Project #06: Web-based CTA App

Complete By: Tuesday December 3rd at noon
Assignment: C# web application using ASP.NET and ADO.NET
Policy: Individual work only, late work *is* accepted (up to 24 hrs late for a penalty of 10%)
Submission: "Mark as Complete" via Codio's Education menu

Background

The goal of Project #06 is to extend an existing web app that retrieves information about CTA L stations and stops. A working application is provided that does 3 things:

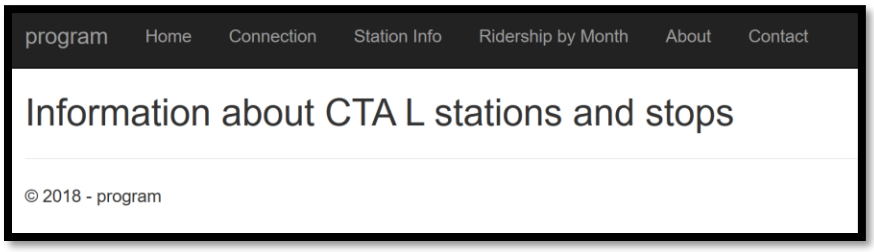
1. Tests the database connection
2. Retrieves information about CTA stations
3. Displays total CTA ridership by month

The app is based on the MVC design pattern, using ASP.NET, C# and ADO.NET. You must work within this design by extending / creating Views and Models. To run the provided application, login to Codio and open **project 06**. Then open a terminal window, change directory to "cta-web", and "dotnet run" to start a web server (*Kestrel*) and load the web app:

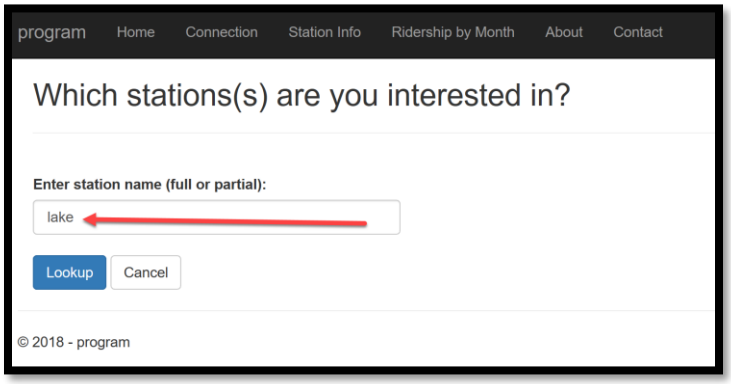
```
codio@slang-dolby:~/workspace$ cd cta-web
codio@slang-dolby:~/workspace/cta-web$ dotnet run
Hosting environment: Development
Content root path: /home/codio/workspace/cta-web
Now listening on: http://[::]:3000
Application started. Press Ctrl+C to shut down.
```

The web server is now running, waiting for clients to connect via a web browser. The simplest way to connect is via the "Box URL" command in the next to last menu in Codio. Drop the menu and select "Box URL":

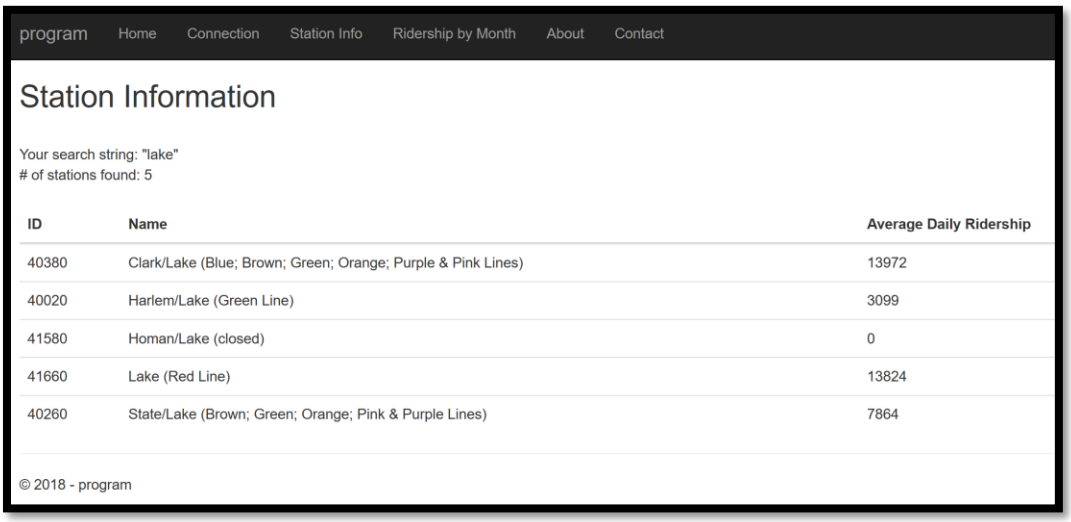
The “Box URL” command will browse to the web app and trigger the loading of the home page:



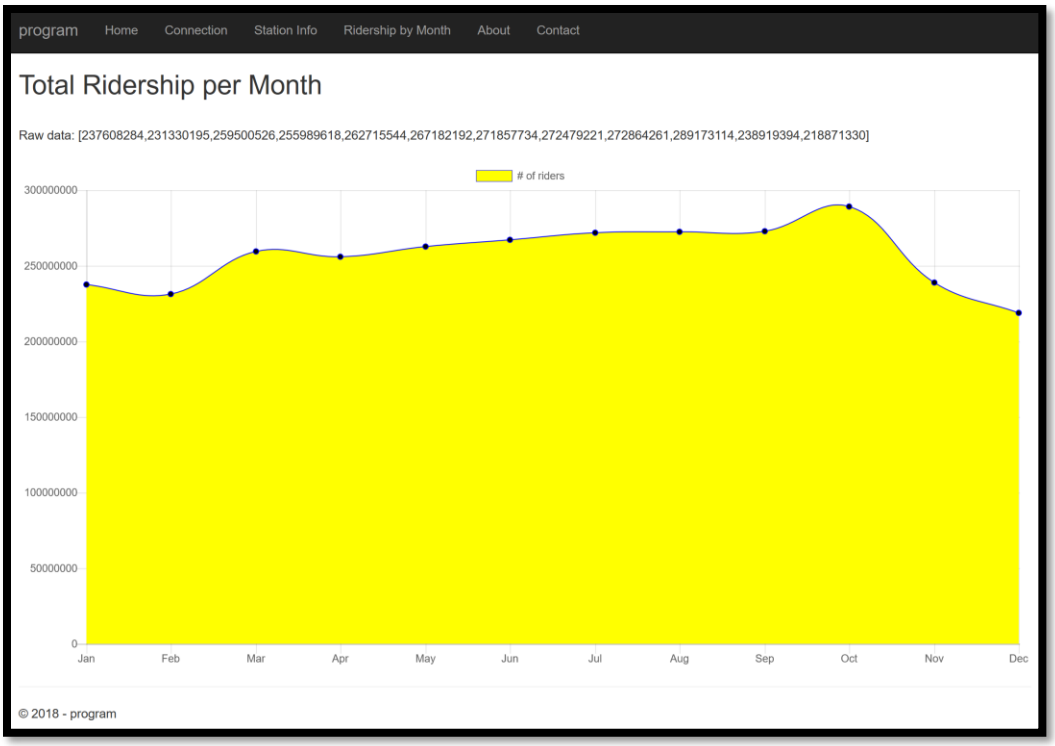
Click on “Connection” to check the connection status to the Azure CTA database. Then click “Station Info”, and you’ll be asked to enter a full or partial station name:



Enter “lake” and click lookup. This will retrieve information about 5 stations, in alphabetical order by station name:



Finally, click “Ridership by Month”. This will produce a graph of total ridership across the CTA L system, by month (next page):



CTA Database

We are working with the CTA database running in Azure. The following is a diagram of the tables in the database.

Stations

StationID	Name
40710	*Chicago (Brown & Purple Lines)*
...	...

Stops

StopID	StationID	Name	Direction	ADA	Latitude	Longitude
30137	*40710*	*Chicago (Kimball-Linden-bound)*	*N*	*1*	*41.89681*	*-87.635924*
30138	*40710*	*Chicago (Loop-bound)*	*S*	*1*	*41.89681*	*-87.635924*
...	...	...	...	...	...	...

CTA

StationOrder

LineID	StationID	Position
2	40890	1
2	40820	2
1	40900	1
1	41190	2
...	...	...

Lines

LineID	Color
4	Brown
5	Purple
6	Purple-Express
...	...

StopDetails

StopID	LineID
30137	4
30137	6
30138	4
30138	6
...	...

Riderships

RiderID	StationID	TheDate	TypeOfDay	DailyTotal
...	...	...	...	...
8199	40170	2001-02-28	W	4206
...	...	...		

Assignment

The assignment is to extend the given web application in the following ways:

1. Modify the existing "About" view and model to display something about yourself.

2. Modify the existing "Contact" view and model to display some sort of contact info (your name or netid is sufficient).

3. Extend the existing "Station Info" view and model by adding 2 more columns of information: # of stops (integer), and how many of these stops are handicap accessible (string value that's either "none", "some", or "all"). Note that information about handicap accessibility is stored in the ADA column of the Stops table (0 means no, 1 means yes). You are required to extend the **Station** class to store this additional information, which the view then retrieves and displays.

   [NOTE: "Station Info" involves 2 views, **GetStationInput** to obtain the station name and **StationInfo** to display the actual information. The corresponding models are **GetStationInputModel** and **StationInfoModel**, along with the **Station** class.]

4. Add a "Lines" view and model to display information about the Stations along a particular Line. The user will enter a Line name and then information about all the stations along that will be displayed. For each station, display the station id, the station name, and the number of stops (entrances) associated with that station in the order that the stations are visited along that line. You are required to define a **Line** class to properly model the information about the line.

   [NOTE: follow the same design as "Station Info", i.e. using one model to contain the information about the Line including the list of Station models, and a new view for the Station model.]

5. Add a "Ridership by Day" view and model. Follow the "Ridership by Month" example, instead grouping the results by which day of the week (Monday, Tuesday, etc.). You can use the DATENAME(WEEKDAY, Date) function to extract the day of the week from the Date field.

6. Add something of your own choice. Suggestions include "Top-10 Stations by Ridership", or viewing ridership by the different days of the week (W vs. A vs. U), or graphing ridership for a particular station by month or year.

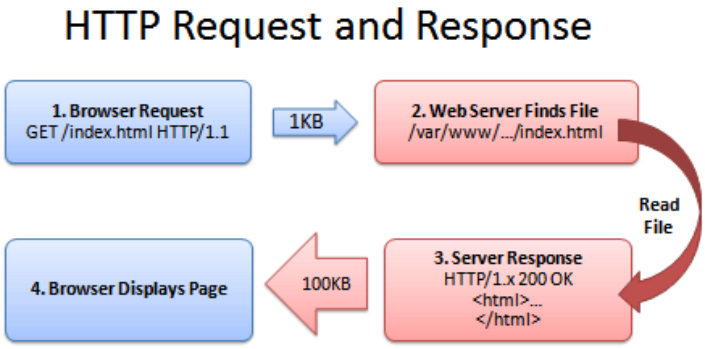
You should plan to work on **Codio** since this is how submissions will be collected (we will not be using Gradescope for this project, nor Blackboard). You can work in outside Codio if you want, but then you'll need to upload your final project to Codio for submission. To work in *Visual Studio for Windows* you'll need to first run *Visual Studio Installer* to add the ASP.NET workload. Then you can login to Codio, export the provided files to your local system, unzip, and then open the provided .csproj file. You should then be able to run (F5) from within Visual Studio.

Getting Started with ASP.NET

ASP stands for "Active Server Pages", and the **.NET** refers to the .NET family of languages: C#, F#, VB, etc. Most web apps today using a combination of technologies, in our case you can think of ASP.NET => HTML + C#. This is a server-side technology, which means that

a. The user ("client") uses a browser to request a web page --- from any platform, using any browser

b. The web server ("server") loads the web page (in our case a mix of HTML and C#), executes the C# to produce more HTML, and returns pure HTML to the client



At this point the client-server interaction is done; the user might request another web page from this site, or surf away to another web site. A simplified visual is shown to the right. The difference is that where you see "read file", think "read file and execute C#".

Since web applications have a very different architecture, we are providing a working application to help you get started. Since there are many moving parts to a MVC-based solution, you'll need to learn your way around app.

1. Login to Codio
2. There are 2 folders: "cta-web" is the provided web app, "__save" is a backup copy.
3. Run the provided app:

```
cd cta-web
dotnet run
```

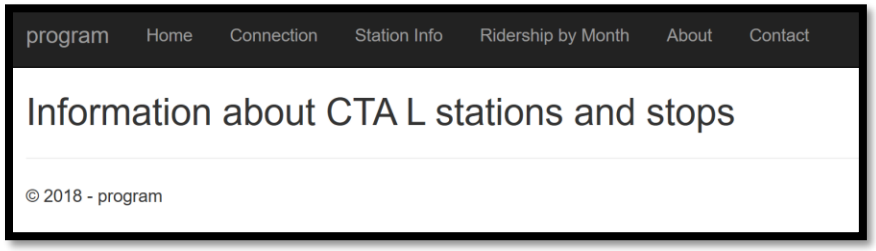
```
codio@slang-dolby:~/workspace$ cd cta-web
codio@slang-dolby:~/workspace/cta-web$ dotnet run
Hosting environment: Development
Content root path: /home/codio/workspace/cta-web
Now listening on: http://[::]:3000
Application started. Press Ctrl+C to shut down.
```

4. Stop and reflect for a second… The database (our data store) is running in Azure. Our web app (our data access and logic tiers) is running in Codio. But a web app requires a web server, so that is *also* what we're running when we say dotnet run --- we are starting a web server (Kestrel), which in turn is hosting our web app (CTA). That's why you must leave the program running in Codio while you run and test your web app. When you need to make changes to your web app, you'll stop the web server by typing Ctrl+C in the terminal window.

5. Assuming the web server is running, the last step is to browse to your web app. In theory your Codio VM is public on the internet, but in reality (due to security concerns) it can only be accessed by you. In

particular, it can only be accessed by the browser you have open, since this is the only one that has login credentials to access Codio. The easiest way to access your web app is to use Codio's "access" menu, which is the next to last menu in Codio:

Drop the menu and select "Box URL" --- this should open a new tab in your browser, and automatically surf to the web app you have running. The first time you'll see a "cookie" dialog across the top, which you should accept. Then you'll see the web app's home page with a menu bar above:



6. At this point, all is well and you have a working web app that you can extend. Close the browser tab containing this web page, and then stop the web server by clicking in the terminal window and pressing Ctrl+C.

   NOTE: if you take a break or walk away from Codio for an extended period of time, the Codio system will timeout and your login credentials will expire. If you find yourself in a situation where the web app does not open, the solution is to logout of Codio, close the browser, reopen the browser, and log back into Codio. Then all should be well.

7. As discussed earlier, the web app is using a Model-View-Controller architecture, where the Controller portion is supplied. Our job is to provide the Models and Views that constitute our app.

8. For example, open the Views sub-folder. The home page is represented by the view "Index.cshtml". Double-click and open this file in the editor:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";

    <h2>Information about CTA L stations and stops</h2>
}
```

The first line tells ASP.NET that this file contains a web page (and hence ASP.NET should build a

controller to handle requests for this page). The second line says there is an underlying model to potentially supply data to this page. The rest of the file should contain HTML, or a mix of HTML and C#. To mix HTML and C#, you surround with @{ … }, as shown above. [NOTE: to comment out "code" in a view, surround with @* … *@]

9. The home page ("Index.cshtml") refers to a model on line 2. By convention, the underlying model is stored in the Models sub-folder, in a file named "Index.cshtml.cs". The file *must* contain a C# class named **IndexModel** (as defined on line 2). The purpose of the model is to supply data needed by the web page, e.g. when the user "gets" the page:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace program.Pages
{
    public class IndexModel : PageModel
    {
        public void OnGet()
        {
           // nothing to do --- home page has no app-specific data:
        }
    }
}
```

When the home page is first loaded, the controller will called the OnGet() function to obtain any data needed by the page. In this case no data is needed, so the function is empty.

10. Next, take a look at the View "Connection.cshtml", which displays the current status of the database connection:

```
@page
@model ConnectionModel
@{
  ViewData["Title"] = "Connection to Database";

  <h1>@Model.Status</h1>
}
```

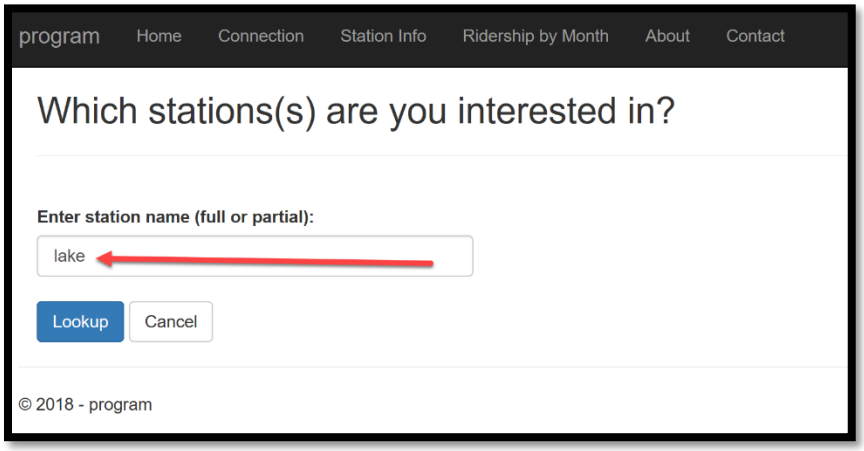
Notice the view refers to @Model.Status, i.e. the **Status** data member of the associated model ConnectionModel. You'll find that in the Model "Connection.cshtml.cs":

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
```

```
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace program.Pages
{
    public class ConnectionModel : PageModel
    {
        public string Status { get; set; }

        public void OnGet()
        {
           if ( DataAccessTier.DB.TestConnection() )
             Status = "Database connection status: good";
           else
             Status = "Database connection status: unreachable";
        }
    }
}
```

In this case the model uses the **DataAccessTier** to ping the database, and reports the result via updating the **Status** data member.

11. Okay, here's a more interesting example: "Station Info". Displaying station information is a two-step process. First the view "GetStationInput" is displayed to obtain input from the user:



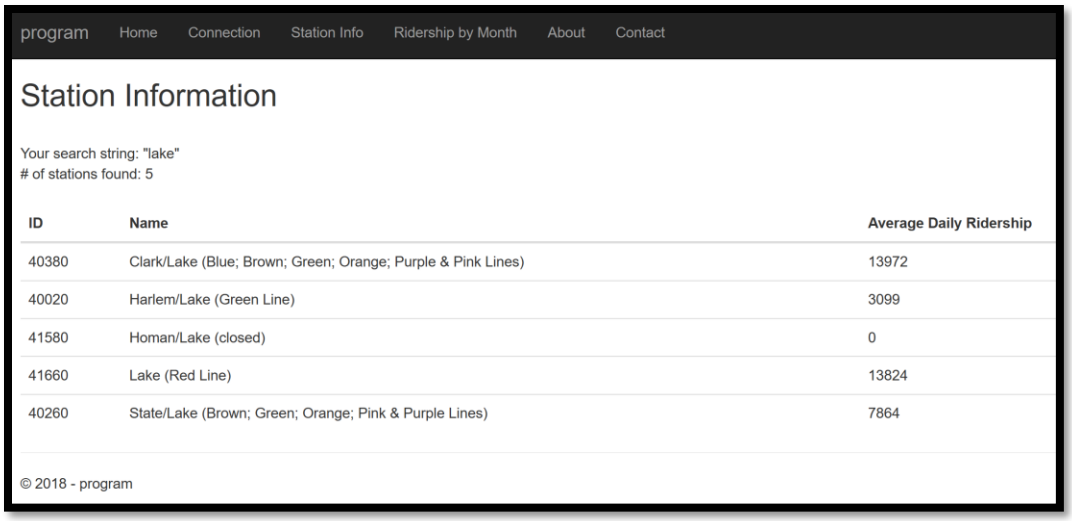
Let's look at the underlying view in detail (you may need to read this section a few times). This is the view "GetStationInput.cshtml":

```
1   @page
2   @model GetStationInputModel
3   @{
4     ViewData["Title"] = "Get Station Input";
5   }
6
7   <div class="container">
8       <h2>Which stations(s) are you interested in?</h2>
9       <hr />
10      <br />
11          <form action="StationInfo" method="get">
12
13          <div class="row">
14            <div class="col-md-4">
15              <label asp-for="Input">Enter station name (full or partial):</label>
16              <input asp-for="Input" class="form-control" />
17            </div>
18          </div>
19
20          <br />
21          <input type="submit" value="Lookup" class="btn btn-primary" />
22          <a class="btn btn-default" href="/">Cancel</a>
23
24      </form>
25  </div>
26
```

The important lines are 11, and 15-16. Line 11 specifies that when the form is submitted (by clicking the Lookup button), the action will be to "get" the "StationInfo" view. Lines 15-16 associate the user's input with a model variable called "Input" (capitalization is important), and causes this "Input" to be passed as a URL parameter when the "get" occurs. Here's the underlying model in "GetStationInput.cshtml.cs, which must specify "Input" as a public data member:

```
namespace program.Pages
{
    public class GetStationInputModel : PageModel
    {
      public string Input { get; set; }

      public void OnGet()
      {
         // no data needed for initial view:
      }

    }//class
}//namespace
```

When the user clicks **Lookup**, the web app will get the "StationInfo" view and pass the input along. Here's what that final web page looks like:



Here's the underlying view "StationInfo.cshtml", which displays the simple data members are simple HTML, displays exception info if one occurred, and displays the list of stations using a table:

```
@page
@model StationInfoModel
@{
  ViewData["Title"] = "Station Information";
}

<h2>Station Information</h2>

<br />
Your search string: "@Model.Input"
<br />
# of stations found: @Model.StationList.Count
<br />
<br />
@{
   if (@Model.EX != null)
    {
          <h3>**ERROR: @Model.EX.Message</h3>
          <br /> <hr /> <br /> <br />
    }
}

<table class="table">
    <thead>
        <tr>
            <th>
                ID
            </th>
            <th>
                Name
            </th>
            <th>
```

```
                Average Daily Ridership
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.StationList)
        {
            <tr>
                <td>
                  @item.StationID
                </td>
                <td>
                  @item.StationName
                </td>
                <td>
                  @item.AvgDailyRidership
                </td>
            </tr>
        }
    </tbody>
</table>
```

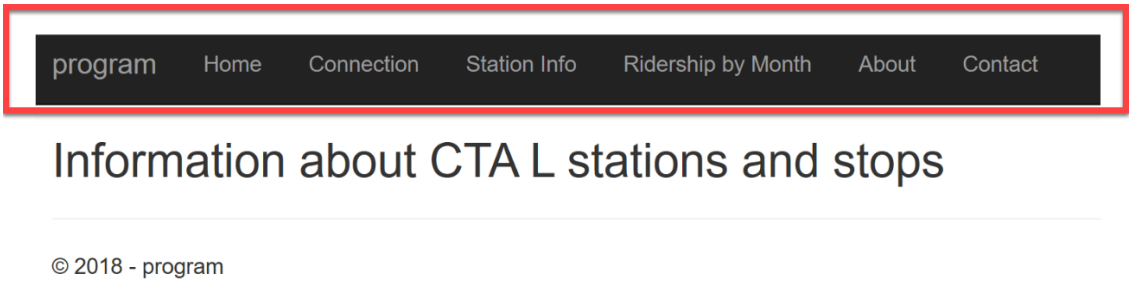
The associated Model in "StationInfo.cshtml.cs" is the C#, ADO.NET and SQL code needed to retrieve the necessary station data from the CTA database and turn this data into a list of Station objects. Here's the code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Data;

namespace program.Pages
{
    public class StationInfoModel : PageModel
    {
        public List<Models.Station> StationList { get; set; }
        public string Input { get; set; }
        public Exception EX { get; set; }

        public void OnGet(string input)
        {
          StationList = new List<Models.Station>();

          // make input available to web page:
          Input = input;

          // clear exception:
          EX = null;

          try
          {
```

```
                // Do we have an input argument?  If not, there's nothing to do:
                if (input == null)
                {
                    // there's no page argument, perhaps user surfed to the page directly?
                }
                else
                {
                // Lookup station(s) based on input, which could be id or a partial name:
                    string sql;

                    // lookup station(s) by partial name match:
                    input = input.Replace("'", "''");

                    sql = string.Format(@"
    SELECT Stations.StationID, Stations.Name, AVG(DailyTotal) AS AvgDailyRidership
    FROM Stations
    LEFT JOIN Riderships ON Stations.StationID = Riderships.StationID
    WHERE Stations.Name LIKE '%{0}%'
    GROUP BY Stations.StationID, Stations.Name
    ORDER BY Stations.Name ASC;
    ", input);

                    DataSet ds = DataAccessTier.DB.ExecuteNonScalarQuery(sql);

                    foreach (DataRow row in ds.Tables[0].Rows)
                    {
                        Models.Station s = new Models.Station();

                        s.StationID = Convert.ToInt32(row["StationID"]);
                        s.StationName = Convert.ToString(row["Name"]);

                        // avg could be null if there is no ridership data:
                        if (row["AvgDailyRidership"] == System.DBNull.Value)
                            s.AvgDailyRidership = 0;
                        else
                            s.AvgDailyRidership = Convert.ToInt32(row["AvgDailyRidership"]);

                        StationList.Add(s);
                    }
                }//else
            }
            catch(Exception ex)
            {
              EX = ex;
            }
            finally
            {
              // nothing at the moment
            }
        }

    }//class
}//namespace
```

12. There's a lot going on in the above Views and Models, you may need to re-read a few times… Where's the **Station** class? In "Station.cs" in Models. You'll want to review that class as well.

13. Interestingly, where is the menu defined that appears across the top of the web page?



Under Views, expand the folder "Shared". Open the file "_Layout.cshtml". In the file you'll see a tag with class="navbar-collapse collapse":

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
       <li><a asp-page="/Index">Home</a></li>
       <li><a asp-page="/Connection">Connection</a></li>
       <li><a asp-page="/GetStationInput">Station Info</a></li>
       <li><a asp-page="/RidershipByMonth">Ridership by Month</a></li>
       <li><a asp-page="/About">About</a></li>
       <li><a asp-page="/Contact">Contact</a></li>
    </ul>
</div>
```

Notice the asp-page attribute refers to the View name, e.g. "/Connection" or "/GetStationInput". When the user clicks the displayed text, e.g. "Connection" or "Station Info", this is how ASP.NET knows which View to load. When you want to add items to the menu, you'll add them here.

14. When it comes time to add features to your web app, feel free to copy existing View and Model files and modify (aka copy-paste). This is a necessary evil as you learn your way around.

Debugging

Debugging is hard in web apps since the C# code runs on the server --- e.g. output from System.Console.WriteLine is not visible. One approach is to print debug to your web page: add data members to the model, and then display these data members in the view using HTML and C#:

```
<br />
Data member contains:  @Model.NameOfDataMember
<br />
```

For an example, review how the Connection view and Connection model work together to display the status of the database connection.

Requirements

You are required to use ASP.NET, ADO.NET, C#, following the MVC design pattern which strictly separates the model from the view. You may use LINQ to SQL if you want to experiement with that technology.

You are required to use **try-catch-finally** to perform basic error handling. If an error occurs, at the very least a meaningful error message (based on the error that occurred) should be displayed in the relevant web page.

Electronic Submission and Grading

The project will be graded entirely on correctness. When you are ready to "submit", login to Codio, drop the Education menu, and select "Mark as Completed". Note that once you do this, it cannot be undone --- so make sure you are ready to submit. [*If you need to undo and continue working, post on Piazza and one of the staff will undo.*]

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

Unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .