THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

DEPARTMENT OF
COMPUTER SCIENCE
Te Tari Rorohiko

2025

Contributors

J. Turner

R. Mercado

V. Moxham-Bettridge

J. Kasmara

THE SOFTWARE DEVELOPMENT LIFECYCLE: EVALUATION

Last week you began building the backend for your application. This week we will continue with your example including some specific functionality and queries for your database. We will look at SQL in more detail and how we can display different types of data.

The Parkway Commute Database Structure

Recall that in CSNeT Level 1 we introduced you to the basic concepts of databases, including Entity-Relationship Diagrams (ERDs); Relational models; and the programming language SQL. In this session we will continue to build on those concepts using the Parkway Commute database.

ERD

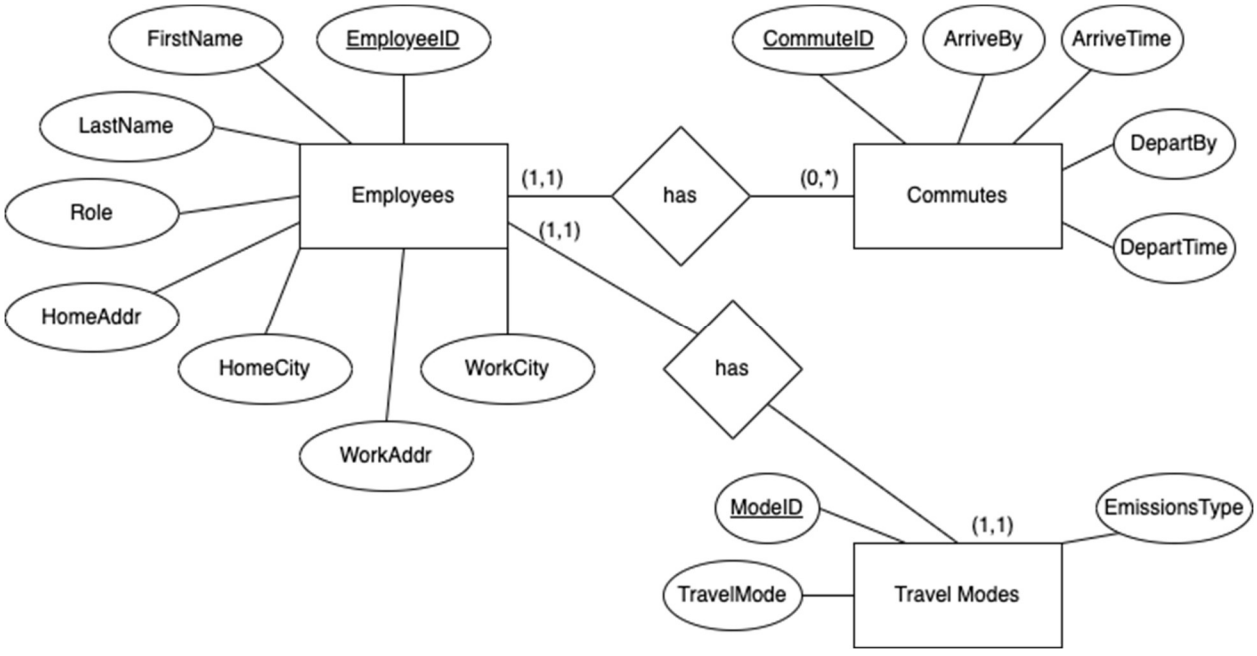

Figure 36: Parkway Commute ERD

First, let's explore the ERD for the Parkway Commute database as depicted in Figure 36. Recall that an entity is depicted in a rectangle box and will later be translated to the name of a table. Every entity has attributes associated with it, or rather pieces of information that we wish to store. Attributes are depicted in ovals, these become the columns of our tables. Underlined attributes represent a primary key which allows us to identify a unique piece of data.

Diamonds in the ERD depict the relationships between two entities, along with their cardinalities (the numbers on the lines). In the ERD for Parkway Commute we can see that employees may have 0 to many (*) commutes while commutes are only associated with one employee. Similarly there is a one-to-one relationship between employees and travel modes i.e. an employee has exactly one travel mode and a travel mode belongs to exactly one employee.

Relational Model

Next, let's look at the relational model for our ERD. Recall that a relational model defines three different aspects of the database design: **relations,** data organised in tables; **attributes,** every table with a fixed set of columns; and **Foreign Keys,** logical links between tables.

We can define our relational model by using the ERD. For every entity there is exactly one relation with it's associated keys and attributes, similarly for a relationship. We can merge relations which have identical keys. Cardinalities determine how keys are placed within relations for relationships of the ERD.

For the Parkway Commute ERD we get the following relational model:

Employees(EmployeeID, FirstName, LastName, Role, HomeAddr, HomeCity, WorkAddr, WorkCity)
Commutes(CommuteID, ArriveBy, ArriveTime, DepartBy, DepartTime)
TravelModes(ModeID, TravelMode, EmissionsType)
has-Employee-Commute(EmployeeID,CommuteID)
has-Employee-TravelModes(ModeID,EmployeeID)

Figure 37: Initial Relational Model

Note that in our initial relational model our relations "has-Employee-Commute" and "has-Employee-TravelModes" have the same IDs as Commutes and TravelModes respectively. This means we can simply merge these relations to get the final relational model:

Employees(EmployeeID, FirstName, LastName, Role, HomeAddr, HomeCity, WorkAddr, WorkCity)
Commutes(CommuteID, ArriveBy, ArriveTime, DepartBy, DepartTime,EmployeeID)
TravelModes(ModeID, TravelMode, EmissionsType,EmployeeID)

Figure 38: Final Relational Model

If you inspected the SQL in the last session closely, you will see that we now have a relational model that matches with the tables defined in the script. Next we will describe how the relation becomes a table in SQL.

Relational Model to SQL

There are many different types of programming languages which use SQL (pronounced "sequel") in their name. For example, there is Oracle-SQL which is developed by the software corporation Oracle or MySQL which is an open source version (also maintained by Oracle). The built in SQL for Visual Studio that you saw last time is called Transact-SQL or T-SQL for short. Unsurprisingly, T-SQL is managed by Microsoft.

While each different SQL language has its own syntax and "quirks" they all operate on the same programming paradigm and use a similar command set. Therefore, once you are familiar with one it is easy to use different versions, provided you use the correct syntax!

1	*Employees(*	`CREATE TABLE "Employees" (`
2	*EmployeeID,*	`"EmployeeID" "int" IDENTITY (1, 1) NOT NULL ,`
3	*FirstName,*	`"FirstName" nvarchar (20) NOT NULL ,`
	LastName,	`"LastName" nvarchar (20) NOT NULL ,`
4	*Role,*	`"Role" nvarchar (30) NULL ,`
5	*HomeAddr,*	`"HomeAddr" nvarchar (60) NULL ,`
6	*HomeCity,*	`"HomeCity" nvarchar (60) NULL ,`
7	*WorkAddr,*	`"WorkAddr" nvarchar (60) NULL ,`
8	*WorkCity)*	`"WorkCity" nvarchar (60) NULL ,`
9		`CONSTRAINT "PK_Employees" PRIMARY KEY CLUSTERED ( "EmployeeID" )`
10		`) GO`

Figure 39: Parkway Commute Relations to T-SQL Table

Let's inspect the relation we created for Employees in our model above (Figure 38) to the T-SQL you were provided in the last session. On line 1 we are creating a new table called Employees. This is followed on lines 2-8 with each of the attributes for the relation, which become columns in the table. The difference between the relational model and the T-SQL programmed version is that we must specify appropriate data types. You can see here that we have "`int`" for integers and `nvarchar (60)` for string data which has up to 60 characters. Note that we also specify which values can be empty (`NULL`) or not-empty (`NOT NULL`).

Line 9 is where the T-SQL differs from the relation. In order to specify the primary key we must add a constraint to the table, unlike in the relational model where we simply underlined the key. Lastly, on line 10 we tell SQL to execute the command.

If you compare the other tables to their relations from the relational model you should notice a similar pattern appearing.

SQL Queries

All SQL queries have a similar format and allow us to ask the database for a specific subset of data. In L1 we focussed on queries which had SELECT, WHERE and FROM clauses. For example, recall the following:

```
SELECT <column_name>
FROM <table_name>
WHERE <search_condition>
GROUP BY <column_name>
HAVING <search_condition>
ORDER BY <column_name>
```

In the above format we select a particular column of data from a specific table in our database. In our case this would be any attribute from Employees, Commutes or TravelModes. Our where condition can include specific parts of information we are looking for, for example, all employees which travel by car and so on. Group by allows us to group data by a certain column. Having is used with a group by clause, for example we might want to know all travel modes that are used by more than 10 employees. Lastly, order by allows us to sort the data by a particular column.

Today the Queries you need will be supplied, however, understanding their format will help with the activities that you are working on today.

Today's Exercise

Last session you only started looking at extending the backend of your Parkway Commute application. This week you are going to imagine that Joe from Parkway has gotten back to you with some suggestions for a dashboard which can show different types of data. He attached a sketch of the layout below.

Figure 40: Joe's Dashboard Ideas

Today's Exercise is to implement something close to Joe's vision. We will start by opening your project from the last session and creating a new blank form. We are going to fill out a form to look like the dashboard idea's provided by Joe. The following figure depicts the different widgets you will need and what names you should give to these. Remember, it is important to follow the naming conventions provided as we will use these later in our code.

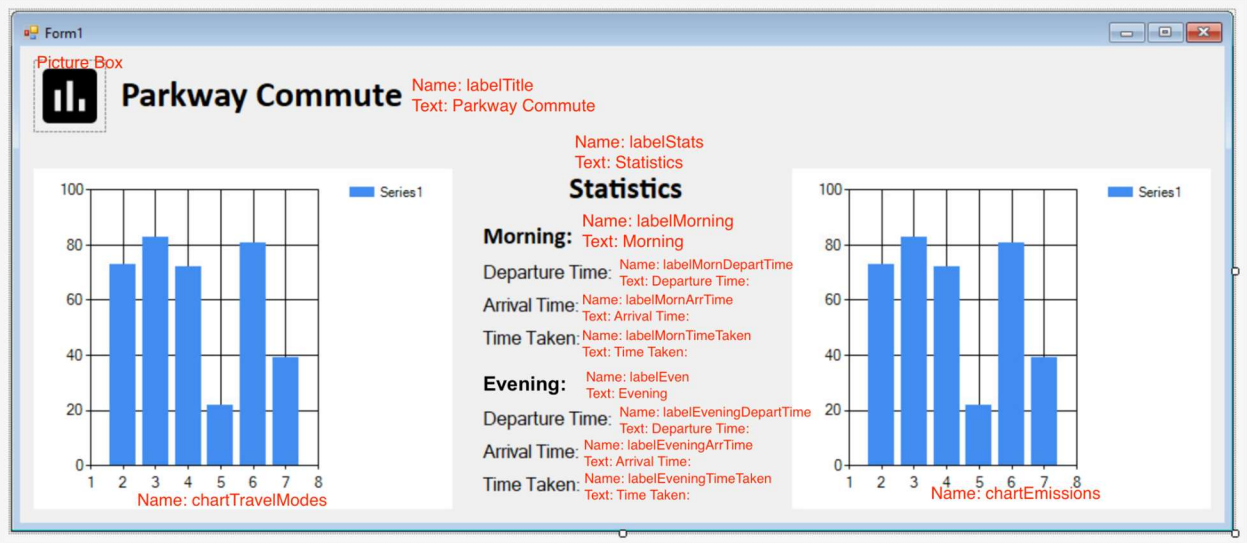

Figure 41: The Windows Form Layout

Next, we will begin with displaying the travel modes. Open the code for the form to view the code editor.

Inside the form 1 constructor method we are going to add the following code:

```
17    public Form1()
```

```
18    {
19          InitializeComponent();
20
21          SqlConnection connection = new SqlConnection(@"Data
            Source=(localdb)\MSSQLLocalDB;Initial
            Catalog=ParkwayCommute;Integrated Security=True");
22          connection.Open();

            …
```

On line 19 we initialise the form as always, however, on line 21 we need to establish a connection to our local database. Here we create an SQL connection object and specify the relevant connection string. On line 22 we open the connection to the database.

Now that we have a relevant connection to the database we can query the database for specific information for our chart. We are going to create a function called getData which will handle the query for us and a second function called displayChart which will show the data on the form.

First, continuing within the form 1 constructor, we are going to add the following code:

```
23          …
24          String query = "SELECT TravelMode, COUNT(TravelMode) as
            Commutes FROM TravelModes GROUP BY TravelMode";
25          DataSet ds = getData(query, connection, "TravelModes");
26          displayChart(chartTravelModes, ds, "TravelModes",
            "TravelMode", "Commutes", "Data by Travel Modes");
27          …
```

On line 24 we create the SQL query which will select all the travel modes from the database and then count the number of those modes. By grouping the modes by the travel mode we ensure we get the number of trips by each mode.

On line 25 we create the query to get the data from the database, here we call our getData function which we will specify next. Line 26 also makes a call to a function, in this case our displayChart function which will show the data on the form.

Next we specify the getData function. Be sure to add this function outside of the Form1 constructor:

```
70    private DataSet getData(string query, SqlConnection connection,
      string datasetName){
71          SqlCommand getTravelModes = new SqlCommand(query,
```

```
            connection);
72          SqlAdapter adapter = new SqlDataAdapter(getTravelModes);
73          DataSet dataset = new Dataset();
74          adapter.Fill(dataset, datasetName);
75          return dataset;
76      }
```

On line 70 we specify the function, this is a private function which means that it is only accessible inside the file. The function returns a DataSet object and takes 3 parameters, the query string, the SQL connection object and a name for the dataset.

On line 71 we create the SqlCommand object, this sets up our query for our specific database connection. Next we set up the adapter which will execute the query on the database and return some data. On line 73 we create the DataSet object which will store the retrieved data in a usable format. On line 74, we tell the adapter to fill our dataset object with the retrieved data, then we return the dataset object for use within our form.

Once we have the data retrieved from the database in a usable format, we need to display the information on the chart. To do this we create another function called displayChart:

```
79      private void displayChart(Chart chart, DataSet dataset, string
        datasetName, string x, string y, string title) {
80          chart.DataSource = dataset.Tables[datasetName];
81          chart.Series[datasetName].XValueMember = x;
82          chart.Series[datasetName].YValueMembers = y;
83          chart.Titles.Add(title);
84          chart.Series[datasetName].ChartType = SeriesChartType.Pie;
85          chart.Series[datasetName].IsValueShownAsLabel = true;
86      }
```

Similarly to our previous function we start by giving the function definition. Again our method is private but this time it does not return anything, so we specify “void” instead of an object type. This function requires several parameters in order to display the chart correctly, these are as follows:

- `chart`: the chart object in the form that we will display data on.
- `dataset`: the data to be displayed on the chart.
- `datasetName`: the name of the dataset (should be the same as we used in our query).
- `x`: the name of the data column for the x series of data.
- `y`: the name of the data column for the y series of data.
- `title`: the title to display for the chart.

If we look at line 26 above where we made our call to this function we can see that we specified: chartTravelModes as the chart object to display the data; that dataset ds as the result of our travel modes query; the name TravelModes as the name of the dataset; the TravelMode column as our x series; the Commutes column as our y series; and lastly the chart title "Data by Travel Modes".

If you have entered the above code correctly, you should get the following output for your travel modes chart:

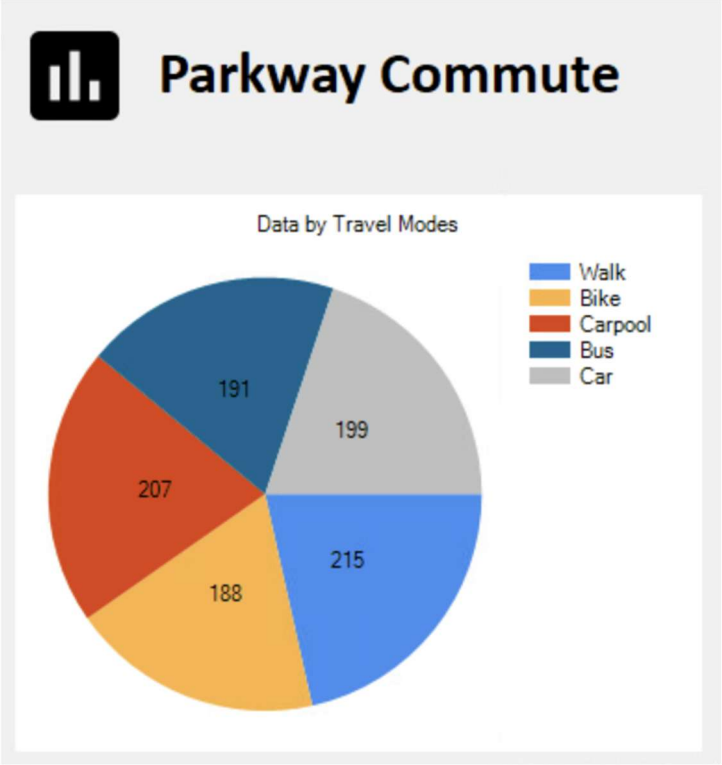

Figure 42: Data by Travel Modes Output

Next, let's work on populating the averages data inside each of the labels. Inside your form1 constructor function we are going to continue with adding a query to get the average arrival and departure time taken.

```
28    …
29    query = "SELECT AVG(ArriveTime), AVG(DepartTime) FROM Commutes";
30    ds = getData(query, connection, "Times");
31    DataRow row = ds.Tables[0].Rows[0];
32    labelMornTimeTaken.Text += " " + row[0].ToString() + " minutes";
33    labelEveningTimeTaken.Text += " " + row[1].ToString() + "
      minutes";
34    …
```

On line 29 we specify the query which will allow us to select the average arrival and departure time taken for a commute from the commutes table. On line 30 as above we create the query to the database using our getData function. This time however, we need to process our data row by row rather than display in a chart to modify the label text.

On line 31 we get the single row of data that should be returned from the database. Next on line 32 we specify the morning time taken to add the number from the database followed by the unit minutes. We do the same for the evening time taken. You should get the following output:

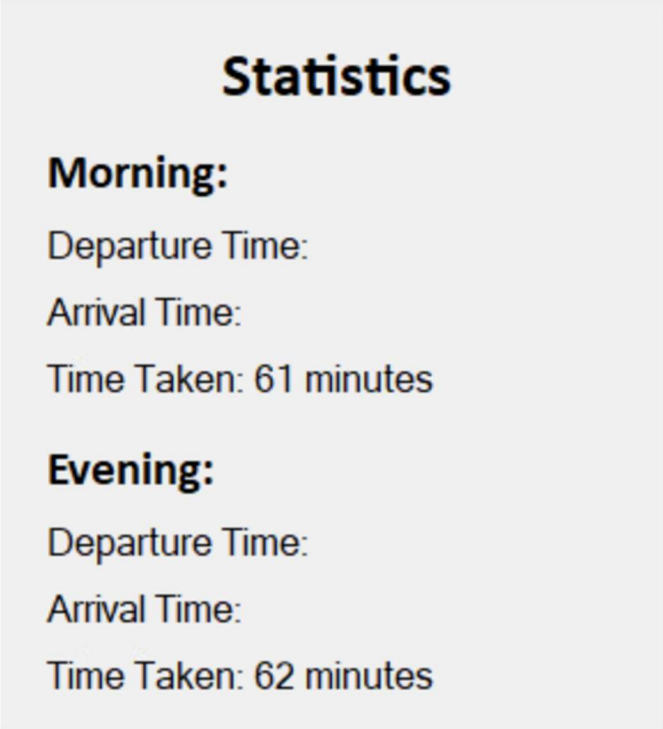

Figure 43: Time Taken Averages

Next we will demonstrate how to get the average departure time for the morning from the database. Following along in your form1 constructor function, add the following two lines:

```
34    …
35    query = "SELECT AVG(DATEPART(hour, DepartBy)) as DepartByHr FROM
      Commutes WHERE DATEPART(hour, DepartBy) < 12";
36    setTimeLabel(labelMornDepartTime, query, connection);
37    …
```

This query is more complicated than the ones that we have seen previously. We tell the database to select the average of the hour part of our date for our DepartBy column. This essentially means taking a string which is formatted as a date and selecting just the hour from it, we then want the average hour of all the data. In our where clause we are only interested in data from the

morning, therefore we specify the average from all times which begin before 12 (note that the database is in 24-hour time).

On line 36 we use another function to specify how to display the data that we retrieve. The data will be returned as either a 1 or 2 digit number for the hour, therefore, we need to display this as a time. Also note that we haven't actually made the query to the database yet! Add the following function to your Form1 code:

```
63    private void setTimeLabel(Label label, string query,
      SqlConnection connection){
64          DataSet ds = getData(query, connection, "Morning hour");
65          DataRow row = ds.Tables[0].Rows[0];
66          label.Text += " " + row[0].ToString().PadLeft(2, '0') +
            ":00";
67    }
```

We start by specifying the function definition. We take as parameters: the label to display the time in, the query we have specified, and the connection to our database. On line 64 we first query the database to get the data returned. As this will return a single number we can process the data rows as we did with our previous query. Lastly, on line 66 we display the average morning departure time by padding our resulting hour string to have at least two numbers followed by ":00" so that the information is displayed in 24 hour time.

If you have added this correctly you should get the following output shown in figure 44.

The next step is to fill out the other averages and populate the data by emissions chart. At this point you should have all the pieces of code to follow as examples to populate the other labels and chart. We leave this to you to complete. When you are finished you should have the labels populated as depicted in figure 45 and the chart shown in figure 46.

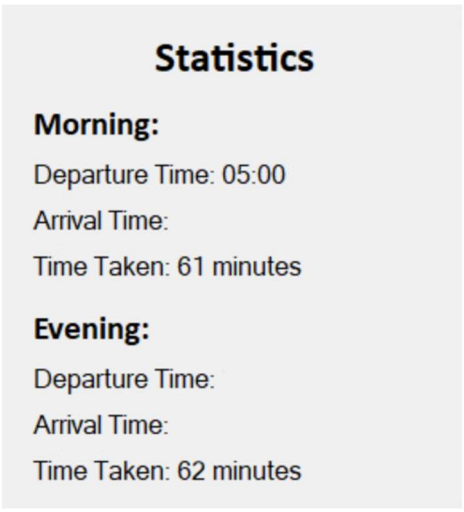

Figure 44: Departure Time Average

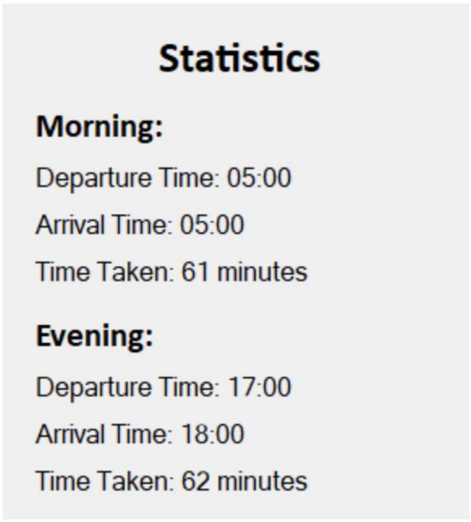

Figure 45: All Averages

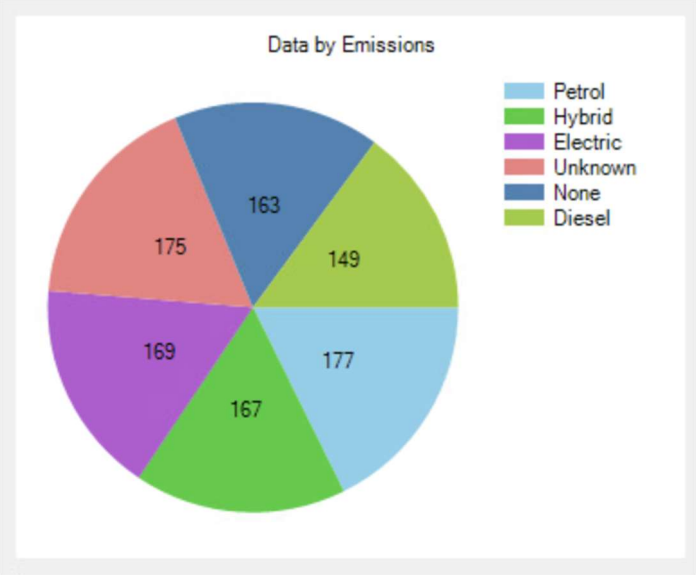

Figure 46: Data By Emissions Chart

Summary

Today's exercise looked at creating a more complex backend for your implementation of the Parkway Commute application. We explored more complex SQL queries and how to display these using charts on a Windows Form application. Next week we will finish working with the software development lifecycle by touching on user testing.

Useful Resources

- MVC Magic!: https://avelonpang.medium.com/mvc-magic-13e37d782cf7
- Create a Windows Form to search data: https://docs.microsoft.com/en-us/visualstudio/data-tools/create-a-windows-form-to-search-data?view=vs-2022
- SQL queries syntax: https://www.w3schools.com/sql/sql_syntax.asp
- Chart Control in Windows Forms Application: https://www.c-sharpcorner.com/UploadFile/1e050f/chart-control-in-windows-form-application/
- SQL COUNT(), AVG() and SUM() Functions: https://www.w3schools.com/sql/sql_count_avg_sum.asp
- SQL Aliases: https://www.w3schools.com/sql/sql_alias.asp
- SQL Group By: https://www.w3schools.com/sql/sql_groupby.asp