**Logging, WCF REST Services, Action Filters: Project Report**

Stephan Moncavage

Grand Canyon University

CST-247: Enterprise Applications III

Michael Pritchard

14 February 2021

GitHub: https://github.com/smoncavage/CST247\_EnterpriseApplications3.git

**CST-247 Activity Guide**

Contents

[Activity 4 2](#_Toc60388495)

[Part 1: Logging with nLog Framework 2](#_Toc60388496)

[Part 2: REST Services using WCF 4](#_Toc60388497)

[Part 3: Action Filters 10](#_Toc60388498)

# 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:

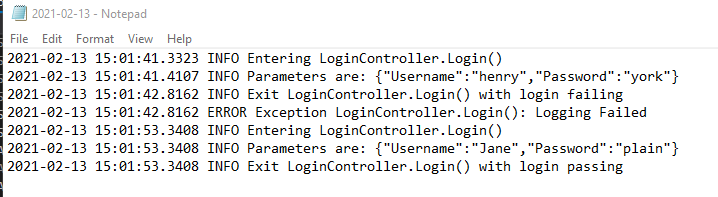
logger.Info(“Exit LoginControlle.DoLogin() with login passing”);

* + 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. 

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);

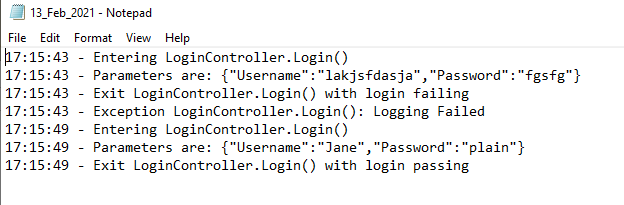
void Warning(string message)

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.
     1. 

**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}")]

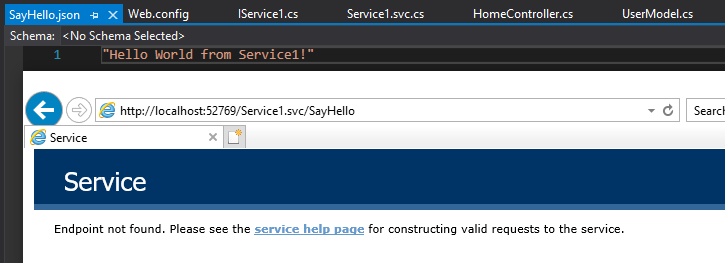
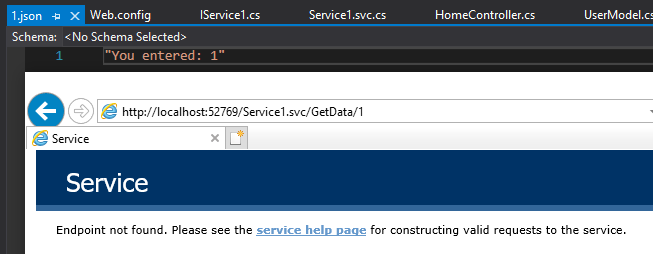
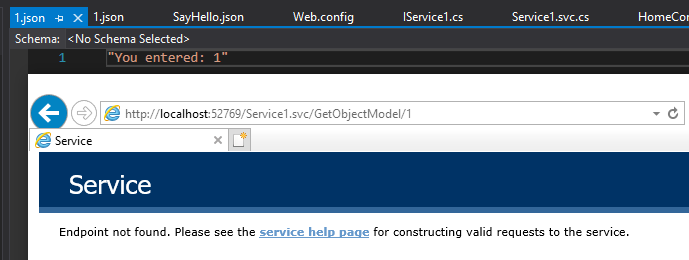
string GetData(string value);

[OperationContract]

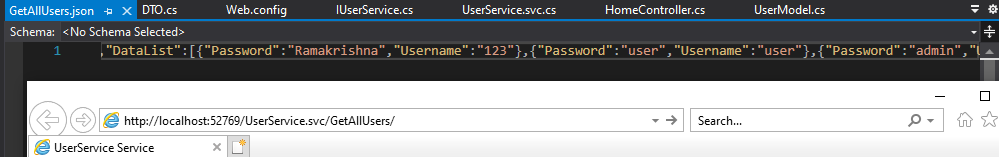
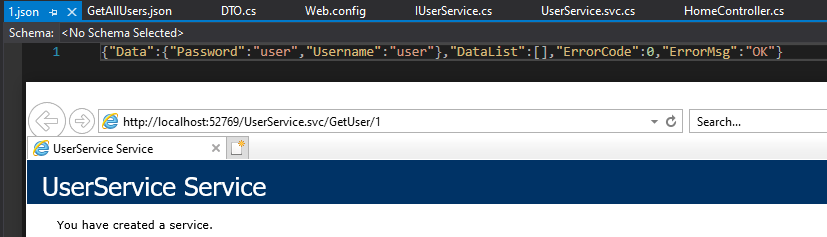
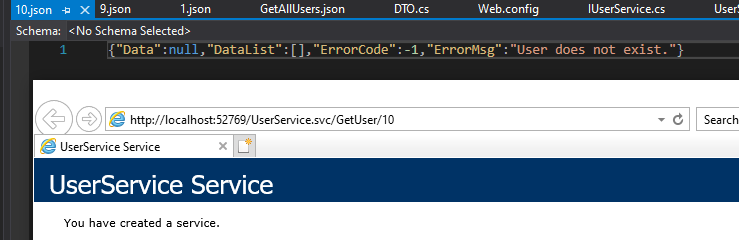
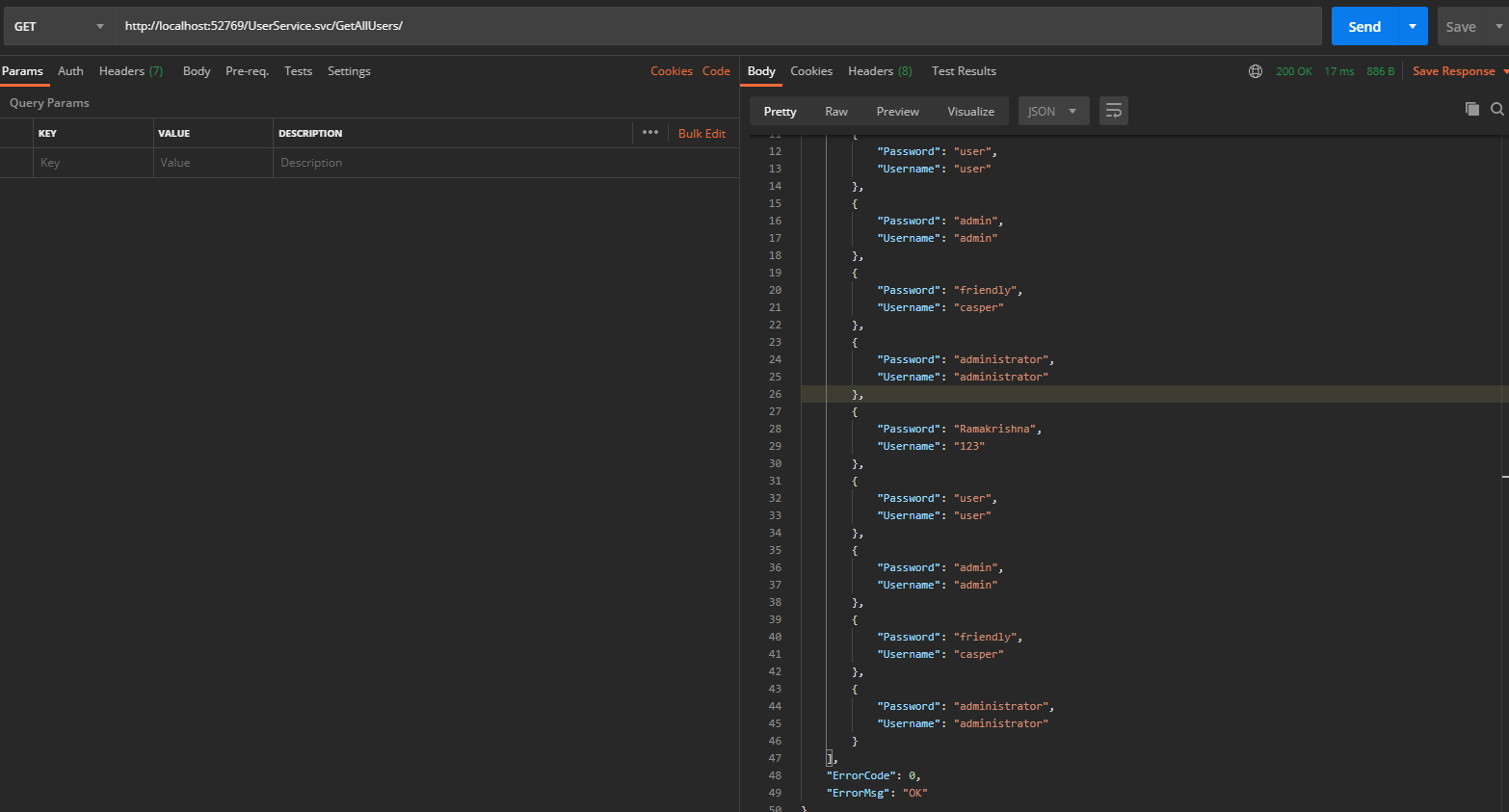
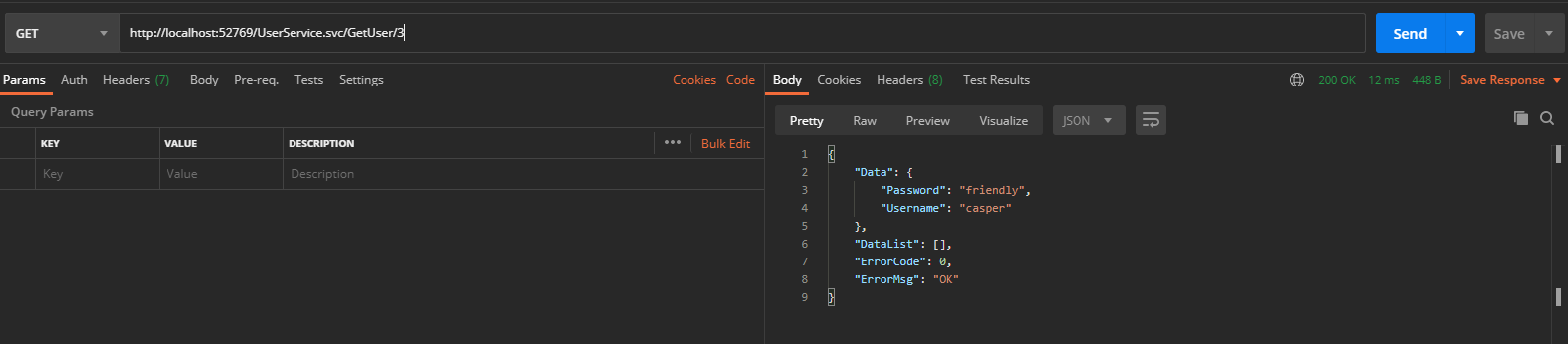
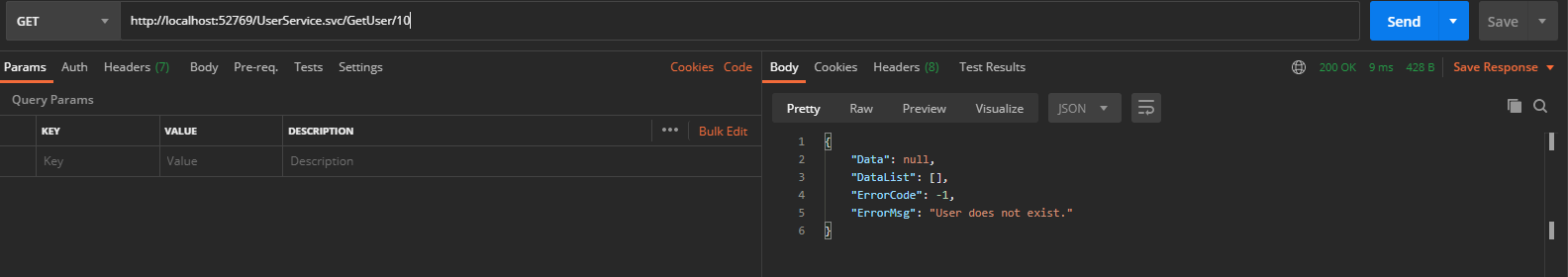
[WebGet(ResponseFormat = WebMessageFormat.Json, UriTemplate = "GetObjectModel/{id}")]

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.
      1. 
   3. Take a screenshot of localhost:[port]/Service1.svc/GetData/1. Label this Screenshot 6.
      1. 
   4. Take a screenshot of localhost:[port]/Service1.svc/GetObjectModel/1. Label this Screenshot 7.
      1. 

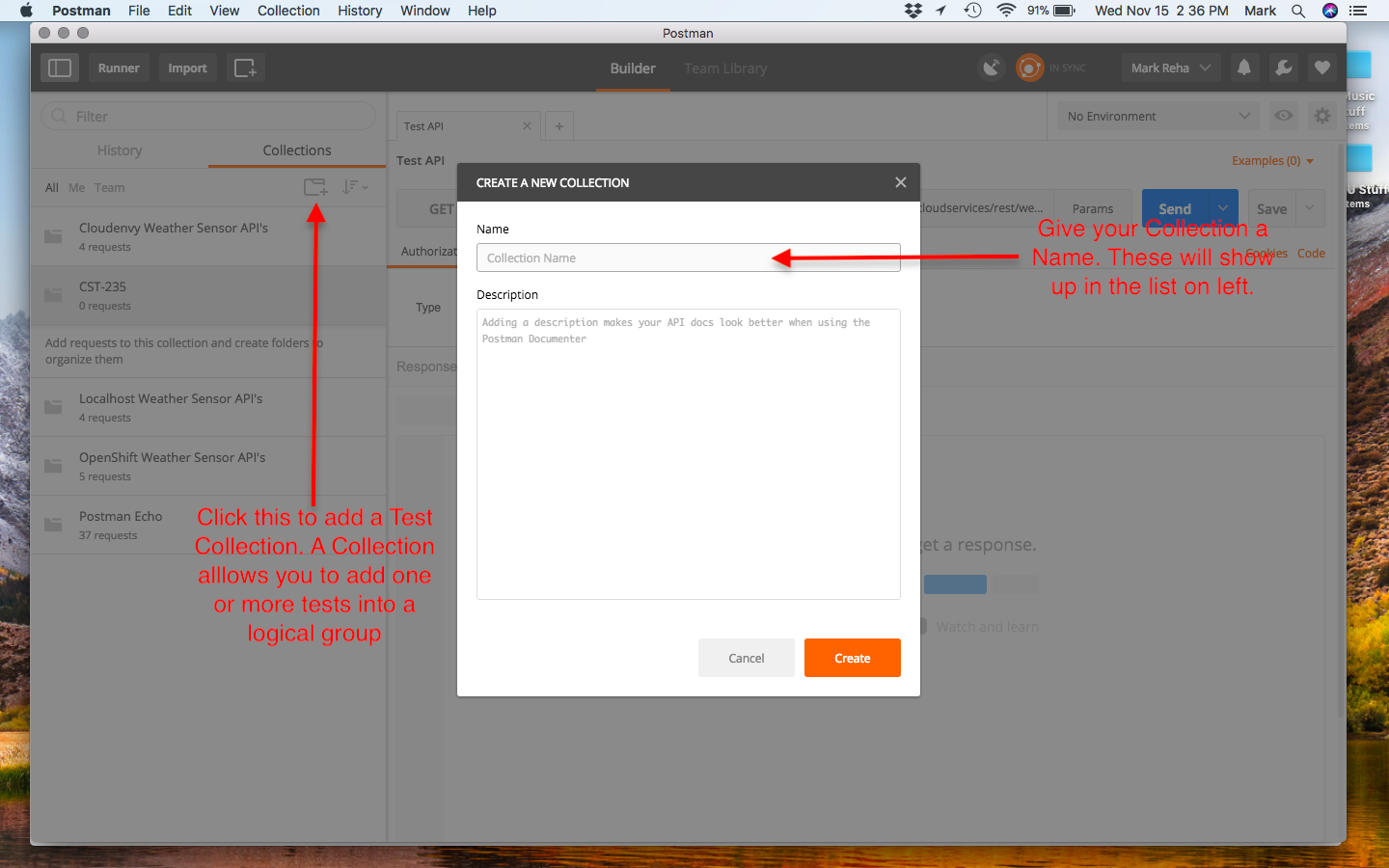
*Section 2: Create a "User" WCF REST Service*

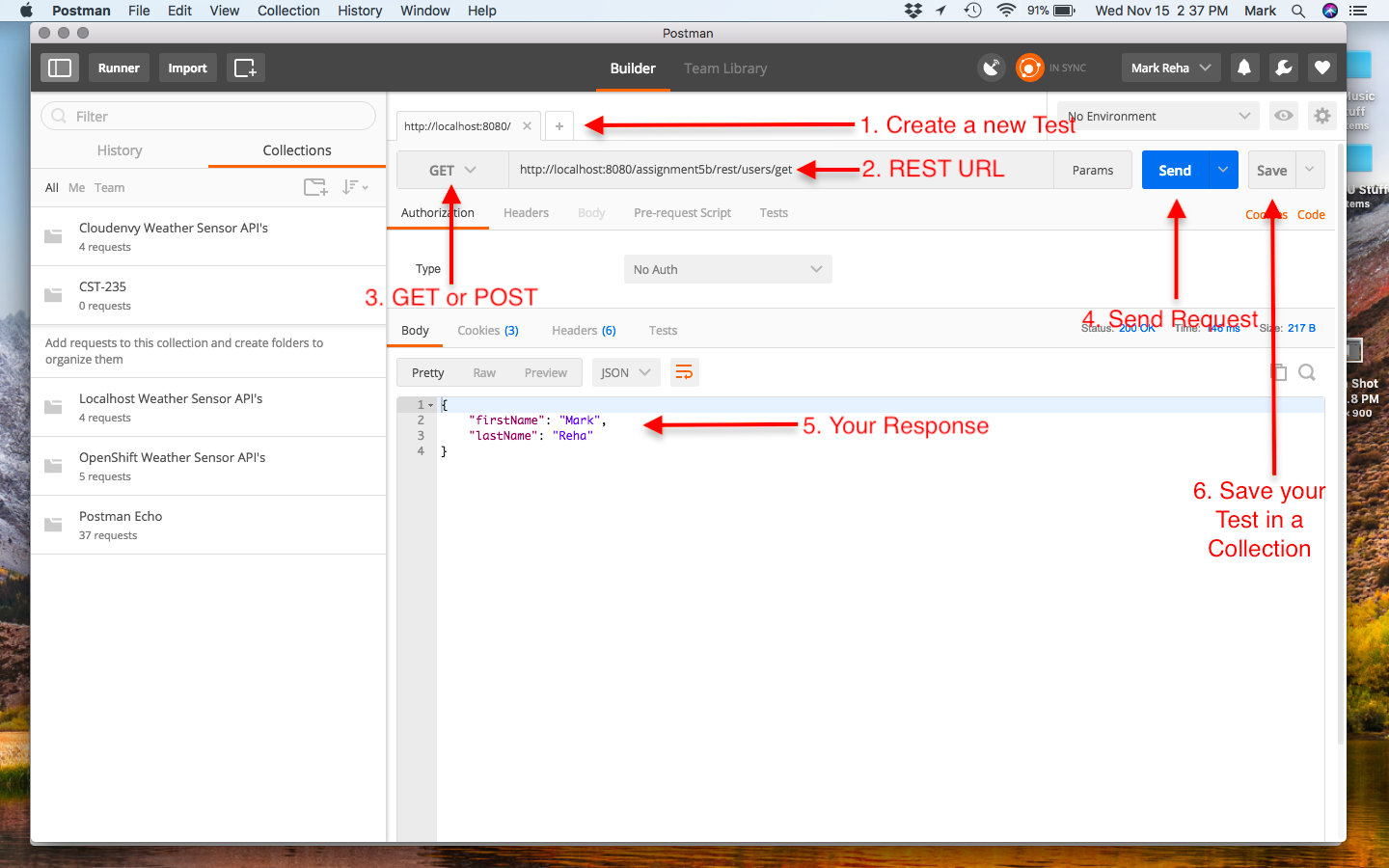
1. Refactor the Service Contract Interface and Service Class:
   1. Open up the Service Contract Interface class. Highlight the class name. Rename to IUserService. Right click. Select the Rename menu. Click OK.
   2. Click on the IService1.cs in your solution. Right click. Select the Rename menu. Enter 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.
      1. 
   3. Take a screenshot of localhost:[port]/UserService.svc/GetUser/1. Label this Screenshot 9.
      1. 
   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.
      1. 
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.
         1. 
      2. Take a screenshot of localhost:[port]/UserService.svc/GetUser/1. Label this Screenshot 12.
         1. 
      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.
         1. 
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?**

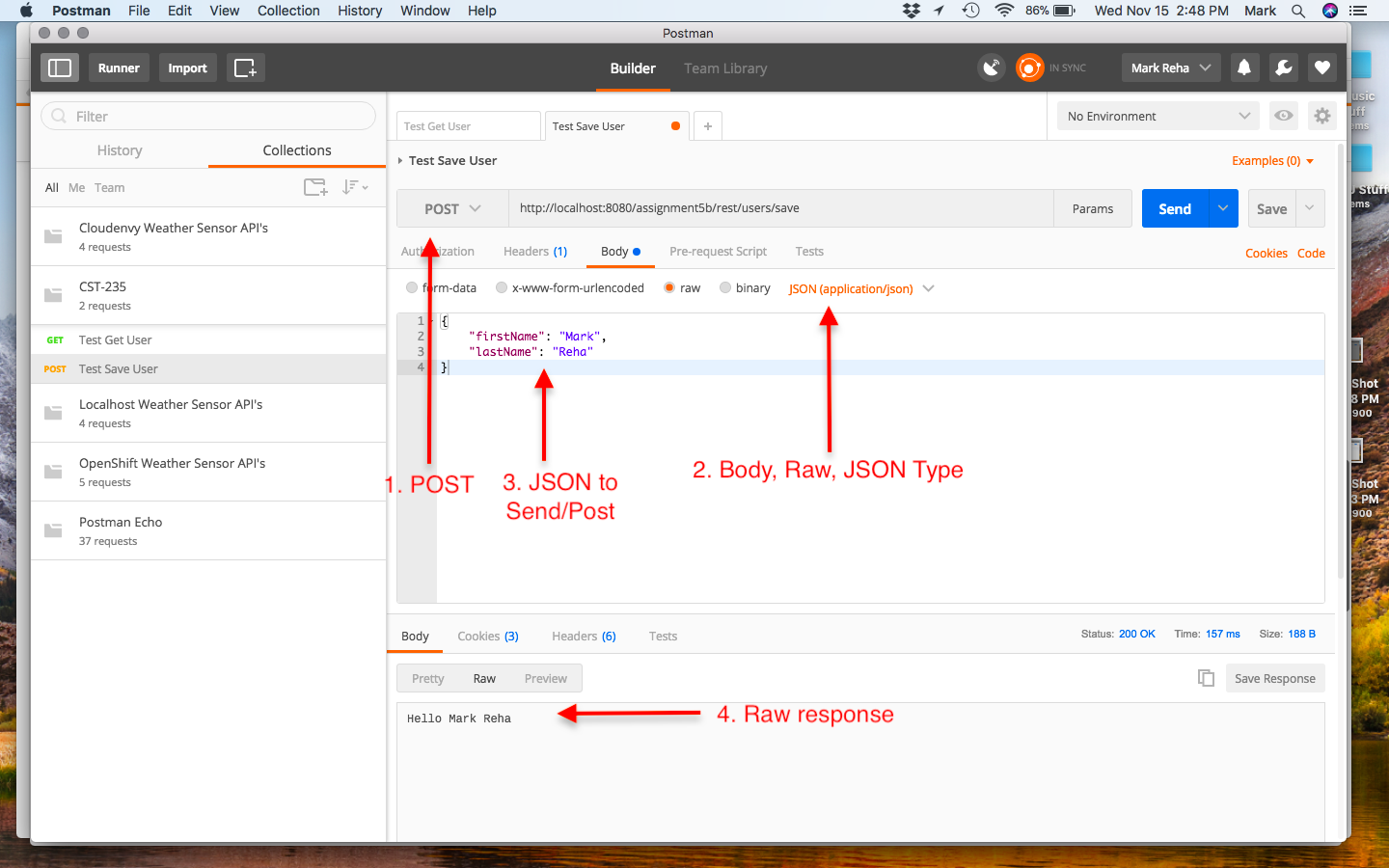
You can always use your browser for sending GET requests, but this is error prone, and you have to write down all your URLs and test scenarios for all your APIs. Postman is a very nice desktop tool for testing your REST APIs. Postman allows you to set up Collections of API Tests, and supports making 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.

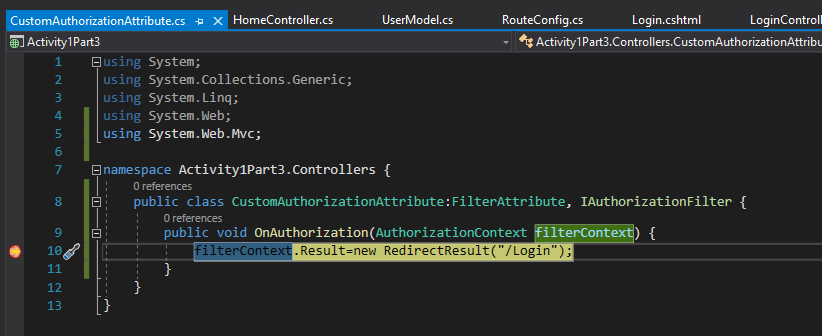
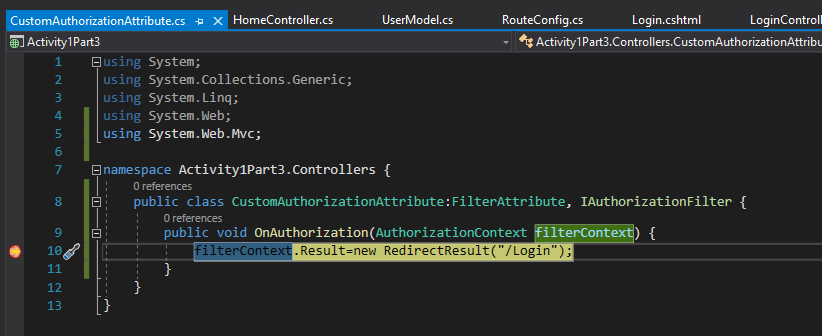
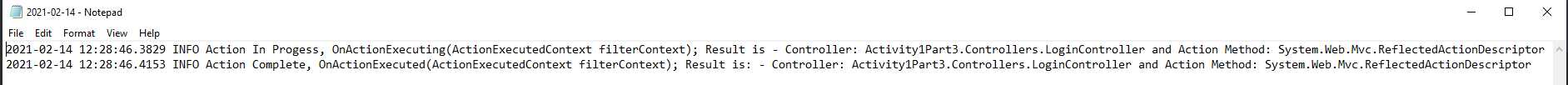
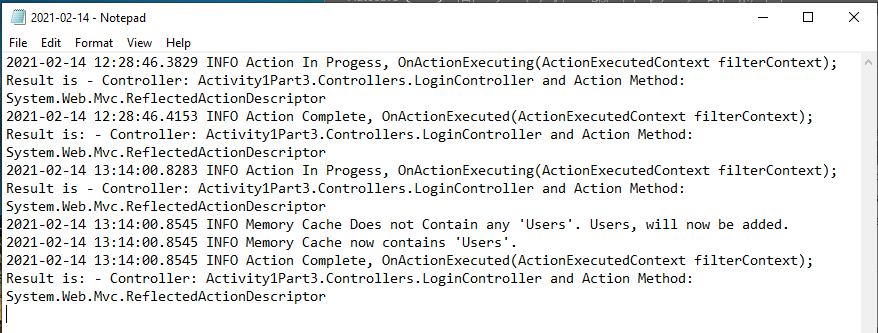
## Part 3: Action Filters

**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.
      1. 
   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.
      1. 
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.
      1. 
      2. 2021-02-14 12:28:46.3829 INFO Action In Progess, OnActionExecuting(ActionExecutedContext filterContext); Result is - Controller: Activity1Part3.Controllers.LoginController and Action Method: System.Web.Mvc.ReflectedActionDescriptor
      3. 2021-02-14 12:28:46.4153 INFO Action Complete, OnActionExecuted(ActionExecutedContext filterContext); Result is: - Controller: Activity1Part3.Controllers.LoginController and Action Method: System.Web.Mvc.ReflectedActionDescriptor
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.
      1. 

**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.