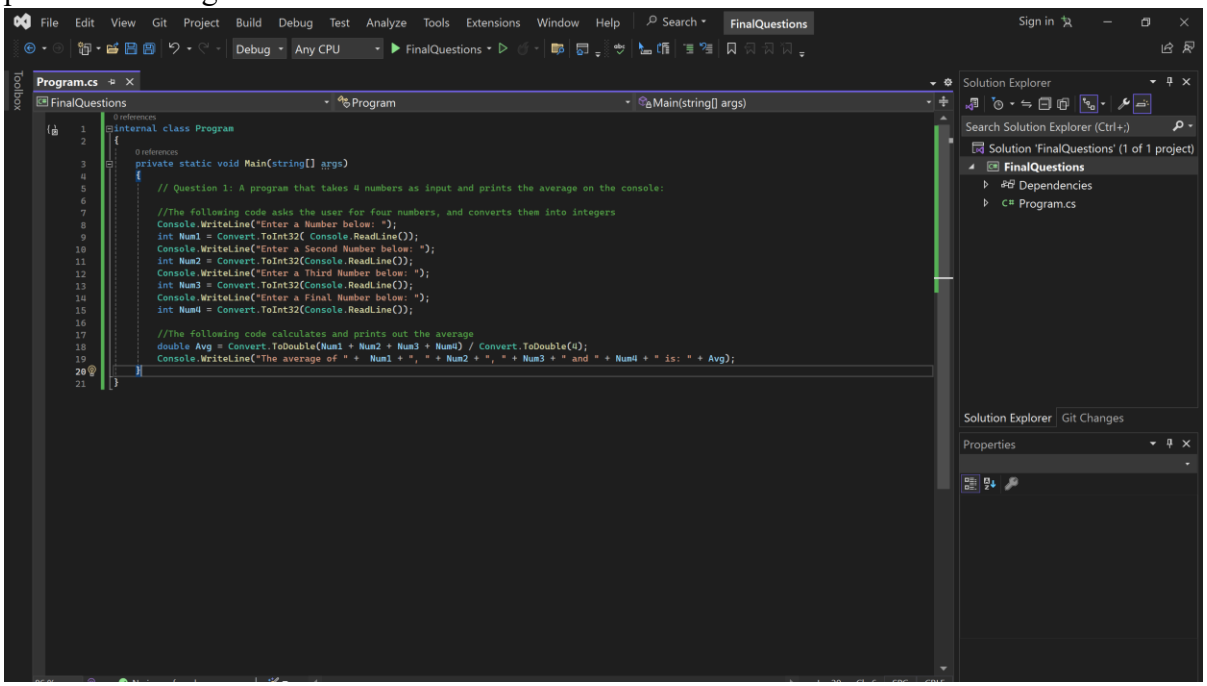
Part 1) Examination:

Q.1 (a) Write a C# Sharp program that takes four numbers as input to calculate and print the average. Paste your code in a word document along with the IDE screenshot of generated output.

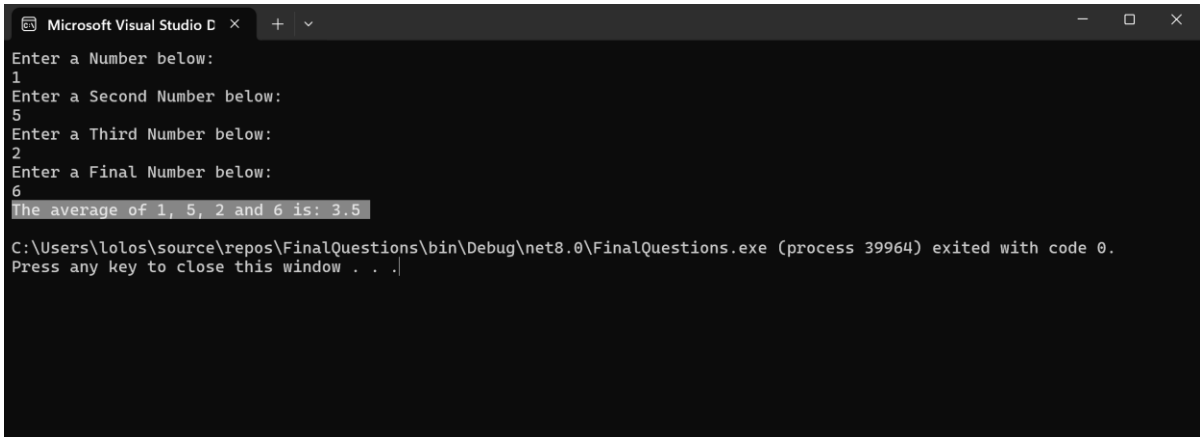
The below screenshot shows my program which takes four input numbers, calculates and prints the average in Visual Studio.



The code seen in the above screenshot is pasted in the box below:

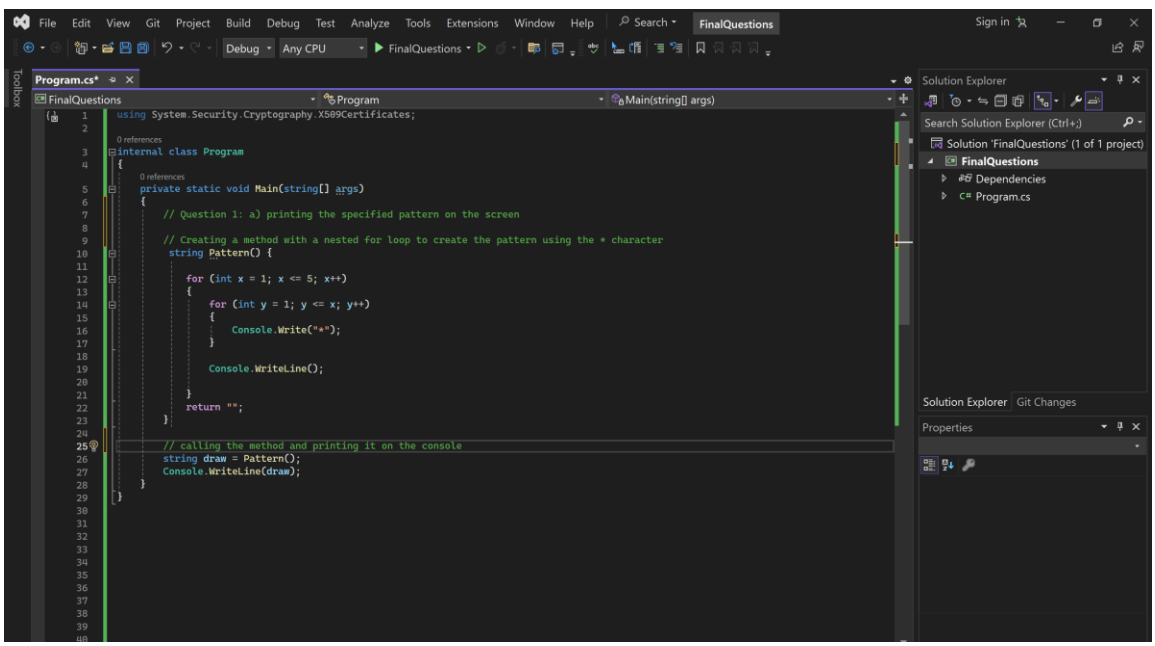
```
// Question 1: A program that takes 4 numbers as input and prints the average on the
console:

//The following code asks the user for four numbers, and converts them into integers
Console.WriteLine("Enter a Number below: ");
int Num1 = Convert.ToInt32( Console.ReadLine());
Console.WriteLine("Enter a Second Number below: ");
int Num2 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Enter a Third Number below: ");
int Num3 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Enter a Final Number below: ");
int Num4 = Convert.ToInt32(Console.ReadLine());

//The following code calculates and prints out the average
double Avg = Convert.ToDouble(Num1 + Num2 + Num3 + Num4) / Convert.ToDouble(4);
Console.WriteLine("The average of " +  Num1 + ", " + Num2 + ", " + Num3 + " and " +
Num4 + " is: " + Avg);
```

The successful output shows how the user can input 4 numbers, and the average is then calculated and printed on the screen.



Q.1 (b) Write a C# Program to display the following pattern: Paste your code in a word document along with the IDE screenshot of generated output.

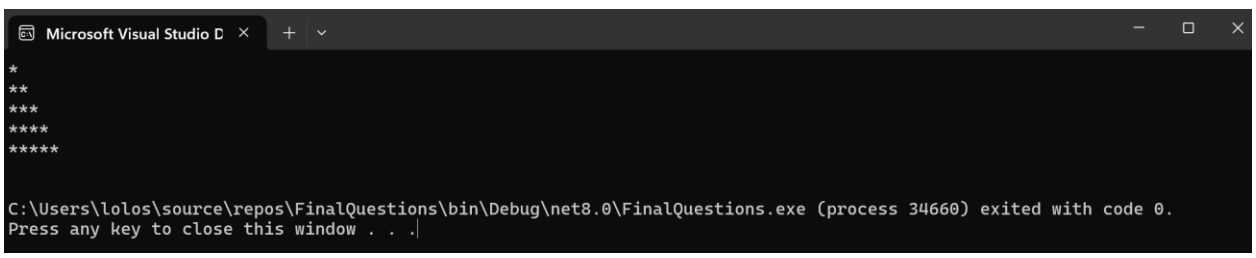
The program I made in Visual Studio that prints the * pattern is shown in the screenshot below. I used a nested for loop to create the pattern using the * as the only thing the console would print. The program is shown below.



The code I used in the above program is pasted into the box below along with the comments for explanation.

```
// Question 1: a) printing the specified pattern on the screen

// Creating a method with a nested for loop to create the pattern using the *
character
 string Pattern() {

    for (int x = 1; x <= 5; x++)
    {
        for (int y = 1; y <= x; y++)
        {
            Console.Write("*");
        }

        Console.WriteLine();

    }
    return "";
}

// calling the method and printing it on the console
string draw = Pattern();
Console.WriteLine(draw);
```

The corresponding output to the console is shown in the screenshot below. Evidently the loop concept worked, and the pattern is successfully printed.



Q.2 (a) What is MS Unit Test? Explain all of the steps to add the MS Unit Test in detail.

MS Unit test is an inbuilt testing framework in Visual Studio that was created by Microsoft and is one of their most simple frameworks. This framework is an excellent way of formulating unit tests in C# due to its simplicity and readability. It efficiently provides a very simple method attribute structure, here we use the [TestClass] and [TestMethod] for performing the unit testing. Along with the test solution in the software, it is evident which tests pass or fail, and helps developers to properly revise and fix the defects in the software. Below are the steps needed to add the MS unit test to a solution:

i. Once you have a solution created, right click on it and press the Add button.
ii. Next click the New Project button and configure the filter results to (languages to C#) - and (Project types to test). Then there will be an option entitled, Unit Test Project.

iii. Once Unit Test Project is selected, you can then enter an appropriate name for the project, typically it would be the name of the previous solution, suffixed by 'Tests'.
iv. When everything is filled out accordingly, click create and it will add the new project to the existing solution.
v. The next set of steps required to configure the unit test framework properly is to ensure that you have added the previous project as a reference to this new unit test project. Do this by …
vi. Navigate to the solution explorer and right click the References button located under the test project.
vii. Then select the project section on the left hand side, and check off the project name of the one previously mentioned, in that solution.
viii. Once finished click the 'OK' button. Now your Unit Test Project has been successfully added and is now ready for some test functions!

Q.2 (b) List and explain any four differences between Static Web Page and Dynamic Web Page. Explain all the features of Dynamic Web Page.

Differences between static and dynamic web pages:

Static and dynamic webpages are two forms of displaying website's content on the internet. These are two separate ways of displaying this content and they have major differences.

The first being, with static webpages displaying content in a fixed and unchanging way, while dynamic webpages change and adapt while in use. For instance, a Static webpage displays stable content. Each user viewing this type of webpage will see the same exact thing. On the other hand, a Dynamic webpage allows content to change for each individual user.

Secondly, they are different in composition. Static web pages are composed of HTML elements, CSS, and JavaScript, all on one document which is delivered by the server in an "as-is" way, while dynamic pages utilize a combination of client-side and server-side technologies to deliver changing and personalized content. This content is usually organized into a database or another content management system which connects to the site's pages.

Thirdly, they are different in how they are edited or how the content is changes. In a static web page, the content only ever changes when the HTML is edited on the coding level. However, dynamic web pages have content that is always changing. An example may be a stock market website, or weather sites whose content is always changing all the time.

Finally, the last difference between static and dynamic pages are how they are used. Static web pages are much simpler when compared to dynamic sites but are very useful for building personal websites to establish online presence. Dynamic webpages are very useful for building websites or pages that require content that is constantly changing due to its dynamic features.

Features of a Dynamic Web Page:

A dynamic webpage has many features that ensure that the content is readily available and delivered. Two interesting features of a dynamic web page are personalized content, and advanced functionality.

The first feature, personalized content is unique to a dynamic web page. Through the use of different web development tools, dynamic web pages can change, and alter according to each user. This is something that does not happen with static pages, since they are exactly the same no matter which user is viewing it. This feature can be done using text, styling, audio/video, and pictures elements, in order to personalize the web page according to each user. Additionally, personalized content on a dynamic web page can be accomplished with the use of cookies, which remember user preferences, and adjust accordingly, and with location data, which can provide the user personalized content based on their location.

Another important and interesting feature of a dynamic web page is advanced functionality. The functionality of a dynamic page is limitless! Dynamic web pages can use buttons, forms, pictures, videos, audio and other elements that enhance its functionality, and offer greater user experience to the user. The various components of a dynamic web page which involve both server-side and client-side programming and components increase the site's functionality and user experience. No doubt dynamic web pages have plenty of interesting features that elevate them from regular, plain static web pages.

Q.3 (a) List all the validator controls and explain any two validators in detail with proper programming examples with screenshots.

RequiredFieldValidator: This validation control is used when you want to make a field mandatory. This control is very useful for required information that must be input by the user before submission. It generates an error message if the field is empty when the user tries to submit the form.

RangeValidator: This validation control validates that the value entered by a user into a field lies between specified values. This can be useful for setting an age, or when the user needs to input a value within specified ranges. For instance, you can set upper and lower boundaries to 1 and 50, showing that the number needs to be in between those numbers. If the number input is not in those ranges, an error message is present.

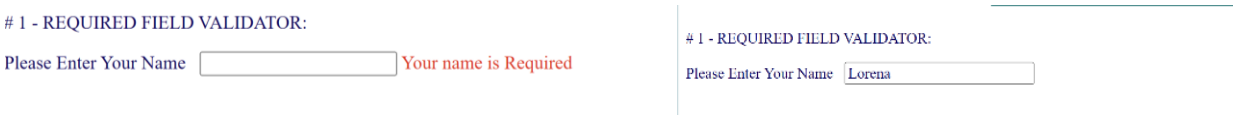
CompareValidator: This validation control compares the values of two inputs. Using comparison operators such as equal, less than, greater than etc. We can compare the values of two inputs and provide an error message if the two inputs do not follow the comparison specifications.

RegularExpressionValidator: The RegularExpressionValidator is a validation control that allows you to specify a pattern of text that the input needs to follow. This is very useful for standard patterns of text that may be a Social Insurance number, emails, postal codes and other similar fields. An example could specify a regular expression to be: "\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*" which sets the pattern for an appropriate email. If the email entered by the user does not follow this pattern, the error message is shown.

CustomValidator: This validator is very useful. It allows you to create your own validation function for something that needs validating and does not fall into one of the above categories.

The two that I have chosen to show an example of are RequiredFieldValidator, and RangeValidator. In the screenshots below you can see how they are in practice in a web form. These are done in VisualSudio by utilizing the ASP validation elements, and combining them with a web form.

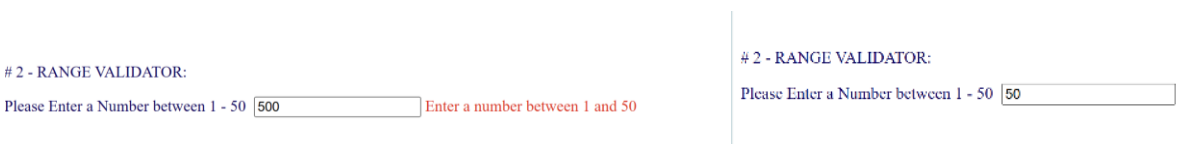
In this first example, the requiredFieldValidator is used by mandating the input of the user for the Name text box. This is evident by how the error message is present when the box is empty, however there is no error message when the box has text in it.



The associated code for this validator is shown below

```
# 1 - REQUIRED FIELD VALIDATOR:</p>
<p class="auto-style1" style="text-align: left">
    Please Enter Your Name  
    <asp:TextBox ID="name" runat="server"></asp:TextBox>
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server" ControlToValidate="name" ErrorMessage="Your name is Required" ForeColor="Red"></asp:RequiredFieldValidator>
</p>
```

The second example, about the RangeValidator, demonstrates this asp validation control by setting upper and lower boundaries to it. Any number within those boundaries is acceptable, if outside the boundaries, an error is present. In the example below, the acceptable ranges are any number between 0 and 50. You can see how when 500 is entered the error message is shown, but when 50 is input, there is no error since it falls within the boundaries.



The associated code for this validator is shown below:

```
<p class="auto-style1" style="text-align: left">
     </p>
<p class="auto-style1" style="text-align: left">
    # 2 - RANGE VALIDATOR:</p>
<p class="auto-style1" style="text-align: left">
    Please Enter a Number between 1 - 50 
    <asp:TextBox ID="range" runat="server"></asp:TextBox>
    <asp:RangeValidator ID="RangeValidator1" runat="server" ControlToValidate="range" ErrorMessage="Enter a number between 1 and 50" ForeColor="Red"
        MaximumValue="50" MinimumValue="1" Type="Integer"></asp:RangeValidator>
</p>
```

Q.3 (b)Explain Try, Catch and Throw Keywords with proper programming example.

Try, catch, and throw are the C# keywords used in error handling, when handling an exception. They are very useful in debugging and security for the program, and they help to ensure that a program is going to run as expected every time.

Try: The try keyword is the start of a try block. It is used to define the block of code. This block holds the code that may throw an exception.

Catch: The catch keyword is used to define a catch block, this block catches the exception specified in the above try block.

Throw: the final part of this block is the throw keyword. The throw keyword is used to manually throw an exception.

An example of all of these in action is shown in the screenshot below. Here I used a system exception along with the try, catch and throw keywords to ensure that a field is not entered as null or empty. If it is, there will be a manual error given to the user.

On the button click, if the first textbox if empty the error will be present and throw the error message as depicted in the bottom of the coding screenshot, within the try block and the catch block.

```
protected void Button1_Click(object sender, EventArgs e)
{
    string first = name.Text;
    string city = town.Text;
    string favColour = colour.Text;
    Color favColor =  Color.FromName(favColour);
    string result = "Hello there, " + first + " " + "from " + city + " welcome to this Dynamic Web Page!!";
    labelInfo.Text = result;
    labelInfo.ForeColor = favColor;
    Label1.ForeColor = favColor;
    Label2.ForeColor = favColor;
    Label3.ForeColor = favColor;
    Label4.ForeColor = favColor;
    Label5.ForeColor = favColor;
    Label6.ForeColor = favColor;

    try
    {
        if (string.IsNullOrWhiteSpace(first))
        {
            throw new NoNullAllowedException("Cannot be empty.");
        }

    }
    catch (NoNullAllowedException ex)
    {
        Label7.Text = "Cannot be empty";
        Label7.ForeColor = Color.Red;

    }
}
```

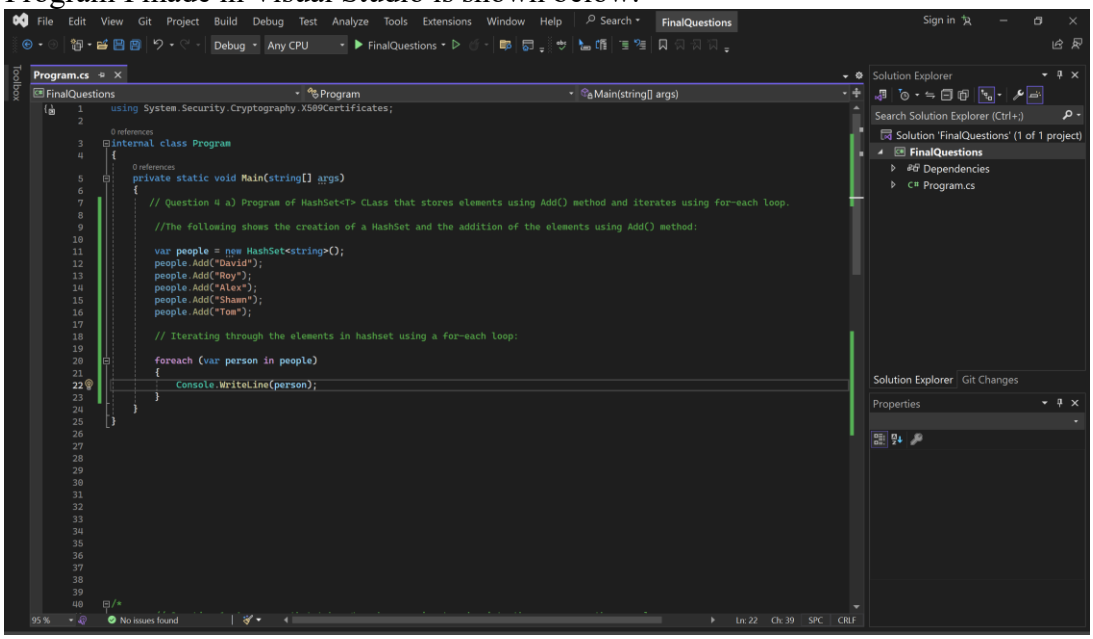
Output:

Evidently it worked, the exception was caught when the field was registered as empty, and the throw message is shown in red,

Q.4 (a) Write a C# program of to create a list of string shown below using HashSet<T> class that stores elements using Add() method and iterates elements using for-each loop.: Paste your code in a word document along with the IDE screenshot of generated output.

Below shows a screenshot of my program demonstrating the creation of a generic HashSet in C#, the addition to it using the Add() method and iterating through it with a for-each loop. The output is shown in the very bottom of the screenshot.

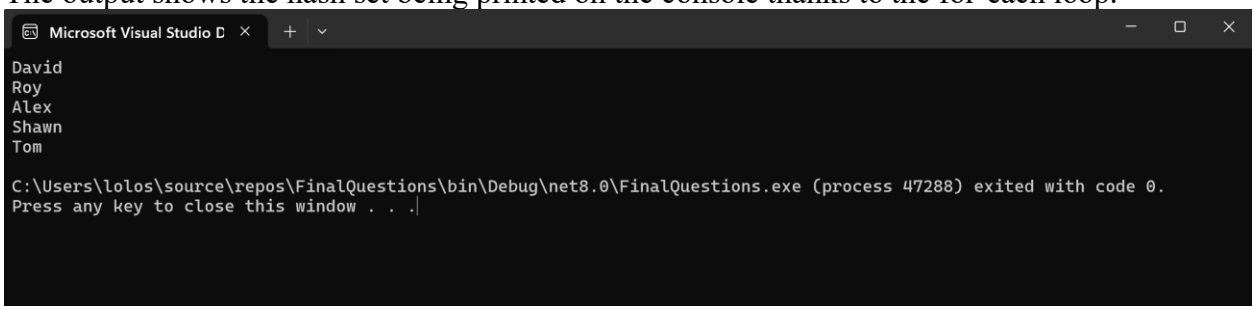
The Program I made in Visual Studio is shown below:



The corresponding code is pasted into the box below along with explanatory comments.

```
// Question 4 a) Program of HashSet<T> CLass that stores elements using Add()
method and iterates using for-each loop.

 //The following shows the creation of a HashSet and the addition of the elements
using Add() method:

 var people = new HashSet<string>();
 people.Add("David");
 people.Add("Roy");
 people.Add("Alex");
 people.Add("Shawn");
 people.Add("Tom");

 // Iterating through the elements in hashset using a for-each loop:

 foreach (var person in people)
 {
     Console.WriteLine(person);
 }
```

The output shows the hash set being printed on the console thanks to the for-each loop.



Q.4 b) List all the sorting algorithms. Explain Bubble sort and Selection Sort algorithm with proper programming examples.

The sorting algorithms are as follows:

i. Bubble Sort
ii. Insertion Sort
iii. Selection Sort
iv. Quick Sort
v. Merge Sort

Bubble Sort: The bubble sorting algorithm is a relatively simple sorting algorithm that repeatedly steps through the given list to be sorted, them compares each pair of items next to each other, and swaps them if they are in the wrong order. This process is repeated until no swaps are needed, which means that the list is completely sorted. Although the bubble algorithm is relatively simple, it is too slow and impractical for most programs. It can be practical if the input is usually in sorted order but may have a few out-of-order elements nearly in position.
The screenshot below shows my programming example demonstrating the Bubble sort algorithm. Evidently, with a given array, this algorithm uses the bubble sorting method to sort the array and print it on screen.

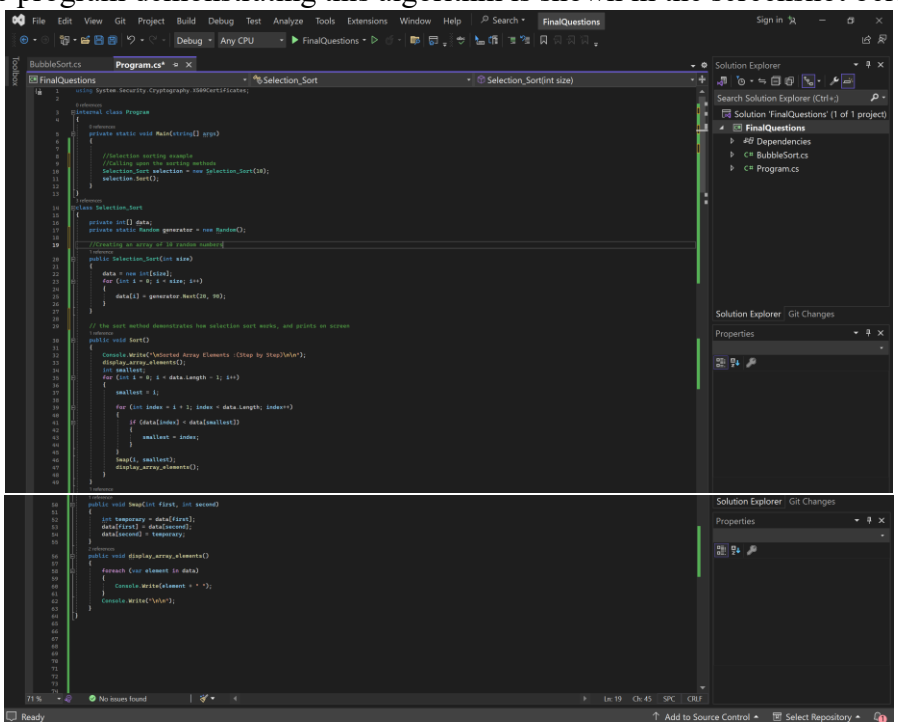
```
using System.Security.Cryptography.X509Certificates;

internal class Program
{
    private static void Main(string[] args)
    {

        //Bubble sorting example
            //Unsorted array
            int[] a = { 3, 0, 2, 5, -1, 4, 1 };
            int t;
            Console.WriteLine("Original array :");
            foreach (int aa in a)
                Console.Write(aa + " ");
            for (int p = 0; p <= a.Length - 2; p++)
            {
                for (int i = 0; i <= a.Length - 2; i++)
                {
                    if (a[i] > a[i + 1])
                    {
                        t = a[i + 1];
                        a[i + 1] = a[i];
                        a[i] = t;
                    }
                }
            }
            Console.WriteLine("\n" + "Sorted array :");
            foreach (int aa in a)
                Console.Write(aa + " ");
            Console.Write("\n");
```

The output of the code above is shown below. Evidently after iterating through the algorithm, the array is sorted, as shown in the screenshot below.

```
Original array :
3 0 2 5 -1 4 1
Sorted array :
-1 0 1 2 3 4 5

C:\Users\lolos\source\repos\FinalQuestions\bin\Debug\net8.0\FinalQuestions.exe (process 31048) exited with code 0.
Press any key to close this window . . .
```

Selection Sort: The selection sort is very similar to the bubble sort. However, the selection sort actually improves on the bubble sort, it does so by making only one exchange for every pass through the list. This facilitates the sorting process by making it faster when compared to bubble sort. What selection sort does, is it takes either the smallest or largest value in the list and places is in its appropriate location in the set. An example is shown in the screenshot below.

The program demonstrating this algorithm is shown in the screenshot below.



The output of this algorithm shows how it iterates through each step of this algorithm, and the random array that was generated ends up fully sorted.

Q.5 (a) Explain Accessor and Mutator in C# Programming with proper programming examples.

Accessors and mutators are sometimes called getter and setter respectively. An accessor or a getter is, when used with a mutator, used to protect the data in your program when creating classes. An accessor returns its value in a variable. On the other hand, a mutator or setter, is the second addition to an accessor. It is too useful in protecting data in a class. A mutator sets or updates values in a variable. An example of accessors and mutators in practice is shown in the screenshots below.

Program demonstrating a Class defining a sentence, with accessor and mutator (Getter and setter) shown below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FinalQuestions
{
    0 references
    internal class GetterAndSetter
    {
        //This class has one attribute, a sentance

        //The get and set keywords are for the accessor and mutators
        2 references
        public string Sentance { get; set; }

        0 references
        public string WriteSentance(string aSentance)
        {
            Sentance = aSentance;
            return Sentance;
        }
    }
}
```

Program that defines the sentence in the program file, and prints it on screen, shown below.

```
using FinalQuestions;
using System.Security.Cryptography.X509Certificates;

0 references
internal class Program
{
    0 references
    private static void Main(string[] args)
    {
        Sentances mySentance = new Sentances();
        mySentance.WriteSentance("Hello, My name is Lorena ");
        Console.WriteLine(mySentance.Sentance);
    }
}
```

The output which utilizes the accessor and mutators and prints the object is shown below.

```
Hello, My name is Lorena

C:\Users\lolos\source\repos\FinalQuestions\bin\Debug\net8.0\FinalQuestions.exe (process 11096) exited with code 0.
Press any key to close this window . . .
```

Q.5 (b) Give a proper definition of Public, Private and Protected and give any four differences between Public, Private and Protected.

Public private and protected are access modifiers. Access Modifiers are the keywords used to define the accessibility level for types and type members. By specifying an access

level, it is possible to control whether they can be accessed in other classes, or the current assembly or other assemblies based on our requirements.

Public: Used to define a type or type member where access is not restricted.

Private: Modifier used to define that the access of that type member is restricted to the containing type.

Protected: Modifier used to specify that access is limited to the containing type or types derived from the containing class.

Differences:

1. The first difference is that in public, any other part of the program can access that member, while this is not true for private and protected members.
2. Protected and public members can be accessed in inherited classes while private cannot.
3. Private is only accessed by the containing class, whereas the other modifiers have more flexible limitations.
4. Private can only be viewed by the containing class, public can be viewed by anyone, and protected can be viewed by an inherited or derived class.

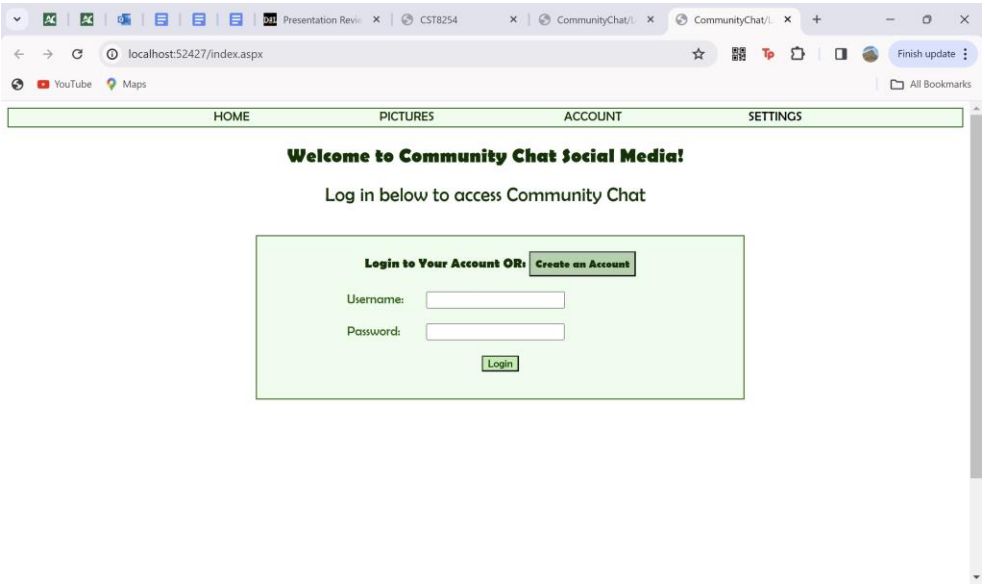
Part 2 – Programming application – Social media site (Community Chat)

The C# application I chose to create is a type of social media that I have called Community Chat. This application has 3 different pages , the index.aspx (which is the startup page) which is a login form to access the application. There is an option here to create an account which directs the user to the second page the accountCreate.aspx. Here there is a form to create and account which gathers the users credentials. Once successfully created they are directed back to login, then they can access the third page, home.aspx. Here is the homepage of the social media platform which allows them to view other's posts, and comment on them.

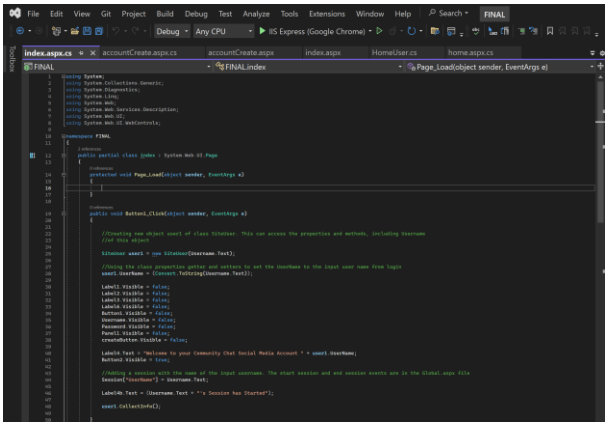
Below are some screenshots of the coding steps, and output of these three pages.

Step 1, Create Index.aspx:

In Microsoft Visual Studio, after creating my ASP.NET web application, and adding the first web form item called index.aspx; I used asp designer controls such as Labels, Pannels, Buttons and TextBoxes ValidationControls and AJAX controls to create the final output of this page meant for logging into the social media as shown below.



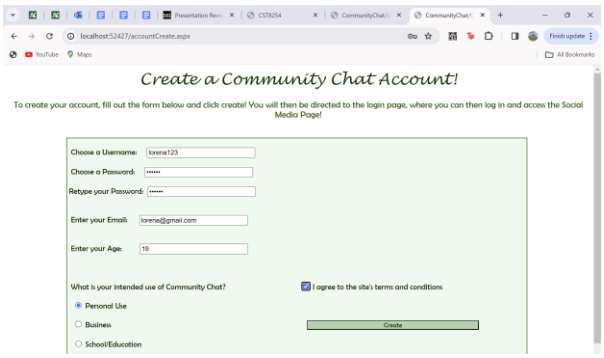
The next step once the design is completed is to add the C# coding which adds functionality to this page. All the coding was done in the index.aspx.cs file, and is shown below.

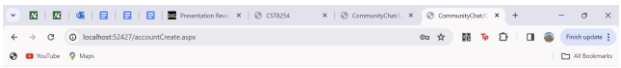


As seen in the above images of the C# coding, the coding done in the Button1_Click event explains what happens after the user logs in. It asynchronously hides the panel containing the login form, and displays a success message using their input username, and a button that redirects them to the home page. This also starts the user's session. In the second screenshot you can see how when the user clicks the second button after logging in, they are redirected to the home page, and a query string hold their username information in the URL. Additionally, the thrid button is the button that redirects the user to the page where they can create an accont.

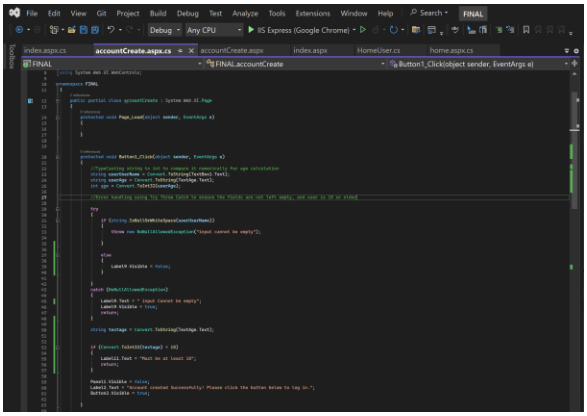
Step 2, create the page accountCreate.aspx

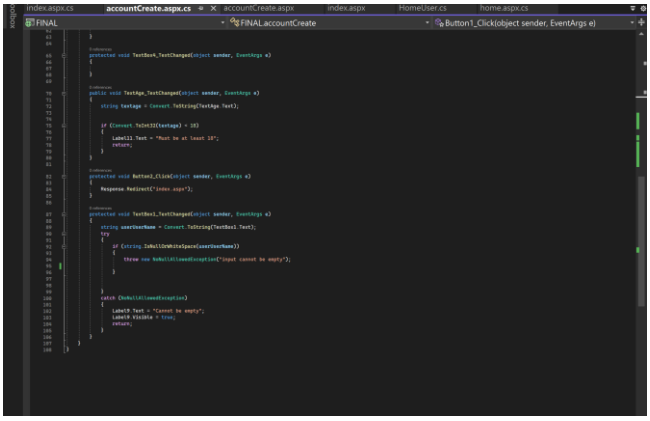
The second step I took after fully creating the first page, and programming it to make it dynamic and functional, I added another web form, calling it accountCreate. After adding it, I used various tools in the designer tab in visual studio to make the page look right. I used tools similar to the last page such as labels, panels, text boxes, radio buttons, check boxes, buttons and validation controls for design and functionality. Once it was looking as it does in the screenshots below, it was time to program some elements and add further functionality.





To add the C# programming I coded button and text elements in the accountCreat.aspsx.cs file. The coding is shown in the following screenshots.

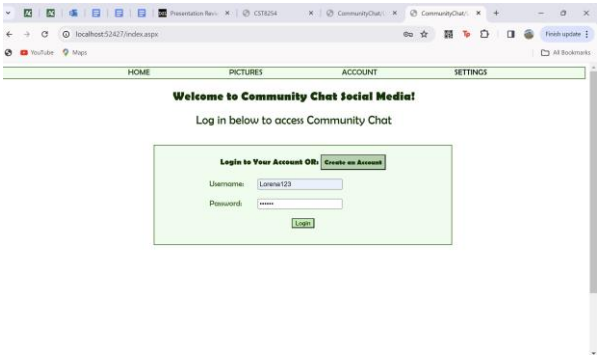


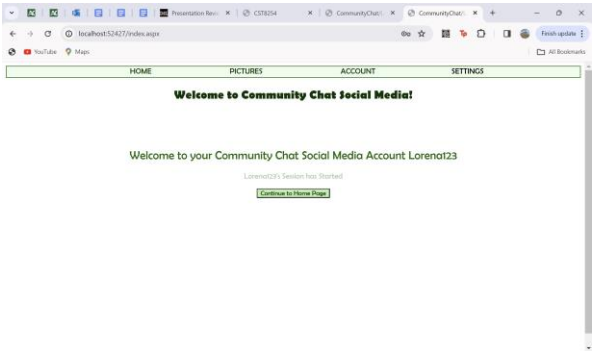


As seen in the above screenshots, the C# coding for this page includes error exception handling, and toggling the visibility of certain elements when the “create” button is clicked. If all the validation succeeds, but the users age is under 18 and/or the username text box is empty, the create button will not work, and there will be errors informing the user of the specifications for these elements (which will be discussed further later on). If the button is successful, the panel will be hidden a message saying that their account was created, and a button directing the user back to the login page will be present. This new button redirects them to log in.

Back to index.aspx

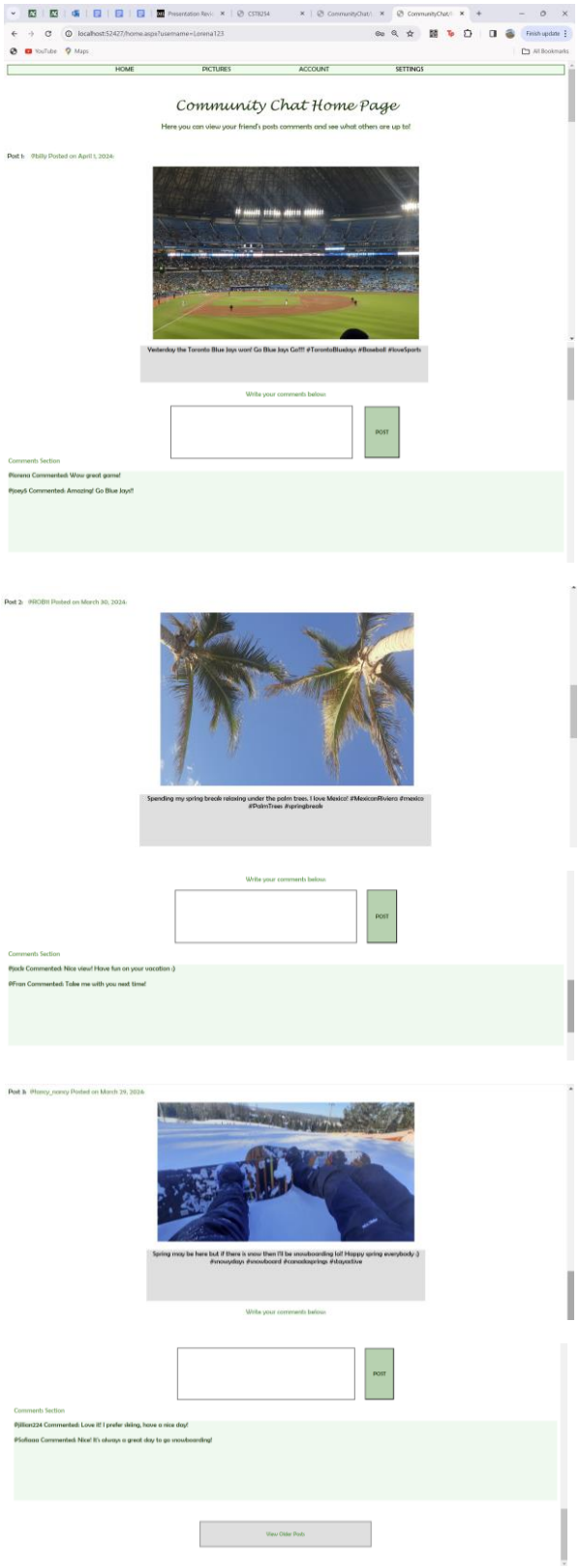
Now the user can log in, and access the homepage.





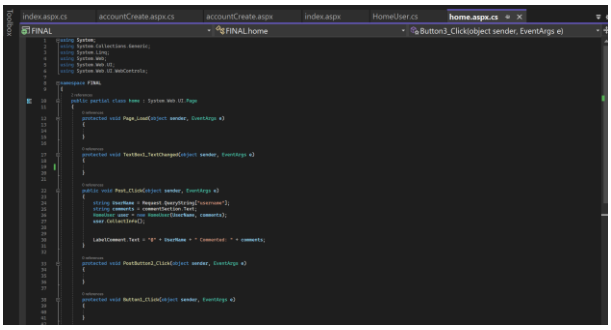
Step 3: Create and program the home.aspx page

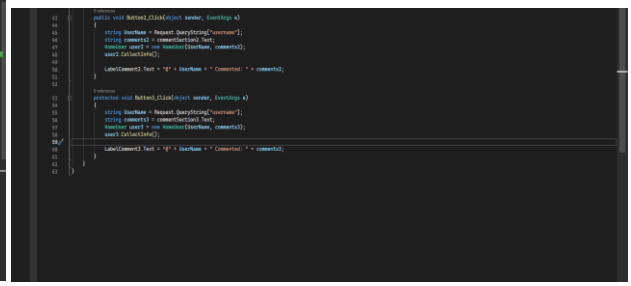
The final step was to create the third page called home.aspx. I used various asp and ajax tools within the designer tab in visual studio to create the looks of this page. The design is shown in the following screenshots.



The programming of all these elements was done in the home.aspx.cs file using C# programming. The main goal of the programming was to extract the user's username from the

query string and allow them to write their comments in the text boxes provided for each post, and have them posted along with the other comments including their username. The programming file is shown in the following screenshots.





As shown in the screenshots, most of the programming was done within the events of each button click. When a post button was clicked, the user's username was extracted from the query string, an object of HomeUser was created the comments were read from the textbox, and the comment is posted under the other comments with the user's username and their comment.

As you can see from the images above, these three pages are all connected, and lead to this home page. I will now go into greater detail about the elements of each page, and the programming behind them.

Page 1) index.aspx:

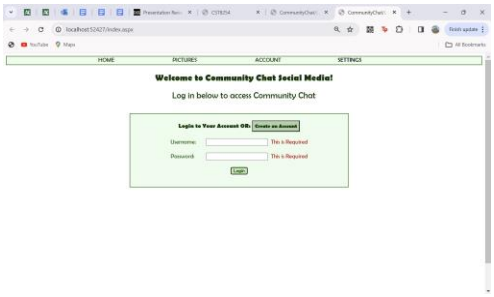
Elements included:

Ajax/Ajax toolkit, Validation Control, State Management, Class and Object Dynamic Web Page.

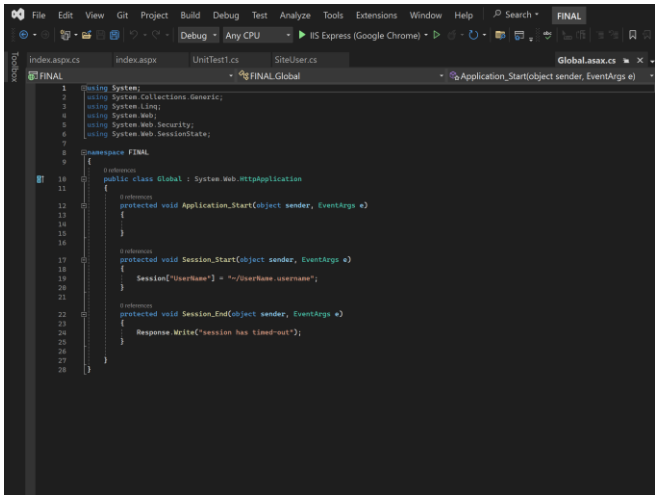
These elements are all within the index.aspx landing page.

Ajax/Ajax toolkit: These elements were used within this page by adding the ScriptManager, and UpdatePannel at the beginning of the page. This means that the page does not refresh when the login button is clicked. However, when the button is clicked and the inputs are valid, it asynchronously updates to hide the form elements, and shows a welcome message with the user's appropriate username, as well as an informative message saying that user's session has started, and a button that will redirect them to the home page. This is all done asynchronously thanks to the Ajax tools.

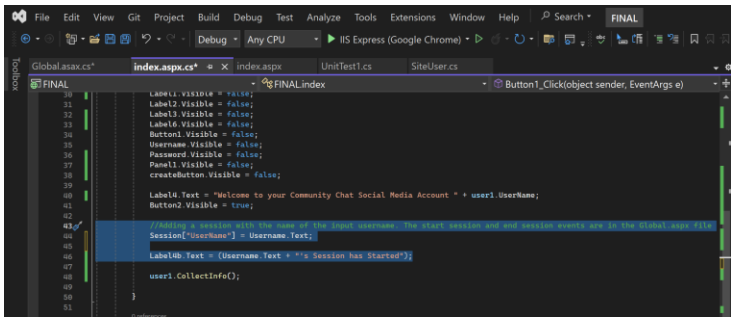
Validation Control: These elements are used in the input controls such as Username and Password text boxes. I used the RequiredField Validator to ensure that these boxes are not left empty upon clicking the login button. If they are empty, an error message saying that the elements are required appears, as shown below.



State Management: The state management is used in two different ways within this page, by starting a session, and by using query strings. Firstly, a session will start with name of the input Username when the login button is clicked. As seen in the screenshot of the Global.asax file, the session starts, and ends when it is times-out.

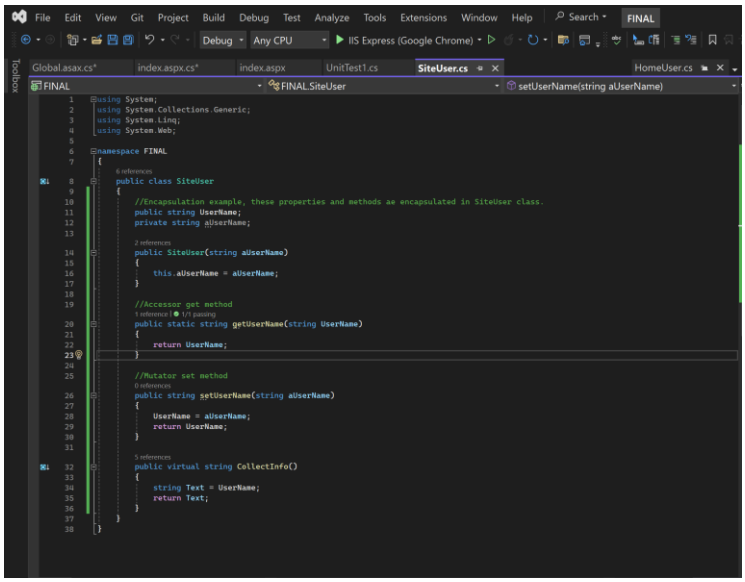


As seen in the index.aspx.cs file shown below, upon logging in, the user's username input is collected, and a session is created. This is notified by the label upon submission saying <username>'s Session Has Started as highlighted in the screenshot below.



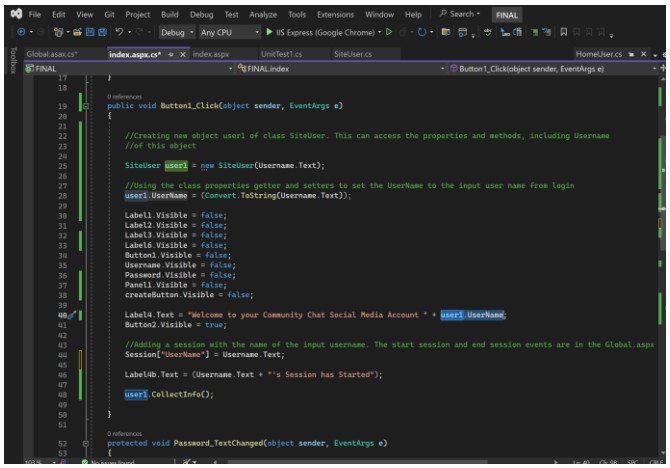
Furthermore, a Query string is used to send over the information of the username from the index page to the home page. The Response.Redirect ("home.aspx?username=" + Username.Text) inserts the current value of the username into the url so that the home page can read it and use that value. This is be elaborated further in the explanation of the home page.

Class and Object: I created a class called SiteUser.cs which has a UserName property, allowing me to create an object of this class, and set its UserName to the username entered in the form. The class is shown below.



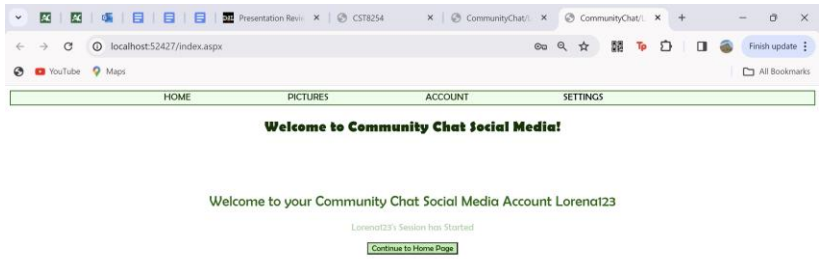
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace FINAL
{
    6 references
    public class SiteUser
    {
        //Encapsulation example, these properties and methods ae encapsulated in SiteUser class.
        public string UserName;
        private string aUserName;

        2 references
        public SiteUser(string aUserName)
        {
            this.aUserName = aUserName;
        }

        //Accessor get method
        1 reference | 0/1 passing
        public static string getUserName(string UserName)
        {
            return UserName;
        }

        //Mutator set method
        0 references
        public string setUserName(string aUserName)
        {
            UserName = aUserName;
            return UserName;
        }

        5 references
        public virtual string CollectInfo()
        {
            string Text = UserName;
            return Text;
        }
    }
}
```

I then created an instance of this class by initializing the object user1 within index.aspx.cs. After creating this user I can then call upon its UserName property, and utilize that further within my code. This is shown in the screenshot below.



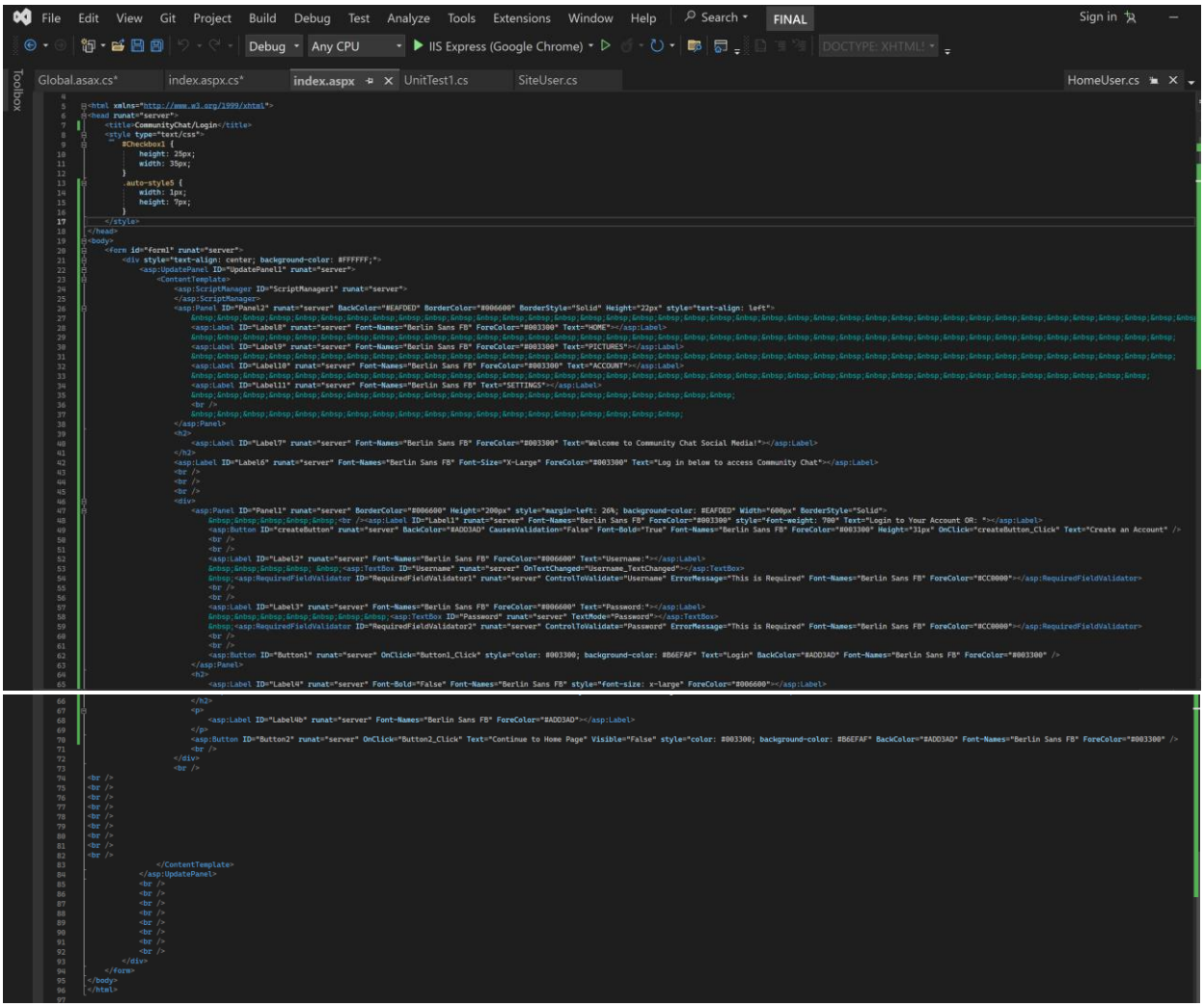
```
    }

    0 references
    public void Button1_Click(object sender, EventArgs e)
    {

        //Creating new object user1 of class SiteUser. This can access the properties and methods, including Username
        //of this object

        SiteUser user1 = new SiteUser(Username.Text);

        //Using the class properties getter and setters to set the UserName to the input user name from login
        user1.UserName = (Convert.ToString(Username.Text));

        Label1.Visible = false;
        Label2.Visible = false;
        Label3.Visible = false;
        Label6.Visible = false;
        Button1.Visible = false;
        Username.Visible = false;
        Password.Visible = false;
        Panel1.Visible = false;
        createButton.Visible = false;

        Label4.Text = "Welcome to your Community Chat Social Media Account " + user1.UserName;
        Button2.Visible = true;

        //Adding a session with the name of the input username. The start session and end session events are in the Global.aspx
        Session["UserName"] = Username.Text;

        Label4b.Text = (Username.Text + "'s Session has Started");

        user1.CollectInfo();

    }

    0 references
    protected void Password_TextChanged(object sender, EventArgs e)
    {
```

Dynamic WebPage: Index.aspx is certainly a dynamic webpage by how it reacts differently for different users. What makes it dynamic is the welcome message. After entering login credentials and clicking login, the page changes to look like the screenshot below:

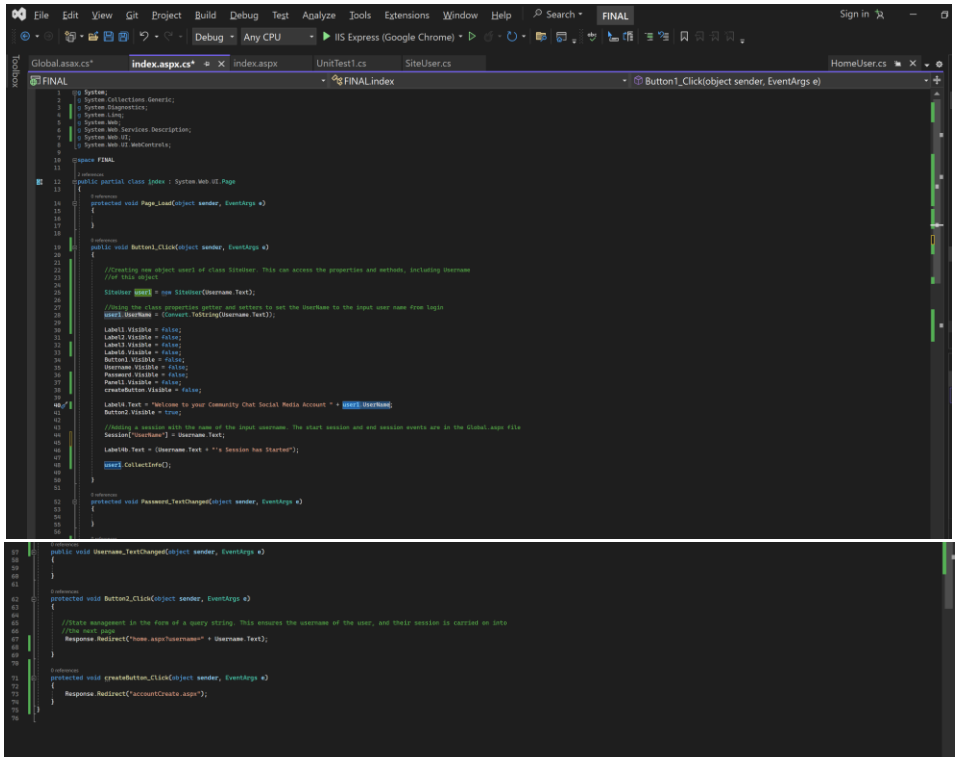


It welcomes the user by name, by using the object user1, and its property UserName. It adds this username to the label, making it more personal and dynamically changing depending on the user. Additionally, it says that this user's session has started, but the label changes depending on the user. My username is lorena123, so it says "lorena123's session has started".

The full source code (index.aspx):



C# coding for events (index.aspx.cs):

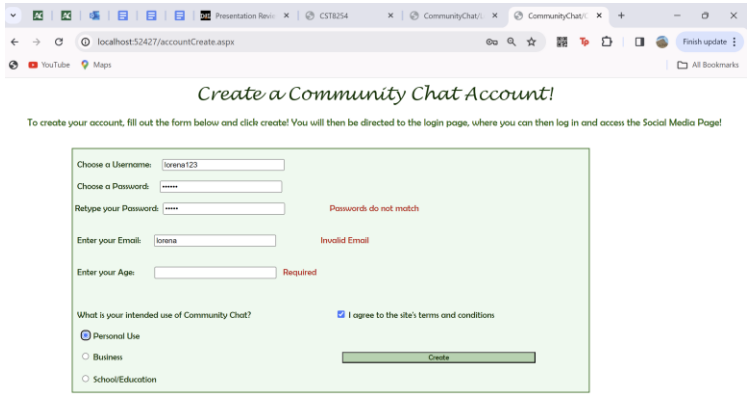


Page 2) accountCreate.aspx

This second page uses elements of Validation Control, Ajax toolkit, Error Handling and Security, and Type Casting.

This is the page the user would go to create an account, and all of the above elements are utilized.

Validation Control: Validation controls are used within the form fields. As shown in the screenshot below the input elements have various validation controls such as RequiredFieldValidator, RegularExpressionValidator, and CompareValidator. In the screenshot below you can see how each of these are used.



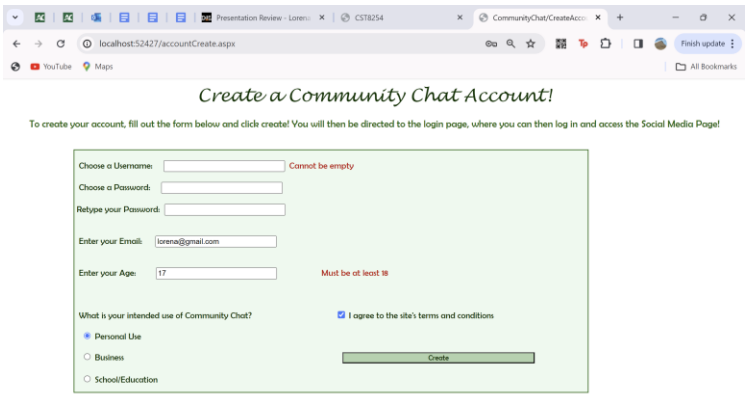
In the above screenshot, you can see how each of the mentioned validator controls are used. The form asks the user to enter the password twice to ensure it is entered properly, using the CompareValidator, I have set it to ensure that the values of these two textboxes are equal. As seen in the above screenshot, if they are not exactly the same, there is a red error message saying that they do not match. The regularExpressionValidator is used in the email field. My input regulat expression was < \w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)* > which is the standard pattern for an email address. If not following that pattern there is an invalid email error message. Lastly, the RequiredFieldValidator (which is used for all the fields) is demonstrated in the Age textbox. If empty, there is an error message saying that it is required in order to submit.

Ajax toolkit: The Ajax toolkit is used in the same was as the index page. By using the ScriptManager and the UpdatePannel elements, I am able to update the page asynchronously without refreshing.

TypeCasting: I used typecasting in this page in order to convert a string value to a 32bit integer. In the screenshot below you can see how I used the Convert.ToInt32 to convert a string value (which is taken from the age textbox) to an integer in order to compare it to a number.

```
0 references
public void TextAge_TextChanged(object sender, EventArgs e)
{
    string textage = Convert.ToString(TextAge.Text);


    if (Convert.ToInt32(textage) < 18)
    {
        Label11.Text = "Must be at least 18";
        return;
    }
}
```

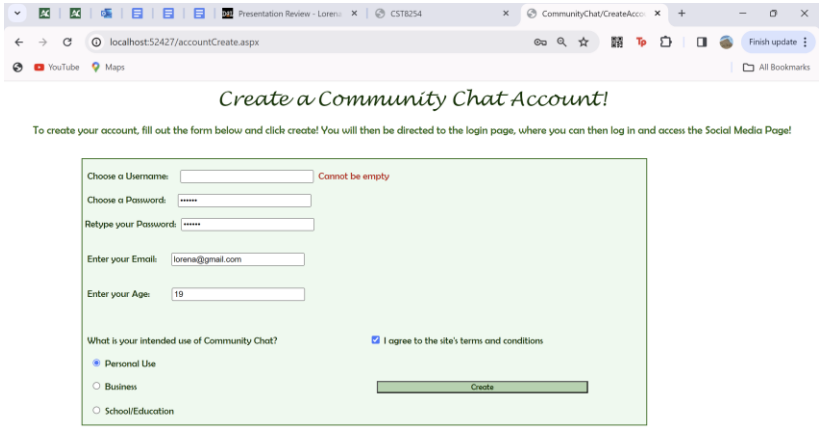
Here we want the user to be at least 18 years old (age of majority) so, by converting the string to an integer I can manually put an error message saying that the user must be 18 years old if the value entered is less than 18. This is demonstrated in the screenshot below.



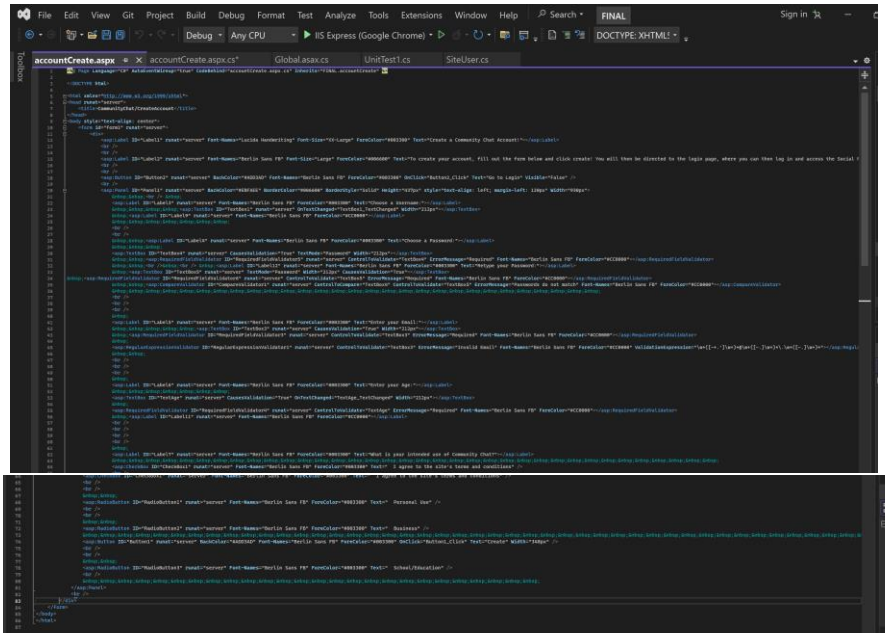
Error Handling and Security: This page also uses Error Handling and Security by using the Try, throw and catch keywords for error handling. In the screenshot below, these keywords are used to handle the exception that the first input in the form should not be empty.

```
0 references
protected void TextBox1_TextChanged(object sender, EventArgs e)
{
    string userUserName = Convert.ToString(TextBox1.Text);
    try
    {
        if (string.IsNullOrWhiteSpace(userUserName))
        {
            throw new NoNullAllowedException("input cannot be empty");

        }


    }
    catch (NoNullAllowedException)
    {
        Label9.Text = "Cannot be empty";
        Label9.Visible = true;
        return;
    }
}
```

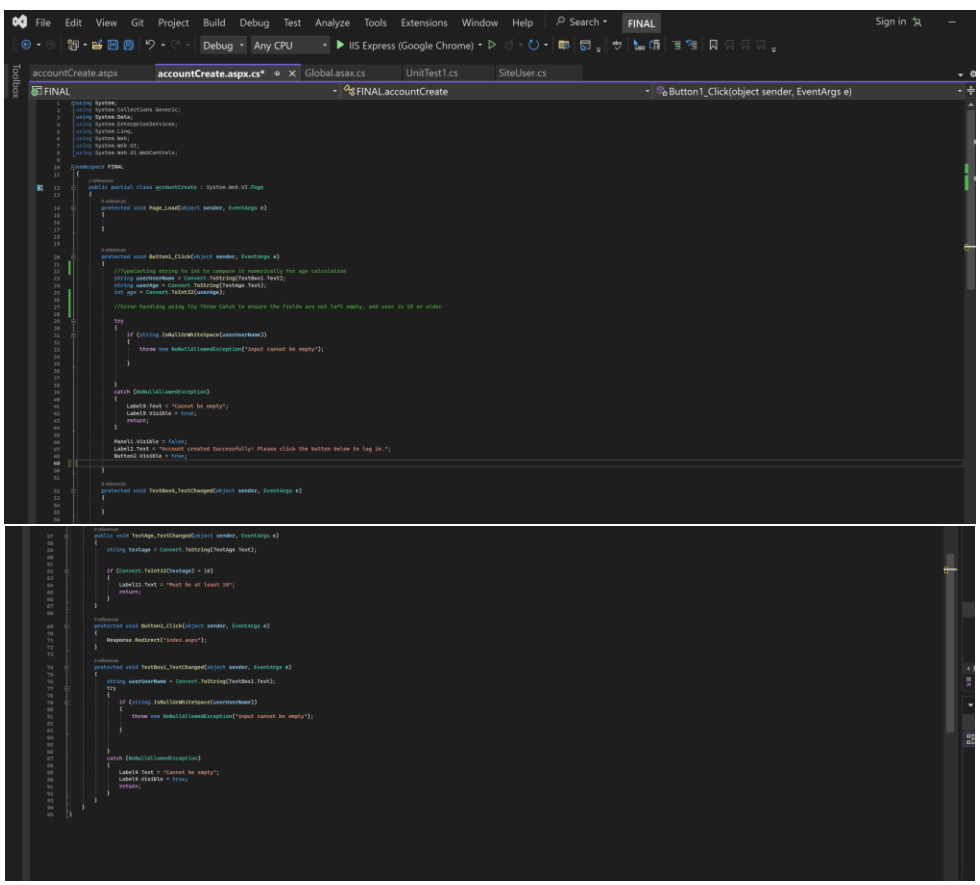
Evidently, in the screenshot above, the try throw and catch keywords are used to ensure that the username text box will not be empty or null. If it is, it throws the exception, and the error message is present as shown in the screenshot below.



Complete Source Code (accountCreate.aspx):



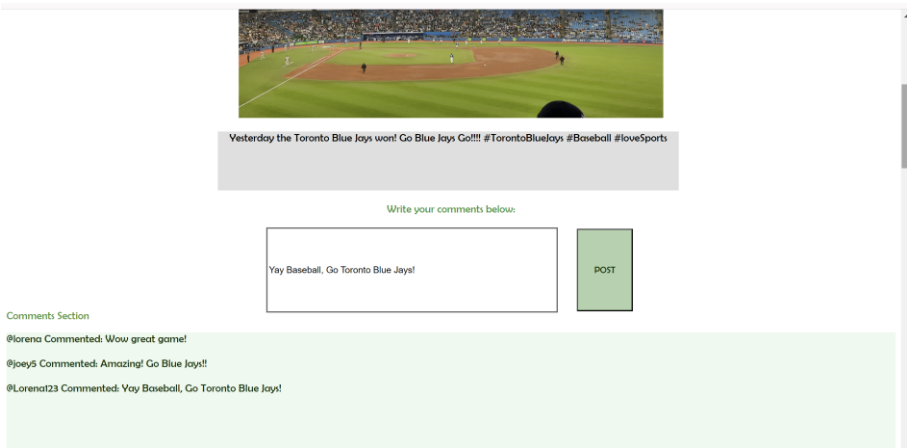
All C# Coding for events (accountCreate.aspx.cs):



Page 3) Home.aspx

This home page is the social media platform. It includes elements of Dynamic web pages, Ajax, Class State Management and Class and Object.

Dynamic Web Pages: This page is certainly dynamic; it changes depending on each user and updates according to their inputs. As you have probably noticed earlier, each "post" has a comments section and an area for the user to write their comments in a text area and post them. This updates the comments section to include what the user wrote. This is demonstrated in the screenshot below. For example, if I wanted to comment on the post about the blue jays baseball I could write, "go blue jays", when I hit the post button it will update the comments section to My comment and each post has this feature!

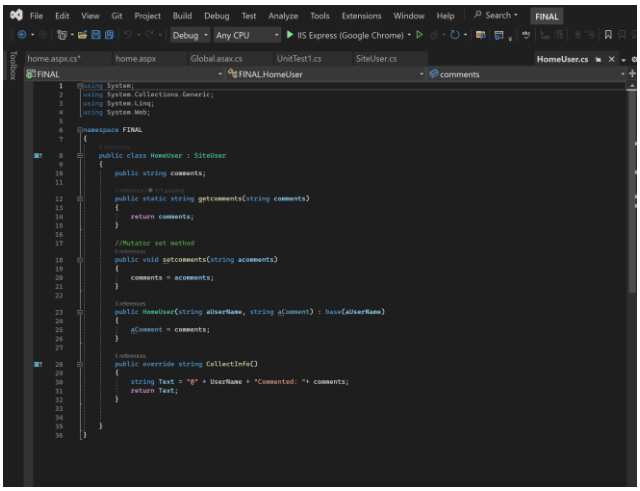


Ajax: Like each other page in this application, the home.aspx page includes the ajax tools of scriptManager and UpdatePannel. These are especially useful in this page since whenever a user wants to make a post the ajax allows the page to update without refreshing the whole page.

State Management: As you may have noticed in the previous screenshot regarding the comments, when a user comments, it does not just post their comment, it posts "@ <username> Commented: <their comments>". Since there is no input asking the user for their username again, I needed to use the State Management Query string to collect their username. Since they entered their username in the login page, and I added it to the query string to send to the home page, I can now read that information in the URL and use it. The code below shows how I extracted the username from the URL which can then be used anywhere else in the code, and this is done for each posts comment section.

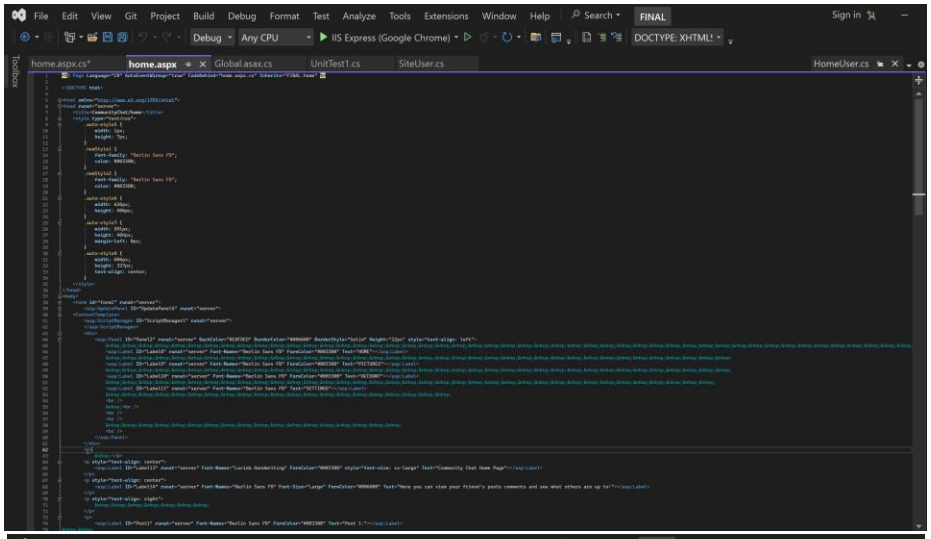
```
0 references
public void Post_Click(object sender, EventArgs e)
{
    string UserName = Request.QueryString["username"];
    string comments = commentSection.Text;
    HomeUser user = new HomeUser(UserName, comments);
    user.CollectInfo();


    LabelComment.Text = "@" + UserName + " Commented: " + comments;
}
```

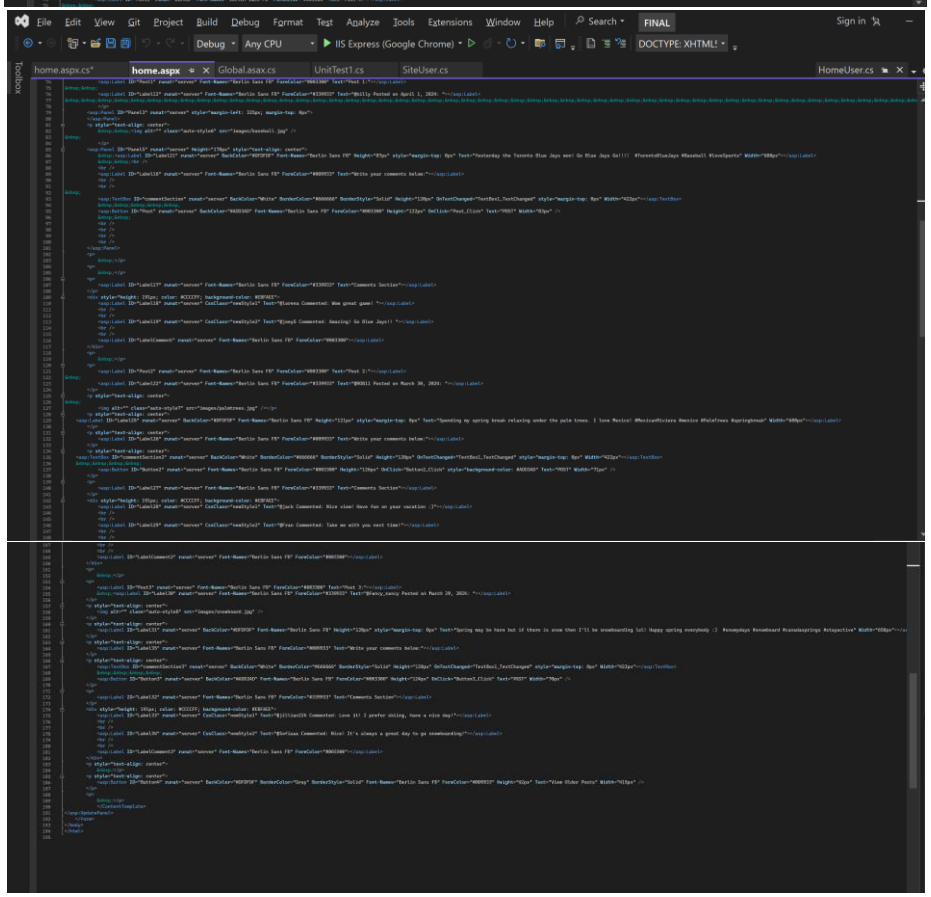
Class and object: As demonstrated in the above screenshot, there is another class that I had created called HomeUser, which has the properties of UserName and comments. In the home.aspx.cs I created an object of that class called user, which utilized these properties. The full class is shown below.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace FINAL
{
    public class HomeUser : SiteUser
    {
        public string comments;

        public static string getcomments(string comments)
        {
            return comments;
        }

        //Mutator set method
        public void setcomments(string acomments)
        {
            comments = acomments;
        }

        public HomeUser(string aUserName, string aComment) : base(aUserName)
        {
            aComment = comments;
        }

        public override string CollectInfo()
        {
            string Text = "@" + UserName + "Commented: "+ comments;
            return Text;
        }


    }
}
```

Full Source Code (home.aspx):





All C# Coding for events (home.aspx.cs):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace FINAL
{
    2 references
    public partial class home : System.Web.UI.Page
    {
        0 references
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        0 references
        protected void TextBox1_TextChanged(object sender, EventArgs e)
        {

        }

        0 references
        public void Post_Click(object sender, EventArgs e)
        {
            string UserName = Request.QueryString["username"];
            string comments = commentSection.Text;
            HomeUser user = new HomeUser(UserName, comments);
            user.CollectInfo();


            LabelComment.Text = "@" + UserName + " Commented: " + comments;
        }

        0 references
        protected void PostButton2_Click(object sender, EventArgs e)
        {

        }

        0 references
        protected void Button1_Click(object sender, EventArgs e)
        {

        }

        0 references
        public void Button2_Click(object sender, EventArgs e)
        {
            string UserName = Request.QueryString["username"];
            string comments2 = commentSection2.Text;
            HomeUser user2 = new HomeUser(UserName, comments2);
            user2.CollectInfo();

            LabelComment2.Text = "@" + UserName + " Commented: " + comments2;
        }

        0 references
        protected void Button3_Click(object sender, EventArgs e)
        {
            string UserName = Request.QueryString["username"];
            string comments3 = commentSection3.Text;
            HomeUser user3 = new HomeUser(UserName, comments3);
            user3.CollectInfo();

            LabelComment3.Text = "@" + UserName + " Commented: " + comments3;
        }
    }
}
```

Encapsulation, Inheritance and Polymorphism: These were done within my two classes, SiteUser and HomeUser. Properties and methods are encapsulated into these classes, such as the properties of UserName or comments. The HomeUser class actually inherits from the SiteUser and they both have a virtual method called CollectInfo(). These are demonstrated in the screenshots below of the classes.

SiteUser.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace FINAL
{
    6 references
    public class SiteUser
    {
        //Encapsulation example, these properties and methods ae encapsulated in SiteUser class.
        public string UserName;
        private string aUserName;

        2 references
        public SiteUser(string aUserName)
        {
            this.aUserName = aUserName;
        }

        //Accessor get method
        1 reference | 19 passing
        public static string getUserName(string UserName)
        {
            return UserName;
        }

        //Mutator set method
        0 references
        public string setUserName(string aUserName)
        {
            UserName = aUserName;
            return UserName;
        }

        5 references
        public virtual string CollectInfo()
        {
            string Text = UserName;
            return Text;
        }
    }
}
```

HomeUser.cs

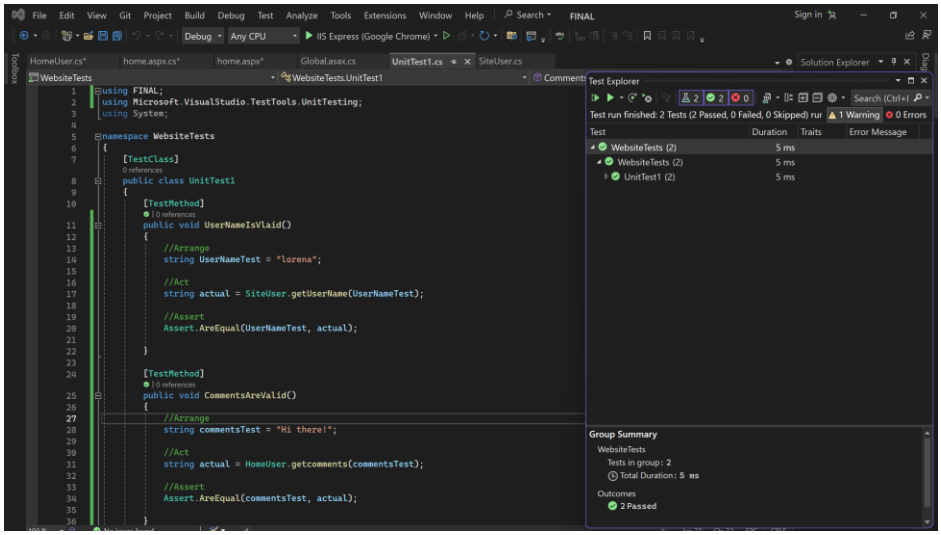
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace FINAL
{
    //Inherits from SiteUser
    0 references
    public class HomeUser : SiteUser
    {
        public string comments;

        1 reference | 1/1 passing
        public static string getcomments(string comments)
        {
            return comments;
        }

        //Mutator set method
        0 references
        public void setcomments(string acomments)
        {
            comments = acomments;
        }

        //Uses the property of Username from SiteUser base class
        3 references
        public HomeUser(string aUserName, string aComment) : base(aUserName)
        {
            aComment = comments;
        }

        5 references
        public override string CollectInfo()
        {
            string Text = "@" + UserName + "Commented: "+ comments;
            return Text;
        }


    }
}
```

Evidently, the HomeUser inherits from the base class SiteUser. It now accesses the SiteUser's property of Username, without having to declare it.

Additionally, the base class SiteUser has a virtual method Of CollectInfo() which return the property values of the object (in this case the username.) In the second screenshot you can see how it is overridden by the inherited class HomeUser, now returning both the username and the comments. This is an example of Polymorphism by how it changes depending on the object. An example of this is within the home.aspx.cs which calls upon the method as an object of HomeUser class, therefore it will return both the Username and comments as shown below.

```
    0 references
    protected void Button3_Click(object sender, EventArgs e)
    {
        string UserName = Request.QueryString["username"];
        string comments3 = commentSection3.Text;
        HomeUser user3 = new HomeUser(UserName, comments3);
        user3.CollectInfo();

        LabelComment3.Text = "@" + UserName + " Commented: " + comments3;
    }
}
```

Testing: The testing done for this project were Unit Tests. By adding a unit test project in this solution I was able to test and debug several different methods and functions. As seen in the screenshot below containing a couple of these tests, I was able to properly ensure that the code would word as expected each time.



The tests were made using the AAA pattern, of arrange, act and assert. By creating these tests and ensuring that they all pass I was able to debug my program more easily and assure that the program works as expected every time.