Documented Design

High-Level Overview

Flow Chart of entire system:

Output: Login Window

Admin window

Input: Login Credentials

Admin

VALID?

No

Exception Thrown

Yes

Output: Account window

Display Tasks

Input: Button Press

Complete Task

Which button was pressed?

Complete Task

Create Task

Output: Create Task

Create Group

Which sort was selected?

Output: Create Group Window

Sort

Sort by chosen factor

This flow chart shows how the program goes through each form from user input. Initially the program launches in Login. If valid, goes to account, from there the user can select from different options such as set tasks.

Login Flowchart:

Input: Login button pressed

Input: Password

Input: Username

Output: Exception

No

Username valid?

Hash Password

Yes

Are the hashes equal?

Get Corresponding Hash

Yes

Output: Exception

Output: Account

No

This flowchart starts, as seen in the previous flowchart, when the program is launched. The user must input their username and password and press login, the corresponding hash is fetched from their username log in the database, and their input is hashed, if the two match, access is granted, if not, the user if informed and the process repeats.

Set Task:

Output: Exception

Input:

Title, Priority, Description, Users, Groups, Date Set, Date Due

No

Valid Input?

Yes

Create Task record in database

No

Is user hierarchy value lower than target?

Output: Task Created

Yes

Is user owner of group?

Create record in HasTaskID

No

Yes

Create record in User\_Groups

The AddTask flow starts when the corresponding button is pressed in the Account window. Once the input is recognised, the validity is checked, if valid, a task is then created with the data given. That task is then to be set to users and groups, subject to the setter being of a higher role that the recipient, and the setter being the owner of the group being set the task.

Create group and sort do not require a flow diagram, creating of a group is simply name, users(in the same method as setting users a task); sort is identifying which sort was selected, and then using a WinForms function, now SQL is needed

Structure Chart

Admin Subsystem:

Admin Form

Add users

Access login window

Create tables in database

Create database

The admin form will allow users with administrative access to create a database, create tables in the database, add new users/ employees, and then return to the login.

Account Subsystem:

Account Form

Access Create Task Window

Access Create Group Window

Complete Tasks

Sort Tasks

Display Tasks

The account form will display a user’s tasks, sort and complete them, and allow the user to access the create group and task windows.

Create Group Subsystem:

Create Group Form

Add to GroupID table

Add Users to group

Add to User\_Groups table

Name Group

The Create Group form will allow users to create a group, name it, add users, and then store this information in the program database.

Create Task Subsystem:

Create Task Form

Name Task

Set Priority

Set Dates

Set Description

Add to TaskID table

Add task to users and groups

Add to Task\_Groups table and HasTaskID

The Create Task form will allow users to create a task, add a name, description, priority, and dates, set this task to users, and then store this information in the program database.

ISPO Chart

|  |  |
| --- | --- |
| Input:  User Details:  Username, Password, Hierarchy  Task Details:  Name, Priority, Description, Date Set, Date Due  Users, Groups  Group Details:  Name, Users  Sort Details  By Priority, By Date Due, By Date Set  Create Database button  Create Tables button | Processes:  Add User  Add Task  Validate Task Setting  Add Group  Get GroupID  Get UserID  Get TaskID  Display Tasks  Sort Tasks  Complete Tasks  Validate Login  Get User Tasks |
| Storage:  Tables:  UserID  TaskID  GroupID  HasTaskID  User\_Groups  Task\_Groups | Outputs:  Login Form  Account Form  Create Task Form  Create Group Form  User Tasks:  TaskID, Title, Priority, Description, Date Due, Date Set  Exception Message Box: Login failed, task setting failed, group creation failed, complete task failed |

Database Design

All data about tasks, groups, and users, will be stores in a database within respective tables. Databases have been chosen as opposed to file storage because of the key feature, which will ensure there is no overlap in certain key fields, such as UserID.

Additionally, relationships between tables can be kept simple (many-to-one or one-to-many) with the creation of middling tables between a many-to-many relationship, these types of relationships can rarely be used effectively in code. However, with query functions such as joins used in conjunction with these middling tables, all appropriate data can be reached, as well as any overlaps, as will be seen in the query section of the design.

The project utilises seven key tables in order to store all necessary data, maintaining correct relationships. The tables, along with their fields, data types, purpose etc are shown below.

UserID:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Constraint | Data Type | Validation | Notes |
| UserID | Primary Key  Auto Increment | Integer |  |  |
| Name | Unique | String | <30 |  |
| Hierarchy |  | Integer | 0<x<100 | Can be updated for uses by different company size |
| HashValue |  | Integer | 128 bits | Generated using MD5 hashing algorithm, cannot be reversed |

The UserID table will hold User information relevant to the system. UserID will be an identifier within the system, however Name can also be used as it has been specified to be unique. Hierarchy is designed to have the most access or highest role at 1, so that as more positions or levels within a company are made, hierarchies can expand outwards, not having to change every other level if a new one is added, which would be the case if the lowest level was 1. The hash value is an encrypted password, and the details of how it is generated will be detailed in the later sections.

TaskID:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Constraint | Data Type | Validation | Notes |
| TaskID | Primary  Auto Increment | Integer |  |  |
| Title | Unique | String | <30 | Can include numbers |
| Priority |  | Integer | <10 |  |
| Description |  | String | <60 | Can include numbers |
| DateDue |  | Date | Date | Selected with integrated calendar |
| DateSet |  | Date | Date | Selected with integrated calendar |

The TaskID table will hold all information on tasks, similar to UserID, the TaskID field will be the primary identifier for records within the code, however, this could be done with the Title field as it is also unique. Priority operates in the same way as hierarchy, with 1 being the most urgent and 9 being the least, if the client feels this is too little or large of a scope, the range can be reduced or increased to fit the need of the client. Although DateDue originally has logic coded to ensure it was always after Date Set:

int result = DateTime.Compare(DateSet, DateDue);

if (result< 0)

return false

else

return true

Upon consulting the client, they opted out of this logic as tasks which take only a few minutes could be delayed in being set, and although I suggested changing the priority in this case, they wished to allow setting of tasks regardless of this.

GroupID:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Constraint | Data Type | Validation | Notes |
| GroupID | Primary Key  Auto Increment | Integer |  |  |
| GroupName | Unique | String | <30 |  |
| OwnerID | Foreign | Integer |  | Reference: UserID |

The GroupID table will hold all relevant information about Groups of Users. Each group needs an owner, since a group can only have one owner, it isn’t necessary to have a middling table because the relationship is many-to-one. However, upon the creation of a group, it makes sense to ensure there actually is a user to be referenced as the owner. The simplest way to do this is to make OwnerID a foreign key, if there is no associated UserID, the record will not be made, and an exception will be thrown.

After these three key tables have been explained, it becomes obvious that the relationships between them are three many-to-many relationships. As previously mentioned, these relationships are incredibly hard to query against and work with, so the following tables were created to solve this issue.

User\_Groups:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Constraint | Data Type | Validation | Notes |
| GroupID | Primary Key  Foreign Key | Integer |  | Reference:  GroupID |
| UserID | Primary Key  Foreign Key | Integer |  | Reference:  UserID |

Task\_Groups

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Constraint | Data Type | Validation | Notes |
| GroupID | Primary Key  Foreign Key | Integer |  | Reference:  GroupID |
| TaskID | Primary Key  Foreign Key | Integer |  | Reference:  TaskID |

HasTaskID

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Constraint | Data Type | Validation | Notes |
| UserID | Primary Key  Foreign Key | Integer |  | Reference:  UserID |
| TaskID | Primary Key  Foreign Key | Integer |  | Reference:  TaskID |

Each of these three tables have a composite key, made up of the two fields. This means that whilst there can be multiple records with the UserID the same (in the User\_Groups table), there cannot be repeats of a record where the UserID and GroupID are the same. This means- for example- a user can be in multiple groups, groups can have multiple members, but a user cannot be part of a group twice. This is particularly useful in the HasTaskID, ensuring that a user can’t be set the same task twice.

Another feature requested by the user was to be able to complete tasks, which would then not appear in the task view. Because tasks can be set to groups, but the tasks are completed individually, the task record can not be simply deleted from the database; as soon as one user has marked the task completed, it would be completed for all. Additionally, looking back at previous tasks wouldn’t be possible, which would most likely be needed for confirmation of certain dates etc.

To solve this problem, the following table has been implemented:

Task\_Completed:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Constraint | Data Type | Validation | Notes |
| TaskID | Primary Key  Foreign Key | Integer |  | Reference:  TaskID |
| UserID | Primary Key  Foreign Key | Integer |  | Reference:  UserID |

The table is of the same format as HasTaskID, but records which tasks have been completed.

When displaying tasks, logic is used to only display tasks that haven’t been completed, as will be explained in the subroutines section.

DDL

The following DDL has been used to create the database as previously explained.

"CREATE TABLE UserID("

+ "UserID INT IDENTITY(1,1),"

+ "Name VARCHAR(30) UNIQUE,"

+ "Hierarchy NUMERIC(2),"

+ "HashValue VARCHAR(60),"

+ "PRIMARY KEY(UserID)"

+ ")"

"CREATE TABLE TaskID("

+ "TaskID INT IDENTITY(1,1),"

+ "Title VARCHAR(30) UNIQUE,"

+ "Priority NUMERIC(1),"

+ "Description VARCHAR(60),"

+ "DateDue DATE,"

+ "DateSet DATE,"

+ "PRIMARY KEY(TaskID)"

+ ")"

CREATE TABLE GroupID("

+ "GroupID INT IDENTITY(1,1),"

+ "GroupName CHAR(30) UNIQUE,"

+ "OwnerID INT,"

+ "PRIMARY KEY(GroupID),"

+ "CONSTRAINT OwnerID FOREIGN KEY (OwnerID) REFERENCES UserID

(UserID)"

+ ")"

CREATE TABLE HasTaskID("

+ "UserID INT,"

+ "TaskID INT,"

+ "CONSTRAINT FKUser FOREIGN KEY (UserID) REFERENCES UserID (UserID),"

+ "CONSTRAINT FKTask FOREIGN KEY (TaskID) REFERENCES TaskID (TaskID),"

+ "PRIMARY KEY(UserID,TaskID)"

+ ")"

"CREATE TABLE User\_Groups("

+ "GroupID INT,"

+ "UserID INT,"

+ "CONSTRAINT UserID FOREIGN KEY (UserID) REFERENCES UserID (UserID),"

+ "CONSTRAINT GroupID FOREIGN KEY (GroupID) REFERENCES GroupID (GroupID),"

+ "PRIMARY KEY(UserID,GroupID)"

+ ")"

"CREATE TABLE Task\_Groups("

+ "GroupID INT,"

+ "TaskID INT,"

+ "CONSTRAINT TaskID FOREIGN KEY (TaskID) REFERENCES TaskID (TaskID),"

+ "CONSTRAINT FK\_Group FOREIGN KEY (GroupID) REFERENCES GroupID (GroupID),"

+ "PRIMARY KEY(TaskID,GroupID)"

+ ")";

"CREATE TABLE Task\_Completed("

+ "TaskID INT,"

+ "UserID INT,"

+ "CONSTRAINT FK\_User FOREIGN KEY (UserID) REFERENCES UserID (UserID),"

+ "CONSTRAINT FK\_Task FOREIGN KEY (TaskID) REFERENCES TaskID (TaskID),"

+ "PRIMARY KEY(UserID,TaskID)"

+ ")"

Constraint names such as FK\_User have been used in place of UserID, as you cannot have two constraints with the same name in the same database, so they have been used as alternative, but similarly descriptive, placeholders.

Relationship Model:

The relationships previously described can be seen in the following model, particularly that no tables have a many-to-many relationship.

Diagram

Description automatically generated

Algorithms and Subroutines

The most used subroutines in the project as a whole are those which query within SQL. The SQL class includes two methods, ExecuteSql, which accepts a string as a parameter, and queries within the database, without returning a result. This is used for insertion of new records, and creation of tables.

The SQL in the project works on SQL Server. The Connection String contains the database which needs to be queried against, UserDatabase.mdb.

The subroutine works by creating a new instance of OleDbConnection, passing the connection string to the object. It then invokes the command method and tries to open a connection with the database. If this connection is established, the query is then executed. If the query syntax is not accepted, or the connection isn’t opened, an exception is thrown to prevent the project runtime crashing from an unhandled exception.

public void ExecuteSql(string Query)

{

using (OleDbConnection connection = new OleDbConnection(CONNECTION\_STRING))

{

using (OleDbCommand command = new OleDbCommand(Query))

{

command.Connection = connection;

try

{

connection.Open();

command.ExecuteNonQuery();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

}

The subroutine is notedly void because it doesn’t return any values from the queries it executes. Applications of this subroutine will be noted later in the design.

ExecuteSqlReturn

public OleDbDataReader ExecuteSqlReturn(string Query)

{

OleDbConnection connection = new OleDbConnection(CONNECTION\_STRING);

connection.Open();

OleDbCommand command = new OleDbCommand();

command.Connection = connection;

command.CommandText = Query;

OleDbDataReader reader = command.ExecuteReader();

return reader;

}

Although this subroutine looks majorly different upon first glance, both open a new instance of OleDbConnection. The actual difference is this function calls the ExecuteReader function, which returns an array. This array is of type OleDbDataReader and works by storing each column of returned data in one index, which then needs to be iterated through for all data to be accessed, as seen in code. There is no try catch, as if failed, the reader will simply be empty, so the try catch is implemented when trying to read the array, not fill it.

One version of SQL execution has been left in a form, not a method is the SQL object, as it is specific to that form. This is the ExecuteSqlDisplay subroutine:

public void ExecuteSqlDisplay(String sSqlString)

{

DataTable dt = new DataTable();

using (OleDbConnection connection = new OleDbConnection(CONNECTION\_STRING))

{

using (OleDbCommand command = new OleDbCommand(sSqlString))

{

using (OleDbDataAdapter dataAdapter = new OleDbDataAdapter(command))

{

command.Connection = connection;

try

{

connection.Open();

dataAdapter.Fill(dt);

DisplayData(dt);

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

}

}

}

This functions exactly the same, for the most part, as the first function. The only difference being after querying, it fills a data table with the data found in the query. This is used to fill the data table in the Account form.

Before displaying the data, the data grid must be set up. For each field wished to be displayed, a column is defined with an index, title, and format. This is done in the SetupGrid subroutine, called when a new account object is created. For example:

dgvTasks.Columns[2].Name = "Date Due";

dgvTasks.Columns[2].DefaultCellStyle.Format = "dd/MM/yyyy";

The data table form ExecuteSqlDisplay is passed to DisplayData as a parameter. The table is then iterated through, and each relevant column is given values from the returned data:

dgvTasks.Rows[n].Cells[1].Value = \_dt.Rows[i][1];

This extract fills the 2nd column (index starts at 0) with the Title, the data from the second column of the parameter. The n variable takes the value of the first non-filled row, so that there is no overlap of data, or missing rows, this value is obtained from Add function.

int n = dgvTasks.Rows.Add();

The filtering of which tasks to actually display is done majorly by two key functions, TasksFromGroups, and PersonalTasks. TaskFromGroups both identifies which groups the user is part of, and which tasks belong to those groups, as will be shown. PersonalTasks then finds which tasks belong to the user-not through groups- and then returns the set not overlapping with TasksFromGroups.

private List<int> TasksFromGroups()

{

string sSqlString = $"SELECT TaskID FROM Task\_Groups INNER JOIN User\_Groups ON Task\_Groups.GroupID = User\_Groups.GroupID WHERE UserID={UserID}";

var reader = Query.ExecuteSqlReturn(sSqlString);

List<int> GroupTasks = new List<int>();

while (reader.Read())

{

GroupTasks.Add(Convert.ToInt32(reader[0]));

}

return GroupTasks;

}

The query executed works by first creating an inner join between Task\_Groups and User\_Groups, where the GroupID field in both the tables is the same. It could look something like:

|  |  |
| --- | --- |
| UserID | GroupID |
| 1 | 2 |
| 2 | 3 |

|  |  |
| --- | --- |
| TaskID | GroupID |
| 5 | 3 |
| 1 | 3 |
| 6 | 1 |

|  |  |
| --- | --- |
| UserID | TaskID |
| 2 | 5 |
| 2 | 1 |

From this newly created temporary table, the records where UserID is the same as the current user’s ID, are selected, and from there, the TaskIDs are returned.

The query used in PersonalTasks is comparatively simple:

SELECT TaskID FROM HasTaskID WHERE UserID={UserID}

Simply returning the tasks which have been set specifically to the user aside from groups. However, the extra logical step comes in overlapping tasks. Since it is possible to be set the same task through a group and specifically, there must be logic to ensure the same task isn’t displayed twice.

This is done by invoking TasksFromGroups inside of PersonalTasks. Tasks returned from the query are stored in a temporary array, which is then iterated through; if a task is in the array returned from TasksFromGroups, it is not added to the final array, if not, then it is added. This means the arrays returned can be displayed simply, instead of having to use logic when they are displayed to avoid duplication.

foreach (int Task in TempIndividualTasks)

{

if (!GroupTasks.Contains(Task))

{

IndividualTasks.Add(Task);

}

}

return IndividualTasks;

The only other hurdle preventing tasks being displayed, is the user’s ability to complete tasks, before a task is displayed, a check must take place to see if it is completed, as explained previously, the task cannot simply be deleted.

When a task is completed, the task is added to the relevant table, along with he UserID of the user who completed it. From there, the subroutine TasksDone returns the array of tasks which have been completed, if any exist, from the database:

SELECT TaskID FROM Task\_Completed WHERE EXISTS (SELECT TaskID FROM Task\_Completed WHERE UserID={this.UserID})

Each task which is attempted to be displayed can then be checked against this array by passing to the IsTaskCompleted array.

Finally, the user has the option to sort their tasks visually by three different constraints:

* Priority
* Date Set
* DateDue

Whilst this obviously can be done with the SORT BY feature of SQL, since queries are being executed and put in arrays constantly, the trouble comes in when this sorting should be done. However, WinForms has an inbuilt sort feature which can be used one the tasks are displayed. An Example:

dgvTasks.Sort(dgvTasks.Columns[4], ListSortDirection.Ascending);

The grid is now sorted by priority, with the most important (value closest to one) being displayed first.

From the Account form, the main features are creating groups and setting tasks.

The majority of creating a group is simple and done by inserting the new values into the GroupID table, with a try catch in case the input was of the wrong format. The only complication of the creation is adding new users to a group.

The decided upon format that was easiest to implement was referencing users, separated by commas. From there, string handling is used to separate users into an array, which each user stored in a separate index of the array. From there, the text is trimmed to avoid space errors, and inserted into User\_Groups.

foreach (string Member in Members)

{

try

{

Member.Trim();

int MemberID = Convert.ToInt32(Member);

AddUserGroup(GroupID, MemberID);

}

catch

{

}

}

Adding tasks is sightly more complicated and requires many subroutines, however, since many of them function is a very similar way to previously explained functions, their explanations will be omitted.

The users and groups that are be set the tasks are subject to the respective constraints: the user setting the task has a higher hierarchy (lower value) that who is being set the task, and the personal assigning the task to the group is that group’s owner.

For the first constraint, CanSetTask accepts two UserIDs as parameters. The GetHierarchy function then returns the hierarchies of the two users. If the first user’s hierarchy is of a lower value than the second’s, then the Boolean value true is returned, if not, false is returned. This can be then used for each target user to deem if the task can be set.

private bool CanSetTask(int UserID, int TargetID)

{

int UserHierarchy = GetHierarchy(UserID);

int TargetHierarchy = GetHierarchy(TargetID);

if ((UserHierarchy != 0) && (TargetHierarchy != 0))

{

if (UserHierarchy <= TargetHierarchy)

{

return true;

}

else

{

return false;

}

}

else

{

return false;

}

}

To determine whether a user can set a task to a group, the function GetGroupOwner is used, passing the GroupID as a parameter, the following SQL query is executed:

SELECT OwnerID FROM GroupID WHERE GroupID={GroupID}

This value is then compared to the current UserID, if matching, the task is added to Task\_Groups.

Security

Since this project requires different accounts for different users, there will need to be some form of security to ensure only valid access to accounts is granted. This is done with a hashing algorithm, specifically in this project, the MD5 algorithm.

The MD5 algorithm accepts strings of any length and acts as a hash function, producing a 128-bit hash value. Hashing is a one-way type of encryption; one cannot decipher the original input from the hash.

The algorithm breaks up the input into 512-bit blocks and pads the message to be a multiple of 512, by adding a single 1, and then as many zeros as needed. The algorithm operates on 4 32-bit words, with 4 stages of 16 operations. The four functions:

Text, letter

Description automatically generated

This algorithm is accessed through the Microsoft Security.Cryptography library, which contains many hashing algorithms. MD% was chosen because, whilst there are vulnerabilities, the hardware requirements are sufficiently low, and since, as of 2019, a quarter of content management systems still use MD5, it proves sufficient for this purpose, and can even be updated later in the project’s lifetime.

The encryptor class contains the Hash method which is where MD5 is accessed.

Upon invoking the method, a new instance of MD5 in instantiated, the hash is computed and returned as a byte array. This is then changed to hexadecimal to be stored as a hash, and returned:

public static string Hash(string text)

{

MD5 md5 = new MD5CryptoServiceProvider();

//compute hash from the bytes of text

md5.ComputeHash(ASCIIEncoding.ASCII.GetBytes(text));

//get hash result after compute it

byte[] result = md5.Hash;

StringBuilder password = new StringBuilder();

for (int j = 0; j < result.Length; j++)

{

//change it into 2 hexadecimal digits

//for each byte

password.Append(result[j].ToString("x2"));

}

return password.ToString();

}

This method is invoked in adding users, and in the login. For the purposes of security, the login logic and the use of MD5 will be shown.

When a new user is added, their password is stored as a HashValue in the UserID table, this is then queried against to retrieve when a user is logging in.

SELECT HashValue FROM UserID WHERE Name='{Username}

Where username is the name entered. If no username or hash is retrieved, null is returned. The password entered by the user is then hashed:

string Hash = GetHash(Password);

Finally, if the two hash values match, a new account object is created, if they do not match, then user is informed through a message box, and if a hash value was not retrieved, the user is informed the username is not stored on the database.

if ((Hash == Password) && (Hash != "null"))

{

var Account = new Account(Username);

Account.Show();

MessageBox.Show("Login Successful!");

}

else if (Hash == "null")

{

MessageBox.Show("Username not found!");

}

else if ((Hash != Password) && (Hash != "null"))

{

MessageBox.Show("Username and Password do not match!");

}

GUI

An important element of the project is the graphical user interface, it must enable the user to access all features, and must be intuitive to use so that extensive training isn’t required to use the system

From research in other similar programs, such as Google Classroom, the following forms have been created. Whilst they aren’t as aesthetic as other programs, largely due to the limitations of google forms (and my designing abilities), I believe they are fit for purpose.

The AddUser from, accessible by admins and those with total control of the system.

Graphical user interface

Description automatically generated

Labels for text boxes

Close form, can be omitted as forms have banner with close and minimize

Text Boxes for user input

Account form accessible after login.

Row headers for the task data

Graphical user interface

Description automatically generated

Radio buttons for task sorting

Links to other forms

Calendar for user organization

Data grid view where task will appear

AddTask from accessible via button in account form

Named group of elements

Graphical user interface

Description automatically generated

Date selector that allows user to pick from calendar

Larger text box due to bigger character allowance of description

Automatically appearing text to show desired input

CreateGroup form accessible via button in account form

Graphical user interface, application

Description automatically generated

Button to execute request

Suggested input as seen in SetTask

Form accessible only by administrators allowing creation of users, database, and tables.

Graphical user interface

Description automatically generated

Links to forms used by regular users as seen previously.

Creates individual tables and fields, can only be done after creation of database.

Creates database which can be seen in software like Access.