**CST-247 Activity Guide**

Contents

[Activity 1 2](#_Toc26852658)

[Part 1: Tools Installation and Validation 2](#_Toc26852659)

[Part 2: Introduction to .NET MVC Controllers and Views 4](#_Toc26852660)

[Part 3: Advanced .NET MVC Models, Controllers, Views 8](#_Toc26852661)

[Activity 2 13](#_Toc26852662)

[Part 1: Introduction to .NET Razor 13](#_Toc26852663)

[Part 2: Advanced .NET Razor 17](#_Toc26852664)

[Part 3: Data Validation 21](#_Toc26852665)

[Part 4: Practicing Agile Scrum 22](#_Toc26852666)

[Activity 3 23](#_Toc26852667)

[Part 1: AJAX, Partial Views, and Partial Page Updates 23](#_Toc26852668)

[Part 2: Visual Studio Debugger 25](#_Toc26852669)

[Activity 4 28](#_Toc26852670)

[Part 1: Logging with nLog Framework 28](#_Toc26852671)

[Part 2: REST Services using WCF 30](#_Toc26852672)

[Part 3: Action Filters 36](#_Toc26852673)

[Activity 5 39](#_Toc26852674)

[Part 1: IoC using Unity 39](#_Toc26852675)

[Part 2: Application of Action Filters, IoC, and Logging 41](#_Toc26852676)

# Activity 1

This activity has multiple parts/assignments. All assignments must be completed prior to documentation submission.

## Part 1: Tools Installation and Validation

**Overview**

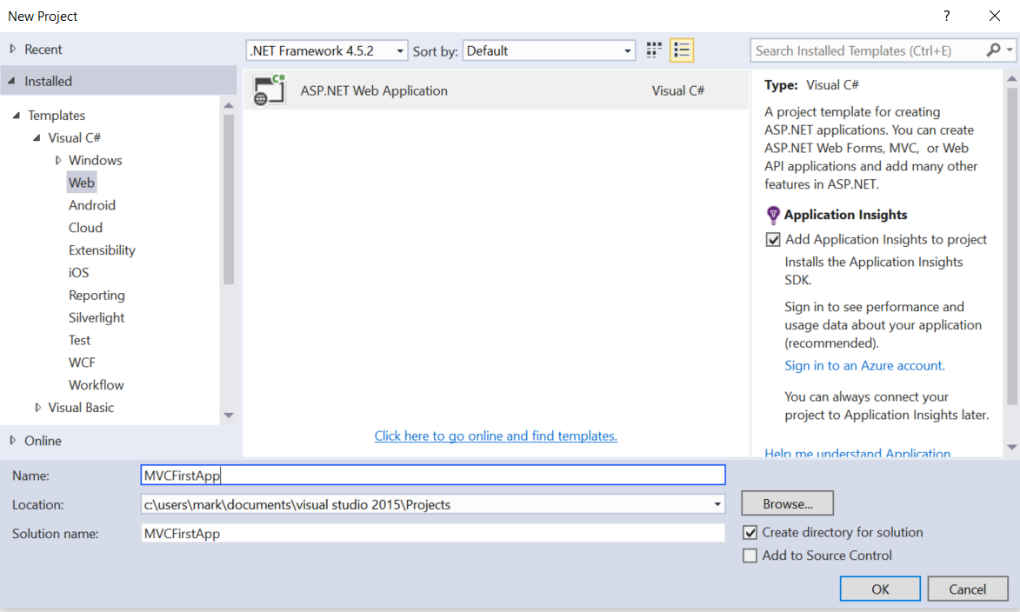
In this activity, students will create a default .NET MVC application in Visual Studio to validate their environment and also become familiar with the .NET MVC project structure.

**Execution**

Execute this activity according to the following guidelines:

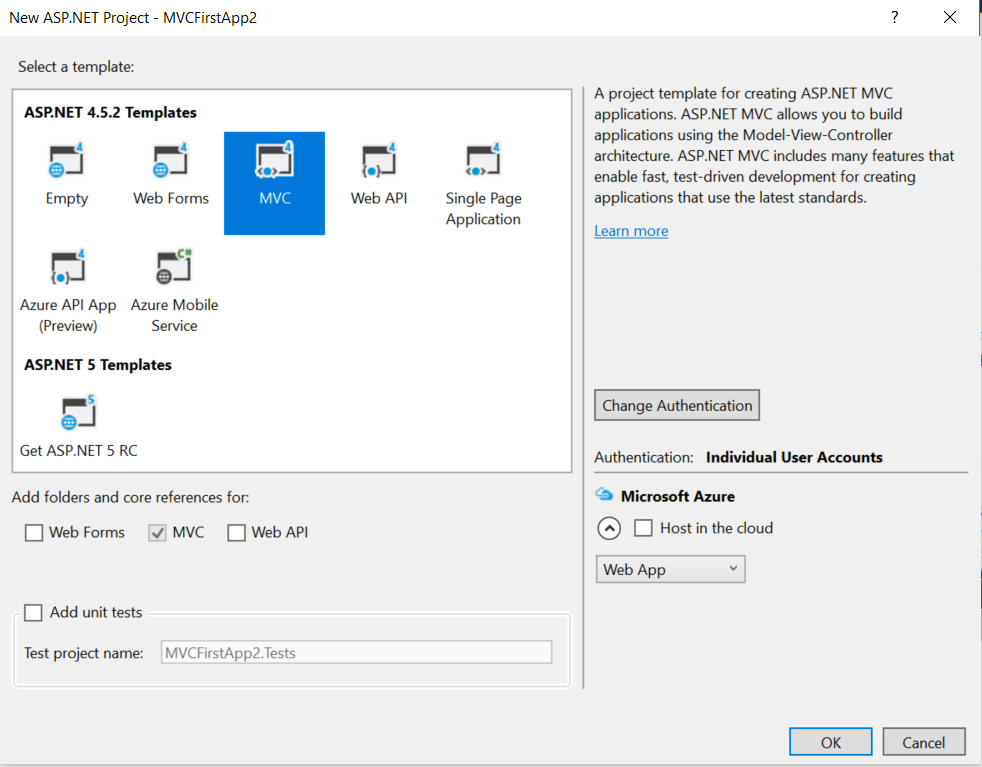
1. Make sure your Visual Studio environment is installed properly:
   1. Go to <https://www.visualstudio.com/downloads/>
   2. Download and install the latest version of the Community Edition
   3. NOTE: If you are running Parallels on a Mac, you will need to disable all the sharing features.
   4. Run the installer. Select Custom and install the Microsoft Web Developer Tools as well as the Microsoft SQL Server Data Tools.
2. Create a default .NET MVC application:
   1. Click File -> New Project
   2. From the left pane, select Templates -> Visual C# -> Web.
   3. In the middle pane, select ASP.NET Web Application.
   4. Enter the project name, MVCFirstApp, in the Name field, and click OK to continue. You will see the following dialog, which asks you to set the initial content for the ASP.NET project (see Figure 1).

**Figure 1**



* 1. To keep things simple, select the MVC template, and check the MVC checkbox in the Add folders and core references section. Click OK to create a basic MVC project with default predefined content. Once the project is created by Visual Studio, you will see a number of files and folders displayed in the Solution Explorer window (see Figure 2).

**Figure 2**



1. Run the application to ensure your environment is working properly:
   1. Inspect the project structure and some of the code that was generated.
   2. You should become familiar with this project type because your team project should be based on this project type.
   3. Resolve the Controllers and View for the Home, About, Contact, and Register menu items. Use breakpoints and the debugger to validate the code paths that handle these menu items.
2. Complete the ASP.NET MVC Tutorials as indicated in the Topic Materials.

**Observations and Notables**

1. A file ‘RouterConfig.cs’ located in the App\_Start folder is used to configure your Controller Routes.
2. All Controllers are located in a folder called Controllers. By convention, the framework follows a Controller naming convention of [Controller Name]Controller and uses a URI within the URL to resolve to the proper Controller. The method called within the Controller is also resolved via the URI with the method Index() used as a default for the root URI.
3. All Views are located in a folder called Views.
4. All Models are located in a folder called Models. In general, all Models are simple C# classes that contain nothing but a set of properties.
5. The file ‘ViewStart.cshtml’ located in the root of the project controls the Layout for each page, and is called by the framework as a default view prior to rendering all Views. This page often is used to reference a common/shared partial View that contains common JavaScript files, such as Bootstrap, and CSS files to control a common application theme. Within a partial View, the tag @RenderBody() is used as a placeholder where the desired View content is rendered.

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Create a project report using a GCU standard Project Header/Cover Page to include a header, your name, course, assignment name, and date.
2. After creating the default .NET MVC project, run the project, and attach a screenshot of your working application in your project report. Label this Screenshot 1.
3. After finishing the ASP.NET MVC Tutorials, identify six things you learned and document them in your project report with details and examples. Provide references and citations.

## Part 2: Introduction to .NET MVC Controllers and Views

**Overview**

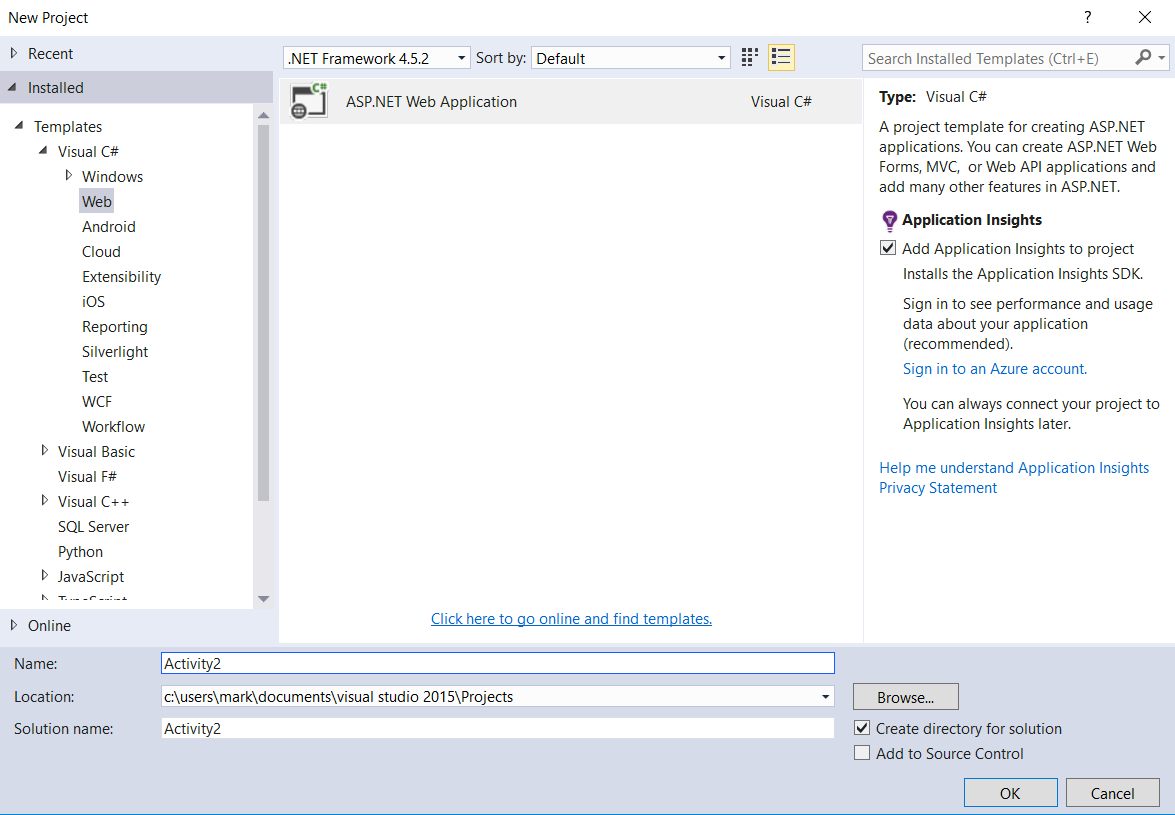
In this activity, students will create an empty .NET MVC application in Visual Studio and use this project as a basis to learn how to create Controllers and basic Views.

**Execution**

Execute this activity according to the following guidelines:

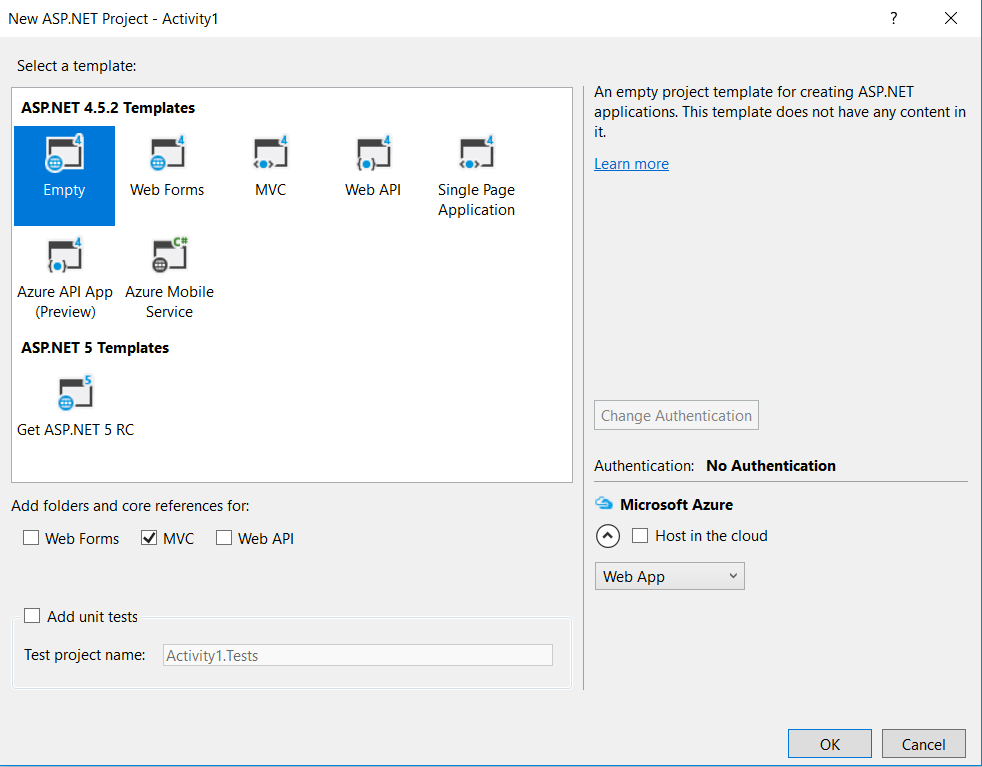
1. Create an empty .NET MVC application:
   1. Click File -> New Project
   2. From the left pane, select Templates -> Visual C# -> Web.
   3. In the middle pane, select ASP.NET Web Application.
   4. Enter the project name, Activity1Part2, in the Name field, and click OK to continue. You will see the following dialog, which asks you to set the initial content for the ASP.NET project (see Figure 1).

**Figure 1**



* 1. To keep things simple, select the Empty template, and check the MVC checkbox in the Add folders and core references section. Click OK to create an empty MVC project with default predefined content. Once the project is created by Visual Studio, you will see a number of files and folders displayed in the Solution Explorer window (see Figure 2).

**Figure 2**



1. Create an empty Test Controller and Route:
   1. Right click on the Controllers Folder.
   2. Select the Add->Controller… menu items. Then select the ‘MVC 5 Controller – Empty’ scaffolding type. Click the Add button. Name your Controller ‘Test,’ and click the Add button. Inspect the Test Controller (as TestController.cs class) within the Controllers folder.
   3. Add a Route for the Test Controller by updating the RouteConfig.cs class located with the App\_Start folder.

routes.MapRoute(

name: "Test",

url: "{Test}",

defaults: new { controller = "Test", action = "Index", id UrlParameter.Optional }

);

* 1. Update the method body of the Index() method of the Test Controller:

return @"<b>Hello World as a string from Index</b>";

* 1. Validate the Project and Test Controller:
     1. Select the Debug->Start Debugging menu options.
     2. You should first get a 404 error for the default localhost:[port] URL.
     3. Enter localhost:[port]/Test as the URL and validate that the Hello World string entered in Step 2d is displayed. The /Test is the URI that maps to the Test Controller and the root Index() method.
     4. Take a screenshot. Label this Screenshot 2.
  2. Add a new Play Action to the Test Controller:
     1. Open the Test Controller class.
     2. Add a Play action:

// GET: /Test/Play

[HttpGet]

public string Play()

{

// Return a string as a View

return @"<b>Hello World as a string from Play</b>";

}

* + 1. Run the Application and access the Play action by using the localhost:[port]/Test/Play as the URL. The Play URI maps to the Play action.
    2. Take a screenshot. Label this Screenshot 3.

1. Create a basic View for the Test Controller.
   1. Add a new Test View Action to the Test Controller:

// GET: /Test/TestView

[HttpGet]

public ActionResult TestView()

{

// Return a 'TestView.cshtml' (View maps to Action method name)

return View();

}

* 1. Click the mouse in a code statement within the body of the Test View Action.
  2. Select the Add View… menu item. Leave the defaults as TestView for the View Name, use an Empty model Template, and without the Partial View and Use Layout options. Click the Add button.
  3. Validate that the Test View (as TestView.cshtml file within the Views/Test folder was created.
  4. Update the Test View body:

<div>

Hello World from my Test View

</div>

* 1. Run the Application and access the Play action by using the localhost:[port]/Test/TestView as the URL. The TestView URI maps to the Play action.
  2. Take a screenshot. Label this Screenshot 4.

1. Complete the ASP.NET MVC Tutorials as indicated in the topic materials.

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Updates to the project report created in Part 1 with all applicable screenshots.
2. After creating the default .NET MVC Project, run the project and attach a screenshot of your working application in your project report. Label this Screenshot 5.
3. After finishing the ASP.NET MVC Tutorials identify six things you learned, and document them in your project report with details and examples. Provide references and citations.

## Part 3: Advanced .NET MVC Models, Controllers, Views

**Overview**

In this activity, students will use .NET MVC Routes, Controllers, Models, and Views to build a simple Login Page using a N-Layer Web Application Architecture.

**Execution**

Execute this assignment according to the following guidelines:

1. Create a new .NET MVC Application:
   1. Select File->New Project menu items. Under the Visual C# Windows templates select Web. From the listed templates select the ASP.NET Web Application template. Name your application Activity1Part3. Click OK.
   2. From the ASP.Net Template dialog select the Empty template, check the MVC option under folders and core references, and uncheck the Host in the Cloud Azure option. Click OK.
2. Create a Login Controller and Route:
   1. Right click on the Controllers Folder.
   2. Select the Add->Controller… menu items. Then select the ‘MVC 5 Controller – Empty’ scaffolding type. Click the Add button. Name your Controller ‘Login’ and click the Add button. Inspect the Login Controller (as LoginController.cs class) within the Controllers folder.
   3. Add a Route for the Test Controller by updating the RouteConfig.cs class located with the App\_Start folder.

routes.MapRoute(

name: "Login",

url: "{Login}",

defaults: new { controller = "Login", action = "Index", id UrlParameter.Optional }

);

* 1. Update the method body of the Index() method of the Login Controller:
     1. Add an [HttpGet] attribute above the Index() method.
     2. Code body: return @"<b>Just a test from Index</b>";
  2. Validate the Project and Login Controller:
     1. Select the Debug->Start Debugging menu options.
     2. You should first get a 404 error for the default localhost:[port] URL.
     3. Enter localhost:[port]/Login as the URL and validate that the Test string entered in Step 2d is displayed. The /Login is the URI that maps to the Login Controller and the root Index() method.

1. Create a User Model class:
   1. Click the Stop Debugging button.
   2. Right click on the Model Folder.
   3. Create a Model C# class by selecting Add New Item menu. Then under the Add New Item dialog select Code under Visual C#, select Class, and enter UserModel for the name. Click Add.
   4. Add two properties—Username and Password—to the UserModel class with both setter and getter methods:
      1. public string Username { get; set; }
      2. public string Password { get; set; }
2. Create a Login View:
   1. Right click on any part of the code body for the Login Controller Index(). Select the Add View menu option. Change the View name to Login, select the Create template, and from the Model class dropdown select the UserModel created in Step 3. Click Add. Inspect the Login Page (as Login.cshtml) within the Views/Login folder.
   2. Update the Login Controller Index() to return an ActionResult, remove the test implementation, and change the code to: return View(“Login”).
   3. Validate the Login Page by running the solution using localhost:[port]/Login as the URL.
3. Bind the Login View and Login Controller with a Controller method:
   1. Click the Stop Debugging button.
   2. Update the Login Controller by adding a Login() method to the controller that handles the Login View Form Post.

[HttpPost]

public String Login(UserModel model)

{

return “Here “ + model.Username + “ and “ model.Password;

}

* 1. Update the Login View to post the form to the Login Controller’s Login() method:

@using (Html-BeginForm(“Login”, “Login”, FormMethod.Post))

* 1. Validate the Login Page by running the solution using localhost:[port]/Login as the URL.

1. Create a Security Data Service class:
   1. Click the Stop Debugging button.
   2. Right click on the Activity1Part3 project name.
   3. Select Add->New Folder menu items. Name the folder Services.
   4. Right click on the Services folder and select Add->New Folder menu items. Name the folder Business.
   5. Right click on the Services folder and select Add->New Folder menu items. Name the folder Data.
   6. Right click on the Services\Data folder and select Add->New Item. Then under the Add New Item dialog select Code under Visual C#, select Class, and enter SecurityDAO for the name. Click Add.
   7. Add a FindByUser() method to the SecurityDAO class.

public bool FindByUser(UserModel user)

{

return true;

}

1. Create a Security Business Service class:
   1. Right click on the Services\Business folder and select Add->New Item. Then under the Add New Item dialog select Code under Visual C#, select Class, and enter SecurityService for the name. Click Add.
   2. Add a Authenticate() method to the SecurityService class.

public bool Authenticate(UserModel user)

{

SecurityDAO service = new SecurityDAO();

return service.FindByUser(user);

}

1. Integrate Security Business Service with Login Controller:
   1. Remove the existing test code from the Login() method of the Login Controller.
   2. Change the return type to ActionResult for the Login() method.
   3. Call the Security Business Service Authenticate() method from the Login() method and save the results of the method call in a local method variable.
   4. Add a conditional statement from the Boolean returned from the Authenticate method to forward to either a View named LoginPassed if the return value is true else forward to a View named LoginFailed if the return value is false. In the LoginPassed View, also pass the model to this View.
   5. Create a pass and fail Login Views:
      1. Right click on the Views\Login folder.
      2. Select Add->View menu items. Select LoginPassed as the View name using an Empty (without model) template. Click the Add button. Add a “login passed” message to the body content of the page.
      3. Select Add->View menu items. Select LoginFailed as the View name using an Empty (without model) template. Click the Add button. Add a “login failed” message to the body content of the page.
   6. Test your Login View from end to end:
      1. Validate the Login Page by running the solution using localhost:[port]/Login as the URL. You should see a Login Passed View displayed. Take a screenshot. Label this Screenshot 6
      2. Change the return value of the FindByUser() method in the SecurityDAO class to false.
      3. Validate the Login Page by running the solution using localhost:[port]/Login as the URL. You should see a Login Failed View displayed. Take a screenshot. Label this Screenshot 7
2. Create a Database Schema and Table:
   1. Open SQL Server Object Explorer by selecting the View->SQL Server Object Explorer menu items. By default, under the SQL Servers, you should see a local DB called MSSQLLocalDB listed. Expand this database server. If this server is not listed:
      1. Click the Add SQL Server icon.
      2. Expand the Local tree.
      3. Select the MSSQLLocalDB.
      4. Select Windows Authentication.
      5. Click the Connect button to add the local database server to the list of database servers in SQL Server Object Explorer.
   2. Right click on the Databases folder and select Add New Database. Enter a Database Name of Test. A new database should be now available under the MSSQLLocalDB databases.
   3. Expand the Test database to display the Tables folder.
   4. Right click on the Tables folder and select Add New table menu option. This will display the Table designer.
   5. In the T-SQL window, change the name of the Table in the Create statement to Users.
   6. Add the following columns:
      1. ID, primary key, type int, no NULL’s, with auto increment (in the T-SQL window add IDENTITY(1,1) to the script.
      2. USERNAME, type nvarchar(50), no NULL’s
      3. PASSWORD, type nvarchar(50), no NULL’s
   7. Click the Update button from the designer. In the Preview Database Updates dialog, click the Update Database button.
   8. In the Output window, validate that the table and columns were created without errors.
   9. Validate your Test Database:
      1. Open SQL Server Object Explorer by selecting the View->SQL Server Object Explorer menu items. By default, under the SQL Servers, you should see a local DB called MSSQLLocalDB listed.
         1. If this database is NOT listed, click the ‘Add SQL Server’ icon.
         2. Expand the Local tree.
         3. Select the MSSQLLocalDB.
         4. Select Windows Authentication.
         5. Click the Connect button to add the local database to the list of databases in SQL Server Object Explorer.
      2. Expand the MSSQLLocalDB database server and under the Databases folder, select the Test database.
      3. Right click on the Test database, and select the New Query menu option.
      4. In the query run, execute the following SQL Statement:

SELECT \* FROM dbo.Users

* + 1. The Message window should display the three columns of the Users table with no data.
    2. Insert data into the Users table by executing the following SQL Statement:

INSERT INTO dbo.Users (USERNAME, PASSWORD) VALUES (‘test’, ‘test’)

* 1. Update the Security DAO using ADO.NET:
     1. Open the SecurityDAO class, and make the following modifications to the FindByUser() method.
        1. Create a SqlConnection using the (localdb)\\MSSQLLocalDB database, the Test database, and using your Windows authentication credentials.
        2. Create a SqlCommand using a SELECT query on the Users table and the SqlConnection instance.
        3. Open the connection returned from the SqlConnection instance.
        4. Call the ExecuteRead() on the SqlCommand instance.
        5. Return true if there were rows returned from reader instance.
        6. Return false if there were no rows returned from the reader instance.
        7. See posted code snippet for complete code implementation as well as the SQL Server and T-SQL resources listed below.
  2. Test the Controller and Views with the browser by navigating to /Login.
  3. Take a screenshot of the login results for a user that exists in the database and a user that does not exist in the database. Label these Screenshots 8 and 9.

**Submission**

Submit the following to the learning management system:

1. Project report that includes:
   1. A GCU standard project header/cover page to include your name, course, assignment name, and date
   2. Screenshots 1-9
   3. Documentation of the things you learned from the ASP.NET MVC tutorials providing details, examples, references, and citations.
   4. URL of your GIT repository
2. Upload your code to the GIT repository.

# Activity 2

This activity has multiple parts/assignments. All assignments must be completed prior to documentation submission.

## Part 1: Introduction to .NET Razor

**Overview**

In this activity, students will use .NET MVC Views to learn some of the Razor Syntax and Razor Helper methods, display a list of C# object models in a standard HTML table, and submit Image Buttons to a Controller.

**Execution**

Execute this activity according to the following guidelines:

*Section 1*

1. Create a new .NET MVC Application:
   1. Select File->New Project menu items. Under the Visual C# Windows templates select Web. From the listed templates select the ASP.NET Web Application template. Name your application Activity2Part1. Click OK.
   2. From the ASP.Net Template dialog, select the Empty template, check the MVC option under folders and core references, and uncheck the Host in the Cloud Azure option. Click OK.
2. Create a Test Controller and Route to test the Razor View:
   1. Right click on the Controllers Folder.
   2. Select the Add->Controller… menu items. Then select the ‘MVC 5 Controller – Empty’ scaffolding type. Click the Add button. Name your Controller ‘Test’ and click the Add button. Inspect the Test Controller (as TestController.cs class) within the Controllers folder.
   3. Add a Route for the Test Controller by updating the RouteConfig.cs class located with the App\_Start folder.

routes.MapRoute(

name: "Test",

url: "{Test}",

defaults: new { controller = "Test", action = "Index", id= UrlParameter.Optional }

);

1. Create a User Model class (to test our Razor View):
   1. Right click on the Models Folder.
   2. Create a Model C# class by selecting Add New Item menu. Then, under the Add New Item dialog, select Code under Visual C#, select Class, and enter UserModel for the name. Click Add.
   3. Add three properties to the UserModel class with both setter and getter methods: Name, Email Address, and Phone Number. Also add a nondefault constructor that will initialize the three properties.
      1. public UserModel(string Name, string Email, string Phone) { };
      2. public string Name { get; set; }
      3. public string EmailAddress { get; set; }
      4. public string PhoneNumber { get; set; }
2. Create a Test View:
   1. Right click on any part of the code body for the Test Controller. Select the Add View menu option. Change the View name to Test, select the Empty template, and from the Model class dropdown select the UserModel created in Step 3. Uncheck the Use Layout option. Click Add. Inspect the Test Page (as Test.cshtml) within the Views/Test folder.
   2. Update the Test Controller Index() to return an ActionResult of View(“Test”).
   3. Validate the Test Page by running the solution using localhost:[port]/Test as the URL.
3. Using the HTML Label Helper:
   1. Create two labels using the Html.Label():
      1. Label 1: Welcome to Razor, centered in bold, 24-point Arial font.
      2. Label 2: My Customers are:, left aligned in bold, 18-point Arial font.
   2. To test your View, refresh your browser when you update your View.
   3. Take a screenshot. Label this Screenshot 1.
4. Using the HTML Action Link Helper:
   1. Create a link labeled ‘Refresh Page’ to this page, using the Html.ActionLink().
   2. To test your View, refresh your browser when you update your View.
   3. Hover your mouse over the hyperlink to display the URL in your status bar.
   4. Take a screenshot. Label this Screenshot 2.
5. Using an iterator to display a list of Object Model objects:
   1. In the Index() of the Test Controller create and initialize a List of UserModels. Pass this list as model data to the Test View.
   2. At the top of the Test View change the model to:

@model List<Activity2Part1.Models.UserModel>

* 1. Create an HTML Table in the View.
  2. Create a Row of Headers in Table: Name, Email Address, Phone Number.
  3. Iterate over the Model and List of UserModel objects using the C# for each construct. For each iteration, render a table row and three columns of data from the Model:

<td>@Html.DisplayFor(modelItem => user.Name)</td>

<td>@Html.DisplayFor(modelItem => user.EmailAddress)</td>

<td>@Html.DisplayFor(modelItem => user.PhoneNumber)</td>

* 1. To test your View, refresh your browser when you update your View.
  2. Take a screenshot. Label this Screenshot 3.

*Section 2*

1. Open solution from Activity 2 Part 1, Section 1.
2. Create a Button Controller to test new Razor View:
   1. Right click on the Controllers Folder.
   2. Select the Add->Controller… menu items. Then select the ‘MVC 5 Controller – Empty’ scaffolding type. Click the Add button. Name your Controller ‘Button,’ and click the Add button. Inspect the Button Controller (as ButtonController.cs class) within the Controllers folder.
3. Create a Button Model class to test the Razor View:
   1. Right click on the Models Folder.
   2. Create a Model C# class by selecting Add New Item menu. Then under the Add New Item dialog select Code under Visual C#, select Class, and enter ButtonModel for the name. Click Add.
   3. Add one property named State ButtonModel class with both setter and getter methods. Also add a nondefault constructor that will initialize the three properties.
      1. public ButtonModel(bool State) { };
      2. public bool State { get; set; }
4. Create a Button View:
   1. Right click on any part of the code body for the Button Controller. Select the Add View menu option. Change the View name to Button, select the Empty template, and from the Model class dropdown, and select the ButtonModel created in Step 3. Uncheck the Use Layout option. Click Add. Inspect the Button Page (as Button.cshtml) within the Views/Button folder.
   2. Update the Button Controller Index() to return an ActionResult of View(“Button”).
   3. Validate the Button Page by running the solution using localhost:[port]/Button as the URL.
5. Add Images for Buttons:
   1. Right click on your Project, select the Add->New Folder menu options, and create a new Folder called Images.
   2. Go to Google Images and search for two Button Images. Using Microsoft Paint (or any installed image editor), resize the button images to 50x50 pixels.
   3. Copy and paste the image files from File Explorer into your Projects Images folder.
6. Using the HTML Label Helper:
   1. Create two labels using the Html.Label():
      1. Label 1: Welcome to Razor, centered in bold, 24-point Arial font.
      2. Label 2: Play with Buttons, left aligned in bold, 18-point Arial font.
   2. To test your View, refresh your browser when you update your View.
7. Display Image Buttons in the View:
   1. Create static class scoped member variable named buttons as a List of type ButtonModel.
   2. In the Index() of the Button Controller add two ButtonModels to the buttons list. Pass this list as model data to the Button View.
   3. At the top of the Button View, change the model to:

@model List<Activity2Part1.Models.ButtonModel>

* 1. Using the Html.BeginForm() Razor method, create a Form that posts to the OnButtonClick method in the Button Controller.
  2. Add two HTML buttons with dynamic Images to the Form:

<button type=”submit” name=”mine” value=”1”>

@if(Model[0].State) <img src=”~/Images/On.png” />

else <img src=”~/Images/Off.png” />

</button><br/>

<button type=”submit” name=”mine” value=”2”>

@if(Model[1].State) <img src=”~/Images/On.png” />

else <img src=”~/Images/Off.png” />

</button>><br/>

1. Update the Button Controller:
   1. Add a new public method OnButtonClick that returns an ActionResult and takes a single method argument named mine of type string.
   2. In the OnButtonClick method implementation, check to see whether the mine argument is a value of “1” by toggling the ButtonModel state in the first element of the buttons list. If the mine argument is a value of “2,” then toggle the ButtonModel state in the second element of the buttons list. Return the Button view, and pass the buttons list model to the view.
2. Test your View:
   1. Validate the Button Page by running the solution using localhost:[port]/Button as the URL.
   2. Toggle the buttons and take at least two screenshots showing different states of your button images. Label these Screenshot 4 and 5.

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Create a project report using a GCU standard project header/cover page to include a header, your name, course, assignment name, and date.
2. For Section 1, add screenshots of the following to your project report:
   1. Screenshot 1 of the HTML Label Helper
   2. Screenshot 2 of the HTML Action Link Helper
   3. Screenshot 3 of the HTML Table rendering a list of Users
3. For Section 2, add screenshots of the following to your project report:
   1. Screenshots 4 and 5 of at least two different button states
4. Upload your code to GIT, and include the URL of your GIT repository in the project report.

## Part 2: Advanced .NET Razor

**Overview**

In this activity, students will use .NET MVC Views to learn Razor Layouts. They will also learn how to use the NuGet Packet Manager to add Bootstrap and jQuery to their existing activity.

**Execution**

Execute this activity according to the following guidelines:

1. We use the code from .NET MVC Application from the Activity 2 Part 1. Open the Activity 2 Part 1 Solution.
2. Create a Default Layout Page:
   1. Inspect the Views menu, and if a shared folder does not exist, complete the following: Right click on the Views folder. Select the Add->New Folder menu options. Create a folder called Shared. This folder will be used to hold your shared layout files.
   2. Right click on the Shared folder and select the Add->New Item menu options. In the Add New Item dialog, select the Web->Razor options. Then select the Layout Page option from the right pane. Then name your layout \_MyDefaultLayout.cshtml. Click the Add button. Validate that your new Layout Page was created in the Shared folder.
   3. Create a Layout Page with the following content:

<!DOCTYPE html>

<html>

<head>

<title>@ViewBag.Title</title>

</head>

<body>

@RenderPage("\_Header.cshtml")

@RenderBody()

<hr/>

@RenderPage("\_Footer.cshtml")

</body>

</html>

1. Create Header and Footer Pages:
   1. Right click on the Shared folder and select the Add->New Item menu options. In the Add New Item dialog, select the Web->Razor options. Then select the Web Page option from the right pane. Then name your layout to \_Header.cshtml. Click the Add button. Validate that your new Header Page was created in the Shared folder.
   2. Remove all the content from the Header Page, and replace it with the following content:

<div align="center">

<h2>Welcome to our Activity</h2>

</div>

* 1. Right click on the Shared folder and select the Add->New Item menu options. In the Add New Item dialog, select the Web->Razor options. Then select the Web Page option from the right pane. Then name your layout to \_Footer.cshtml. Click the Add button. Validate that your new Footer Page was created in the Shared folder.
  2. Remove all the content from the Footer Page and replace with the following content:

<style>

.footer {

text-align: center;

position: absolute;

bottom: 0;

width: 100%;

}

</style>

<div class="footer">

<h5>Copyright &copy; My Own Company Name</h5>

</div>

1. Update the Test View to use the new Layout Page:
   1. Remove all top-level html, header, and body tags.
   2. Update the Layout Specification to:

@{

ViewBag.Title = “My App”;

Layout = "~/Views/Shared/\_MyDefaultLayout.cshtml";

}

* 1. Validate the Test Page by running the solution using localhost:[port]/Test as the URL.
  2. Take a screenshot. Label this Screenshot 6.

1. Add Bootstrap and jQuery to your Application:
   1. Right click on your project and select the Manage NuGet Packages menu item.
   2. Select the Browse menu from the NuGet Package Manager dialog.
   3. In the search box, enter Bootstrap. Select the ‘bootstrap by Twitter’ package. Then click the Install button to add Bootstrap (and jQuery) into your Solution. Validate by inspecting the Content folder and Scripts folder in your Solution.
   4. NOTE: This same process can be used to update any packages in your Solution, but rather than selecting the Browse menu, you select the Updates menu from the NuGet Package Manager dialog.
   5. Update the Layout Page to add the Bootstrap and jQuery libraries:

<!DOCTYPE html>

<html>

<head>

<title>@ViewBag.Title</title>

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link href="~/Content/Site.css" rel="stylesheet" type="text/css" />

<link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />

<script src="~/Scripts/jquery-1.10.2.min.js"></script>

<script src="~/Scripts/bootstrap.min.js"></script>

</head>

<body>

@RenderPage("\_Header.cshtml")

<div class="container body-content">

@RenderBody()

</div>

<hr />

@RenderPage("\_Footer.cshtml")

</body>

</html>

* 1. Update the Header Page to add a Menu Bar—similar code can also be found from our first MVC app built from Activity #1 in the \_layout.cshtml file:

<div class="navbar navbar-inverse navbar-fixed-top">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

@Html.ActionLink((string)ViewBag.Title, "", "Test", new { area = "" }, new { @class = "navbar-brand" })

<a href="<http://www.gcu.edu>" class="navbar-brand" target="\_blank">GCU</a>

</div>

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav navbar-right">

<a href="<http://www.google.com>" class="navbar-brand" target="\_blank">Google</a>

<a href="<http://www.yahoo.com>" class="navbar-brand" target="\_blank">Yahoo</a>

</ul>

</div>

</div>

</div>

* 1. Resize your browser. Notice how your application is now responsive.
  2. Validate the Test Page by running the solution using localhost:[port]/Test as the URL.
  3. Take a screenshot. Label this Screenshot 7.

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Updates to the project report created in Part 1.
2. Add screenshots of the following to your project report:
   1. Screenshot 6 of the Test View rendered with a layout, header, and footer.
   2. Screenshot 7 of the Test View rendered with Menu Bar using Bootstrap.
3. Upload your code to GIT, and include the URL of your GIT repository in the project report.

## Part 3: Data Validation

**Overview**

In this activity, students will use .NET MVC Data Validation Framework to add validation logic to your Object Model.

**Execution**

Execute this activity according to the following guidelines:

1. Use the code from .NET MVC Application from Activity 1 Part 3. Open the Activity1Part3 Solution.
2. Add Data Validation Rules to the User Model:
   1. Add the following Data Annotations above the Username Property:

[Required]

[DisplayName("User Name")]

[StringLength(20, MinimumLength = 4)]

[DefaultValue("")]

* 1. Add the following Data Annotations above the Password Property:

[Required]

[DisplayName("Password")]

[StringLength(20, MinimumLength = 4)]

[DefaultValue("")]

1. Update the Login Controller to handle Data Validation:
   1. Add the following code to the Login Controller prior to calling the Business Service:

// Validate the Form POST

if (!ModelState.IsValid)

return View("Login");

1. Inspect the Login View for Data Validation Razor Tags:
   1. Note the use of the Html.ValidationMessagesFor() Razor method.
2. Test the Data Validation Rules:
   1. Validate the Login Page by running the solution using localhost:[port]/Login as the URL.
   2. Take a screenshot of validation errors on each field of the Login Form. Label this Screenshot 8.
   3. Take a screenshot of a successful login. Label this Screenshot 9.

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Updates to the project report created in Part 1.
2. Add screenshots of the following to your project report:
   1. Screenshot 8 of the Login View rendered with data validation errors.
   2. Screenshot 9 of the Login response rendered with a successful login.
3. Upload your code to GIT, and include the URL of your GIT repository in the Project Report.

## Part 4: Practicing Agile Scrum

**Overview**

In this activity, students will practice Scrum and work on their team project. They will emulate a “compressed” 3-week Sprint over the course of this week as well as perform Scrum Daily Standups, Sprint Planning, a Burn Down Chart, and a Retrospective on their team project.

**Execution**

Execute this assignment according to the following guidelines:

1. This week will be used for working on your milestone project.
2. You are expected to complete Daily Standups, Sprint Planning, a Sprint Burn Down Chart, and Retrospective.
3. Post the following in a Word document for this activity to LoudCloud:
   1. Results of your Daily Standup (who did what over the course of the week)
   2. Sprint Plan for this week
   3. Sprint Burn Down Chart for this week
   4. Results of Retrospective for this week

**Submission**

Submit the following to the learning management system:

**Parts 1-3**

1. Project Report that includes:
   1. A GCU standard project header/cover page to include your name, course, assignment name, and date
   2. Screenshots 1-9
   3. URL of your GIT repository
2. Upload your code to the GIT repository.

**Part 4**

1. A Word document to include:
   1. Results of your Daily Standup (who did what over the course of the week)
   2. Sprint Plan for this week
   3. Sprint Burn Down Chart for this week
   4. Results of Retrospective for this week

# Activity 3

This activity has multiple parts/assignments. All assignments must be completed prior to documentation submission.

## Part 1: AJAX, Partial Views, and Partial Page Updates

**Overview**

In this activity, students will learn how to build Razor Forms, AJAX enabled Razor Forms, Partial Views, and Partial Page Updates.

**Execution**

Execute this activity according to the following guidelines:

1. Create a Model named CustomerModel with the following properties. Also create a nondefault constructor that initializes all the class properties.

int ID

string Name

int Age

1. Create a default Customer Controller.
2. Create an empty Customer View with no layout and no model.
3. Create a Layout page \_Layout in the Shared directory that contains the following:
   1. @Renderpage("\_Header.cshtml"). Create this Page so it simply displays the following two lines:
      1. Title 'Welcome to Activity 3 Part 1' in H2 formatting.
      2. Current Time 'Current Time is @DateTime.Now in H2 formatting.
   2. @RenderBody()
4. Create a Partial page \_CustomerDetails in the Shared directory that contains the following:
   1. A Model declaration to use the CustomerModel.
   2. Title 'Customer Details' using a Razor Label.
   3. @Html.DisplayNameFor Razor Bind to the Model's Name property.
   4. @Html.DisplayNameFor Razor Bind to the Model's Age property.
5. Modify the Customer Controller:
   1. Create a class scoped member variable ‘customer’ that is a List of type CustomerModel.
   2. Create a default constructor method. Create a List of Customer Model object models within the constructor implementation.
   3. Update Index() to create a C# Tuple with a customer list and first customer from the list as data then pass the Tuple to the Customer View.
   4. Create OnSelectCustomer() that takes a single string method argument (the customer from the Radio Button value) and that creates a C# Tuple with a customer list and customer passed in customer ID from the list as data, then pass the Tuple to the Customer View. The method should conform to an HTTP Post method.
6. Create a non AJAX Form without Partial Page updates:
   1. Change the Model declaration to use the Tuple created in the previous step.
   2. Change the HTML page title to a value set in the ViewBag.Title.
   3. Change the Layout to the layout created in the previous step.
   4. Add a Label '“Please select a customer”' using a Razor Label.
   5. Create a Razor Form that calls the CustomerController.OnSelectCustomer1, that uses a POST, and whose form ID is ‘myform.’
   6. Iterate over the Tuple Item 1 (list of Customer Models) to add a Razor Radio Button for each customer setting the name of the Radio Button group to Customer, the Customer ID as its value, with an onchange handler that submits the ‘myform.’ Also use a Razor Label tag to display the customer name.
   7. Add Partial page using @Html.Partial("\_CustomerDetails", ModelItem2) specifying the Customer Details Partial View and Tuple Item 2 (CustomerModel). Place this Razor tag inside a div tag with an ID of ‘customerInfo.’
   8. Run the application:
      1. Select each of the Customers, and make sure Customer Details Partial Page gets updated.
      2. Note how the time changes, indicating a full-page refresh.
      3. Take a screenshot. Label this Screenshot 1.
7. Create an AJAX Form with Partial Page updates:
   1. Using NuGet, make sure support for Microsoft jQuery Unobtrusive Ajax is installed (search for unobtrusive).
   2. In the \_Layout, add two script references in the HTML header to jquery.js and jquery.unobtrusive-ajax.js.
   3. Create a Razor variable ‘ajaxOptions’ at the top of the Customer View that creates in instance of AjaxOptions that sets the following properties:

HttpMethod = “POST”

InsertionMode = InsertionMode.Replace

UpdateTargetId = “customerInfo”

* 1. Change the Html.BeginForm to Ajax.BeginForm and insert the ‘ajaxOptions’ into the third parameter of the AjaxBegin.Form().
  2. Update the Customer Controller by changing the implementation of the OnSelectCustomer() to returns a ‘\_CustomerDetails’ PartialView with data of a CustomerModel obtained from the list of Customers.
  3. Run the application:
     1. Select each of the Customers, and make sure Customer Details Partial Page gets updated.
     2. Note how the time DOES NOT change, indicating no full-page refresh.
     3. Take a screenshot. Label this Screenshot 2.

1. Create an AJAX JavaScript callback handler:
   1. Update the Customer Controller adding a new controller method GetMoreInfo() that takes a string customer ID as an argument, returns a string, and conforms to a HTTP Post method. The method can just return any desired string value.
   2. Update the Razor variable ‘ajaxOptions’ at the top of the Customer View with the following property:

OnSuccess = “ajaxSuccess”

* 1. Create an inline script in the View that makes a jquery post request to Customer Controller to the GetMoreInfo().

$.post('@Url.Action("GetMoreInfo", "Customer")', { customer: 2 }, function (data) { alert(data); });

* 1. Run the application:
     1. Select each of the Customers, and make sure Customer Details Partial Page gets updated, as well as the AJAX callback alert displayed in JavaScript with your desired message.
     2. Take a screenshot. Label this Screenshot 3.

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Create a project report using a GCU standard project header/cover page to include a header, your name, course, assignment name, and date.
2. Add screenshots of the following to your project report:
   1. Screenshot 1 of the Non AJAX Form
   2. Screenshot 2 of the AJAX Form with Partial Page Updates
   3. Screenshot 3 of the AJAX JavaScript Alert callback
3. Upload your code to GIT and include the URL of your GIT repository in the Project Report.

## Part 2: Visual Studio Debugger

**Overview**

In this activity, students will learn how use the Visual Studio Debugger. Use the code from Activity 2 Part 3.

**Execution**

Execute this assignment according to the following guidelines:

1. Open the solution for Activity 2 Part 3.
2. Setting Breakpoints:
   1. Open the LoginController class.
   2. Set a breakpoint on the ModelState.IsValid() line in doLogin().
   3. Set a breakpoint on the SecurityService service = new SecurityService() line in doLogin().
   4. Run the application in debug mode with known good log in credentials.
   5. Take a screenshot of a breakpoint from Step 2a. Label this Screenshot 4.
   6. Click the Continue button.
   7. Take a screenshot of a breakpoint from Step 2b. Label this Screenshot 5.
   8. Click the Continue button.
3. Inspecting Variables:
   1. Set the same breakpoints from Step 2a and Step 2b.
   2. Run the application in debug mode with known good log in credentials.
   3. Inspect the UserModel argument passed to the doLogin() by hovering over the variable.
   4. Take a screenshot from Step 3c. Label this Screenshot 6.
   5. Inspect the variables in the Locals Window.
   6. Take a screenshot from Step 3e. Label this Screenshot 7.
   7. Remove all breakpoints.
4. Stepping into a Function, over a Function, and out of a Function:
   1. Open up the LoginController class.
   2. Set a breakpoint on the service=Authenticate(model) line in doLogin().
   3. Run the application in debug mode (with known good log in credentials).
   4. Verify that code stopped at the breakpoint set in Step 4b.
   5. Click the Step Into icon from the debugger toolbar or enter the F11 key.
   6. Inspect the UserModel argument passed to the Authenticate() by hovering over the variable.
   7. 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().
   8. Take a screenshot from Step 4g. Label this Screenshot 8.
   9. Open up the SecurityDAO class.
   10. Set a breakpoint on the line of code in the FindByUser() that calls ExecuteReader().
   11. Click the Continue button. Verify the code stopped at the breakpoint set in Step 4j.
   12. Take a screenshot from Step 4k. Label this Screenshot 9.
   13. Step through the code using Step Over until the return statement is reached.
   14. Click the Step Over icon from the debugger toolbar or enter the Shift F11 key until the return value from the Authenticate() has been set.
   15. Click the Step Over icon from the debugger toolbar or enter the Shift F11 key.
   16. Inspect the Authenticate() return value by hovering over the variable.
   17. Take a screenshot from Step 4p. Label this Screenshot 10.
   18. Click the Continue button.
5. Setting Conditional Breakpoints:
   1. Remove all breakpoints (see option from the Debug menu).
   2. Open up the LoginController class.
   3. Set a breakpoint on the line after the service=Authenticate(model) line in doLogin() and where the return value has been set.
   4. Click the “cog” that will be displayed when you hover over breakpoint “dot.”
   5. Click the Conditions checkbox. Set a conditional statement when the return value from Authenticate() is true. Uncheck the Actions checkbox. Click the close button.
   6. Run the application in debug mode.
   7. Verify that code stopped at the breakpoint set in Step 5e.
   8. Take a screenshot from Step 5g. Label this Screenshot 11.
   9. Click the Continue button.
   10. Click the “cog” that will be displayed when you hover over breakpoint “dot.”
   11. Leave the Condition checkbox checked. Check the Actions checkbox. Enter a message “I am here!!” in the ‘Log a message to Output Window.’ Check the Continue execution checkbox. Click the close button.
   12. Run the application in debug mode.
   13. Verify the message “I am here!!” got logged in the Output Window.
   14. Take a screenshot from Step 5m. Label this Screenshot 12.
   15. Click the Continue button.
6. Inspecting the Call Stack:
   1. Open up the SecurityDAO class.
   2. Set a breakpoint on the line of code in return statement in the FindByUser().
   3. Run the application in debug mode.
   4. Verify that code stopped at the breakpoint set in Step 6b.
   5. Inspect the Call Stack Window.
   6. Take a screenshot from Step 6e. Label this Screenshot 13.
   7. Click the Continue button.

**Extra Practice**

1. From the Analyze menu, select the Run Code Analysis menu option. Inspect the warnings returned from the analysis. Why should you fix the warnings in your code?
2. From the Analyze menu, select the Calculate Code Metrics menu option. Inspect the results returned from the analysis. Why is this an important tool to run on your code?

**Submission**

Submit the following to the learning management system:

1. Project Report that includes:
   1. A GCU standard project header/cover page to include your name, course, assignment name, and date
   2. Screenshots 1-13
   3. URL of your GIT repository
2. Upload your code to the GIT repository.

# Activity 4

This activity has multiple parts/assignments. All assignments must be completed prior to documentation submission.

## Part 1: Logging with nLog Framework

**Overview**

In this activity, students will learn how to add logging using the NLog Logging Framework to the previous activity. They will also build a reusable Logging Utility class based on the NLog Logging Framework API’s, which also implements the Singleton Design Pattern. Use the code from Activity 2 Part 3.

**Execution**

Execute this activity according to the following guidelines:

1. Add NLog Logging Framework to Activity 2 Part 3:
   1. Open up the Solution from Activity 2 Part 3.
   2. Right click on your solution and select the Manage NuGet Packages menu options to bring up the NuGet Package Manager.
   3. Select the Browse tab from the NuGet Package Manager.
   4. Search for NLog.
   5. Install both the NLog and NLog.Config packages.
   6. Validate that there is a NLog.config file in the root of your project.
2. Configure NLog:
   1. Open the NLog.config file in the root of your project.
   2. Setup a Logging Target:

<target xsi:type="File" name="myAppLoggerTarget" fileName="${basedir}/logs/${shortdate}.log" layout="${longdate} ${uppercase:${level}} ${message}" />

* 1. Set up a Logging Rule:

<logger name="myAppLoggerRules" minlevel="Debug"

writeTo="myAppLoggerTarget" />

1. Add Logging to the Application using NLog:
   1. Add NLog to all the classes using NLog;
   2. Add a private static class scoped variable to the class using NLog

private static Logger logger = LogManager.GetLogger("myAppLoggerRules");

* 1. Add the following logging statements into your Login Controller:
     1. Add an info level logger statement when the DoLogin() is called:

logger.Info(“Entering LoginController.DoLogin()”);

* + 1. Add an info level logger statement when the user name and password is read from DoLogin(). You must import System.Web.Script.Serialization tp get the JSON serialization support:

logger.Info(“Parameters are: new JavaScriptSerializer().Serialize(model);

* + 1. Add an info level logger statement when login passes:

logger.Info(“Exit LoginControlle.DoLogin() with login passing”);

* + 1. Add an info level logger statement when login fails:

logger.Info(“Exit LoginController.DoLogin() with login failing”);

* + 1. LoginController: add an error level logger statement when an exception occurs:

logger.Error(“Exception LoginController.DoLogin()” . e.Message);

* 1. Run your application through a successful and failed login.
  2. 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.
  3. Take a screenshot of your log file when login passes and fails. Label these Screenshots 1 and 2.

1. Create a Logging Facade in C#:
   1. NOTE: All code is published as a code snippet in an announcement.
   2. Create a Utility folder under the Service folder in your solution.
   3. Create an ILogger Interface Class in C# that supports the following contract:

void Debug(strng message);

void Info(string message);

void Warning(string message)

void Error(string message);

* 1. Create a MyLogger1 Logger Implementation Class in C#:
     1. Singleton Design Pattern:
        1. Private class constructor.
        2. GetInstance() method that does lazy initialization on a private static instance of the class and returns the instance.
     2. GetLogger() that does lazy initialization on a private instance of the NLog Logger and returns the instance.
     3. Implements the ILogger interface methods.

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

private static MyLogger1 logger = MyLogger1.GetInstance();

* 1. 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.
  2. Take a screenshot of your log file when login passes and fails. Label these Screenshots 3 and 4.

**Extra Practice**

1. Research the Singleton Design Pattern. How was this design pattern implemented in Step 4?
2. Research the Façade Design Pattern. How was this design pattern implemented in Step 4?
3. Create another implementation of a Logging Utility that uses different Logging Targets and Logging Rules.
4. Why would creating a wrapper façade over an existing Logging Framework be beneficial and recommended in an application architecture?

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Create a project report using a GCU standard project header/cover page to include a header, your name, course, assignment name, and date.
2. Add screenshots of the following to your project report:
   1. Screenshots 1 and 2 from Step 3f
   2. Screenshots 3 and 4 from Step 5c
3. Upload your code to GIT, and include the URL of your GIT repository in the project report.

## Part 2: REST Services using WCF

**Overview**

In this activity, students will learn how design and build a REST(ful) Service using Windows Communication Foundation (WCF). They will also implement the Data Transfer Object Design Pattern. Use the code from Activity 2 Part 3, as well as the Browser and Postman to test the REST(ful) Services.

**Execution**

Execute this activity according to the following guidelines:

*Section 1: Create a "Hello World" WCF REST Service*

1. Open the Activity 2 Part 3 solution.
2. Right click on the top-level solution folder, click Add->New Project, select the WCF project type, select WCF Service Application type, name your project HelloWorldService, and click OK.
3. Create a Hello World Service:
   1. Remove all the methods in the Service Contract Interface and Service Class.
   2. Add the SayHello(), GetData(), and GetObjectModel() methods to the Service Contract Interface (see code snippet in announcement):

[OperationContract]

[WebGet(ResponseFormat = WebMessageFormat.Json, UriTemplate = "SayHello/")]

string SayHello();

[OperationContract]

[WebGet(ResponseFormat = WebMessageFormat.Json, UriTemplate = "GetData/{value}")]

string GetData(string value);

[OperationContract]

[WebGet(ResponseFormat = WebMessageFormat.Json, UriTemplate = "GetObjectModel/{id}")]

CompositeType GetObjectModel(string id);

* 1. Implement the Service Contract Interface methods in the Service Implementation Class.

1. Add the Service and behaviors to the Web.config (see code snippet in announcement).
2. Test your Hello World Service using your Browser:
   1. Click on the Project Name and click the Run Chrome toolbar icon.
   2. Take a screenshot of localhost:[port]/Service1.svc/SayHello. Label this Screenshot 5.
   3. Take a screenshot of localhost:[port]/Service1.svc/GetData/1. Label this Screenshot 6.
   4. Take a screenshot of localhost:[port]/Service1.svc/GetObjectModel/1. Label this Screenshot 7.

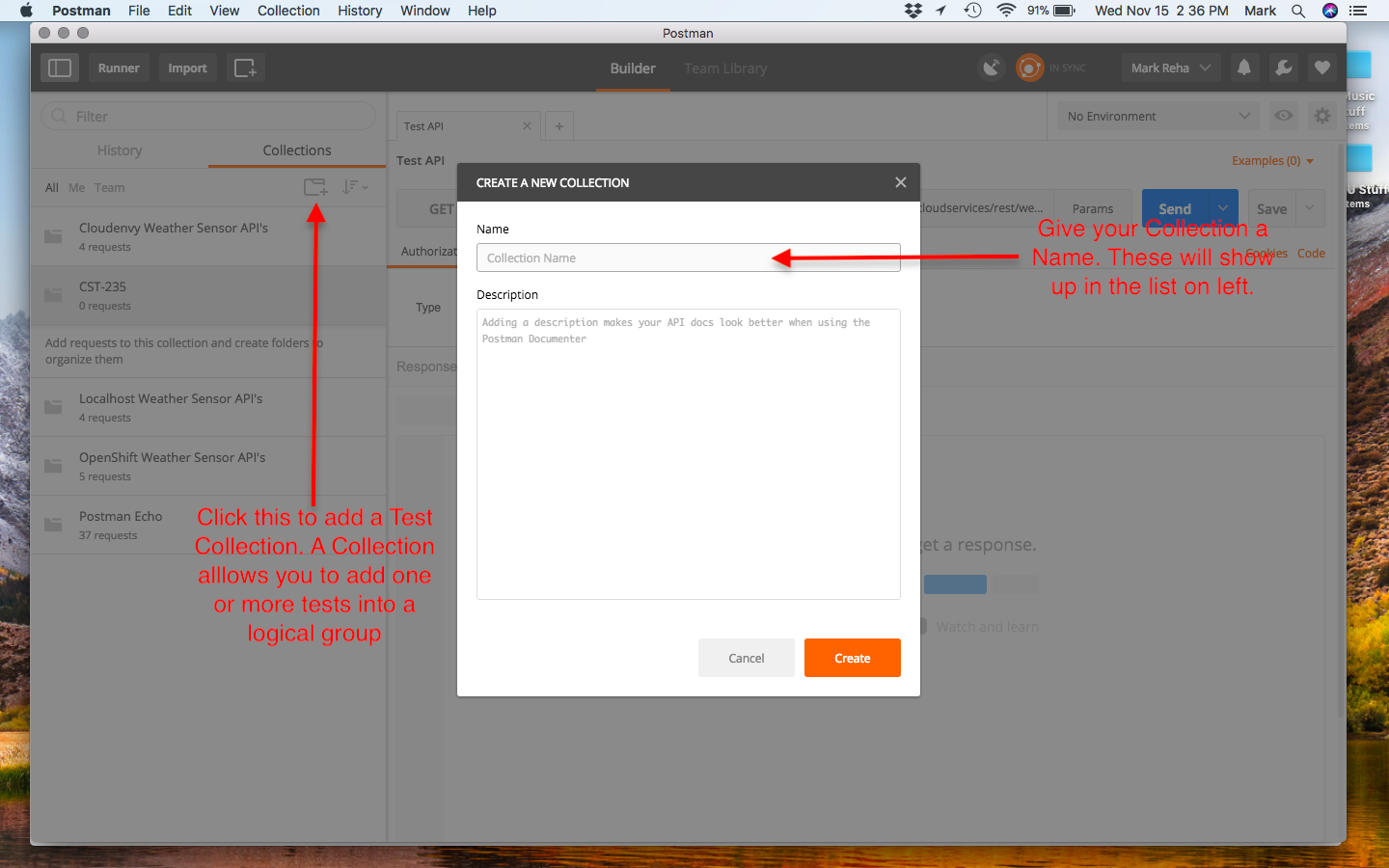
*Section 2: Create a "User" WCF REST Service*

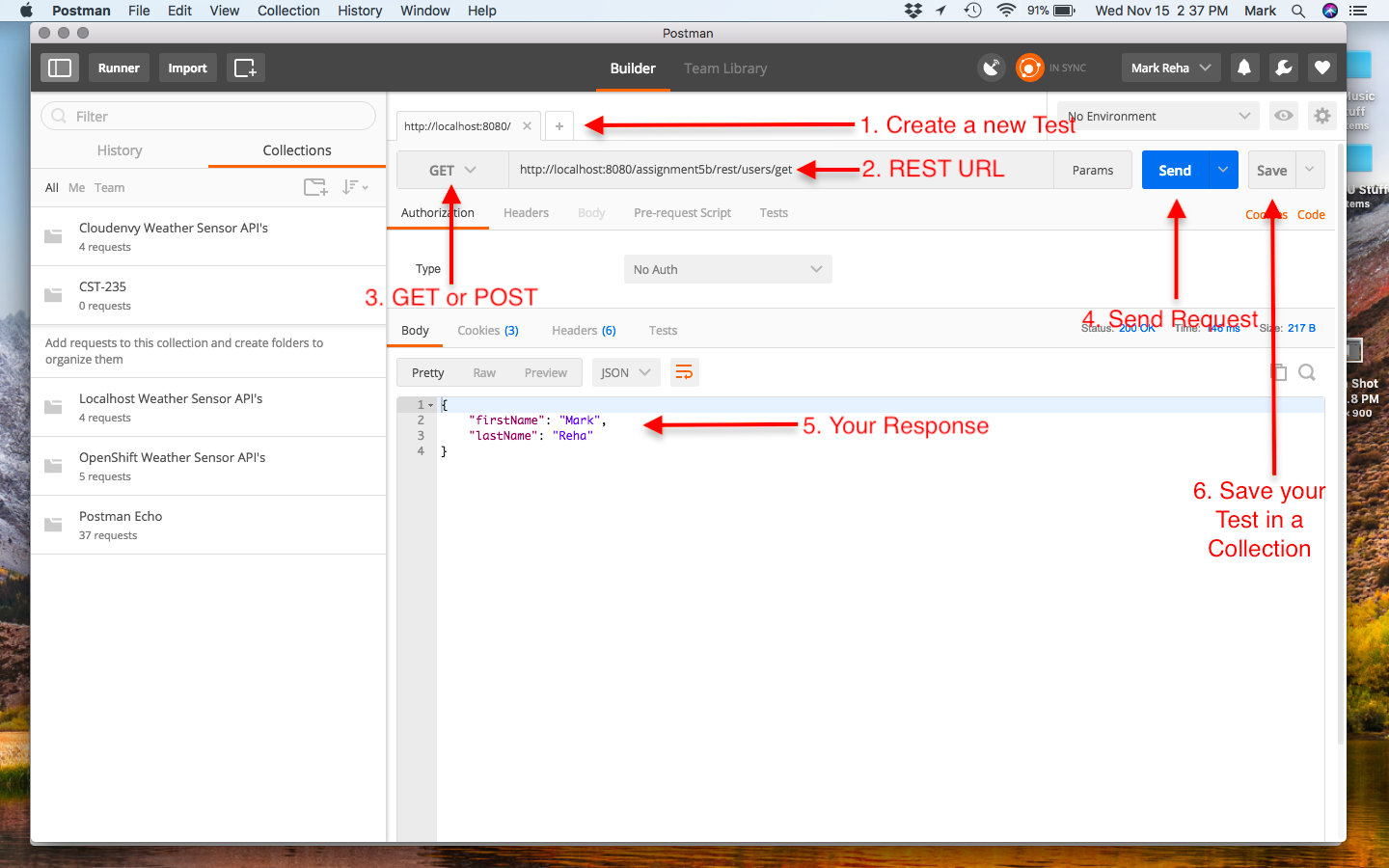
1. Refactor the Service Contract Interface and Service Class:
   1. Open up the Service Contract Interface class. Highlight the class name. Rename to IUserService. Right click. Select the Rename menu. Click OK.
   2. Click on the IService1.cs in your solution. Right click. Select the Rename menu. Enter IUserService.
   3. Open up the Service class. Highlight the class name. Rename to UserService. Right click. Select the Rename menu. Click OK.
   4. Click on the Service1.svc in your solution. Right click. Select the Rename menu. Enter UserService. Update the interface this class implements to IUserService.
   5. Right click on UserService.svc and select the View Markup menu option. Rename Service1 to UserService.
   6. Open up web.config. Change the Service1 class to UserService. Change the IService1 interface to IUserService.
   7. Test your all your Service methods to ensure the Service URI is now localhost:[port]/UserService.svc/
2. Add a DTO class (see code snippet in announcement):
   1. Create a new C# class called DTO.
   2. Add properties of ErrorCode of type int, ErrorMsg of type string, and Data of type List<UserModel>.
   3. Make the DTO class serializable by adding a [DataContract] annotation at the class level. DataContract is in System.Runtime.Serialization.
   4. Make the DTO class properties serializable by adding a [DataMember] annotation for each property.
3. Add a reference to the UserModel (see code snippet in announcement):
   1. Right click, Add->Reference, select Project, click Activity2Part3 (or whatever project has implemented the UserModel class).
   2. Make the UserModel class serializable by adding a [DataContract] annotation at the class level. DataContract is in System.Runtime.Serialization.
   3. Make the UserModel class properties serializable by adding a [DataMember] annotation for each property.
4. Add new Service methods the Service Interface (see code snippet in announcement):
   1. DTO GetUser(string id) mapped to UriTemplate "GetUser/{id}".
   2. DTO GetAllUsers() mapped to UriTemplate "GetAllUsers/".
5. Update the Service Class:
   1. Add a class scoped variable 'users' of type List<UserModel>.
   2. Add a Constructor to the Service Class.
   3. Initialize the 'users' with test Users.
6. Implement the new methods the Service Class (see code snippet in announcement):
   1. GetUser(string id) should create an instance of DTO with an error of 0, message of "OK," and the user from the list if ID is in bounds, otherwise return a DTO populated with an error of -1, message of "User Does Not Exist," and null users list.
   2. GetAllUsers() should create an instance of DTO with an error of 0, message of "OK", and the users list.
7. Test your User Service using your Browser:
   1. Click on the Solution Name and click the Run Chrome toolbar icon.
   2. Take a screenshot of localhost:[port]/UserService.svc/GetAllUsers/. Label this Screenshot 8.
   3. Take a screenshot of localhost:[port]/UserService.svc/GetUser/1. Label this Screenshot 9.
   4. Take a screenshot of localhost:[port]/UserService.svc/GetUser/4 (or a number that exceeds the count in the users list). Label this Screenshot 10.
8. Test your User Service using Postman:
   1. Download Postman from <https://www.getpostman.com>. Refer to the Appendix below for installation instructions as needed.
   2. Click on the New Collection icon. Enter a desired Collection name. Click the Create button.
   3. Click the new Collection name in the Collection list.
   4. Click the Add Requests link. Enter a desired Request name. Click the Save to button.
   5. Click the new Request name listed within the Collection.
   6. From the Request enter the following information:
      1. GET Command
      2. Request URL
      3. Click the Save button
   7. Repeat Steps 8d-f for the following URLs, and click the Send button to test each URL:
      1. Take a screenshot of localhost:[port]/UserService.svc/GetAllUsers/. Label this Screenshot 11.
      2. Take a screenshot of localhost:[port]/UserService.svc/GetUser/1. Label this Screenshot 12.
      3. Take a screenshot of localhost:[port]/UserService.svc/GetUser/4 (or a number that exceeds the count in the users list). Label this Screenshot 13.
9. Options for the User Service:
   1. To return XML instead of JSON change the ResponseFormat in the Interface Service class to WebMessageFormat.Xml.
   2. See the HOWTO Guide for changing your Service URI to a friendly URI (with the Service.svc URI).

**Appendix**

**How do I make a GET Request with Postman?**

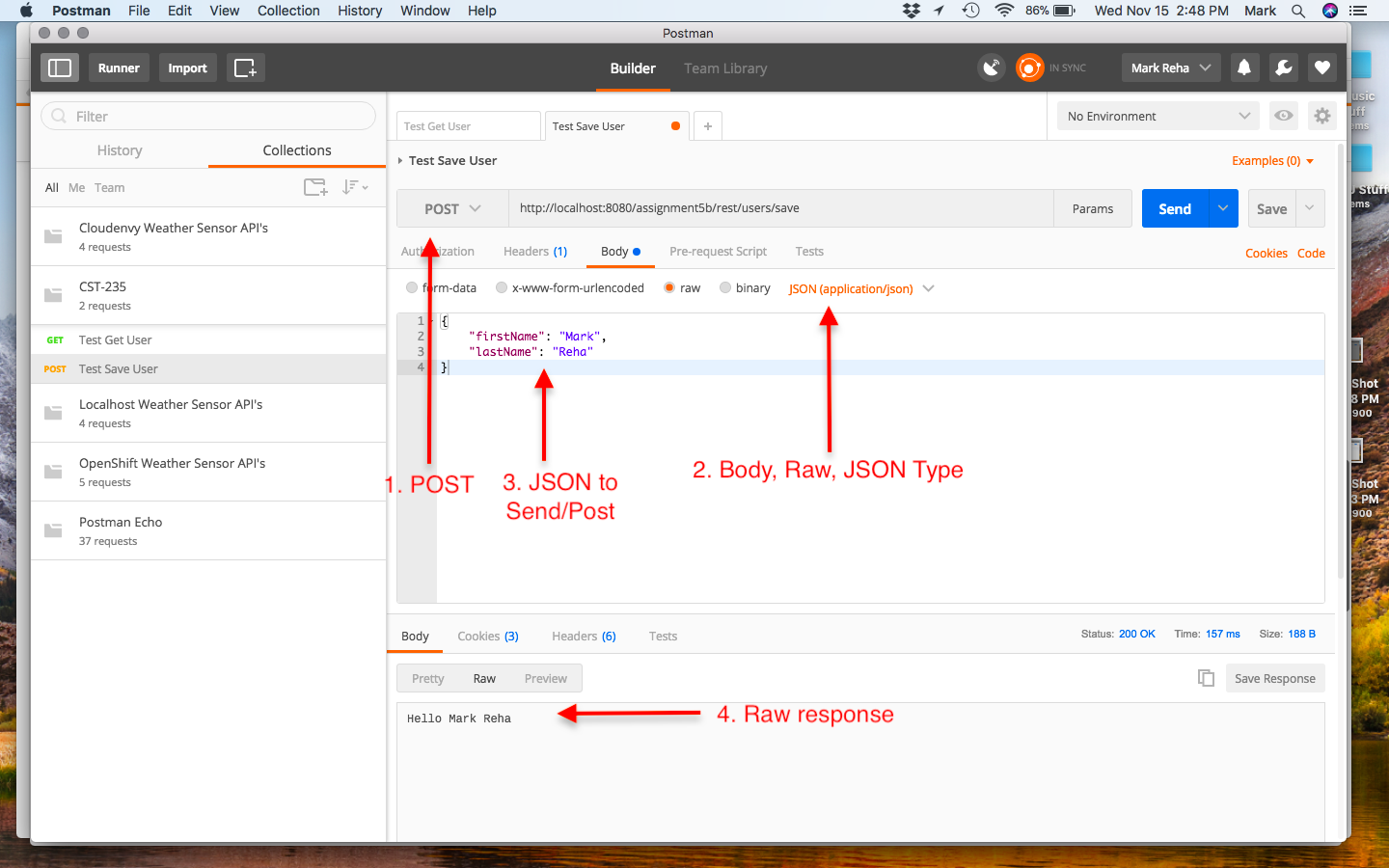
You can always use your browser for sending GET requests, but this is error prone, and you have to write down all your URLs and test scenarios for all your APIs. Postman is a very nice desktop tool for testing your REST APIs. Postman allows you to set up Collections of API Tests, and supports making HTTP GET and POST requests. If you sign up for an account with Postman, you can even sync your Collections and Tests across development environments and team members. Postman even allows you to create unit tests and validate that you get the proper responses. This capability is available in the Tests tab of the UI. Reference the Postman home page for more information. See the examples below.

****

****

**How do I make a POST Request with Postman?**

Postman easily supports making HTTP POST requests, too. This cannot be done using your browser, or without writing a client page. See the example below.

****

**Extra Practice**

1. Research the Data Transfer Object Design Pattern. How was this design pattern implemented in Part 2 Step 2?
2. Why is the Data Transfer Object beneficial and recommended in an application architecture?
3. What would you need to change in your solution to serialize your data to XML?
4. What would you need to change in your solution to use the SOAP binding?

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Updates to the project report created in Part 1.
2. Add screenshots of the following to your project report:
   1. Screenshots 5-7 from Part 1 Step 5
   2. Screenshots 8-10 from Part 2 Step 7
   3. Screenshots 11-13 from Part 2 Step 8
3. Upload your code to GIT, and include the URL of your GIT repository in the project report.

## Part 3: Action Filters

**Overview**

In this activity, students will learn how to use Action Filters, and how to use a Data Cache in both a Controller and Windows Communication Foundation (WCF) Service. Use the code from Activity 4 Part 2.

**Execution**

Execute this assignment according to the following guidelines:

1. Creating a Security Authorization Filter (see code snippets in the announcement):
   1. Create a C# class call CustomAuthorizationAttribute in the Controllers folder.
   2. This class should extend from the FilterAttribute and implement the IAuthorizationFilter interface.
   3. Implement the OnAuthorization() method that redirects to the /Login URI.
   4. Add new method in the Login Controller called Protected that returns a string that is decorated with HttpGet and CustomAuthorization attributes.
   5. Navigate to the /Login/Protected URI and test that the Login Page is displayed.
   6. Take a screenshot of a breakpoint that is stopped at your redirect code in the CustomAuthorizationAttribute class. Label this Screenshot 14.
   7. Create a new Test Controller that returns a string from Index() that is decorated with the HttpGet and CustomAuthorization attributes.
   8. Navigate to the /Test URI and test that the Login Page is displayed.
   9. Take a screenshot of a breakpoint that is stopped at your redirect code in the CustomAuthorizationAttribute class. Label this Screenshot 15.
2. Creating an Action Filter (see code snippets in the announcement):
   1. Create a C# class call CustomActionAttribute in the Controllers folder.
   2. This class should extend from the FilterAttribute and implement the IActionFilter interface.
   3. Implement the OnActionExecuted() method that logs the Controller Name and Action Method Name.
   4. Implement the OnActionExecuting() method that logs the Controller Name and Action Method Name.
   5. Decorate the Login Controller class with the CustomAction attribute.
   6. Navigate to the /Login URI and validate that the entry and exit log messages are displayed in your log file.
   7. Take a screenshot of your log file. Label this Screenshot 16.
3. Creating a Memory Data Cache (see code snippets in the announcement):
   1. Right click on Project, select Add->Reference menu items, select Assemblies, in the Search enter System.Runtime.Caching, check the Assembly, and click OK.
   2. Add new method in the Login Controller called GetUsers that returns a string (JSON serialized instance of a List<UserModel>).
   3. Get an instance of the MemoryCache.Default.
   4. Check for a cache entry of "Users".
   5. If cache entry is null, then create a List of UserModel, and put in the "Users" cache entry with an expiration policy of 60s. Use the Logger Service to log this event.
   6. If cache entry is not null, then use the Logger Service to log this event.
   7. Serialize the List of UserModel to JSON, and return the serialized string from the GetUsers action.
   8. Run the application at /Login/GetUsers validating that the cache works properly, and that the cache can be refreshed every 60s.
   9. Take a screenshot of the log file, showing when the List of Users was created (cache entry miss), when the Users are retrieved from the cache, and when the cache has expired. Label this Screenshot 17.
4. Creating a WCF Data Cache (see code snippets in the announcement):
   1. Add a [AspNetCacheProfile("CacheFor60Seconds")] attribute to the GetAllUser() method in the IUserService service interface class.
   2. Add a caching section to the system.web section of the Web.config file within the HelloWorldService project.
   3. Set a breakpoint in the implementation of the GetAllUsers() method in the UserService.svc.cs class.
   4. Access the GetAllUsers() endpoint, validating that the breakpoint hits with the first access, does not hit the breakpoint on subsequent accesses for 60s, and then breaks after 60s.
   5. Take a screenshot when the breakpoint is hit, and when the breakpoint is not hit. Label this Screenshot 18.

**Extra Practice**

1. Research whether the default MemoryCache in .NET is threadsafe. Why does this matter in a .NET application?
2. Research other Caching Frameworks that are available in .NET. What did you find, and can you add them with the NuGet Package Manager to your application?
3. What design precautions do you need to make when incorporating data caching into your architecture?

**Submission**

Submit the following to the learning management system:

1. Project Report that includes:
   1. A GCU standard project header/cover page to include your name, course, assignment name, and date
   2. Screenshots 1-18
   3. URL of your GIT repository
2. Upload your code to the GIT repository.

# Activity 5

This activity has multiple parts/assignments. All assignments must be completed prior to documentation submission.

## Part 1: IoC using Unity

**Overview**

In this activity, students will use Inversion of Control (IoC) container to inject a Logging Service into a Controller. They will also explore how to build a Logging Service and leverage Dependency Injection using the Unity Framework.

**Execution**

Execute this activity according to the following guidelines:

1. Add Unity Framework to Solution:
   1. Right click on Project, and select Manage NuGet Packages menu to open NuGet Package Manager.
   2. Select the Browse tab, and search for Unity.
   3. Install Unity by Microsoft package.
   4. Select the Browse tab, and search for Unity.Mvc.
   5. Install Unity.Mvc by Microsoft package.
2. Create a Logger Service:
   1. Add public specification to the ILogger Interface in the Utility namespace.
   2. Copy MyLogger1 class to MyLogger2 class from the Utility namespace.
   3. Remove static from Logger class variable.
   4. Remove static MyLogger class instance variable.
   5. Remove private constructor.
   6. Remove GetInstance() method.
   7. Update GetLogger() to use Logger class variable.
3. Constructor Injection:
   1. Register the Logging Service with Unity:
      1. Open UnityConfig file located in the App\_Start folder of your project.
      2. Add the following code to the Application\_Start() to register the Logging Service with Unity.

container.RegisterType<ILogger, MyLogger2>(new ContainerControlledLifetimeManager());

* 1. Create a Test Logging Service Controller:
     1. Add a new Controller called TestLoggingService1Controller.
     2. Create a private read-only class scoped variable called logger of type ILogger.
     3. Create a public nondefault constructor that takes an ILogger method argument.
     4. Save the constructor method argument to the private logger class variables.
     5. Change the index() method to return a string.
     6. In the implementation of the Index() call the Info() on the logger, and return a test string response.
     7. Run the application and invoke the controller.
     8. Verify that the logger statement shows up in the Log file. Take a Screenshot. Label this Screenshot 1.

1. Property Injection:
   1. Create a Test Logging Service Controller:
      1. Add a new Controller called TestLoggingService2Controller.
      2. Create a class scoped property variable called logger of type ILogger.

[Dependency]

public ILogger logger { get; set; }

* + 1. Change the Index() method to return a string.
    2. In the implementation of the Index() call the info() on the logger, and return a test string response.
    3. Run the application and invoke the controller.
    4. Verity that the logger statement shows up in the Log file. Take a Screenshot. Label this Screenshot 2.

1. Method Parameter Injection:
   1. Create a Test Service:
      1. Create a new interface class ITestService in the Business namespace.
      2. Add a method Initialize(ILogger logger).
      3. Add a method TestLogger().
      4. Create an implementation class, TestService, in the Business namespace that implements the ITestService interface.
      5. Add a private class scoped variable logger of type ILogger.
      6. Implement the Initialize():

[InjectionMethod]

public void Initialize(ILogger logger)

{

this.logger = logger;

}

* + 1. Implement the Test():

public void TestLogger()

{

logger.Info("Test Logging in TestService.TestLogger() invoked.");

}

* 1. Register the Test Service with Unity:
     1. Open UnityConfig file located in the App\_Start folder of your project.
     2. Add the following code to the Application\_Start() to register the Logging Service with Unity.

container.RegisterType<ITestService, TestService>();

container.RegisterType<ITestService>(new InjectionMethod("Initialize", new MyLogger2()));

* 1. Create a Test Logging Service Controller:
     1. Add a new Controller called TestLoggingService3Controller.
     2. Create a private read-only class scoped variable called logger of type ILogger.
     3. Create a private read-only class scoped variable called service of type ITestService.
     4. Create a public non-default constructor that takes an ILogger method argument and a ITestService method argument.
     5. Save the constructor method arguments to the private logger class variables.
     6. Change the Index() method to return a string.
     7. In the implementation of the Index() call the Info() on the logger, then call the Test() on the service, and return a test string response.
     8. Run the application, and invoke the controller.
     9. Verify that the logger statement shows up in the Log file. Take a Screenshot. Label this Screenshot 3.

**Documentation**

All documentation will be submitted at the end of the activity to the learning management system. Ensure documentation of the following:

1. Create a project report using a GCU standard project header/cover page to include a header, your name, course, assignment name, and date.
2. Add screenshots of the following activities to your project report:
   1. Screenshot 1 of the Log message from Step 3
   2. Screenshot 2 of the Log message from Step 4.
   3. Screenshot 3 of the Log message from Step 5.
3. Upload your code to GIT, and include the URL of your GIT repository in the Project Report.

## Part 2: Application of Action Filters, IoC, and Logging

**Overview**

In this activity, students will apply previous activities to their team project. They will build a Security Filter, Logging Service, and use the Unity (IoC) Framework. These components will be integrated with their team project.

**Execution**

Execute this assignment according to the following guidelines:

1. Security Filter (reference Activity 4 Part 3: Action Filters):
   1. Implement a Security Authorization Filter to handle a redirect to your login page when a user has not been authenticated—this can be done by running a check on your security token in the session, and if not valid, simply calling “filterContext.Result = new RedirectResult("/Login");” from within your Filter.
2. Logging Service (reference Activity 4 Part 1 and Activity 5 Part 1):
   1. Create a Logging Service Interface class (with debug(), info(), warn(), and error() methods).
   2. Create a Logging Service Implementation class (use NLog).
3. Register Logging Service (reference Activity 5 Part 1):
   1. Bind a singleton to the Logging Service.
4. Team Project Integration:
   1. Add the Security Filter to your team project.
   2. Add the Logging Service to your team project.
   3. Create nondefault constructor() for your Login Controller and inject the Logging Service into the Controller. Add try catch exception blocks to the Login Controller. In the exception block, use the Logging Service to log the exception message at the error log level. Note, you should repeat this for all Controllers when you submit your final team project.

**Submission**

Submit the following to the learning management system:

**Part 1**

1. Project Report that includes:
   1. A GCU standard project header/cover page to include your name, course, assignment name, and date
   2. Screenshots 1-3
   3. URL of your GIT repository
2. Upload your code to the GIT repository.

**Part 2**

1. Zip up the code for the following activities to your project report:
   1. Security Filter
   2. Logging Service
   3. Login Controller