ASP.NET Core Activity 6

Part 1 The Debugger

Goal and Directions:

In this activity, you will learn how use the **Visual Studio Debugger**. We will demonstrate the following debug options:

1. Set a breakpoint
2. Examine the state of variables during a breakpoint
3. Step Over, Step In, and Step Out of code execution
4. Set a conditional breakpoint
5. Log messages with a breakpoint
6. Examine the call stack of an application

You will need the code from a previous activity, the login handler application.

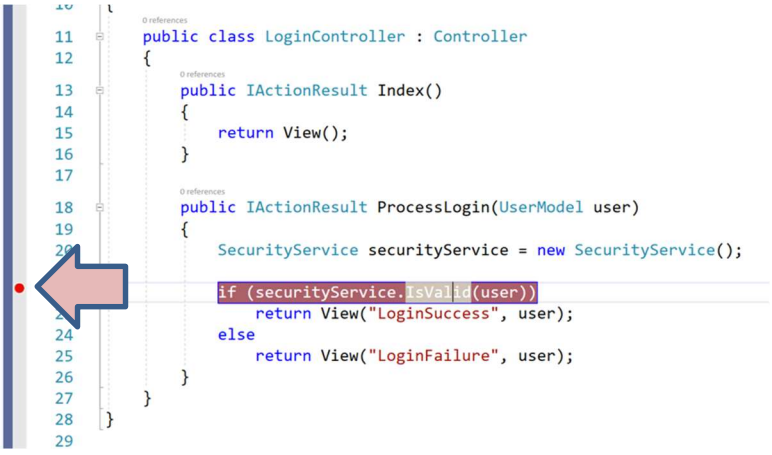
About Debugging

Often the source of a program error occurs during runtime even though all of the syntax of a program may compile correctly. In order to find and diagnose these errors, the process of debugging an app will reveal logical errors in the code. Debugging and breakpoints allow you to view the value, or contents, of variables during the program execution.

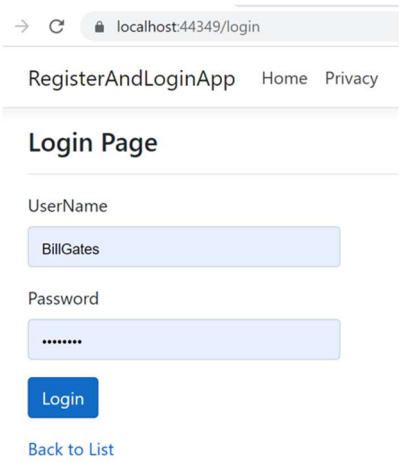
Set a Breakpoint

1. Open the solution for **Register and Login App** that we did in a previous lesson.

2. Open up the LoginController class.
3. Set a breakpoint on the **securityService.IsValid** line in the ProcessLogin method.



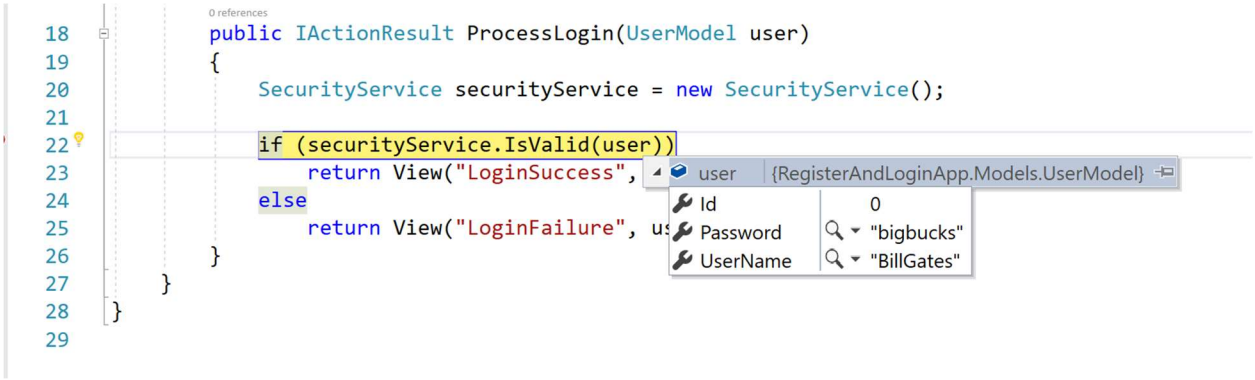
4. Run the application in debug mode. **Login** with known good log in credentials.

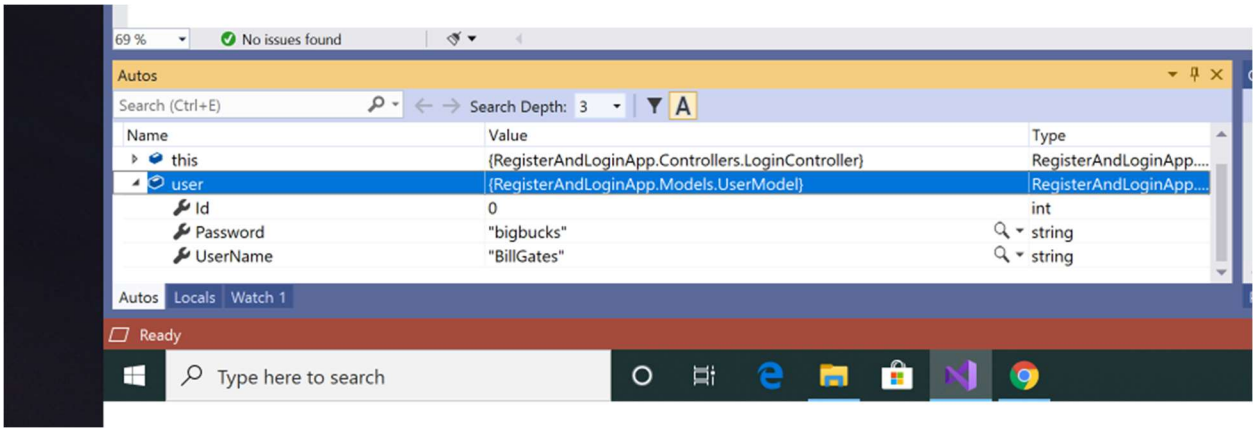


The application should **pause execution** and Visual Studio will show you the code in **debug mode**.

In debug mode, you can **examine the values** of variables and classes used on the page.

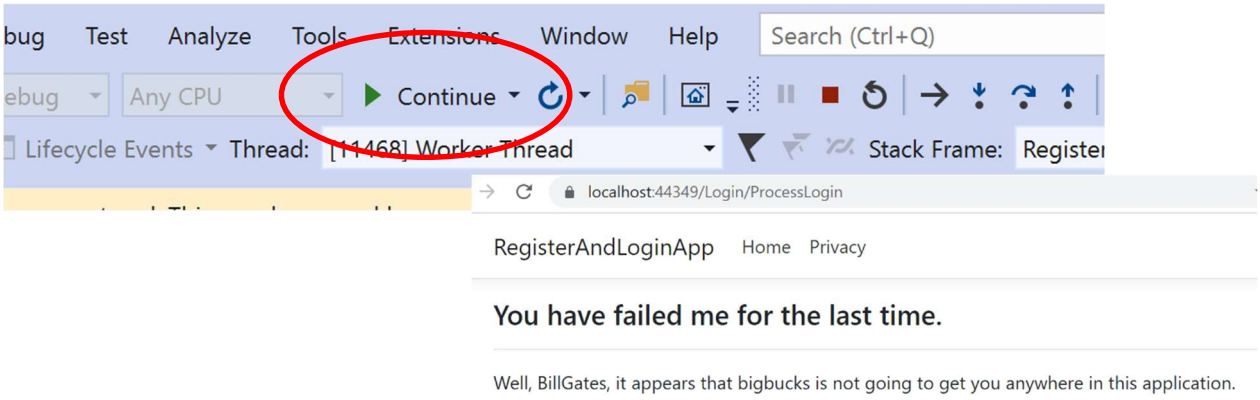
For example, you should be able to hover over the **user** variable and display its contents. You can see this either in the code or in the "autos" window below the code.





5. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

6. Click the **Continue** button. The program will resume execution and display the login success or failure view.

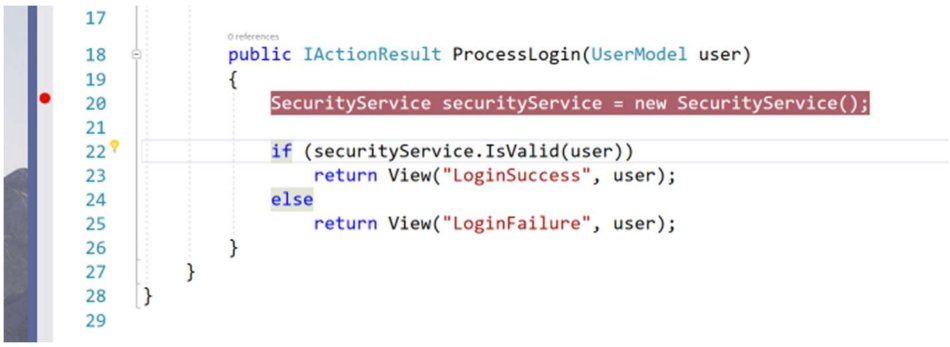


7. Click the **Stop** button.
8. **Remove** all breakpoints.
9. You should be able to run the application normally again without any debug pauses.

Stepping into a Function, over a Function, and out of a Function.

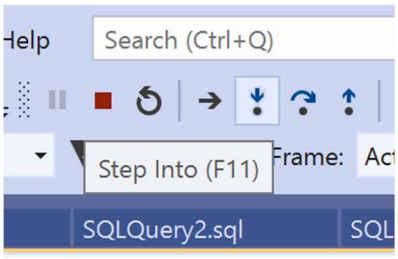
The debugging process allows you to view the values of classes and methods as they are called. The "Step Into" process allows you to see more details of the code execution.

1. Open up the **LoginController** class.
2. Set a **breakpoint** in Login() where the security service is instantiated.



3. Run the application in debug mode.
4. Login.
5. Verify that code stopped at the breakpoint.

6. Click the **Step Into** icon from the debugger toolbar or enter the F11 key. The code will then "step into" the Security Service class since this is the method being accessed at the moment.



7. Inspect the **UserModel** argument passed to **IsValid**() by hovering over the variable.

```
11      public class SecurityService
12      {
13          SecurityDAO securityDAO = new SecurityDAO();
14
15          public bool IsValid(UserModel user)
16          {
17              return securityDAO.FindUserByNa
18          }
19      }
20  }
21
```

user {RegisterAndLoginApp.Models.UserModel}
Id 0
Password "bigbucks"
UserName "BillGates"

8. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

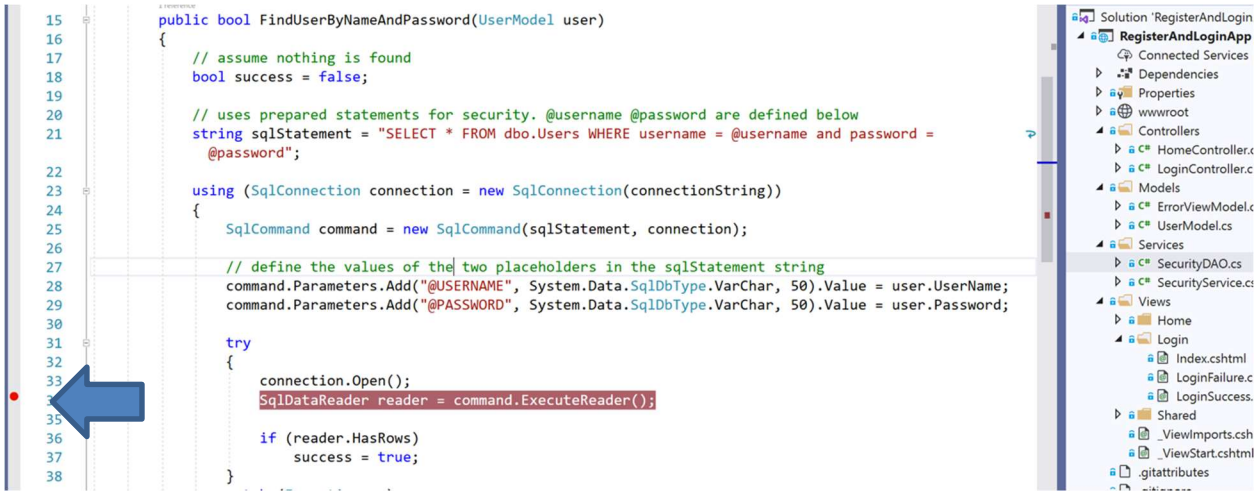
9. Click the **Step Over** icon from the debugger toolbar or enter the F10 key. Continue stepping over code statements until the line of code that creates an instance of the SecurityDAO(). You should be able to view the properties of the service object. For example, you can see the "Data Source" property connection string.

```
18      public IActionResult ProcessLogin(UserModel user)
19      {
20          SecurityService securityService = new SecurityService();
21
22          if (securityService.IsValid(user))
23              reti
24          else
25
26          }
27      }
28  }
29
```

securityService {RegisterAndLoginApp.Services.SecurityService}
securityDAO {RegisterAndLoginApp.Services.SecurityDAO}
connectionString "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=Test;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificat

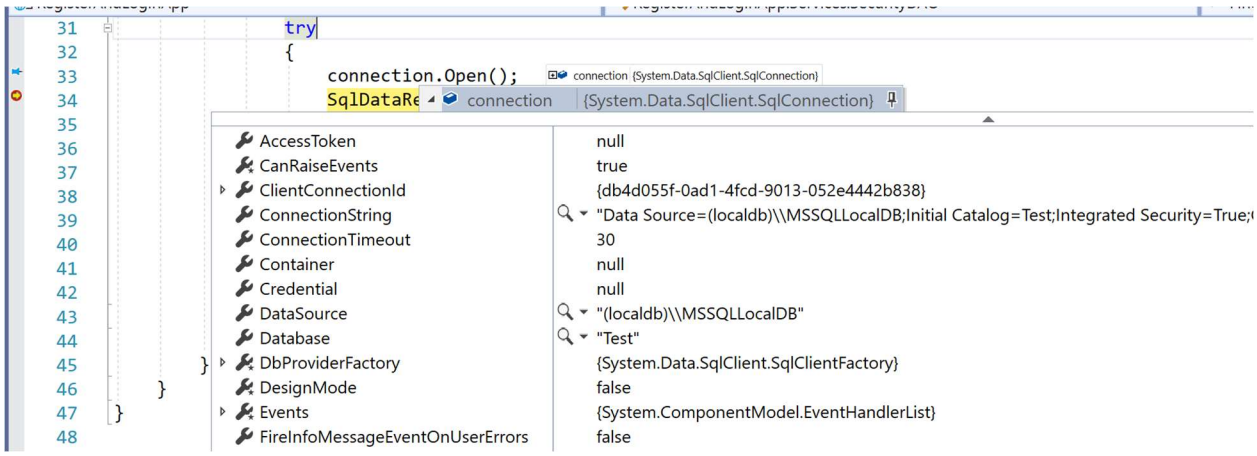
10. Stop the program.

11. In the SecurityDAO file, set another breakpoint on the line of code in the FindByUser() that calls ExecuteReader().



12. Run the program again. Verify the code stopped at both of the set breakpoints.

13. Inspect the properties of the connection. You should notice that dozens of properties exist in complex classes such as connection, which will help you diagnose problems with code.



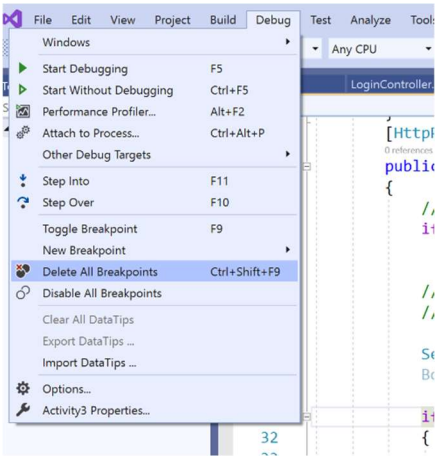
14. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

15. Step through the code (using Step Over) until the return statement is reached. You should be able to determine whether or not the user was found in the database.

```
32                  {
33                      connection.Open();      connection {System.Data.SqlClient.SqlConnection}
34                      SqlDataReader reader = command.ExecuteReader();
35
36                      if (reader.HasRows)
37                          success = true;      success false
38                  }
39                  catch (Exception ex)
40                  {
41                      Console.WriteLine(ex.Message);
42                  };
43              }
44              return success;  ≤ 3ms elapsed
45          }                    success  false
46      }
47  }
48
```

16. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Setting Conditional Breakpoints



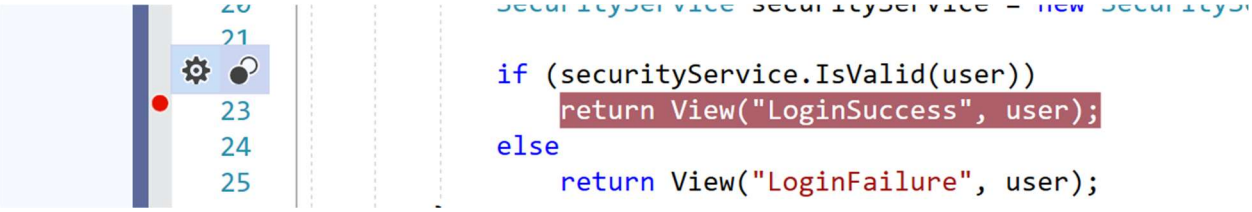
You can pause the execution of the program when certain conditions are met, which will save you time when running loops or long execution processes.

1. Stop the program and remove all breakpoints (see option from the Debug menu).
2. Open up the LoginController class.
3. Set a breakpoint on the line **return View("LoginSuccess", user)**, which in this example is line 23.

This breakpoint should only occur if the login credentials are valid.

```
0 references
18          public IActionResult ProcessLogin(UserModel user)
19          {
20              SecurityService securityService = new SecurityService();
21
22              if (securityService.IsValid(user))
23                  return View("LoginSuccess", user);
24              else
25                  return View("LoginFailure", user);
26          }
27      }
28  }
29
```

4. Click the "cog" that will be displayed when you hover over breakpoint "dot."



5. Click the **Conditions** checkbox.
6. Set a conditional statement. In this example, the breakpoint will stop only when the username is "Bill Gates."
7. Click the close button.

8. You can also put in a **conditional breakpoint** using C# code as shown here.

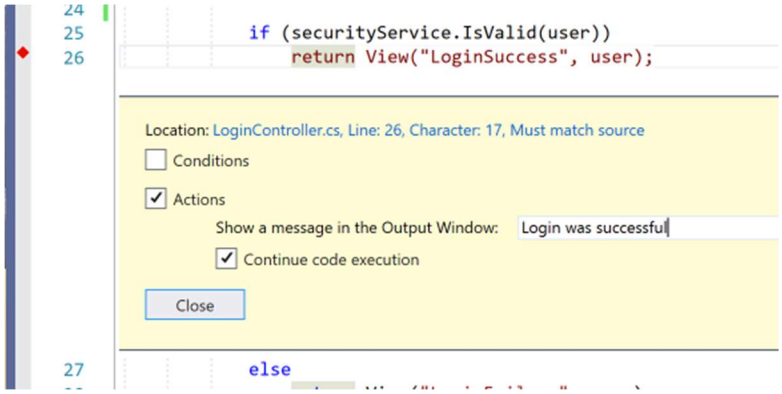
```
0 references
18      public IActionResult ProcessLogin(UserModel user)
19      {
20          SecurityService securityService = new SecurityService();
21
22          if (user.UserName == "Bill Gates")
23              System.Diagnostics.Debugger.Break();
24
25          if (securityService.IsValid(user))
26              return View("LoginSuccess", user);
```

9. Run the application in debug mode.
10. Verify that code stopped at the breakpoint only when the login was successful. Login failure should not cause the program to pause.

11. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

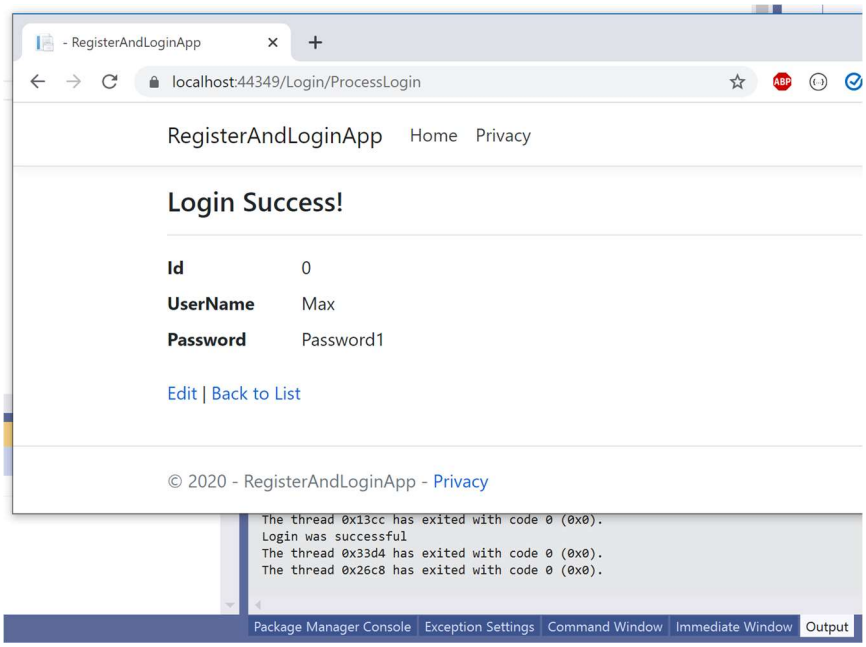
Logging Debugger Messages

1. On the "Success" breakpoint, created in the previous steps, click the "cog" that will be displayed when you hover over breakpoint "dot."
2. Uncheck the conditions box.
3. Check the Actions checkbox. Enter a message "Login was successful" in the 'Log a message to Output Window.'
4. Check the Continue execution checkbox.
5. Click the close button.



6. Run the application in debug mode.
7. Verify the message was logged in the Output Window.
8. Click the Continue button.

9. Take a screenshot of the app at this stage. Paste it into a Word document and caption the image with a brief explanation of what you just demonstrated.

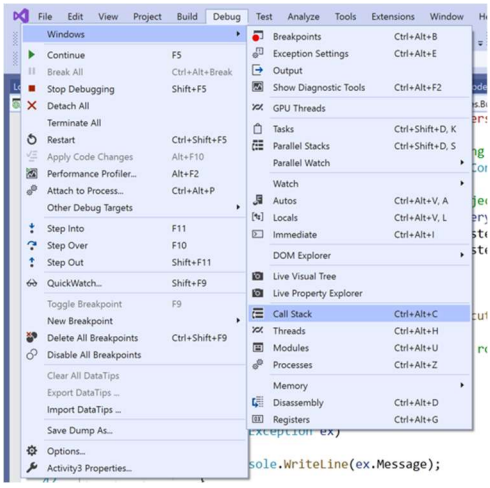


Inspecting the Call Stack:

1. Open up the **SecurityDAO** class.
2. Set a breakpoint on the line of code in return statement for the FindByUser().

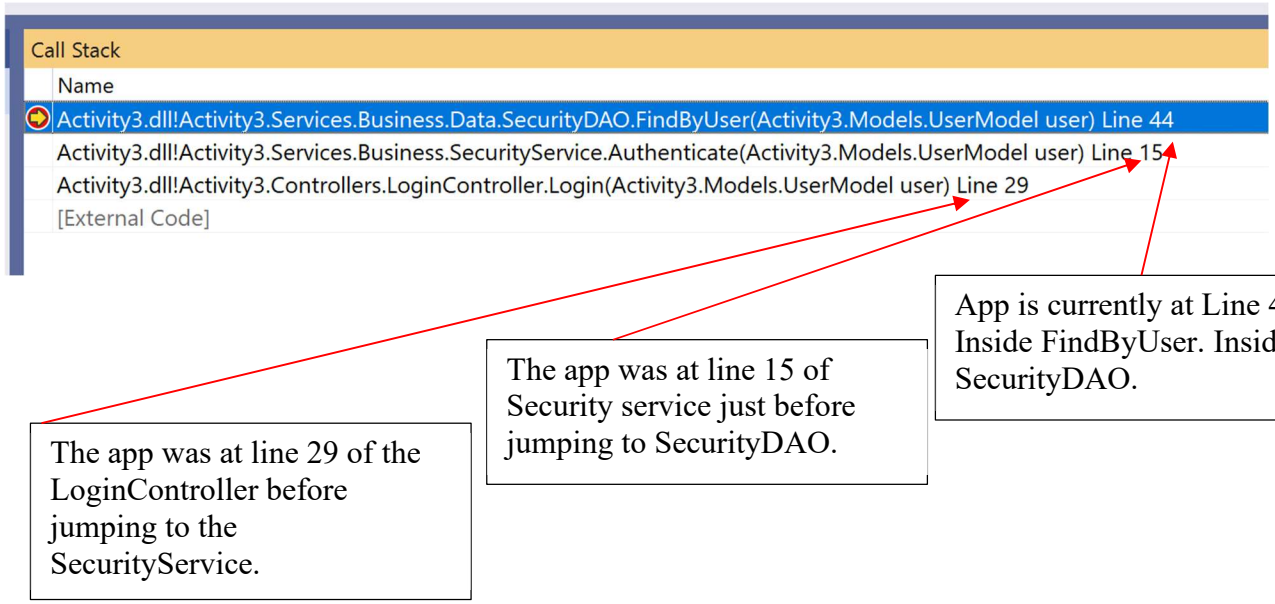
```
            Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False";

        public bool FindByUser(UserModel user)
        {
            // start by assuming that nothing found in this query.
            bool succes = false;

            // Provide the query string with a parameter placeholder.
            string queryString = "select * from dbo.users where username = @username and password = @password";

            // Create and open the connection in a using block. This ensures that all resources will be closed and dis
              when the code exits.
            using (SqlConnection connection =  new SqlConnection(connectionString))
            {
                // Create the Command and Parameter objects.
                SqlCommand command = new SqlCommand(queryString, connection);
                command.Parameters.Add("@USERNAME", System.Data.SqlDbType.VarChar, 50).Value = user.Username;
                command.Parameters.Add("@PASSWORD", System.Data.SqlDbType.VarChar, 50).Value = user.Password;
                try
                {
                    connection.Open();
                    SqlDataReader reader = command.ExecuteReader();

                    // if the query finds at least one row in the table, then the user exists. return true
                    if (reader.HasRows)
                        succes = true;
                    reader.Close();
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
            return succes;
        }
    }
}
```

3. Run the application in debug mode.
4. Verify that code stopped at the breakpoint set.
5. Choose the Debug menu > Windows > Call stack. Inspect the Call Stack Window.

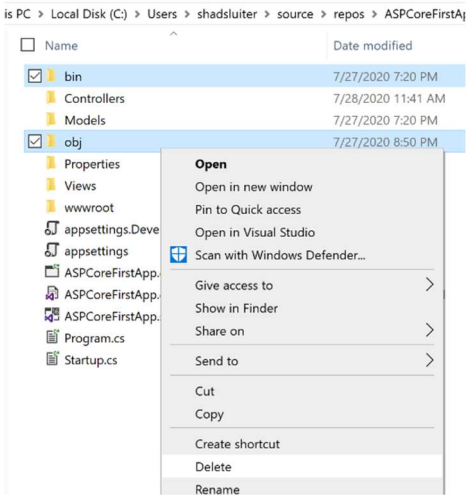


By using the **Call Stack** window, you can view the function or procedure calls that are currently on the stack. **The Call Stack window shows the order in which methods and functions are getting called.** The call stack is a good way to examine and understand the execution flow of an app.

6. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.



Deliverables:



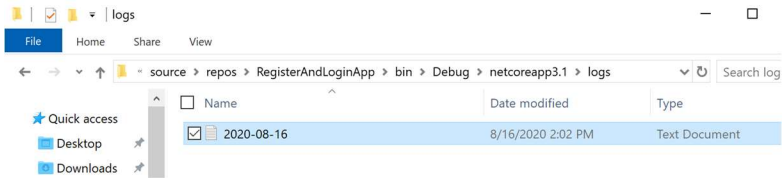
1. This activity has multiple parts. Complete all parts before submitting.
2. Submit a Microsoft Word document with screenshots of the application being run at each stage of development. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
4. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.

2 NLog

Goals and Directions:

- **nLog Framework** – In this activity, you will learn how add logging using the NLog Logging Framework to the previous activity.
- **Singleton Pattern** – You will also build a reusable Logging Utility class, which also implements the Singleton Design Pattern.

The end product of this exercise is to **create a text log file** generated by the app in the /logs subfolder of the project. This file will be a record of events triggered during the execution of the application. Logging is similar to Debugging in that errors and events can be tracked during troubleshooting or auditing events.

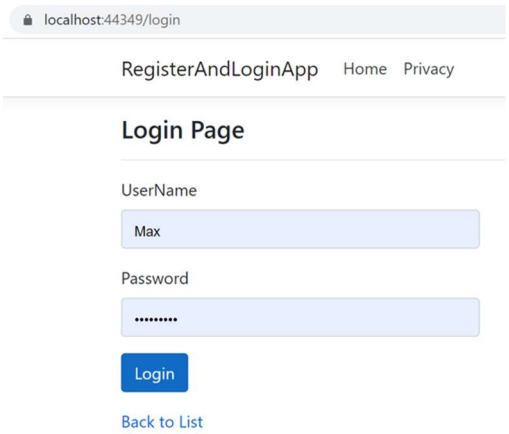


The log files will be stored in the logs folder of the app.

The log file will contain events we configure and test in this exercise.

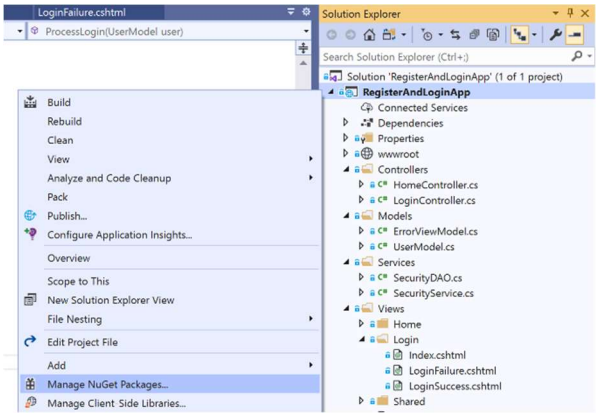
Pre-flight checklist

You will need the project open from the Registration and Login app, which was created in a previous lesson.

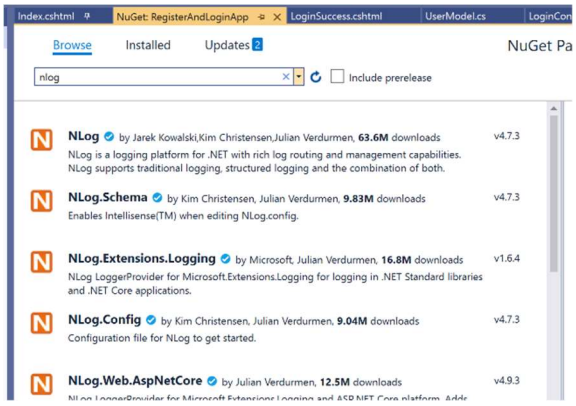


Add NLog Logging Framework the project

1. Right-click on your solution and select the **Manage NuGet Packages** menu options to bring up the NuGet Package Manager.

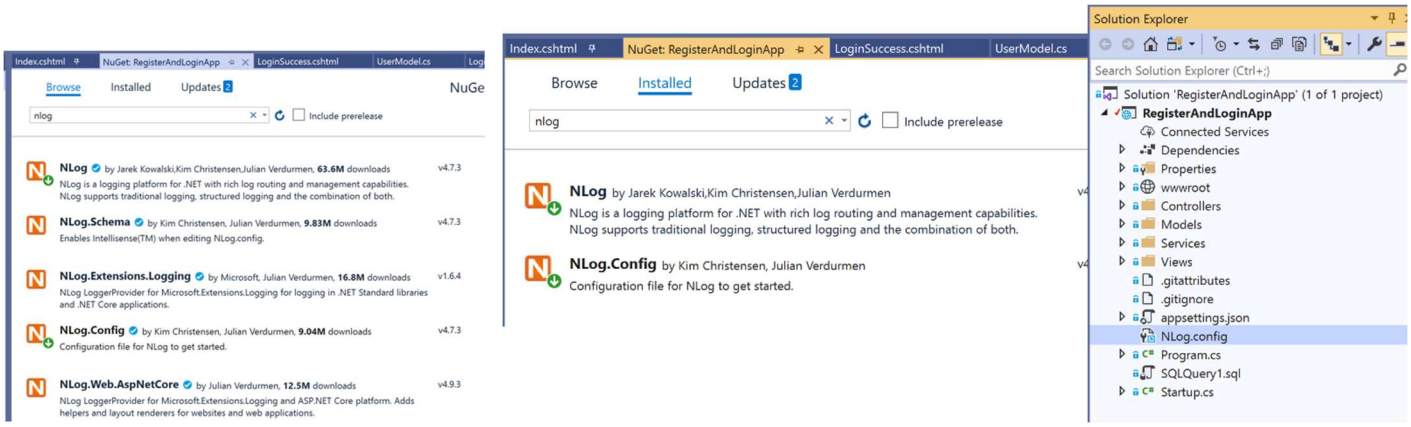


2. Select the **Browse** tab from the NuGet Package **Manager**.
3. Search for NLog.
4. Install both the **NLog** and **NLog.Config** packages.



5. Validate that the installation was successful.

   The packages should show a green arrow. The packages should be listed under the "Installed" tab. NLog.config is a new file in the root of the project.



Configure NLog:

1. Open the NLog.config file in the root of your project.
2. Setup a Logging Target. Uncomment the example and change the name.

```
25
26      <!--
27      Write events to a file with the date in the filename. -->
28      <target xsi:type="File" name="LoginAppLoggerTarget" fileName="${basedir}/logs/${shortdate}.log"
29              layout="${longdate} ${uppercase:${level}} ${message}" />
30
31    </targets>
```

Choose a descriptive name. There can be multiple targets: files, databases, or 3rd party external systems.

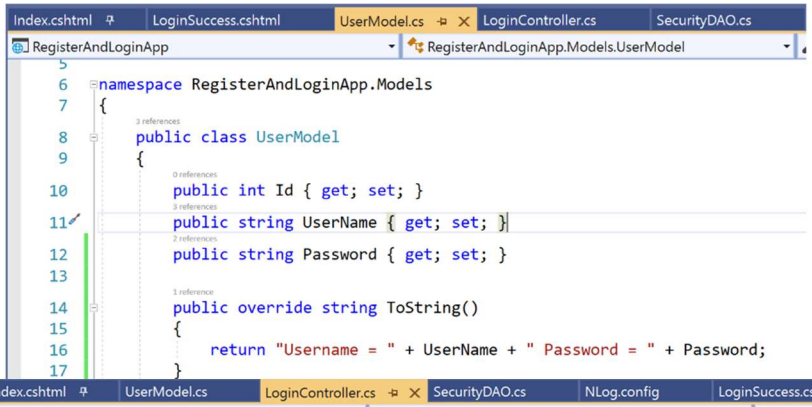
3. Setup a **Logging Rule**. Uncomment the example given and change the name to match the target.

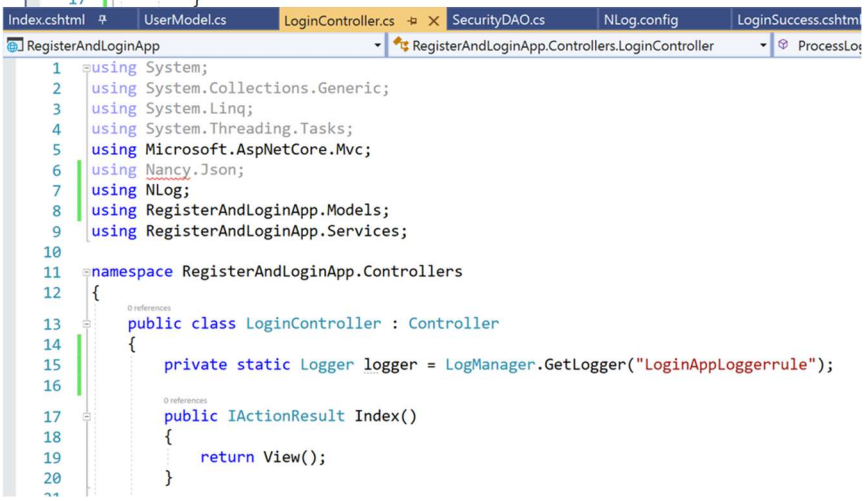
```
26      <!--
27      Write events to a file with the date in the filename. -->
28      <target xsi:type="File" name="LoginAppLoggerTarget" fileName="${basedir}/logs/${shortdate}.log"
29              layout="${longdate} ${uppercase:${level}} ${message}" />
30
31    </targets>
32
33    <rules>
34      <!-- add your logging rules here -->
35      <!--  Write all events with minimal level of Debug (So Debug, Info, Warn, Error and Fatal, but not Trace)  to
        "f" -->
36      <logger name="LoginAppLoggerrule" minlevel="Debug" writeTo="LoginAppLoggerTarget" />
37
38    </rules>
39  </nlog>
40
```

The logging rule writes to the target defined above.

4. Add a ToString method to the UserModel.

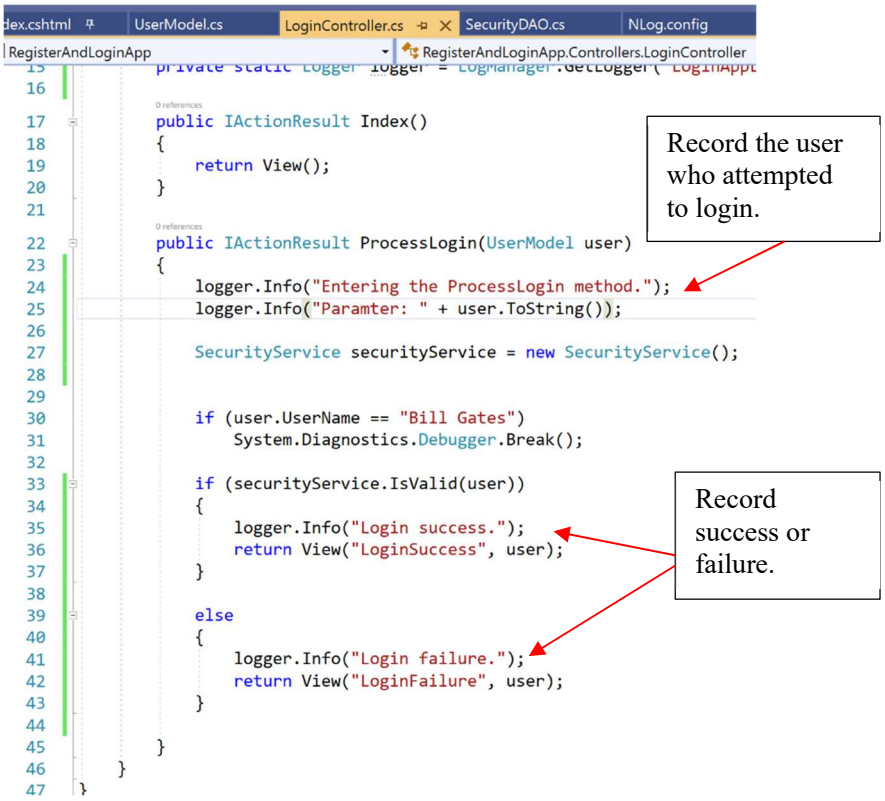
5. Add Logging to the Application using NLog:





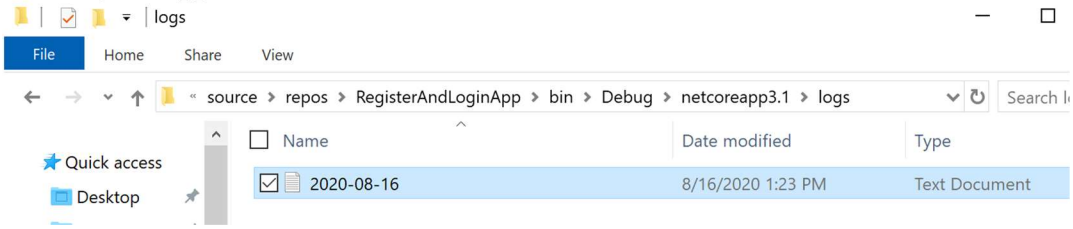
6. Add logging events to the LoginController as shown.

- Post a message whenever a login attempt is made.
- Post the name and password of the login attempt.
- Post a message for either a success or a fail login.

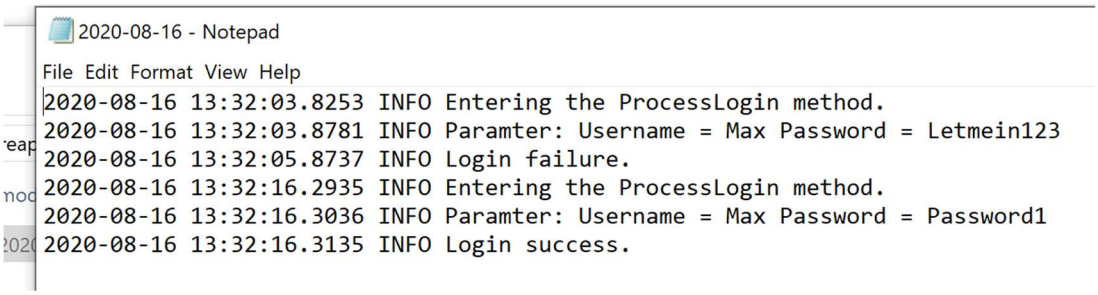


7. Run your application through a successful and a failed login.

8. Using Windows File Explorer and a standard text editor, inspect the contents of application log file located in the 'logs' directory inside the bin/Debug/netcoreapp3.1 of your application.



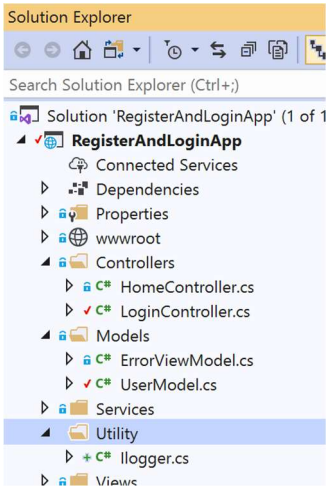
9. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

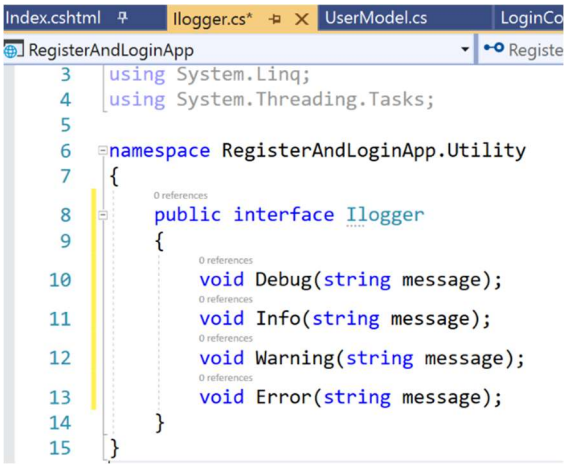


Singleton Pattern
The following code utilizes a Singleton design pattern. If Singleton is new to you, read this page: https://code-maze.com/singleton/

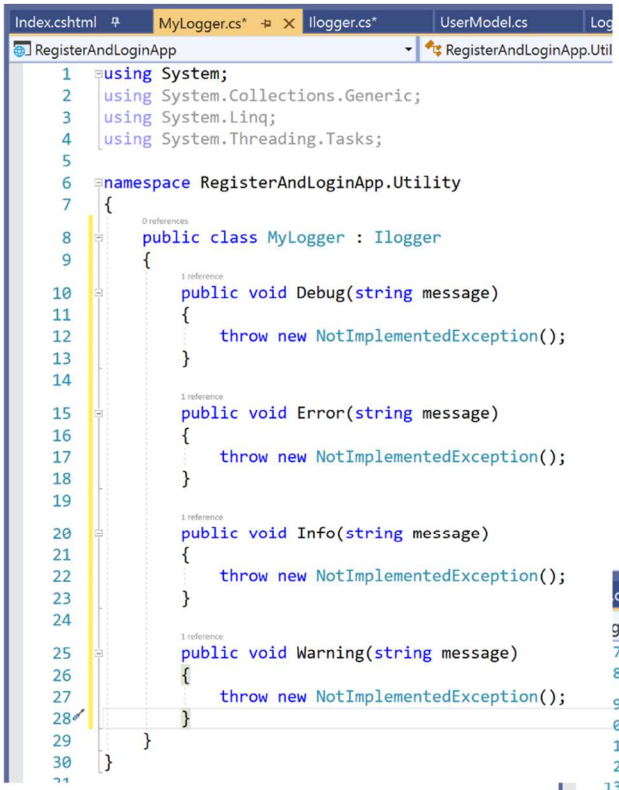
Create a Logging Facade in C#:

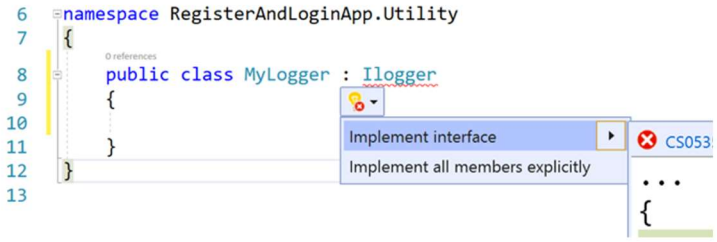
1. Create a Utility folder under the Service folder in your solution.
2. Create a new class, Ilogger, inside the folder (uppercase i).
3. Make the Ilogger class an interface that has the following four method names.





4. Create a new class, **MyLogger**, that implements the **Ilogger** interface.





Create a **MyLogger** class that:

1. Implements the Ilogger interface.
2. Uses the singleton Design Pattern:
    a. Private class constructor
    b. GetInstance() method that does lazy initialization on a private static instance of the class and returns the instance.
    c. GetLogger() that does lazy initialization on a private instance of the NLog Logger and returns the instance.

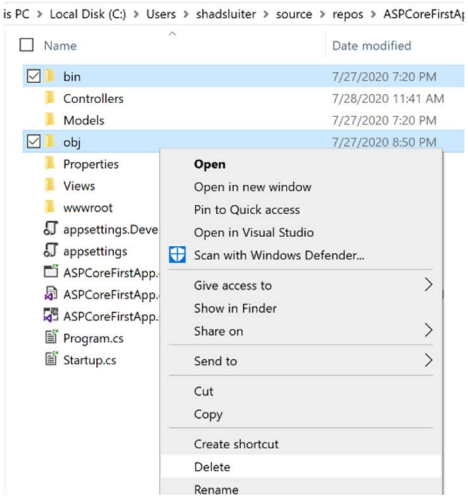
3. Refactor the **LoginController** to use the Logging Facade (ensuring that NLog is not required in the LoginController).
4. Replace the class scoped Logger variable with an instance of the new Logging Utility class. Invoking the Logger Façade can be done as follows:

```
namespace RegisterAndLoginApp.Controllers
{
    public class LoginController : Controller
    {

        public IActionResult Index()
        {
            return View();
        }

        public IActionResult ProcessLogin(UserModel user)
        {
            MyLogger.GetInstance().Info("Entering the ProcessLogin method.");
            MyLogger.GetInstance().Info("Paramter: " + user.ToString());

            SecurityService securityService = new SecurityService();


            if (user.UserName == "Bill Gates")
                System.Diagnostics.Debugger.Break();

            if (securityService.IsValid(user))
            {
                MyLogger.GetInstance().Info("Login success.");
                return View("LoginSuccess", user);
            }

            else
            {
                MyLogger.GetInstance().Info("Login failure.");
                return View("LoginFailure", user);
            }

        }
    }
}
```

5. Using Windows File Explorer and a standard text editor, inspect the contents of application log file located in the 'logs' directory within the root folder of your application.
6. Take a **screenshot** of the app showing a pass and fail login attempt. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Deliverables:

1. This activity has multiple parts. Complete all parts before submitting.
2. Submit a Microsoft Word document with screenshots of the application being run at each stage of development. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
4. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.



Part 3 Action Filters

Goal and Directions:

In this activity, you will learn how to:

- Use Action Filters to modify the behavior of a page depending on the status of a variable.

Part 1 - Filters

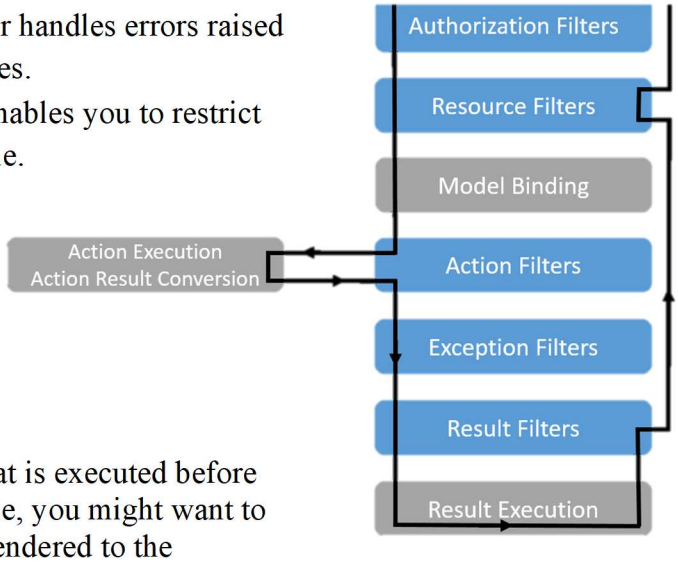
ASP.NET methods generally have a one-to-one relationship with UI controls, such as clicking a button, submitting a form, or clicking a link. However, many times we would like to perform some action before or after a particular operation. ASP.NET MVC uses **Filters** to add pre- and post-action behaviors on the controller's action methods.

Types of Filters

Every event has to pass through a gauntlet of possible filters in the course of being executed.

Action Filters − Action filters are used to implement logic that gets executed before and after a controller action executes. Action filters are one of the most commonly used filters to perform additional data processing, or manipulating the return values, cancelling the execution of action, or modifying the view structure at run time. ASP.NET MVC provides the following action filters

- Output Cache − This action filter caches the output of a controller action for a specified amount of time.
- Handle Error − This action filter handles errors raised when a controller action executes.
- Authorize − This action filter enables you to restrict access to a particular user or role.



Authorization Filters − Authorization filters are used to implement authentication and authorization for controller actions. These can be used to check for a valid login before showing a particular page.

Result Filters − Result filters contain logic that is executed before and after a view result is executed. For example, you might want to modify a view result right before the view is rendered to the browser.

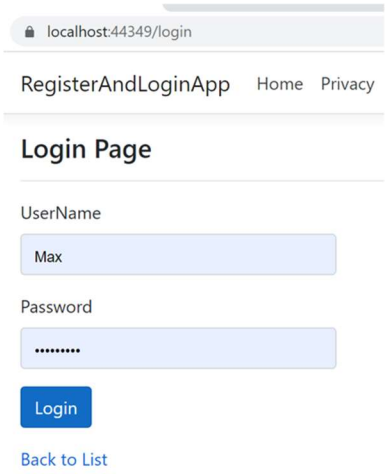
Exception Filters − Exception filters are the last type of filter to run. You can use an exception filter to handle errors raised by either your controller actions or controller action results. You can also use exception filters to log errors.

Custom Filters – You also can create your own custom action filters. For example, you might want to create a custom action filter in order to implement a **custom authentication system**, which is precisely what we will do in the following tutorial. Or, you might want to create an action filter that modifies the view data returned by a controller action.

Create Protected Methods in a Controller

You will need the code from the Registration and Login application done in a previous lesson.

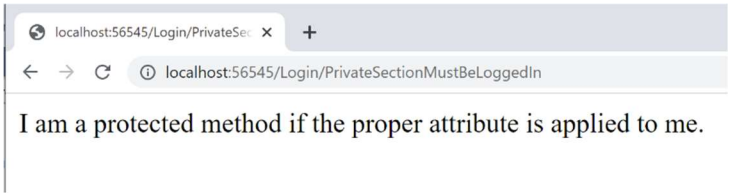
To demonstrate the use of a custom authorization filter, we will create a page (actually just a simple line of text) that is viewable only by users who have logged in.



1. In the LoginController, create a new method called **PrivateSectionMustBeLoggedIn**. This will become an action that only logged in users will be able to access. Give it an appropriate name and message similar to the example below.

```
13  public class LoginController : Controller
14  {
15
        0 references
16      public IActionResult Index()
17      {
18          return View();
19      }
20
        0 references
21      public IActionResult PrivateSectionMustBeLoggedIn()
22      {
23          return Content("I am a protected method if the proper attribute is applied to me.");
24      }
```

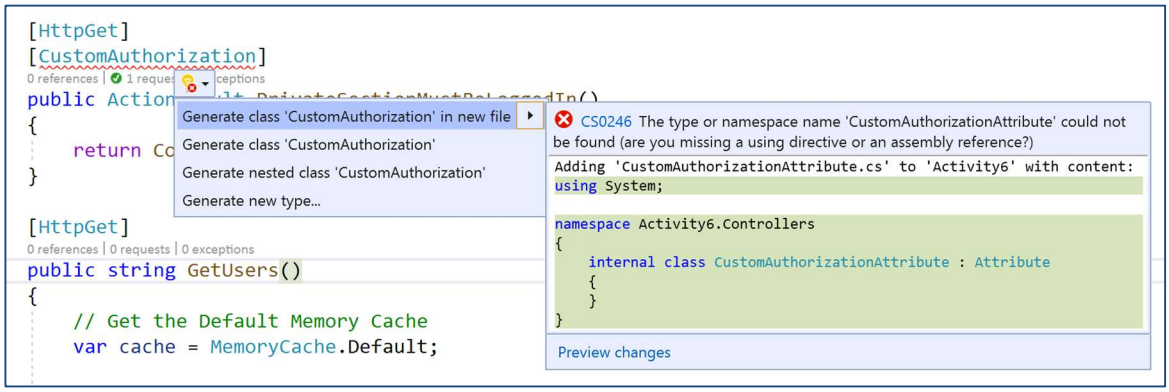
2. Run the application and you will see that the method is NOT protected yet. You should be able to access the page even without a valid login.



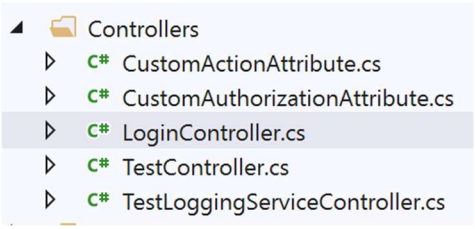
3. Now let's add a new attribute above the controller method. Call it **[CustomAuthorization].** This attribute will be defined in the next step in order to automatically reroute the user to the login page if not authenticated yet.

```
54
55          [HttpGet]
56          [CustomAuthorization]
            0 references | 1 request | 0 exceptions
57          public ActionResult PrivateSectionMustBeLoggedIn()
58          {
59              return Content("I am a protected method if the pro
60          }
61
```

4. The [**CustomAuthorization**] has not yet been defined so you should see a red line under the text. Let's click the suggestion button and choose the option to Generate the class called **CustomAuthorizationAttribute**.

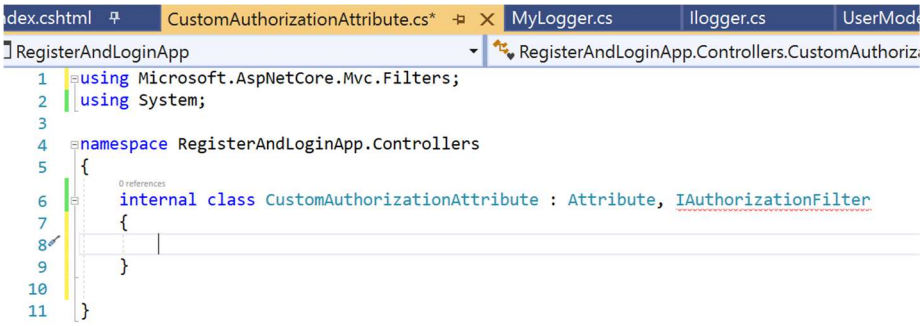


You should see the new file in the controllers folder called **CustomAuthorizationAttribute**.cs



Code the Security Authorization Filter

1. This class should implement the **IAuthorizationFilter** interface.



2. Implement the **OnAuthorization**() method that redirects to the /Login URI. You will have to import Microsoft.AspNetCore.Mvc.Filters

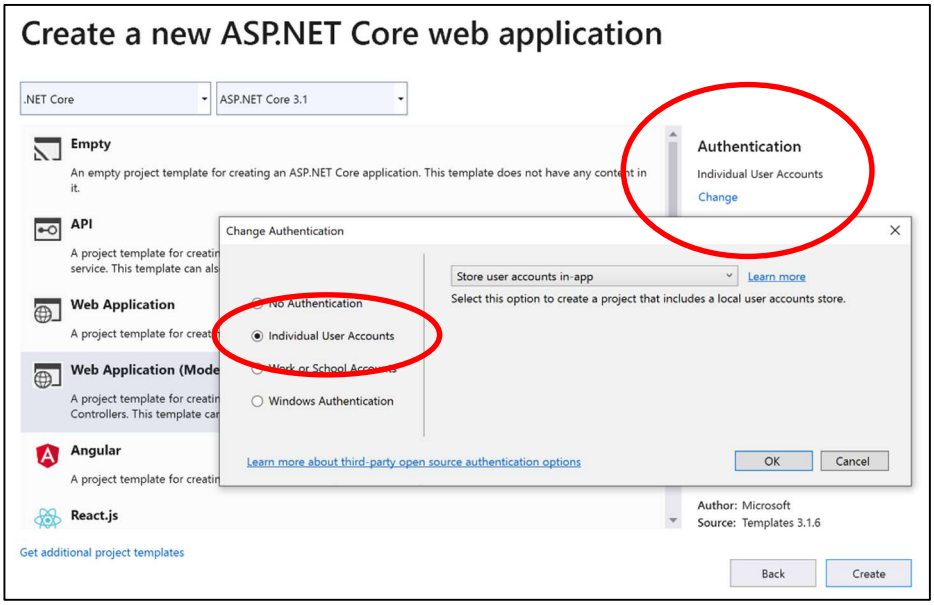
```
Index.cshtml | CustomAuthorizationAttribute.cs | MyLogger.cs | Ilogger.cs | UserModel.cs
RegisterAndLoginApp | RegisterAndLoginApp.Controllers.CustomAuthorizati
1  using Microsoft.AspNetCore.Mvc;
2  using Microsoft.AspNetCore.Mvc.Filters;
3  using System;
4
5  namespace RegisterAndLoginApp.Controllers
6  {
       0 references
7      internal class CustomAuthorizationAttribute : Attribute, IAuthorizationFilter
8      {
           0 references
9          public void OnAuthorization(AuthorizationFilterContext context)
10         {
11             context.Result = new RedirectResult("/login");
12         }
13     }
14 }
```

3. Attempt to navigate to the **/Login/PrivateSectionMustBeLoggedIn** URI. The browser should automatically reroute you to the **Login Page** since you are not yet authenticated.

Authentication Options

Authentication in ASP.NET is a big subject that is well-documented on their ASP.NET documentation site. For an automatic authentication scheme, choose an **authentication option** (Individual User Accounts) when creating the project. This will automatically create the user tables, login and registration screens, and password reset by email. It also integrates easily with Google, Facebook, and Twitter sign-on. So, what's not to like?

Choosing an authentication option automatically sets up the application to use **Entity Framework (EF)** for the database access. EF is a database mapping tool in the category of an **ORM (Object Relational Mapper)**, which is a four-letter word to some people. Entity Framework hides the details of writing SQL statements, which has benefits such as rapid development time, and also drawbacks, including



poorly-understood “black box” Data Access Objects (DAO), which can be challenging to troubleshoot when inevitable performance problems arise.

Setup Session Variables

In the case of this project, we have already implemented a user registration system, so we will continue with a different approach. One method for determining a valid login is to set a **session variable** upon successful authentication.

What is a Session?

A session is a global variable stored on the server. Each session is assigned a unique id that is used to retrieve stored values. Whenever a session is created, a cookie containing the unique session id is stored on the user’s computer and returned with every request to the server. If the client browser does not support cookies, the unique session id is displayed in the URL. Sessions have the capacity to store relatively large data compared to cookies.

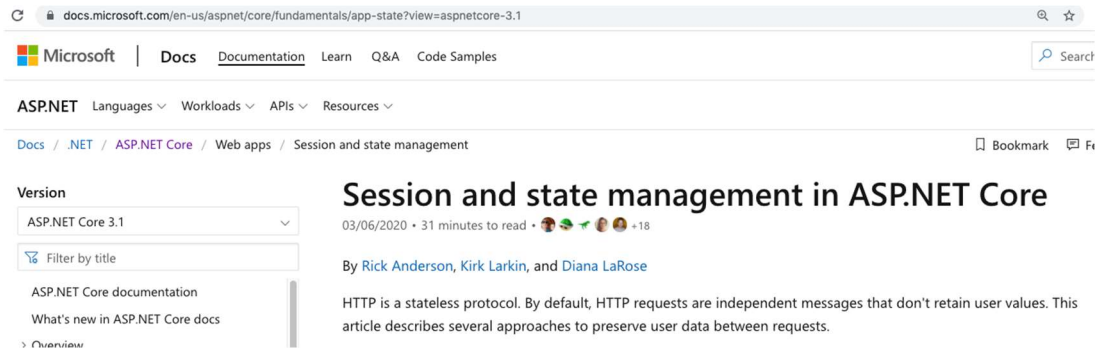
The session values are automatically deleted when the browser is closed. If you want to store the values permanently, then you should store them in the database.

Cookies vs. Sessions

Cookies are client-side files that contain user information, whereas Sessions are server-side files that contain user information.

- A cookie is not dependent on session, but Session is dependent on Cookie.
- Cookie expires depending on the lifetime you set for it, while a Session ends when a user closes his/her browser.
- The maximum cookie size is 4KB, whereas in session, you can store as much data as you like.
- Cookie does not have a function named cookie.remove(), while in Session, you can use session.remove(“keyname”) to unset some session key name. Deleting a cookie is a variation on modifying it. You cannot directly remove a cookie because the cookie is on the user's computer. However, you can have the browser delete the cookie for you. The technique is to create a new cookie with the same name as the cookie to be deleted, but to set the cookie's expiration to a date earlier than today.

In order to use session variables, there are some configuration steps to perform. This code example comes from Microsoft’s official documentation.



1. In the **Startup**.cs file, add the following lines to the **Configuration** method.

```
21          public IConfiguration Configuration { get; }
22
23          // This method gets called by the runtime. Use this method t
24          public void ConfigureServices(IServiceCollection services)
25          {
26              // new - copied from microsoft documentation
27              services.AddDistributedMemoryCache();
28
29              // new - copied from microsoft documentation
30              services.AddSession(options =>
31              {
32                  options.IdleTimeout = TimeSpan.FromSeconds(10);
33                  options.Cookie.HttpOnly = true;
34                  options.Cookie.IsEssential = true;
35              });
36
37              services.AddControllersWithViews();
38
39
40          }
41
```

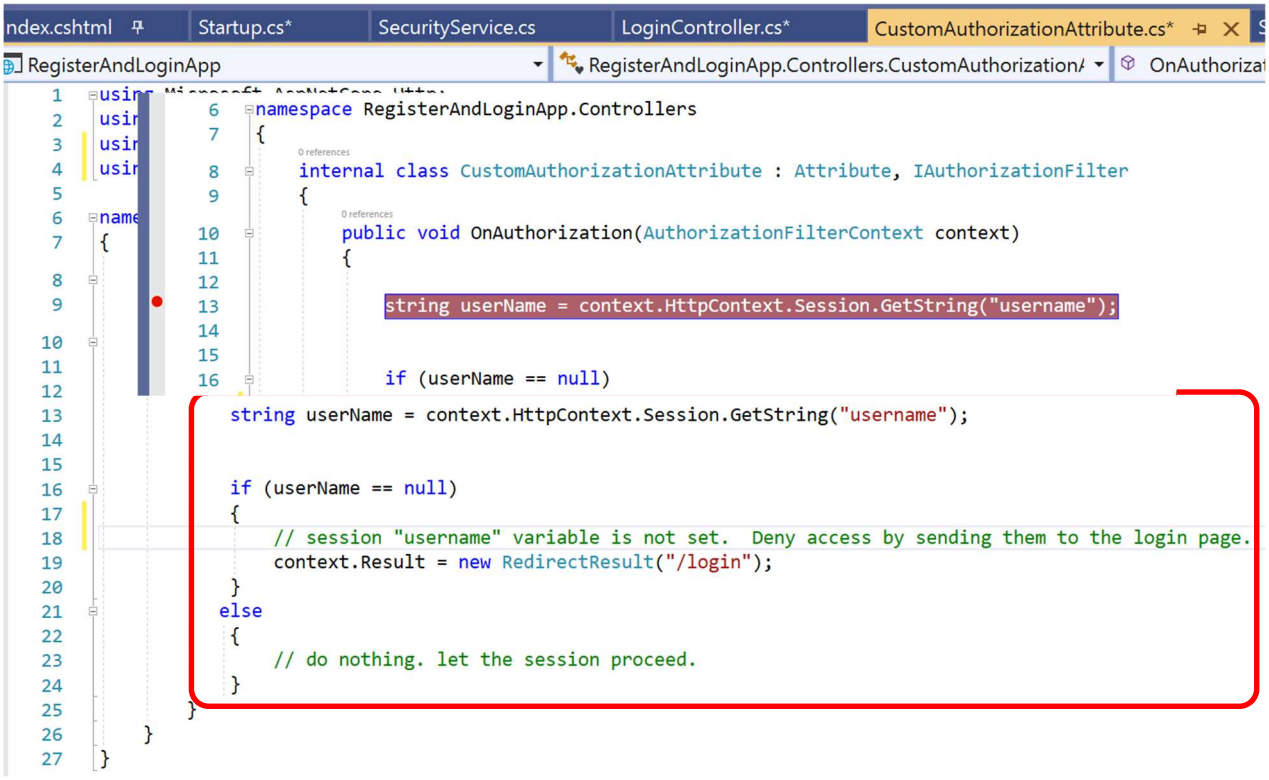
2. Also, in **the Startup.cs** file, add the following lines to the **Configure** method.

```
        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        0 references
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                // The default HSTS value is 30 days. You may want to change this for production scenarios,
                  hsts.
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            // new - copied from microsoft documentation
            app.UseSession();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
```

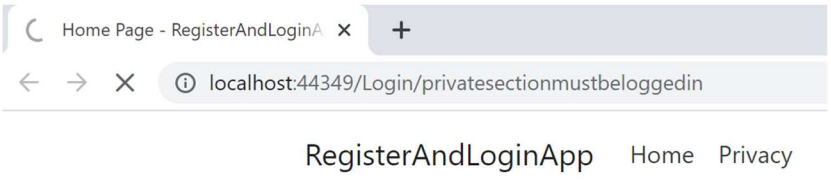
4. In the **LoginController**, set a new session variable called “username” when a user logs in successfully. Also cancel any existing session value if the login fails. You will have to include the http library when using **SetString**: Microsoft.AspNetCore.Http

```
            if (securityService.IsValid(user))
            {
                MyLogger.GetInstance().Info("Login success.");

                // remember who is logged in
                HttpContext.Session.SetString("username", user.UserName);

                MyLogger.GetInstance().Info("Leaving the ProcessLogin method.");
                return View("LoginSuccess", user);
            }

            else
            {
                MyLogger.GetInstance().Info("Login failure.");
                MyLogger.GetInstance().Info("Leaving the ProcessLogin method.");

                // cancel any existing valid login
                HttpContext.Session.Remove("username");
                return View("LoginFailure", user);
            }

        }
```

5. In the CustomAuthorizationAttribute file, check the value of the session “username” to see if the login has occurred or not.

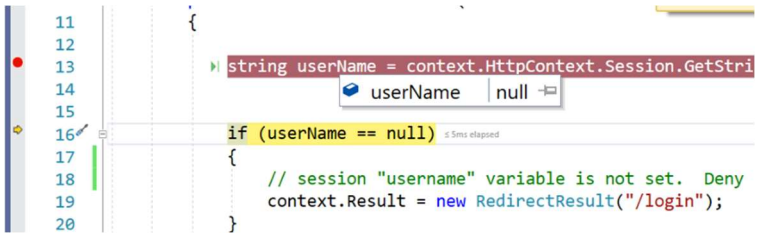


6. Set a breakpoint at the username variable (line 13). We will inspect this value when running the program.
7. Run the application and navigate to the protected method of the controller.



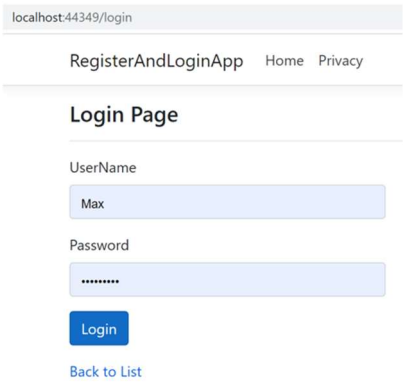
Execution of the application halts when it encounters the breakpoint.

8. Examine the value of **username** in the debugger screen. Step over one line. You should see that the value of username is null.



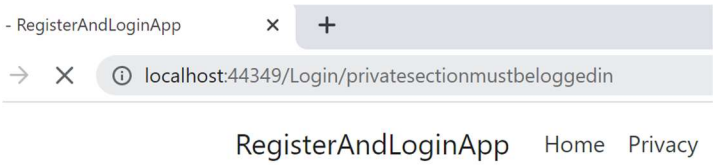
9. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

10. Click the Continue button.



The app should redirect you to the login page since username was null.

11. Login with valid name and password.
12. Go to the protected URL again.



The debugger halts the program again.

13. Inspect the value of the **username** variable. Step over one line if necessary. The value should now be set to the username that you logged in with.

14. Take a screenshot of the app at this stage. Paste it into a Word document and caption the image with a brief explanation of what you just demonstrated.

15. Click the Continue button.

The app should allow you to view the protected content since the session “username” contains a value.

Another example for Logging

In the following instructions, we will create an action filter to perform logging events. This will remove clutter code from the method and make the filter reusable for other methods in our program.

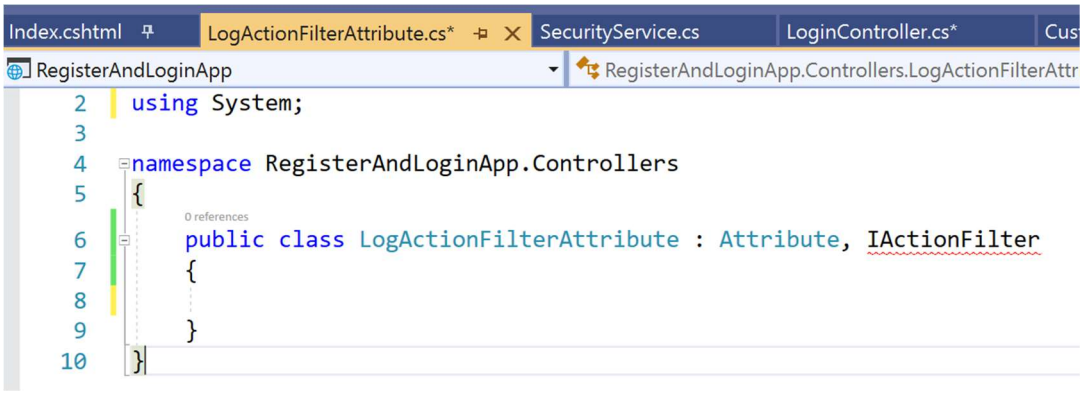
1. In the **LoginController**, add the following [action filter] attribute to the **Login** method. Remove the **MyLogger** statements indicated.

```
30          [LogActionFilter]
31          public IActionResult ProcessLogin(UserModel user)
32          {
33
34              SecurityService securityService = new SecurityService();
35
36              //MyLogger.GetInstance().Info("Entering the ProcessLogin method.");
37              //MyLogger.GetInstance().Info("Paramter: " + user.ToString());
38
39              if (user.UserName == "Bill Gates")
40                  System.Diagnostics.Debugger.Break();
41
42              if (securityService.IsValid(user))
43              {
44                  MyLogger.GetInstance().Info("Login success.");
45
46                  // remember who is logged in
47                  HttpContext.Session.SetString("username", user.UserName);
48
49                  //MyLogger.GetInstance().Info("Leaving the ProcessLogin method.");
50                  return View("LoginSuccess", user);
51              }
52
53              else
54              {
55                  MyLogger.GetInstance().Info("Login failure.");
56
57                  //MyLogger.GetInstance().Info("Leaving the ProcessLogin method.");
58
59                  // cancel any existing valid login
60                  HttpContext.Session.Remove("username");
61                  return View("LoginFailure", user);
62              }
63
64          }
65      }
66  }
```

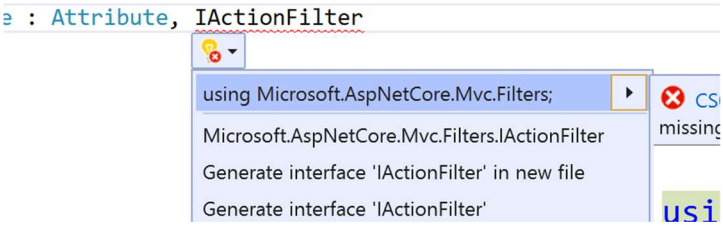
New Action Filter

Comment or delete these logger statements.

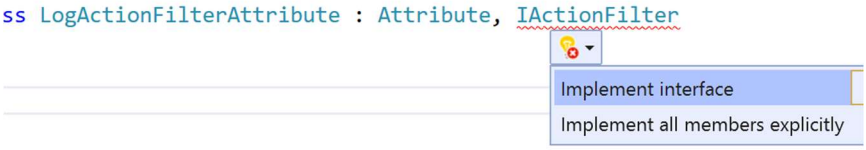
2. To accommodate the new LogActionFilter attribute, the compiler will suggest you create a new file called LogActionFilterAttribute. Create this in a new file.
3. Add the IActionFilter interface to the class.



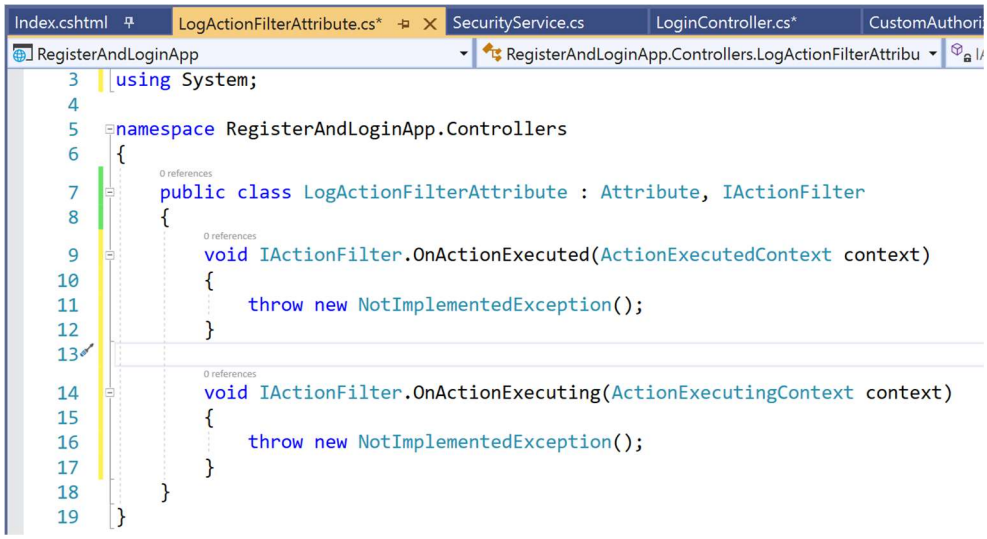
4. Import the library for MMV Filters

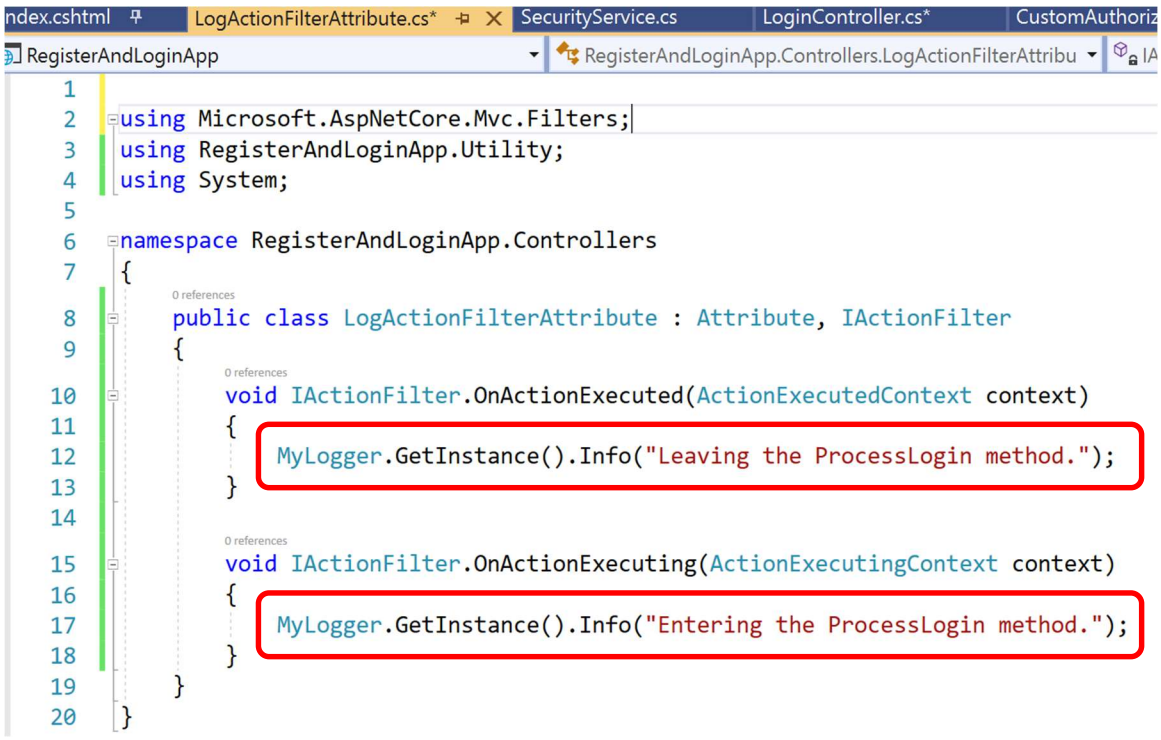


5. Implement the methods for IActionFilter.



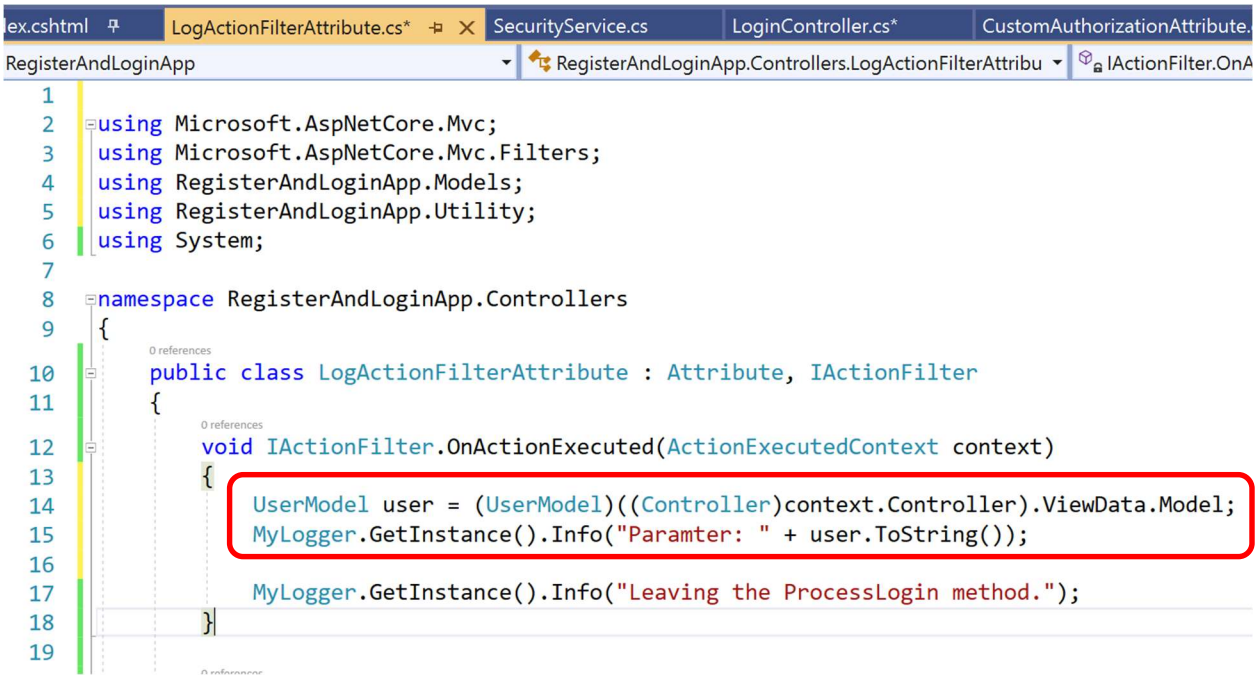
You should have two new methods, one for when the method is finished, the other for when the method is just starting.



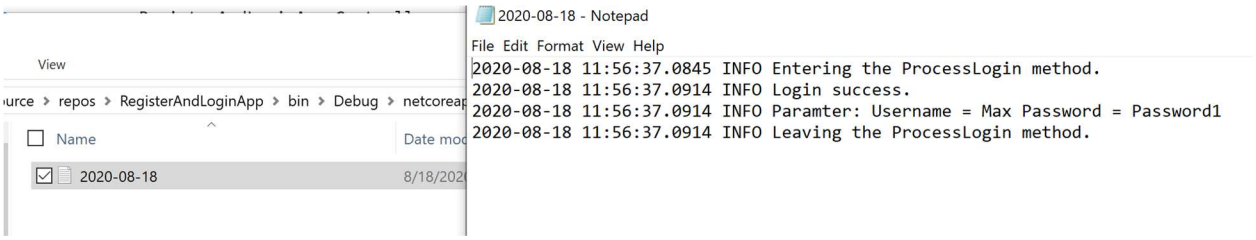


6. Add some logging events to these methods.

7. Add the ability to log the contents of the user variable (i.e., the model) of the Login event. The model is stored in the context variable. To get to the model, there is some tricky casting to be done to get there.



8. Run the application and verify that the logging events are still occurring.



9. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Deliverables:

1. Submit a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
2. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
3. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.