can go from quiet to extremely busy in a matter of hours..will staff get the time to submit dependency levels while bells are ringing and patients need attention?

9.2. Course Material Researched

Principles of Databases JDBC & Stored Procedures - Dr. David Sinclair

Principles of Databases Normalisation - Dr. David Sinclair

Views - Dr David Sinclair

CA218 - Lecture 4 - SQL Notes - Dr. Mark Roantree

CA218 - Lecture 8 - Normalisation Notes - Dr. Mark Roantree

CA214 Systems Analysis Notes - https://www.computing.dcu.ie/~renaat/ca2/ca214over.html

CA4004 Soft. Eng.:Process,Principles & Methods - Dr.Paul Clarke - https://loop.dcu.ie/course/view.php?id=50109

CA269 - Object Oriented Programming - Charlie Daly

9.3. Research References

[1]
https://data.oireachtas.ie/ie/oireachtas/committee/dail/32/joint_committee_on_health/submissions/2019/2019-11-13_opening-statement-phil-ni-sheaghdha-general-secretary-irish-nurses-and-midwives-organisation-inmo_en.pdf
(last accessed 06/05/2021)

[2]
https://www.inmo.ie/Home/Index/217/13597
(last accessed 06/05/2021)

[3]
https://www.researchgate.net/publication/6261377_Measuring_the_workload_of_community_nurses_in_Ireland_A_review_of_workload_measurement_systems
(last accessed 06/05/2021)

[4]
https://docs.oracle.com/
(last accessed 06/05/2021)

[5]
https://docs.oracle.com/javase/tutorial/uiswing/components/passwordfield.html
(last accessed 06/05/2021)

[6]
https://examples.javacodegeeks.com/android/core/ui/password/android-password-field-example/

(last accessed 06/05/2021)

[7]
https://www.codegrepper.com/code-examples/java/arrays.fill+in+java
(last accessed 06/05/2021)

[8]
https://coderanch.com/t/345317/java/clear-JPanel-repainting
(last accessed 06/05/2021)

[8]
https://www.oracle.com/ie/tools/technologies/netbeans-ide.html
(last accessed 06/05/2021)

[9]
https://netbeans.apache.org/
(last accessed 06/05/2021)

[10]
https://hackr.io/blog/best-java-ides
(last accessed 06/05/2021)

[11]
https://www.edureka.co/blog/netbeans-tutorial/
(last accessed 06/05/2021)

[12]
http://www.java2s.com/Tutorials/Java/Swing_How_to
(last accessed 06/05/2021)

[13]
https://docs.oracle.com/javase/7/docs/api/java/awt/event/MouseListener.html
(last accessed 06/05/2021)

[14]
https://bits.netbeans.org/html+java/1.7/org/netbeans/html/context/spi/Contexts.html
(last accessed 06/05/2021)

[15]
https://www.daniweb.com/programming/software-development/threads/347876/joptionpane-unwanted-repeated-iterations
(last accessed 06/05/2021)

[16]
https://developer.android.com/
(last accessed 06/05/2021)

[17]

https://github.com/PhilJay/MPAndroidChart
(last accessed 06/05/2021)

[18]
https://stackoverflow.com/questions/61066069/how-to-customize-pie-chart-using-mpandroidchart-in-android
(last accessed 06/05/2021)

[19]
https://stackoverflow.com/questions/16901331/jfreechart-customizing-pie-chart-labels-for-each-pie
(last accessed 06/05/2021)

[20]
https://medium.com/dhruvishadarji/pie-chart-using-mpandroidchart-library-61b4d003f11
(last accessed 06/05/2021)

[21]
http://www.ing.iac.es/~docs/external/java/jfreechart/org/jfree/
(last accessed 06/05/2021)

[22]
https://www.tutorialspoint.com/jfreechart
(last accessed 06/05/2021)

[23]
https://www.javatpoint.com/jfreechart-tutorial
(last accessed 06/05/2021)

[24]
https://stackoverflow.com/questions/16901331/jfreechart-customizing-pie-chart-labels-for-each-pie
(last accessed 06/05/2021)

[25]
https://www.prostrategy.ie/microsoft-sql-server/
(last accessed 06/05/2021)

[26]
https://www.sqlservertutorial.net/getting-started/what-is-sql-server/
(last accessed 06/05/2021)

[27]
https://www.sqlshack.com/
(last accessed 06/05/2021)

[28]

https://www.w3schools.com/sql
(last accessed 06/05/2021)

[29]
https://docs.microsoft.com/en-us/sql
(last accessed 06/05/2021)

[30]
https://blogs.oracle.com/sql
(last accessed 06/05/2021)

[31]
https://www.sqlservertutorial.net/sql-server-basics
(last accessed 06/05/2021)

[32]
https://www.tutorialspoint.com/java-sql-date-valueof-method-with-example
(last accessed 06/05/2021)

[33]
https://www.w3schools.com/sql/sql_stored_procedures.asp
(last accessed 06/05/2021)

[34]
https://docs.oracle.com/javase/tutorial/jdbc
(last accessed 06/05/2021)

[35]
https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html
(last accessed 06/05/2021)

[36]
https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html
(last accessed 06/05/2021)

[37]
https://stackoverflow.com/questions/15739159/android-jdbc-jtds-connection-to-mssql
(last accessed 06/05/2021)

[38]
https://stackoverflow.com/questions/4393766/differences-between-ms-sql-microsofts-jdbc-drivers-and-jtdss-driver
(last accessed 06/05/2021)

[39]
https://www.b4x.com/android/forum/threads/how-to-connect-to-sql-server-using-jtds-1-3-1-jar.111446/
(last accessed 06/05/2021)

[40]
http://jar.fyicenter.com/2736_What_Is_jTDS-JDBC_Driver_for_SQL_Server_.html
(last accessed 06/05/2021)

[41]
https://www.codeproject.com/Tips/1054377/Direct-Access-to-SQL-Server-From-Android
(last accessed 06/05/2021)