Hands-On Lab

Enhancing Asp.Net MVC Applications

Lab version: 1.0.0

Last updated: 3/12/2009



# Contents

[Overview 3](#_Toc224120009)

[Exercise 1: Using Form Posting in a MVC Web Application 5](#_Toc224120010)

[Task 1 – Using the ActionName Attribute 6](#_Toc224120011)

[Task 2 – Creating a Custom Model Binder 7](#_Toc224120012)

[Task 3 – Using the UpdateModel and TryUpdateModel Helper Methods 11](#_Toc224120013)

[Exercise 1: Verification 14](#_Toc224120014)

[Verification 1 14](#_Toc224120015)

[Verification 2 23](#_Toc224120016)

[Verification 3 26](#_Toc224120017)

[Exercise 2: Using Partial Views with ASP.NET AJAX 29](#_Toc224120018)

[Task 1 – Implementing the Customer Search Logic 29](#_Toc224120019)

[Task 2 – Adding AJAX to the Application 32](#_Toc224120020)

[Exercise 2: Verification 36](#_Toc224120021)

[Verification 1 36](#_Toc224120022)

[Exercise 3: Implementing Action Filters 37](#_Toc224120023)

[Task 1 – Implementing Action Filters 37](#_Toc224120024)

[Exercise 3: Verification 40](#_Toc224120025)

[Verification 1 40](#_Toc224120026)

[Summary 43](#_Toc224120027)

Overview

* 1. ASP.NET MVC enables you to build Model View Controller (MVC) applications by using the ASP.NET framework. ASP.NET MVC is an alternative, not a replacement, for ASP.NET Web Forms that offers the following benefits:
  + Clear separation of concerns
  + Testability - support for Test-Driven Development
  + Fine-grained control over HTML and JavaScript
  + Intuitive URLs
  1. This lab demonstrates some of the features available to enhance a sample ASP.NET MVC application.
  2. In form posting scenarios, the features that will be implemented in the sample application are:
  + The usage of the action name attribute to create controller's action methods with different names than the default ones
  + Model binders, which bind complex objects to input forms automatically
  + Helper methods to update a model object from the input forms' fields
  1. Then, the application will be refactored to make use of ASP.NET AJAX. By taking advantage of AJAX, the responsiveness and performance of the application is improved, because the rendering of an entire page is avoided when the update of only a certain region in a page is needed.
  2. Finally action filters to log the execution of the controller’s action methods will be implemented. An action filter is an attribute that you can apply to a controller action or an entire controller that modifies the way in which the action is executed.
  3. **Note:** For more information about ASP.NET MVC, see the following topics:

- The [Official ASP.NET MVC home page](http://www.asp.net/mvc/) where you will find tutorials, videos, code samples, and more.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Use form posting
  + Use ASP.NET AJAX within MVC using partial views
  + Implement Action Filters
  1. **Note:** This hands-on lab assumes that the user is familiar with ASP.NET MVC. For introductory concepts of ASP.NET MVC, see Introduction to ASP.NET MVC.



# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2008 SP1
  + Microsoft ASP.NET MVC 1.0
  + Microsoft SQL 2005 or Microsoft SQL 2008 (Express edition or above)
  + Adventure Works sample database
    - For Microsoft SQL 2005: [AdventureWorksLT.msi](http://www.codeplex.com/MSFTDBProdSamples/Release/ProjectReleases.aspx?ReleaseId=4004)
      1. **Note**: The msi just copies the sample database files to your file system; you have to manually attach the database to the SQL Server. For more information, see [How to: Attach a Database (SQL Server Management Studio)](http://technet.microsoft.com/en-us/library/ms190209.aspx).
      2. *Tip for Windows Vista Users*: Consider installing the database files to other location than "C:\Program Files\", as this this folder has extra security and you might not be able to save changes in project files as they will be read-only.
    - For Microsoft SQL 2008: [AdventureWorks 2008 sample databases](http://www.codeplex.com/MSFTDBProdSamples/Release/ProjectReleases.aspx?ReleaseId=18407)
      1. **Note:** The msi file will automatically install all the sample databases in your SQL Server. However, you will only use the **AdventureWorksLT** database.

# Setup

* 1. All the requisites for this lab are verified using the Dependency Checker. To make sure that everything is correctly configured, follow these steps:
  2. **Note:** To perform the setup steps you need to run the scripts in a command window with administrator privileges.
  3. Run the Dependency checker for the Training Kit if you haven't done it previously. To do this, run the **CheckDependencies.cmd** script located under the **%TrainingKitInstallationFolder%\Labs\enhancingAspNetMvcApp\Setup** folder. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.
     1. **Note:** For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **CheckDependencies.cmd** file launches the Visual Studio installer file that installs the code snippets.

Exercises

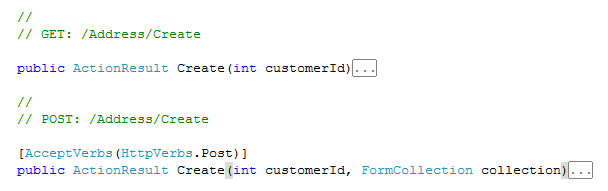
* 1. The following exercises compose this Hands-On Lab:
  2. Using Form Posting in a MVC Web
  3. Using Partial Views with ASP.NET AJAX
  4. Implementing Action Filters
  5. Estimated time to complete this lab: **90 minutes**.
  6. **Note:** Each exercise is accompanied by a starting solution. Some code sections are missing from these solutions, which will be completed through each exercise. Therefore the starting solutions will not work if you run them directly.
  7. Inside each exercise you will also find an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Exercise 1: Using Form Posting in a MVC Web Application

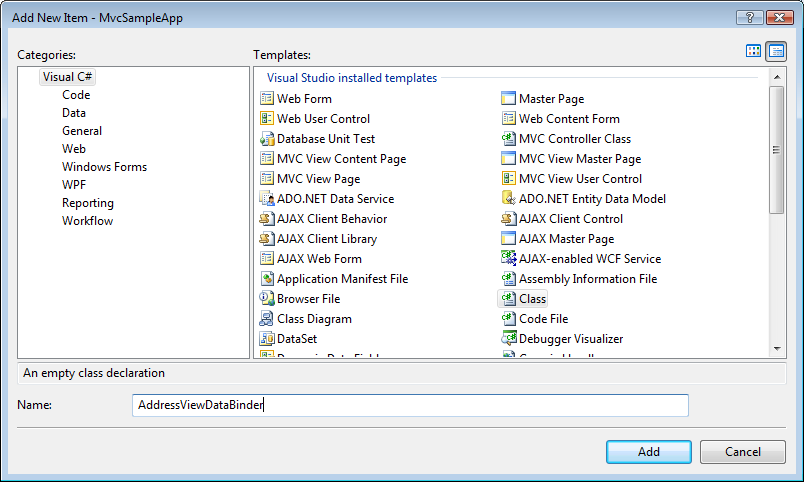
* 1. In this exercise, you will learn how to work with form posting scenarios in a Web application created with ASP.NET MVC.
  2. ASP.NET MVC 1.0 includes the following new features:
  + The ability to publish a single action URL and dispatch it differently depending on the HTTP Verb (GET or POST)
  + Model Binders that allow rich parameter objects to be constructed from form input values
  + Helper methods that enable incoming form input values to be mapped to existing model object instances within action methods
  1. **Note:** Form posting scenarios are those where a <form> section is defined on the HTML page which contains all the input fields. The client requests to the server are done with the POST verb (usually by declaring a **submit** button type inside the <form> tag).
  2. In the following tasks you will learn how each of these features can be implemented on a form posting scenario and the benefits of using them.

Task 1 – Using the ActionName Attribute

In this task you will use the **ActionName** attribute to allow the method name implementation on your controller class to be different from the published URL.

* 1. **Note:** For example, rather than having two overloaded **Create** methods in your controller, you may instead want to have the POST method named **Save**. In this case you can apply the **[ActionName("Create")]** attribute in order to keep the same URL for both actions but with different method names.
  2. Open Microsoft Visual Studio 2008 from **Start | All Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008**.
  3. Open the solution file **MvcSampleApp.sln** located under the **enhancingAspNetMvcApp\Ex01-** **FormPosting\begin** folder.
  4. Open the **AddressController** class. To do this, in the **Solution Explorer**, double click the **AddressController.cs** file, located inside the **Controllers** folder.
  5. Notice that the class contains two **Create** overloaded action methods.
     1. 
     2. Figure
     3. Viewing the overloads of Create method
     4. **Note:** Typically, MVC applications have two action methods to handle create or update scenarios. One for returning the html view that displays the initial empty form, and another to save the inputted data when the form is submitted.
     5. The downside to using two actions in those scenarios is that you need to publish two URLs from your site. For example, "*/Address/Create*" and "*/Address/Save*". The problem with this comes when you need to redisplay the HTML form due to an input error, because the URL of the redisplayed form in the scenario will be "*/Address/Save*" instead of "*/Address/Create*". This is because **Save** was the URL the form was posted to. To summarize, you are in the **Create** address and submit the form, and then you are redirected to the **Save** URL where the data is validated and errors are shown. After that, if there were errors while saving the data, you will be redirected again to the **Create** URL to fix the inputs. One of the problems with this approach is that requires two additional redirections.
  6. Notice that the second action method has the **AcceptVerbs** attribute applied with a **Post** value passed as a parameter. The **AcceptVerbs** attribute allows specifying how the action methods will be dispatched. For example, you can declare an action method that will be called on GET scenarios and another one that will be called on POST scenarios.
     1. **Note:** The usage of the **AcceptVerbs** attribute permits publishing just a single URL (in this case: */Address/Create*). Besides, the handling automatically varies based on the incoming HTTP verb.
     2. **Note:** Both action methods have the same name—**Create**—but different overloads. The method used in POST scenarios has an additional parameter of type **FormCollection**. This parameter is not used in the code, as it is only provided to have a different method signature.
  7. Use the **ActionName** attribute to allow changing the **Create** (for POST) method name. This allows you to prescind from the overloads, and remove the unused parameter from the method signature. To do this, replace the **Create** action method and attribute (for POST) with the following code:
     1. C#
     2. **[AcceptVerbs(HttpVerbs.Post), ActionName("Create")]**
     3. **public ActionResult SaveNew(int customerId)**
     4. **Note:** By adding the **ActionName** attribute, despite that the method name is **SaveNew()**,the action method will handle requests for **Create** actions in POST operations**.**

Task 2 – Creating a Custom Model Binder

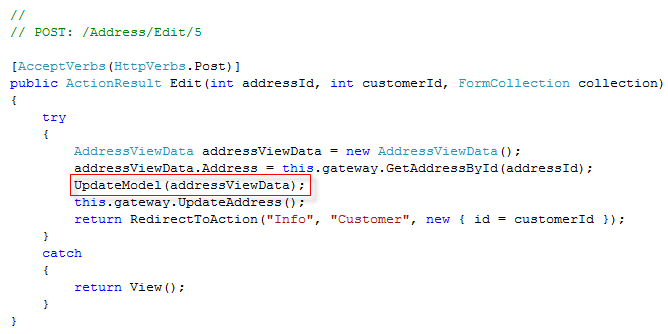
* 1. In this task you will learn how to implement a custom Model Binder which automatically constructs rich parameter objects from form input values. In contrast with previous versions of ASP.NET MVC in which the user had to manually construct the model object (for example, *product* or *address*) based on the input values (specified as method arguments); this new feature provides a way for complex types to be de-serialized from the incoming HTTP input, and passed to a **Controller** action method as arguments or populated through the **UpdateModel** / **TryUpdateModel** methods (this is shown in the current exercise). In other words, the action methods now receive the full model object rather than the input values as separated arguments.
  2. Model Binders can be registered at 4 different levels, and its scope will be different depending where they were registered:
  3. As an attribute applied to the Controller's action method's argument. In this case the model binder will only be used when de-serializing the HTTP input to the action method argument.
  4. As an attribute applied to the type of a model. When registered at this level, the Model Binder will be used each time an instance of the type has to be de-serialized.
  5. On the **ModelBinders.Binders** dictionary on the **Application\_Start** method of the **MvcApplication** class. As on the previous level, defining a Model Binder on this level will affect the application globally, automatically using this binder each time an instance of the model type needs to be de-serialized.
  6. As the default binder, a binder can be registered to be used when a type-specific model binder can't be found, and in this case the registration affects the application globally.
  7. **Note:** Model Binders also provide support for handling input exceptions, and make it easier to redisplay forms when errors occur (without requiring the end-user to have to re-enter all their data again).
  8. Create a folder named **Binders**. To do this, in the **Solution Explorer**, right click the **MvcSampleApp** project, point to **Add**, click **New Folder** and set the folder name to **Binders**.
  9. Create the **AddressViewDataBinder** class in the **Binders** folder. To do this, right click the **Binders** folder, point to **Add** and select **Class.** Set the class name to **AddressViewDataBinder**.
     1. 
     2. Figure
     3. Adding the AddressViewDataBinder class
  10. Open the **AddressViewDataBinder.cs** file and replace the using statements at the top of the file with the following ones:
      1. C#
      2. **using System.Web.Mvc;**
      3. **using MvcSampleApp.Models;**
      4. **using MvcSampleApp.ViewData;**
  11. Change the class signature to implement the **IModelBinder** interface, as shown in the following code.
      1. C#
      2. public class AddressViewDataBinder **: IModelBinder**
      3. {
      4. }
      5. **Note:** Model binders in ASP.NET MVC are classes that implement the **IModelBinder** interface, and can be used to help manage the binding of types to input parameters. A model binder can be written to work against a specific object type, or can alternatively be used to handle a broad range of types. The **IModelBinder** interface allows you to unit test binders independent of the web-server or any specific controller implementation.
  12. Add the implementation of the **BindModel** method that will be the entry point for the model binder.
      1. (Code Snippet – *Enhancing Asp.Net MVC Lab – AddressViewDataBinder BindModel Method*)
      2. C#
      3. public class AddressViewDataBinder : IModelBinder
      4. {
      5. **public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)**
      6. **{**
      7. **AddressViewData addressViewData = (AddressViewData)bindingContext.Model;**
      8. **if (addressViewData.Address == null)**
      9. **{**
      10. **addressViewData.Address = new Address();**
      11. **}**
      12. **addressViewData.CustomerId = GetValueFromProvider<int>(bindingContext, "CustomerId");**
      13. **return RetrieveAddressViewDataFromContext(addressViewData, bindingContext);**
      14. }
  13. Add the **RetrieveAddressViewDataFromContext** method that will populate the **AddressViewData** view data from the binding context.
      1. (Code Snippet – *Enhancing Asp.Net MVC Lab – AddressViewDataBinder RetrieveAddressViewDataFromContext Method*)
      2. C#
      3. public class AddressViewDataBinder : IModelBinder
      4. {
      5. ...
      6. **private static AddressViewData RetrieveAddressViewDataFromContext(AddressViewData addressViewData, ModelBindingContext bindingContext)**
      7. **{**
      8. **if (bindingContext.PropertyFilter("Address.AddressLine1"))**
      9. **{**
      10. **addressViewData.Address.AddressLine1 = GetValueFromProvider<string>(bindingContext, "Address.AddressLine1");**
      11. **}**
      12. **if (bindingContext.PropertyFilter("Address.AddressLine2"))**
      13. **{**
      14. **addressViewData.Address.AddressLine2 = GetValueFromProvider<string>(**
      15. **bindingContext, "Address.AddressLine2");**
      16. **}**
      17. **if (bindingContext.PropertyFilter("Address.City"))**
      18. **{**
      19. **addressViewData.Address.City = GetValueFromProvider<string>(bindingContext, "Address.City");**
      20. **}**
      21. **if (bindingContext.PropertyFilter("Address.CountryRegion"))**
      22. **{**
      23. **addressViewData.Address.CountryRegion = GetValueFromProvider<string>(**
      24. **bindingContext, "Address.CountryRegion");**
      25. **}**
      26. **if (bindingContext.PropertyFilter("Address.PostalCode"))**
      27. **{**
      28. **addressViewData.Address.PostalCode = GetValueFromProvider<string>(bindingContext, "Address.PostalCode");**
      29. **}**
      30. **if (bindingContext.PropertyFilter("Address.StateProvince"))**
      31. **{**
      32. **addressViewData.Address.StateProvince = GetValueFromProvider<string>(**
      33. **bindingContext, "Address.StateProvince");**
      34. **}**
      35. **return addressViewData;**
      36. **}**
      37. }
  14. Add the **GetValueFromProvider** helper method that gets the values from the binding context as a string and converts them to the appropriate type. This is shown in the following bolded code.

(Code Snippet – *Enhancing Asp.Net MVC Lab – AddressViewDataBinder GetValueFromProvider Method*)

* + 1. C#
    2. public class AddressViewDataBinder : IModelBinder
    3. {
    4. ...
    5. **private static T GetValueFromProvider<T>(ModelBindingContext bindingContext, string key)**
    6. **{**
    7. **ValueProviderResult valueProviderResult;**
    8. **bindingContext.ValueProvider.TryGetValue(key, out valueProviderResult);**
    9. **if (valueProviderResult != null)**
    10. **{**
    11. **return (T)valueProviderResult.ConvertTo(typeof(T));**
    12. **}**
    13. **return default(T);**
    14. **}**
    15. }
  1. Register the model binder in the **Global.asax** file. To do this, open the **Global.asax** file, and add the following bolded code inside the **Application\_Start** method.
     1. C#
     2. protected void Application\_Start()
     3. {
     4. RegisterRoutes(RouteTable.Routes);
     5. **ModelBinders.Binders.Add(typeof(AddressViewData), new AddressViewDataBinder());**
     6. }
     7. **Note:** ASP.NET MVC also supports the ability to register binders at application startup using the **ModelBinders.Binders** collection. This is useful when you want to use a type written by a third party (that you can't annotate) or if you don't want to add a binder attribute annotation on your model object directly. Additionally, binders can be registered at other levels within an MVC application, such as a parameter attribute, or as an attribute on the type of the parameter being passed to the action method.

Task 3 – Using the UpdateModel and TryUpdateModel Helper Methods

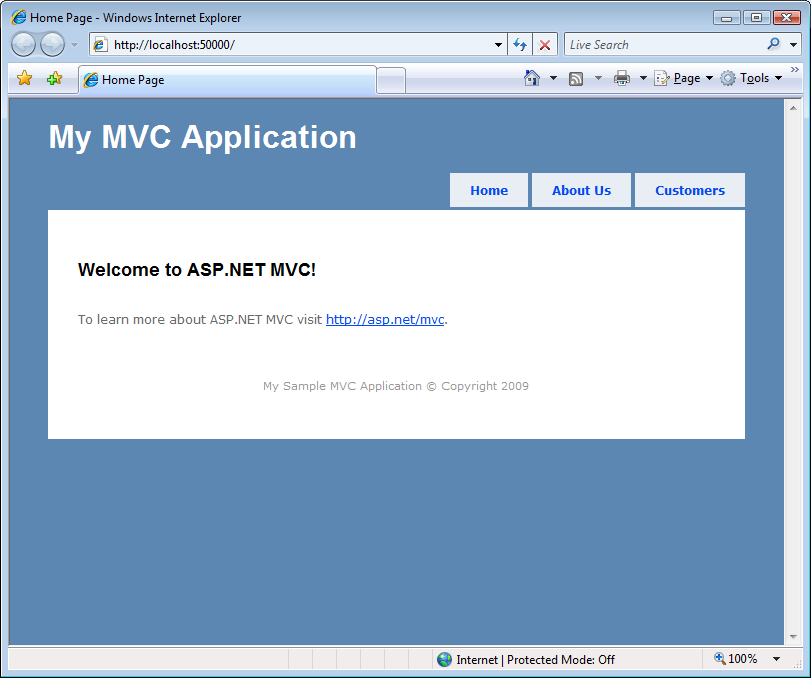
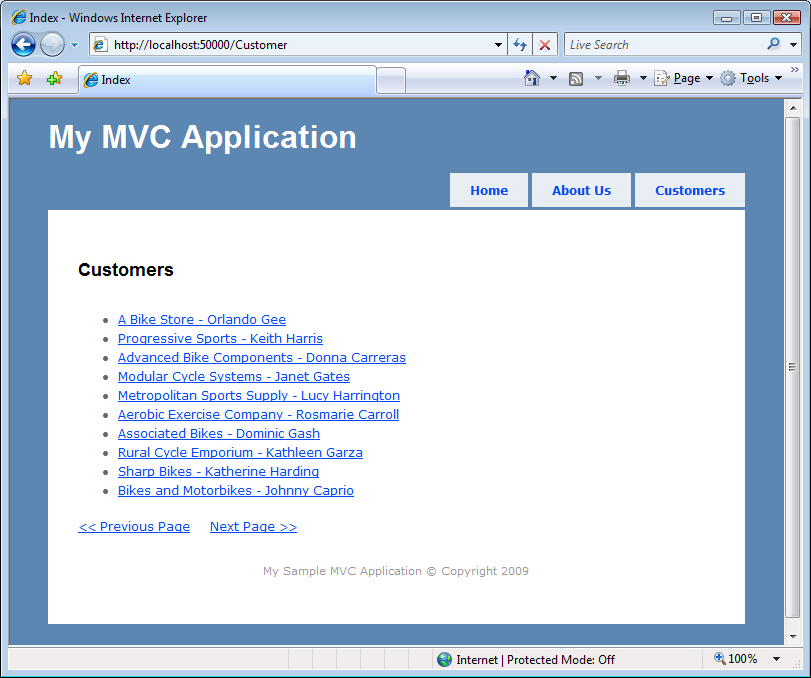
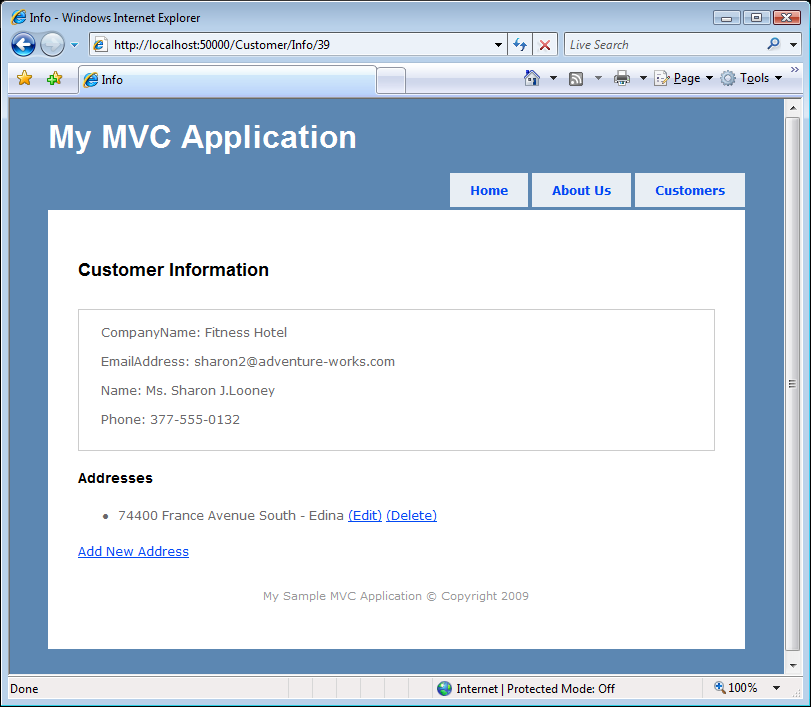
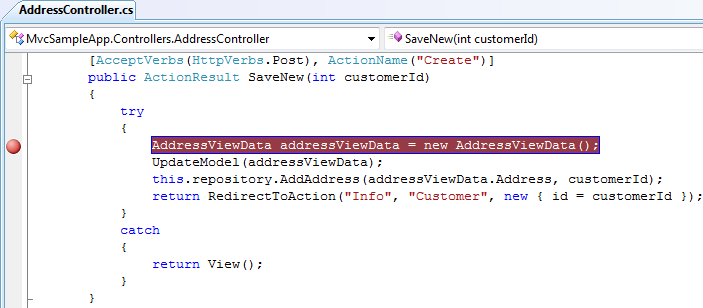
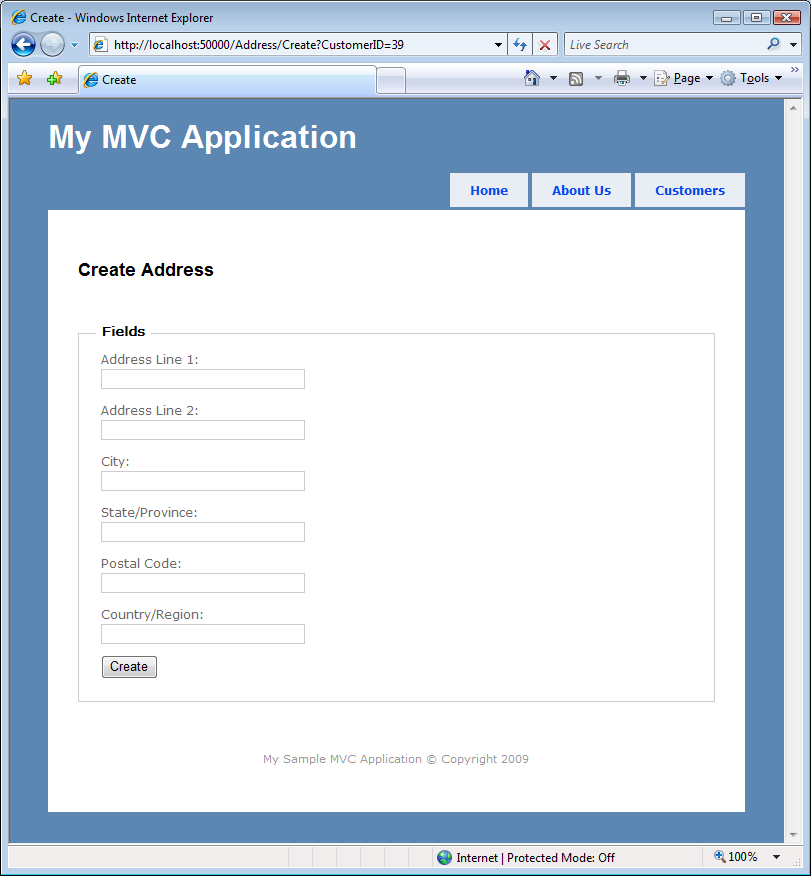
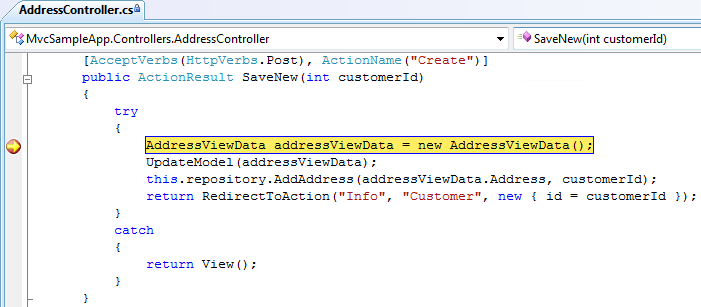
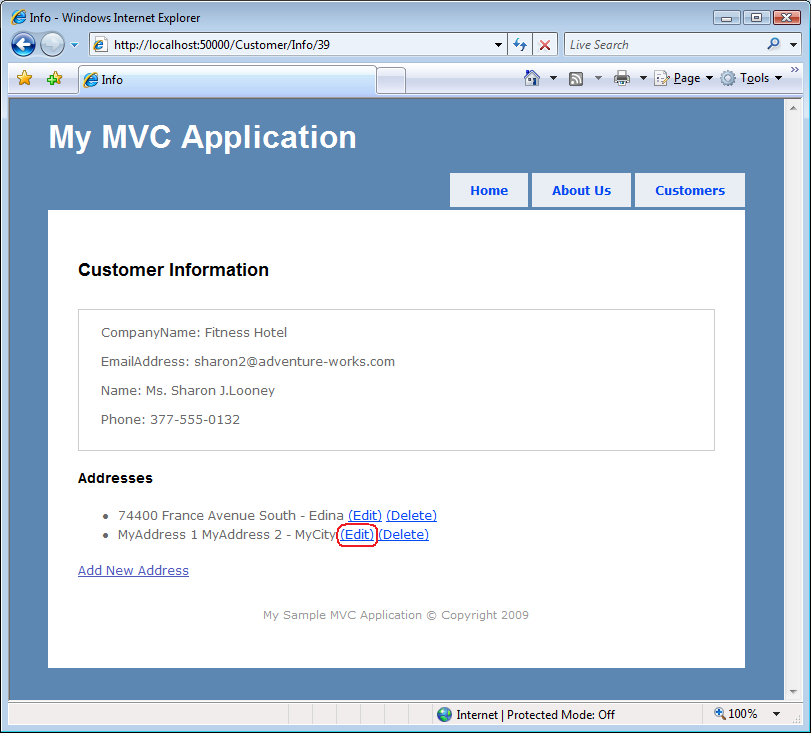
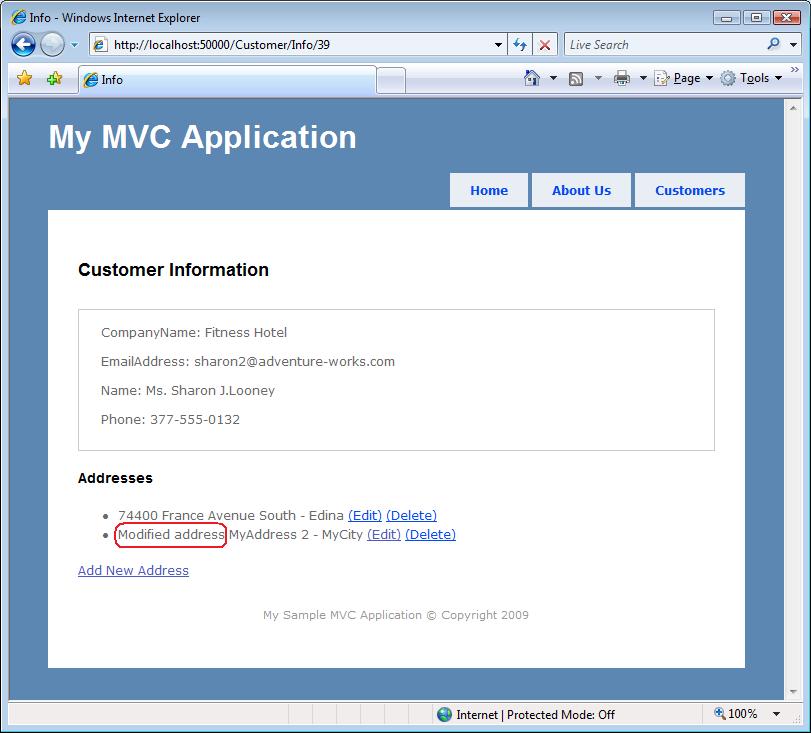
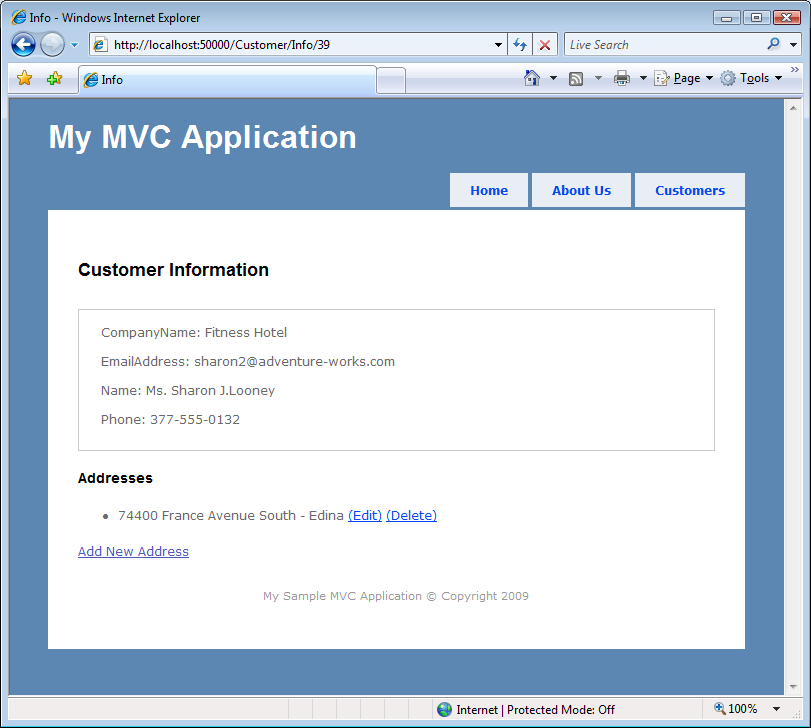
In this task you will learn how to use the **UpdateModel** and **TryUpdateModel** helper methods that enable incoming form input values to be mapped to existing model object instances within action methods.

* 1. **Note:** The **ModelBinder** support explained previously is great for scenarios where you want to instantiate new objects and pass them in as arguments to a controller action method. There are also scenarios, though, when you want to be able to bind input values to existing object instances that you own retrieving/creating yourself within the action method. For example, when enabling an edit scenario for an existing product in the database, you might want to use an ORM to retrieve an existing product instance from the database first within your action method, then bind the new input values to the retrieved product instance, and then save the changes back to the database.
  2. Open the **AddressController** class. To do this, in the **Solution Explorer**, double click the **AddressController.cs** file, located inside the **Controllers** folder.
  3. Look for the **Edit** action method that handles POST operations (specifically the one that has the **AcceptVerbs(HttpVerbs.Post)** attribute applied).
     1. Notice that the **UpdateModel** helper method is used in the method’s body.
     2. 
     3. Figure
     4. Calling the UpdateModel method in the Edit action method
     5. **Note:** The **UpdateModel** helper method allows you to pass in an existing object instance as the first argument, and optionally as a second argument you pass in a security white-list of specific properties you want to update using the form post values. The **UpdateModel** method will attempt to update all of the properties in the list. If you do not provide this list, the default behavior is to update all the properties of the model. If any of them fail, the method will raise an exception.
  4. Replace the line of code where the **UpdateModel** method is invoked, with a **TryUpdateModel** call and store the result value in a **Boolean** variable. Also create a variable to store the value providers, which can be get from the **FormCollection** instance. The updated code is shown in the following bolded code:
     1. C#
     2. [AcceptVerbs(HttpVerbs.Post)]
     3. public ActionResult Edit(int addressId, int customerId, FormCollection collection)
     4. {
     5. try
     6. {
     7. AddressViewData addressViewData = new AddressViewData();
     8. addressViewData.Address = this.repository.GetAddressById(addressId);
     9. **IDictionary<string, ValueProviderResult> valueProviders = collection.ToValueProvider();**
     10. **bool updateModelResult = TryUpdateModel(addressViewData, null, null, new[] { "Address.CountryRegion" }, valueProviders);**
     11. }
     12. catch
     13. {
     14. return View();
     15. }
     16. }
     17. **Note:** The **TryUpdateModel** helper method has the same purpose as the **UpdateModel** method. The major difference between them is that the **TryUpdateModel** returns a **Boolean** value of false if the mapping of properties fails, and true if everything worked well, while **UpdateModel** does not have a return value, and always throws an exception when the mapping fails.
     18. Nevertheless, they both work the same way, updating all properties whose mapping does not fail even if some property update do fail.  
         Choose whichever method works best with your error handling preferences.
     19. Notice that you are using an overload of the **TryUpdateModel** method which accepts the following parameters:
     + **Model**: The model instance to update from the input form. In this case, the addressViewData view data.
     + **Prefix**: Prefix to use when looking up the values in the value provider.
     + **IncludeProperties**: The list of properties to include in the update, if different than null **TryUpdateModel** will update only the listed properties.
     + **ExcludeProperties**: The list of properties to explicitly exclude from the update. These properties will be excluded, even if they are in the **IncludeProperties** list.
     + **ValueProviders**: A dictionary of value providers which are used internally to convert the values from the request to the properties of **Model**.
     1. **Note:** In the preceding code the **CountryRegion** property was passed to the **TryUpdateModel** method in the **ExcludeProperties** parameter. Therefore changes to this property will not be impacted in the model. This could be, for example, as the result of a business rule where you cannot change the country of an existing address.
  5. Add a conditional clause below the previously inserted code to impact the changes in the database or not, depending on the result of the **TryUpdateModel** method. To do this, replace the two lines of code where the **UpdateAddress()** and **RedirectToAction()** methods are invoked, with the following code:
     1. C#
     2. **if (!updateModelResult)**
     3. **{**
     4. **return View(addressViewData);**
     5. **}**
     6. **else**
     7. **{**
     8. **this.repository.UpdateAddress();**
     9. **return RedirectToAction("Info", "Customer", new { id = customerId });**
     10. **}**
     11. **Note:** As seen in the preceding code, when the **TryUpdateModel** operation returns true—no errors in the mapping—you will be redirected to the customer info page after the database is updated.

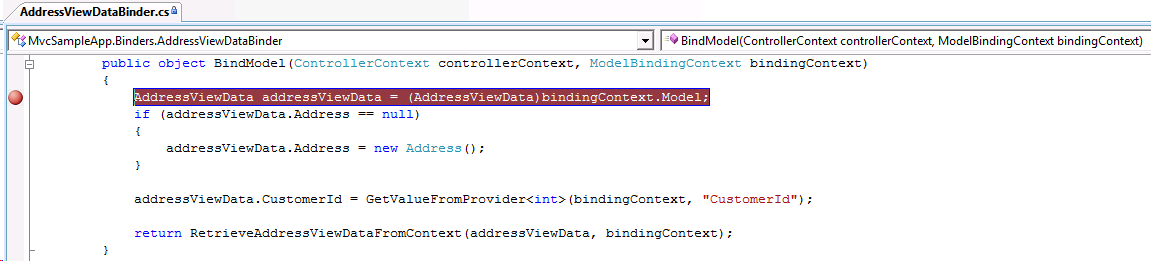
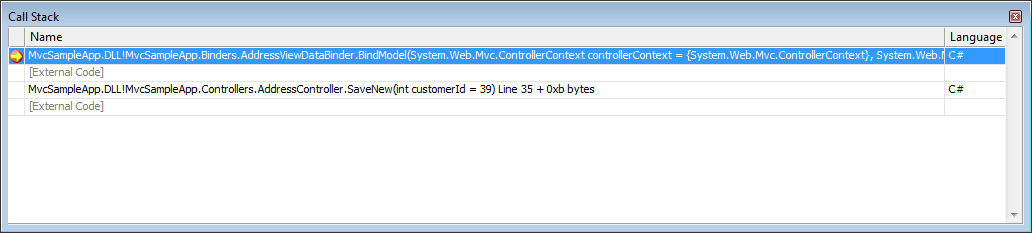
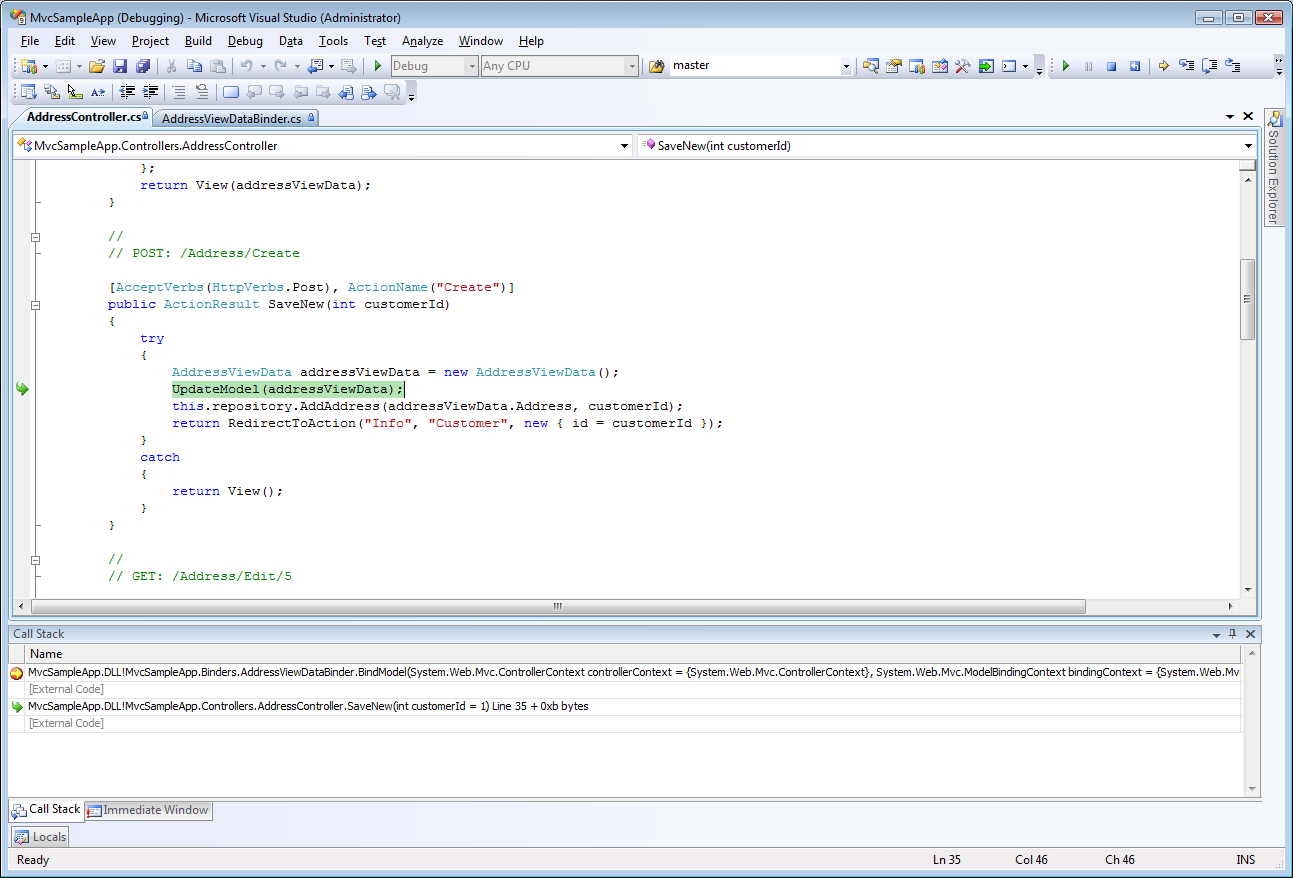
## Exercise 1: Verification

* 1. In order to verify that you have correctly performed all steps of exercise one, proceed as follows:

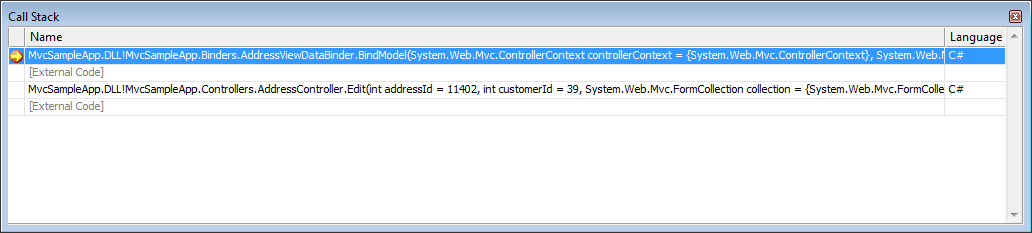
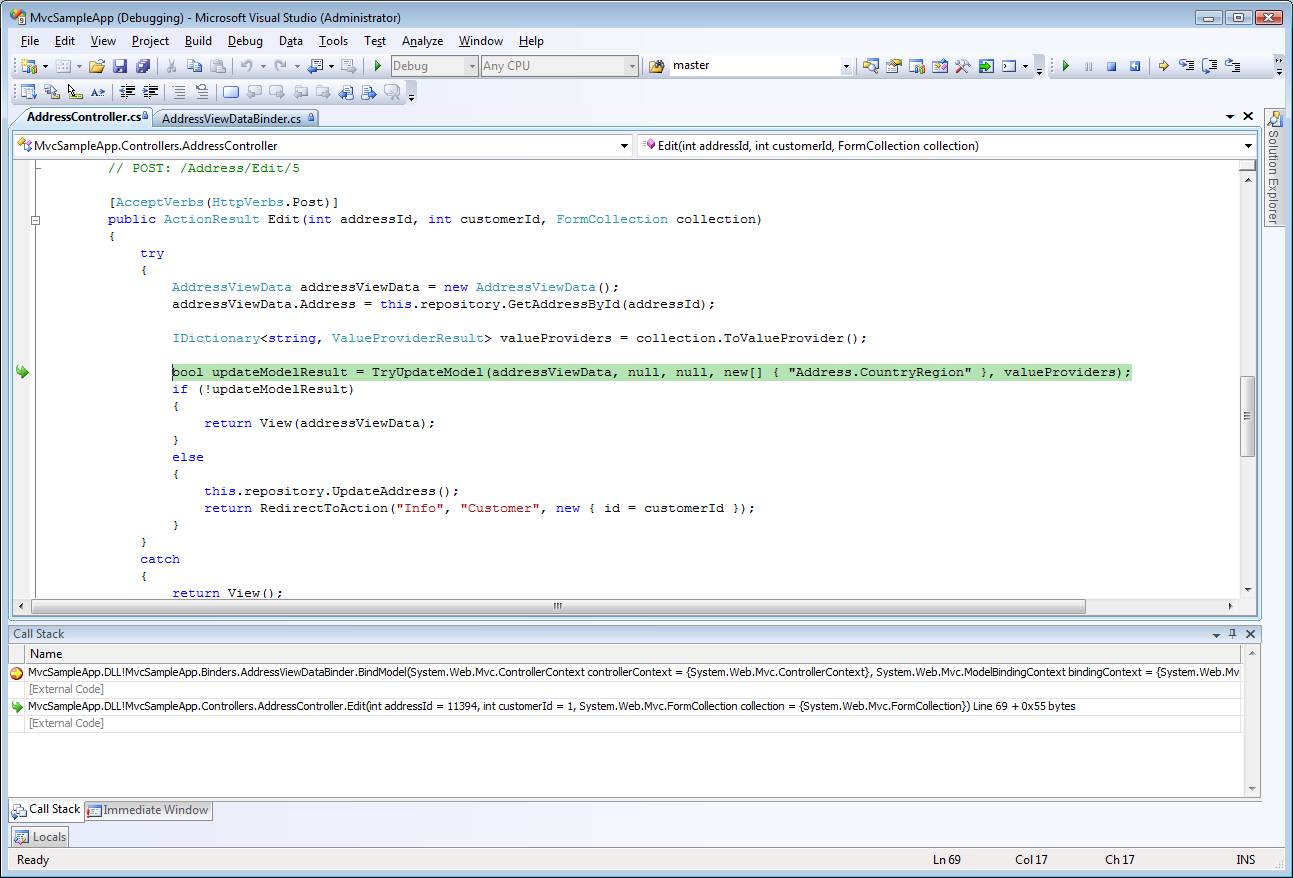
### Verification 1

* 1. You will execute CRUD operations with the simple application to make sure that they work in the same fashion as the starting solution, but with the newly implemented features.
  2. To test the MVC Web Application you need to start a new instance of the **MvcSampleApp** project. To do this, in **Solution Explorer** right-click **MvcSampleApp** project, point to **Debug** and select **Start New Instance**.
     1. **Note:** If the dialog **Debugging Not Enabled** appears, select **Modify the Web.config file to enable debugging** and click **OK**.
  3. You will be directed to[**http://localhost:50000/**](http://localhost:50000/). This is the home page of the MVC Web Application, and it is generated by the ASP.NET MVC framework by calling Home Controller’s **Index** methodand then rendering the Home controller’s **Index** View.
     1. **Note:** In this case **Index** method only renders the view, since no data is needed by the Home controller’s **Index** View.
     2. 
     3. Figure
     4. MVC Sample Application home page
  4. To browse customers, click **Customers** link on page header. You will be redirected to the following address in the web browser <http://localhost:50000/Customers> which invokes **Index** method on the **CustomerController**. The following output should appear. Previous and Next Page links are provided to navigate through all customers. The page showed by default is page one.
     1. 
     2. Figure
     3. Customers Index page
  5. To view a specific customer’s details, click on its name. For example if you click the "Fitness Hotel - Sharon Looney" customer (on the third page of the list), you will be redirected to <http://localhost:50000/Customers/Info/39> (where "39" is the **CustomerId**). The **Info** method of the **CustomerController** controller is invoked, which retrieves customer data, and pass it to the **Info** view when rendering it. The details on the given customer and its currently associated addresses will be shown. The output will be similar to the following.
     1. 
     2. Figure
     3. Customer Info page
  6. Set a breakpoint on the first line of the **SaveNew** method of the **AddressController** controller, as shown in Figure 7:
     1. 
     2. Figure
     3. Breakpoint in SaveNew method
  7. To create a new address for the selected customer, click the **Add New Address** link. You will be redirected to <http://localhost:50000/Address/Create?CustomerID=??> (where **??** is the customer ID.
     1. 
     2. Figure
     3. Create Address page
  8. Complete the required information and click the **Create** button. The application will stop on the breakpoint set on the **SaveNew** method, verifying that the **ActionName** attribute is directing the Create action when posting to that method.
     1. 
     2. Figure
     3. Application stopped at the breakpoint
  9. Remove the breakpoint and press **F5** to continue with the application execution.
  10. To edit the newly created address click the **(Edit)** link, located at the right of the new address, as shown Figure 7.
      1. 
      2. Figure
      3. Edit the newly created address
  11. Change the **Address Line 1** field value to "Modified address" and click **Save**. Verify that the address has been updated.
      1. 
      2. Figure
      3. Address updated
  12. To delete the created address click the **(Delete)** link, located next to the address. Verify that the address has been deleted.
      1. 
      2. Figure
      3. Address deleted

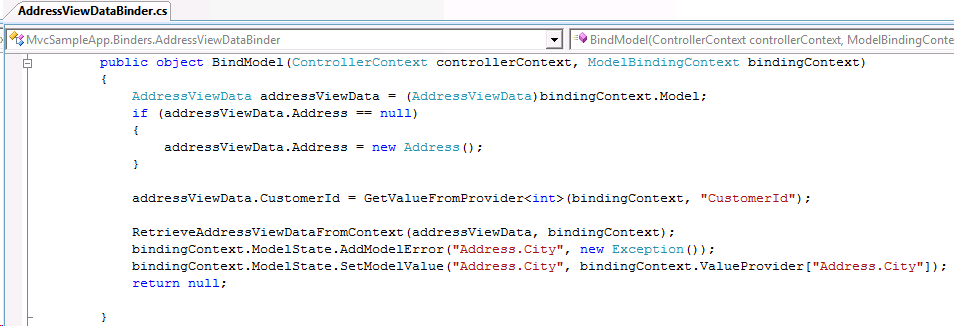
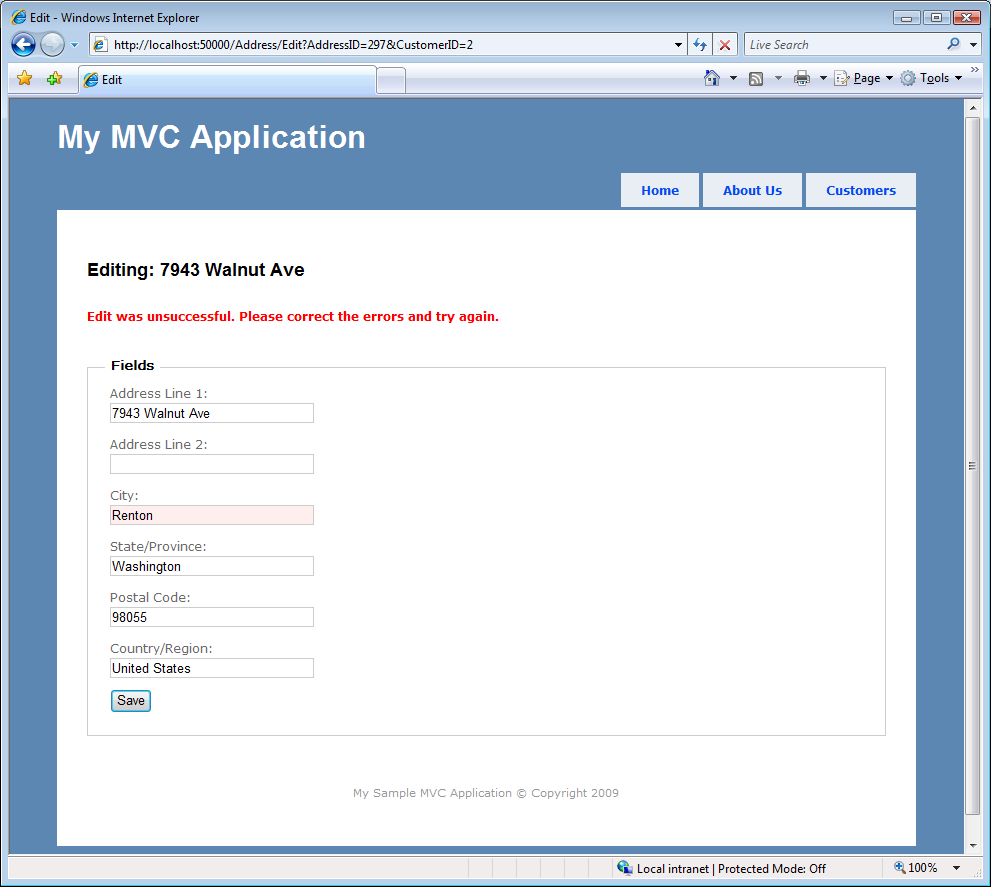
### Verification 2

* 1. In this verification you will check that the **AddressViewDataBinder** binder is being used by the ASP.NET MVC framework to update the model from the Create and Edit pages' fields.
  2. Start a new instance of the **MvcSampleApp** project. Right-click **MvcSampleApp** project in **Solution Explorer,** point to **Debug** and select **Start New Instance**. Alternatively you can continue using the instance used in the previous verification.
  3. Open the Customer controller’s **Index** view with the list of customers. To do this, click the **Customers** link in the page header.
  4. To view a specific customer’s details, click its name. The details on the given customer and its currently associated addresses will be shown.
  5. To create an address of the selected customer, click the **Add New Address** link.
  6. Set a breakpoint on the first line of the **AddressViewDataBinderBindModel** method:
     1. 
     2. Figure
     3. Breakpoint on BindModel method
  7. Complete the required information and click **Create**. Verify that application stops on the breakpoint.
  8. Check in the **Call Stack** that the **UpdateModel** method called from the **SaveNew** method of the **AddressController** controller has invoked the **BindModel** method of the **AddressViewDataBinder** class. To access the Call Stack window, click the **Debug** menu, point to **Windows** and click **Call Stack**.
     1. The call stack window output should look like the following:
     2. 
     3. Figure
     4. Visual Studio 2008 Call Stack window
  9. Check that the invoking method was **UpdateModel.** To do this, double-click the following line:
     1. **MvcSampleApp.DLL!MvcSampleApp.Controllers.AddressController.SaveNew**.
     2. 
     3. Figure
     4. UpdateModel on the call stack
  10. Press **F5** to continue with application execution.
  11. Edit the newly created address. To do this, click the **(Edit)** link, located next to the new address.
  12. Click **Save** and verify that application stops on the breakpoint. This step does not require data modification.
  13. Check in the **Call Stack** that the **TryUpdateModel** method called from the **Edit** method of the **AddressController** controller has invoked the **BindModel** method of the **AddressViewDataBinder** class.

The output of the Call Stack window should look like the following.

* + 1. 
    2. Figure
    3. Visual Studio 2008 Call Stack window
  1. Check that the invoking method was **TryUpdateModel**. To do this, double-click the following line:
     1. **MvcSampleApp.DLL!MvcSampleApp.Controllers.AddressController.Edit**
     2. 
     3. Figure
     4. TryUpdateModel on the call stack
  2. Press **F5** to continue with application execution.
  3. Delete the created address by clicking the **(Delete)** link, located next to the address created for this verification.

### Verification 3

* 1. In this last verification you will check the behavior of the **TryUpdateModel** helper method when an error occurs in the binding.
  2. Open the **AddressViewDataBinder.cs** file. To do this, double click the file in the **Solution Explorer**.
  3. Add the following using statement at the top of the file:
     1. C#
     2. using System;
  4. Introduce an error in the model using the **AddModelError** method. To do this, replace the last line of the **BindModel** method of the **AddressViewDataBinder** class with the following bolded code.
     1. (Code Snippet – *Enhancing Asp.Net MVC Lab – BindModel*)
     2. C#
     3. public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)
     4. {
     5. AddressViewData addressViewData = (AddressViewData)bindingContext.Model;
     6. if (addressViewData.Address == null)
     7. {
     8. addressViewData.Address = new Address();
     9. }
     10. addressViewData.CustomerId = GetValueFromProvider<int>(bindingContext, "CustomerId");
     11. **RetrieveAddressViewDataFromContext(addressViewData, bindingContext);**
     12. **bindingContext.ModelState.AddModelError("Address.City", new System.Exception());**
     13. **bindingContext.ModelState.SetModelValue("Address.City", bindingContext.ValueProvider["Address.City"]);**
     14. **return null;**
     15. }
     16. The following figure shows the updated code that provokes an error.
     17. 
     18. Figure
     19. Modified AddressViewDataBinder.BindModel method
  5. Build the solution and start a new instance of the **MvcSampleApp** project. Right-click the **MvcSampleApp** project in **Solution Explorer,** point to **Debug** and select **Start New Instance**.
  6. Open the Customer controller’s **Index** view with the list of customers. To do this, click the **Customers** link in the page header.
  7. To view a specific customer’s details, click its name. The details on the given customer and its currently associated addresses will be shown.
  8. Edit an address by clicking the **(Edit)** link. Change the values of the form's fields and click **Save**.
  9. Verify that the **City** field is highlighted and that the error message is being shown. Notice that the changed values remain on the form after the error is shown.
     1. 
     2. Figure
     3. Edit page with error
  10. Stop the application.
  11. Rollback the changes made on the **BindModel** method of the **AddressViewDataBinder** class by replacing the last four lines of the method with the original line, as shown in the following code:
      1. C#
      2. public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)
      3. {
      4. AddressViewData addressViewData = (AddressViewData)bindingContext.Model;
      5. if (addressViewData.Address == null)
      6. {
      7. addressViewData.Address = new Address();
      8. }
      9. addressViewData.CustomerId = GetValueFromProvider<int>(bindingContext, "CustomerId");
      10. **return RetrieveAddressViewDataFromContext(addressViewData, bindingContext);**
      11. }

Exercise 2: Using Partial Views with ASP.NET AJAX

* 1. In this exercise, you will learn how to make the application more responsive by adding ASP.NET AJAX support. That way, the rendering of an entire page is avoided, when an update is needed in only a certain region in a page.
  2. You will implement a customer filter functionality, which results are shown in the customer list. When a search is performed only this list is rendered and not the whole page.
  3. Also the navigation between pages will be done using AJAX.

Task 1 – Implementing the Customer Search Logic

* + 1. In this task you will implement customer search functionality to filter customers and update the customer list asynchronously.
  1. Open the solution file **MvcSampleApp.sln** located under the **enhancingAspNetMvcApp\Ex02-PartialViews\begin** folder. Alternatively you might continue using the solution obtained after completing the previous exercise.
  2. Open the **AdventureWorksRepository** class, located inside the **Models** folder.
  3. Add logic to implement the customer search functionality. To do this, paste the following method in the **AdventureWorksRepository** class.
     1. (Code Snippet – *Enhancing Asp.Net MVC Lab – AdventureWorksRepository GetFilteredCustomers Method*)
     2. C#
     3. **public IEnumerable<Customer> GetFilteredCustomers(string filterPattern, int page, int size)**
     4. **{**
     5. **return this.context.Customer.Where(c => (c.CompanyName + " - " + c.FirstName + " " + c.LastName).Contains(filterPattern)).OrderBy(c => c.CustomerID).Skip(page \* size).Take(size);**
     6. **}**

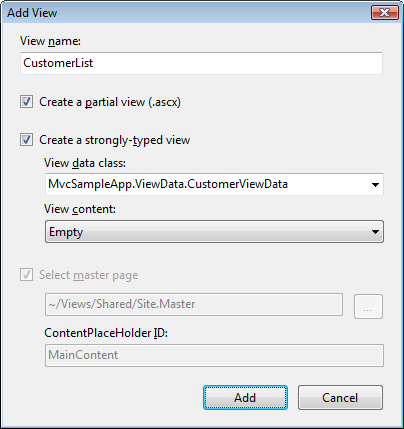
**Note:**  The **GetFilteredCustomers** method shown in the preceding code, searches the database for the provided filter pattern in the concatenation of the **CompanyName**, **FirstName** and **LastName** fields of the customer table.

* 1. Add the **CustomerFilter** property, in the **CustomerViewData** class. To do this, paste the following bolded code inside the class, which is located in the **ViewData** folder.
     1. (Code Snippet – *Enhancing Asp.Net MVC Lab – CustomerViewData CustomerFilter Property*)
     2. C#
     3. public class CustomerViewData
     4. {
     5. public IEnumerable<Customer> Customers
     6. {
     7. get;
     8. set;
     9. }
     10. public int PreviousPage
     11. {
     12. get;
     13. set;
     14. }
     15. public int NextPage
     16. {
     17. get;
     18. set;
     19. }
     20. **public string CustomerFilter**
     21. **{**
     22. **get;**
     23. **set;**
     24. **}**
     25. }
  2. Open the **CustomerController** class. To do this, in the **Solution Explorer**, double click the **CustomerController.cs** file, located inside the **Controllers** folder.
  3. Update the **Index** method, removing the page parameter because the Index method will not be in charge of pagination. The following code shows the updated **Index** method.
     1. (Code Snippet – *Enhancing Asp.Net MVC Lab – CustomerController Index Action*)
     2. C#
     3. **public ActionResult Index()**
     4. **{**
     5. **var viewData = new CustomerViewData();**
     6. **viewData.Customers = this.repository.GetCustomers(0, 10);**
     7. **viewData.NextPage = 1;**
     8. **viewData.PreviousPage = 0;**
     9. **return View(viewData);**
     10. **}**
  4. Add the **FilterCustomers** action method that will be invoked when the search form is submitted. To do this, add the following code in the **CustomerController** class.
     1. (Code Snippet – *Enhancing Asp.Net MVC Lab – CustomerController FilterCustomers Action*)
     2. C#
     3. **public ActionResult FilterCustomers(string customersFilter)**
     4. **{**
     5. **var viewData = new CustomerViewData();**
     6. **viewData.Customers = this.repository.GetFilteredCustomers(customersFilter, 0, 10);**
     7. **viewData.NextPage = 1;**
     8. **viewData.PreviousPage = 0;**
     9. **viewData.CustomerFilter = customersFilter;**
     10. **return PartialView("CustomerList", viewData);**
     11. **}**
     12. **Note:** Partial Views are used to extract common pieces of UI shared across the application or, as in this case, when only a part of the View needs to be updated after an AJAX request to the server.
  5. Add the **ChangeCustomersPage** action method that will be in charge of the pagination of the customer list. To do this, add the following code in the **CustomerController** class.
     1. (Code Snippet – *Enhancing Asp.Net MVC Lab – CustomerController ChangeCustomersPage Method*)
     2. C#
     3. **public ActionResult ChangeCustomersPage(string customersFilter, int currentPage)**
     4. **{**
     5. **var viewData = new CustomerViewData();**
     6. **if (string.IsNullOrEmpty(customersFilter))**
     7. **{**
     8. **viewData.Customers = this.repository.GetCustomers(currentPage, 10);**
     9. **}**
     10. **else**
     11. **{**
     12. **viewData.Customers = this.repository.GetFilteredCustomers(customersFilter, currentPage, 10);**
     13. **}**
     14. **viewData.NextPage = currentPage + 1;**
     15. **viewData.PreviousPage = (currentPage <= 0) ? 0 : currentPage - 1;**
     16. **viewData.CustomerFilter = customersFilter;**
     17. **return PartialView("CustomerList", viewData);**
     18. **}**

Task 2 – Adding AJAX to the Application

* 1. In this task you will add the required JavaScript scripts and refactor the customers Index view to use AJAX.
  2. Open the **Site.Master** page. The file is located in the **Views\Shared** folder.
  3. Add the required JavaScript scripts to add AJAX functionality to the application. To do this, add the following bolded code inside the **<head>** tag of the view master page.
     1. HTML
     2. <head runat="server">
     3. **<script src="../../Scripts/MicrosoftAjax.js" type="text/javascript"></script>**
     4. **<script src="../../Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>**
     5. **<script src="../../Scripts/jquery-1.3.1.min.js" type="text/javascript"></script>**
     6. <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
     7. <link href="../../Content/Site.css" rel="stylesheet" type="text/css" />
     8. </head>
     9. **Note:** All three of these files are included in the **Scripts** folder of a new ASP.NET MVC application.
     10. The inclusion of the required JavaScript files in the application’s view master page is a good practice if you plan to use AJAX in multiple pages in your application. That way, the JavaScript files will be included in all of the pages in the application automatically.
  4. Open the customers **Index** page. To do this, in the **Solution Explorer**, double click the **Index.aspx** file, located inside the **Views\Customer** folder.
  5. Add a new AJAX form to filter customers. The form will have a textbox and a submit button. To do this, add the following bolded code below the Customer heading.
     1. ASP.NET
     2. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     3. <h2>
     4. Customers</h2>
     6. **<% using (Ajax.BeginForm("FilterCustomers", new AjaxOptions() { UpdateTargetId = "divCustomerList" }))**
     7. **{ %>**
     8. **<label for="customersFilter">**
     9. **Search:</label>**
     10. **<%= Html.TextBox("customersFilter") %>**
     11. **<input type="submit" value="Go" />**
     12. **<% } %>**
     13. <ul>
     14. <% foreach (var customer in Model.Customers)
     15. { %>
     16. <li>
     17. <%= Html.ActionLink(customer.CompanyName + " - " + customer.FirstName
     18. + " " + customer.LastName, "Info", new { id = customer.CustomerID }) %>
     19. </li>
     20. <% } %>
     21. </ul>
     22. <%=Html.ActionLink("<< Previous Page", "Index", new { page = ViewData.Model.PreviousPage }) %>&nbsp;&nbsp;&nbsp;&nbsp
     23. <%=Html.ActionLink("Next Page >>", "Index", new { page = ViewData.Model.NextPage }) %>
     24. </asp:Content>

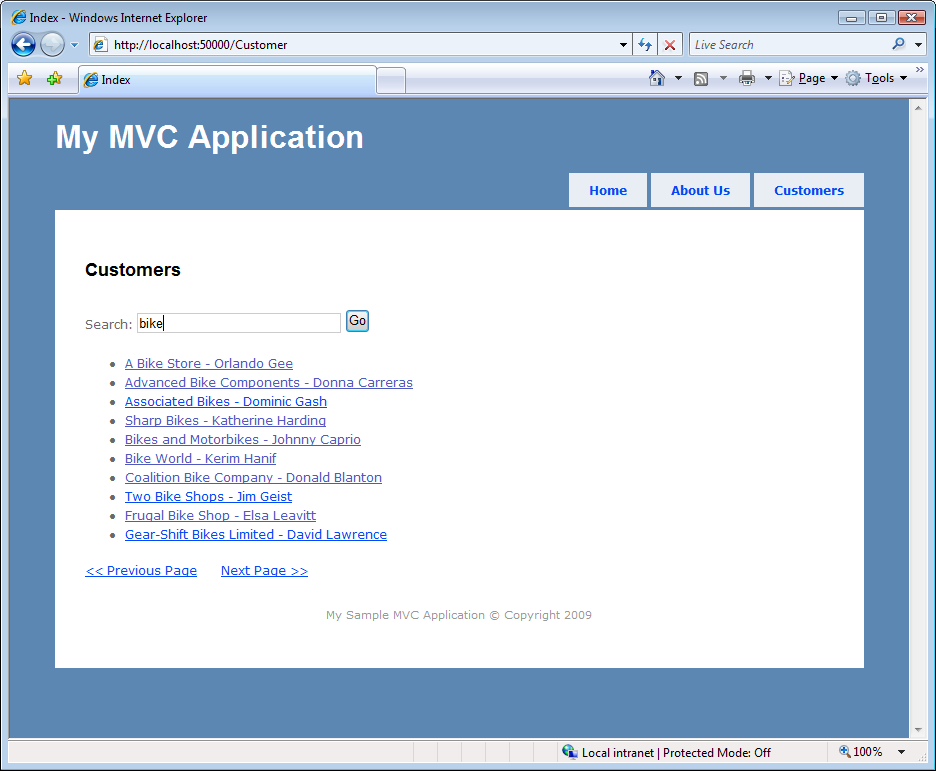
**Note:** The **Ajax.BeginForm** helper method is used to create an AJAX form, which submission is done asynchronously. In this overload, the method takes the following parameters: the action name that is called on submitting and an array containing the Ajax options. In the preceding code, the **filterCustomers** action method and the Ajax options, containing the **UpdateTargetId** are passed as parameters.

* + 1. The **UpdateTargetId** AJAX option is used for defining the region that will be updated as a result of the AJAX requests.
  1. Create a new partial view in the **Views\Customer** folder. To do this, right-click the **Views\Customer** folders, point to **Add** and click **View**. In the **Add View** window, type **CustomerList** as the view name, select the **Create a partial View (.ascx)** and **Create a strongly-typed view** checkboxes, and then select **MvcSampleApp.ViewData.CustomerViewData** in the **View data class** combo box. Click **Add** to insert the view.
     1. 
     2. Figure
     3. The Add View dialog
  2. Separate the part of the view that you want to update asynchronously (the customer list) into a separate partial view (view user control). The section of the Index view that displays the table of customers has to be moved into the **CustomerList** partial view. To do this, cut from the **Index.aspx** file the *<ul>* element that renders the customer list and the Html.ActionLinks calls, and paste it inside the **CustomerList.ascx** under a new *<div>* element with **divCustomers** as id. The code in the **CustomerList.ascx** should look like the following code:
     1. ASP.NET
     2. <%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<MvcSampleApp.ViewData.CustomerViewData>" %>
     3. **<div id="divCustomers">**
     4. **<ul>**
     5. **<% foreach (var customer in Model.Customers)**
     6. **{ %>**
     7. **<li>**
     8. **<%= Html.ActionLink(customer.CompanyName + " - " + customer.FirstName + " " + customer.LastName, "Info", new { id = customer.CustomerID }) %>**
     9. **</li>**
     10. **<% } %>**
     11. **</ul>**
     12. **<%=Ajax.ActionLink("<< Previous Page", "ChangeCustomersPage", new { currentPage = Model.PreviousPage, customersFilter = Model.CustomerFilter }, new AjaxOptions() { UpdateTargetId = "divCustomers" })%> &nbsp;&nbsp;&nbsp;&nbsp**
     13. **<%=Ajax.ActionLink("Next Page >>", "ChangeCustomersPage", new { currentPage = Model.NextPage, customersFilter = Model.CustomerFilter }, new AjaxOptions() { UpdateTargetId = "divCustomers" })%>**
     14. **</div>**
     15. **Note:** The content of the partial view is wrapped in a div section since all of it will need to be updated when doing an AJAX request through the Previous Page / Next Page action links.
  3. In the **Index** view, replace the code that was moved with a call to the partial view. To do this, use the **RenderPartial** method, as shown in the following bolded code.
     1. ASP.NET
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage<MvcSampleApp.ViewData.CustomerViewData>" %>
     3. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     4. Index
     5. </asp:Content>
     6. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     7. <h2>
     8. Customers</h2>
     9. <% using (Ajax.BeginForm("FilterCustomers", new AjaxOptions() { UpdateTargetId = "divCustomerList" }))
     10. { %>
     11. <label for="customersFilter">
     12. Search:</label>
     13. <%= Html.TextBox("customersFilter") %>
     14. <input type="submit" value="Go" />
     15. <% } %>
     17. **<div id="divCustomerList">**
     18. **<% Html.RenderPartial("CustomerList", Model); %>**
     19. **</div>**
     20. </asp:Content>
     21. **Note:**  The **RenderPartial** method is used for rendering the partial view with an empty view data and the given model.
     22. The call to the **RenderPartial** method is located inside the div tags, to define the section of the page that will be updated on AJAX requests.

## Exercise 2: Verification

In order to verify that you have correctly performed all steps of exercise one, proceed as follows:

### Verification 1

* 1. You will run the application and perform a search to verify that instead of re-rendering the whole page, only the customer list is updated.
  2. To test the MVC Web Application you need to start a new instance of the **MvcSampleApp** project. To do this, in **Solution Explorer** right-click **MvcSampleApp** project, point to **Debug** and select **Start New Instance**.
  3. Open the Customer controller’s **Index** view with the list of customers. To do this, click the **Customers** link in the page header.
  4. Type *"bike"* in the **Search** textbox to filter the customers and click **Go**. The resulting screen is shown in the following figure.
     1. 
     2. Figure
     3. The filtered customers list
     4. **Note:** Notice that only the customer list is updated when you click the button, and not the whole page. You can realize that by looking that the filter string still remains on the text box after submitting the form.
  5. Click the **Next Page** link to navigate to the next page of the customer list.
     1. **Note:** Notice that when navigating to another page only the list is updated and not the whole page, as happened on the previous step.

Exercise 3: Implementing Action Filters

* 1. In this exercise, you will learn how to implement an action filter. An action filter is an attribute that you can apply to a controller action or an entire controller that modifies the way in which the action is executed.
  2. The ASP.NET MVC framework supports four different types of filters:
  3. **Authorization filters** – Implements the **IAuthorizationFilter** attribute.
  4. **Action filters** – Implements the **IActionFilter** attribute.
  5. **Result filters** – Implements the **IResultFilter** attribute.
  6. **Exception filters** – Implements the **IExceptionFilter** attribute.
  7. Filters are executed in the order listed above.

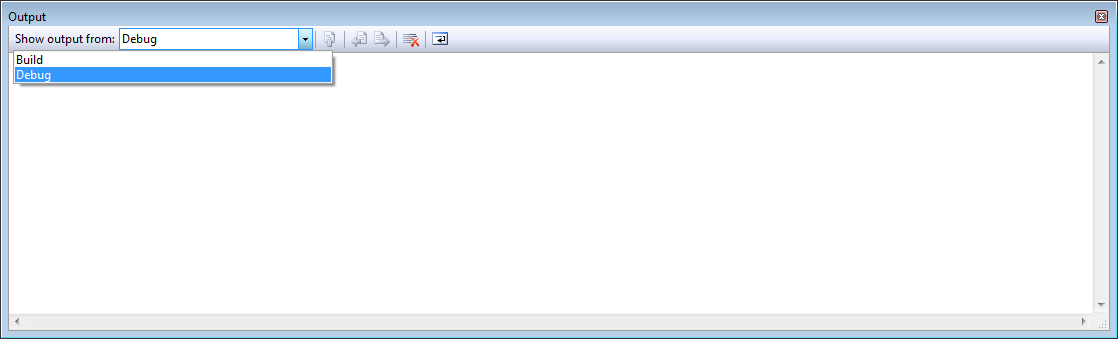
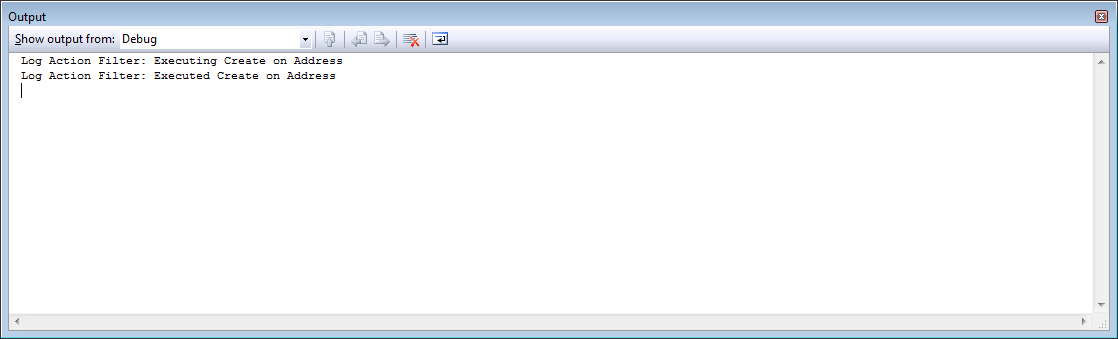
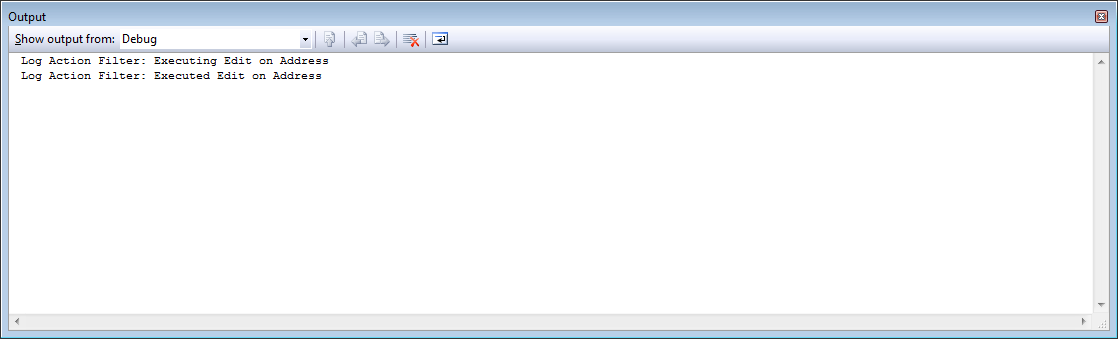
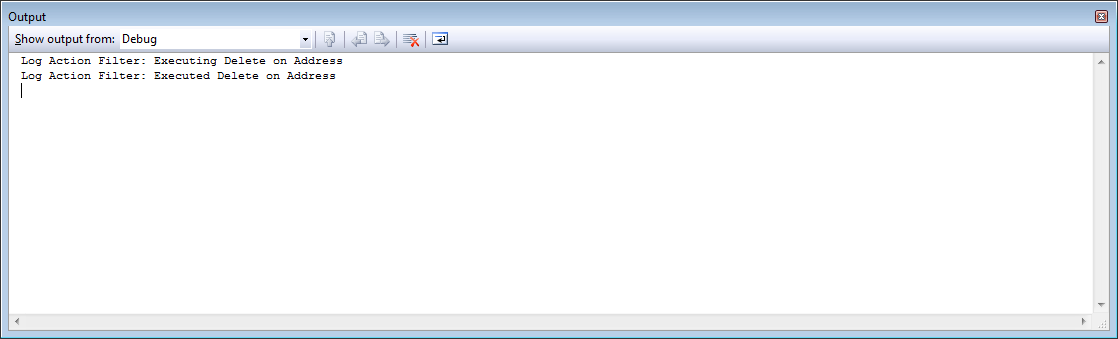
Task 1 – Implementing Action Filters

* + 1. In this task you will create an action filter to log the execution of the desired controller’s action methods.
  1. Open the solution file **MvcSampleApp.sln** located under the **enhancingAspNetMvcApp\Ex03-ActionFilters\begin** folder. Alternatively you might continue using the solution obtained after completing the previous exercise.
  2. Create a folder named **ActionFilters**. To do this, in the **Solution Explorer**, right click the MvcSampleApp project, point to **Add**, click **New Folder** and set the folder name to **ActionFilters**.
  3. Create the **LogActionFilterAttribute** class in the **ActionFilters** folder. To do this, right click the **ActionFilters** folder, point to **Add**, click **Class**, then set the class name to **LogActionFilterAttribute**.
  4. Open the **LogActionFilterAttribute.cs** file and replace the using statements at the top of the file with the following ones.
     1. C#
     2. **using System.Diagnostics;**
     3. **using System.Globalization;**
     4. **using System.Web.Mvc;**
     5. **using System.Web.Routing;**
  5. Change the class signature to inherit from the **ActionFilterAttribute** class, as shown in the following code.
     1. C#
     2. public class LogActionFilterAttribute : **ActionFilterAttribute**
     3. {
     4. }
     5. **Note:** The base **ActionFilterAttribute** class has the following methods that you can override:
     6. **OnActionExecuting** – This method is called before a controller action is executed.
     7. **OnActionExecuted** – This method is called after a controller action is executed.
     8. **OnResultExecuting** – This method is called before a controller action result is executed.
     9. **OnResultExecuted** – This method is called after a controller action result is executed.
  6. Add the **LogEntry** method in the **LogActionFilterAttribute** class. This method obtains information from the route data and writes it on the **Output** window. To do this, paste the following code into the **LogActionFilterAttribute** class.
     1. (Code Snippet – *Enhancing Asp.Net MVC Lab – LogActionFilterAttribute LogEntry Method*)
     2. C#
     3. **private static void LogEntry(string executionStep, RouteData routeData)**
     4. **{**
     5. **string controller = routeData.Values["controller"] as string;**
     6. **string action = routeData.Values["action"] as string;**
     7. **string entry = string.Format(CultureInfo.CurrentUICulture, "Log Action Filter: {0} {1} on {2}", executionStep, action, controller);**
     9. **Debug.WriteLine(entry);**
     10. **}**
  7. Override the **OnActionExecuting** and **OnActionExecuted** methods from the base class to call the **LogEntry** method created in the previous step, which will be responsible for logging the controller actions execution. To do this, add the following code in the **LogActionFilterAttribute** class.
     1. (Code Snippet – *Enhancing Asp.Net MVC Lab – LogActionFilterAttribute Methods*)
     2. C#
     3. **public override void OnActionExecuting(ActionExecutingContext filterContext)**
     4. **{**
     5. **LogEntry("Executing", filterContext.RouteData);**
     6. **}**
     7. **public override void OnActionExecuted(ActionExecutedContext filterContext)**
     8. **{**
     9. **LogEntry("Executed", filterContext.RouteData);**
     10. **}**
  8. Open the **AddressController.cs** file, and add the following using statement:
     1. C#
     2. **using MvcSampleApp.ActionFilters;**
  9. In the **AddressController** class, add the **LogActionFilter** attribute to every action method that you wish to log. In this case, the **Create** (POST), **Edit** (POST), and **Delete** action methods. The following code shows the attribute applied to every action that will be logged.
     1. C#
     2. public class AddressController : Controller
     3. {
     4. private AdventureWorksRepository repository = new AdventureWorksRepository();
     5. public ActionResult Create(int customerId)
     6. {
     7. …
     8. }
     9. **[LogActionFilter]**
     10. [AcceptVerbs(HttpVerbs.Post), ActionName("Create")]
     11. public ActionResult SaveNew(int customerId)
     12. {
     13. …
     14. }
     15. public ActionResult Edit(int addressId, int customerId)
     16. {
     17. …
     18. }
     20. **[LogActionFilter]**
     21. [AcceptVerbs(HttpVerbs.Post)]
     22. public ActionResult Edit(int addressId, int customerId, FormCollection collection)
     23. {
     24. …
     25. }
     26. **[LogActionFilter]**
     27. public ActionResult Delete(int addressId, int customerId)
     28. {
     29. …
     30. }
     31. }

## Exercise 3: Verification

In order to verify that you have correctly performed all steps of exercise one, proceed as follows:

### Verification 1

* 1. You will run the application and will create, edit and delete an address verifying that the logger is writing on the debug console when the controller's actions are invoked.
  2. To test the MVC Web Application you need to start a new instance of the **MvcSampleApp** project. To do this, in **Solution Explorer** right-click **MvcSampleApp** project, point to **Debug** and select **Start New Instance**.
  3. Open the Customer controller’s **Index** view with the list of customers. To do this, click the **Customers** link in the page header.
  4. Open a customer’s details page by clicking on its name.
  5. To create a new address for the selected customer, click the **Add New Address** link.
  6. Complete the form's input data and click on the **Create** button.
  7. Open the **Output** window on Visual Studio. To do this, go to the **View** menu and click **Output**. In the **Output** window, select **Debug** on the **Show output from** combo box as shown in the figure below.
     1. 
     2. Figure
     3. Selecting the Debug pane on the Output window
  8. Verify that a log entry was added for the **Create** action of the **AddressController** controller, when it was being executed and after its execution finished.
     1. 
     2. Figure
     3. Log entries on AddressController.Create action execution
  9. Edit the newly created address by clicking the **(Edit)** link, located at the right of the new address and click **Save** once you are on the edit form.
  10. Verify that a log entry was added for the **Edit** action of the **AddressController** controller, when it was being executed and after its execution finished.
      1. 
      2. Figure
      3. Log entries on AddressController.Edit action execution
  11. Delete the newly created address by clicking the **(Delete)** link, located at the right of the new address.
  12. Verify that a log entry was added for the **Delete** action of the **AddressController** controller, when it was being executed and after its execution finished.
      1. 
      2. Figure
      3. Log entries on AddressController.Delete action execution

Summary

* 1. In this Hands-On Lab, you have learnt how to create controller's action methods with different names than the default ones, how to create a custom model binder to bind complex objects to input forms and how to use the **UpdateModel** / **TryUpdateModel** helper methods to update a model object from the input forms' fields. You have also learnt how to add AJAX functionality to your application, to increase its responsiveness and performance. Finally, you have learnt how to implement action filters to modify the way in which the actions are executed.