Hands-On Lab

Build RESTful API’s with ASP.NET Web API

Lab version: 1.2.0

Last updated: 3/20/2012

* 1. 

Contents

[Overview 3](#_Toc319921197)

[Exercise 1: Create a Read-Only Web API 7](#_Toc319921198)

[Task 1 – Create the API Project 7](#_Toc319921199)

[Task 2 – Create the Contact Manager API Controllers 8](#_Toc319921200)

[Task 3 – Create the Contact Models and Augment the Contact Controller 14](#_Toc319921201)

[Task 4 – Extracting Functionality into a Service Layer and Injecting it using MEF 2.0 17](#_Toc319921202)

[Exercise 2: Create a Read/Write Web API 26](#_Toc319921203)

[Task 1 – Open the Web API Project 26](#_Toc319921204)

[Task 2 – Add Data-Persistence Features to the Contact Repository Implementation 26](#_Toc319921205)

[Exercise 3: Consume the Web API from an HTML Client 29](#_Toc319921206)

[Task 1 – Modify the Index View to Provide a GUI for Displaying Contacts 29](#_Toc319921207)

[Task 2 – Modify the Index View to Provide a GUI for Creating Contacts 31](#_Toc319921208)

[Summary 35](#_Toc319921209)

Overview

In recent years, it has become clear that HTTP is not just for serving up HTML pages. It is also a powerful platform for building Web APIs, using a handful of verbs (GET, POST, and so forth) plus a few simple concepts such as *URIs* and *headers*.

ASP.NET Web API is a set of components that simplify HTTP programming. Because it is built on top of the ASP.NET MVC runtime, Web API automatically handles the low-level transport details of HTTP. At the same time, Web API naturally exposes the HTTP programming model. In fact, one goal of Web API is to *not* abstract away the reality of HTTP. As a result, Web API is both flexible and easy to extend.

In this hands-on lab, you will use Web API to build a simple REST API for a contact manager application. You will also build a client to consume the API.

The REST architectural style has proven to be an effective way to leverage HTTP — although it is certainly not the only valid approach to HTTP. The contact manager will expose the following RESTful methods.

This lab requires a basic understanding of HTTP, REST, and assumes you have a basic working knowledge of HTML, JavaScript, and jQuery.

* 1. **Note:** The ASP.NET Web site has an area dedicated to the ASP.NET Web API framework at <http://asp.net/web-api>. This site will continue to provide late-breaking information, samples, and news related to Web API, so check it frequently if you’d like to delve deeper into the art of creating custom Web API’s available to virtually any device or development framework.

# Objectives

* 1. In this hands-on lab, you will learn how to:
  + Implement a RESTful Web API
  + Call the API from an HTML client
  + Use the Microsoft Managed Extensibility Framework (MEF 2.0) to create a loosely-coupled and reusable componentized architecture

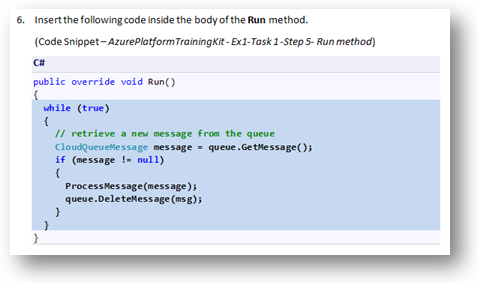
# Prerequisites

* 1. The following is required to complete this hands-on lab:
  + [Microsoft Visual Studio 11 Beta](http://www.microsoft.com/visualstudio/11/)
  + Windows PowerShell (for setup scripts – already installed on Windows 7 and Windows Server 2008 R2)

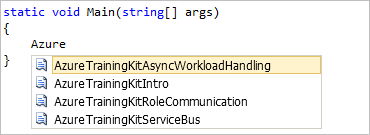
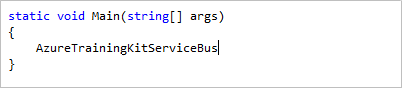
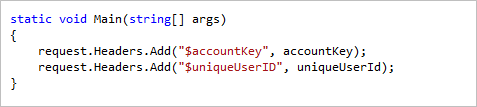
# Setup

* 1. Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of that code is provided as Visual Studio Code Snippets, which you can use from within Visual Studio to avoid having to add it manually.
  2. To install the code snippets:
  3. Open a Windows Explorer window and browse to the lab’s **Source\Setup** folder.
  4. Double-click the **Setup.cmd** file in this folder to install the Visual Studio code snippets.

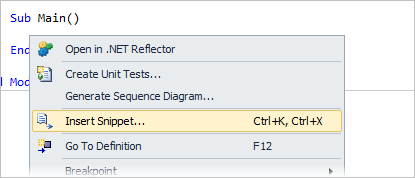
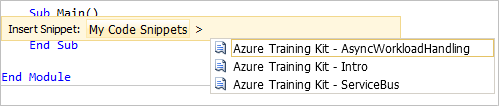
# Using the Code Snippets

* 1. With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.
  2. 
  3. Figure
  4. Using Visual Studio code snippets to insert code into your project

#### To add a code snippet using the keyboard (C# only)

* 1. Place the cursor where you would like to insert the code.
  2. Start typing the snippet name (without spaces or hyphens).
  3. Watch as IntelliSense displays matching snippets' names.
  4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
  5. Press the Tab key twice to insert the snippet at the cursor location.
  6. 
  7. Figure
  8. Start typing the snippet name
  9. 
  10. Figure
  11. Press Tab to select the highlighted snippet
  12. 
  13. Figure
  14. Press Tab again and the snippet will expand

#### To add a code snippet using the mouse (C#, Visual Basic and XML)

* 1. Right-click where you want to insert the code snippet.
  2. **Select Insert Snippet** followed by **My Code Snippets**.
  3. Pick the relevant snippet from the list, by clicking on it.
  4. 
  5. Figure
  6. Right-click where you want to insert the code snippet and select Insert Snippet
  7. 

Figure

* 1. Pick the relevant snippet from the list, by clicking on it

# Exercises

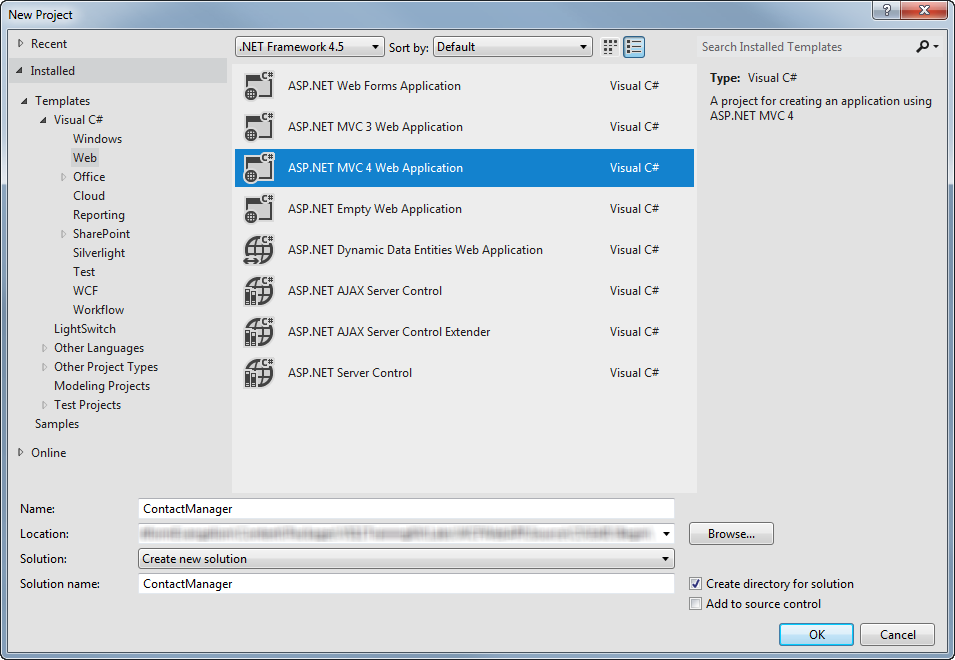
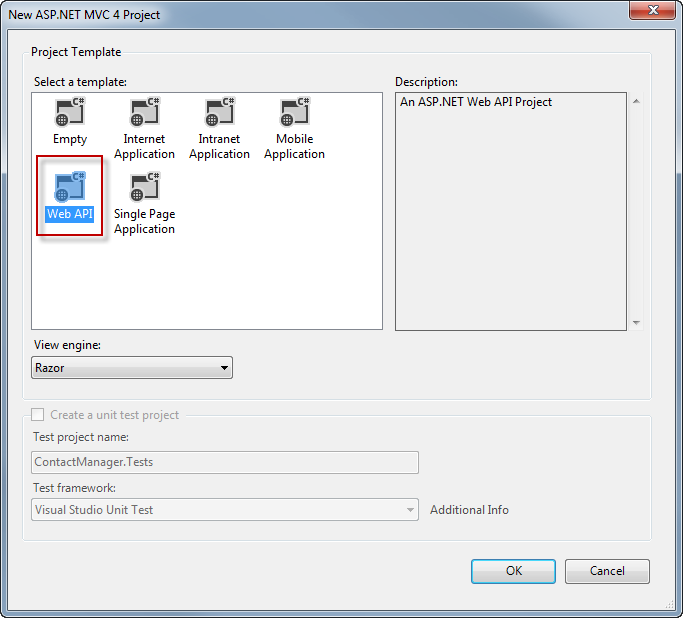
* 1. This hands-on lab includes the following exercise:
  2. Create a Read-Only Web API
  3. Extend the Read-Only Web API so that it persists data
  4. Consume the Web API from an HTML Client
  5. Estimated time to complete this lab: 60 **minutes**.
  6. **Note:** When you first start Visual Studio, you must select one of the predefined settings collections. Every predefined collection is designed to match a particular development style and determines window layouts, editor behavior, IntelliSense code snippets, and dialog box options. The procedures in this lab describe the actions necessary to accomplish a given task in Visual Studio when using the **General Development Settings** collection. If you choose a different settings collection for your development environment, there may be differences in these procedures that you need to take into account.

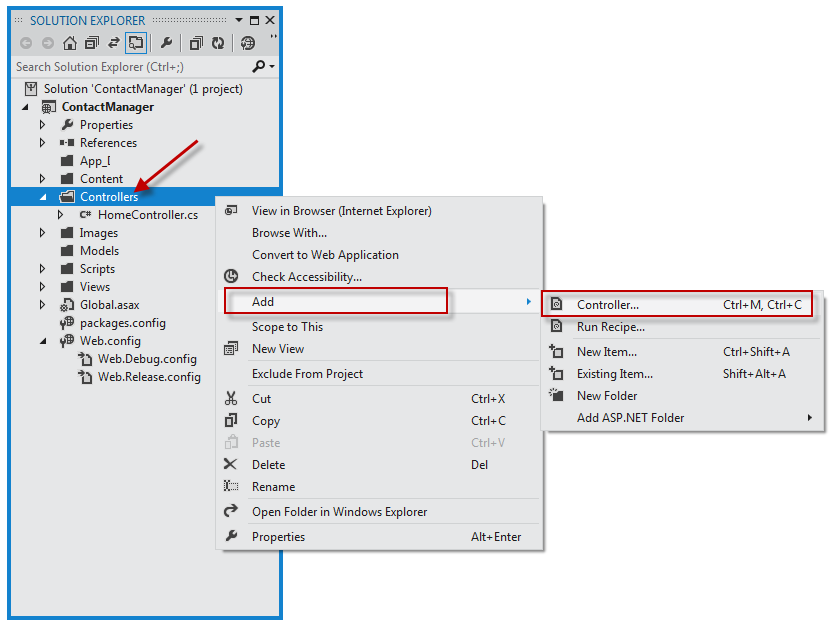
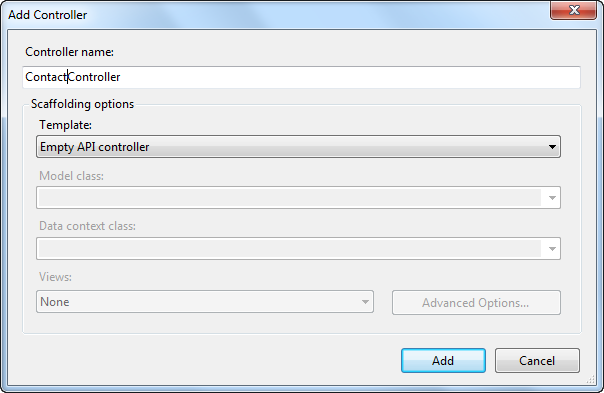
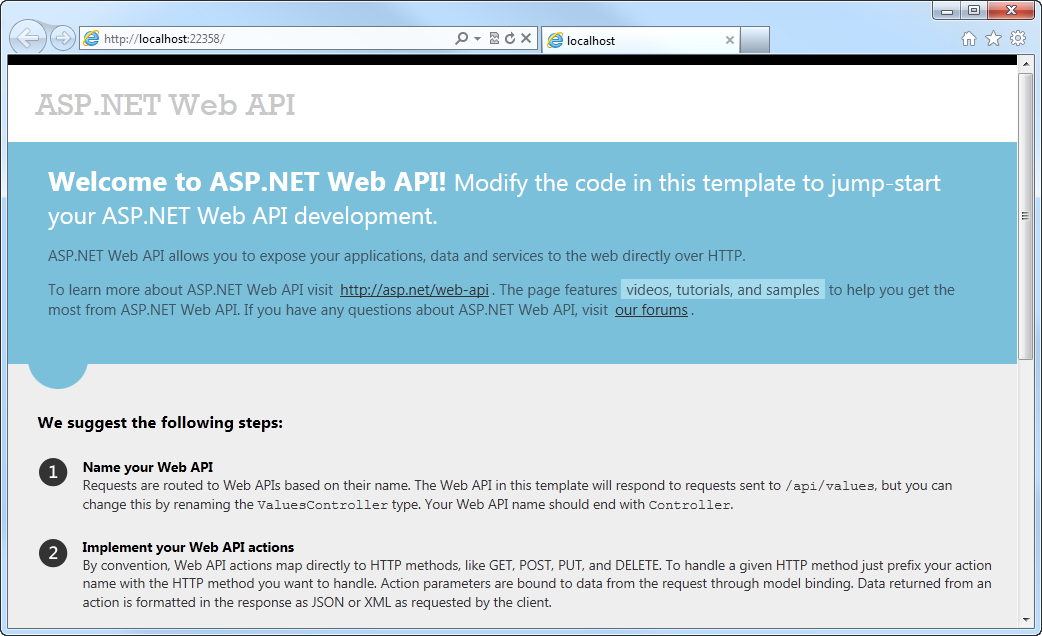
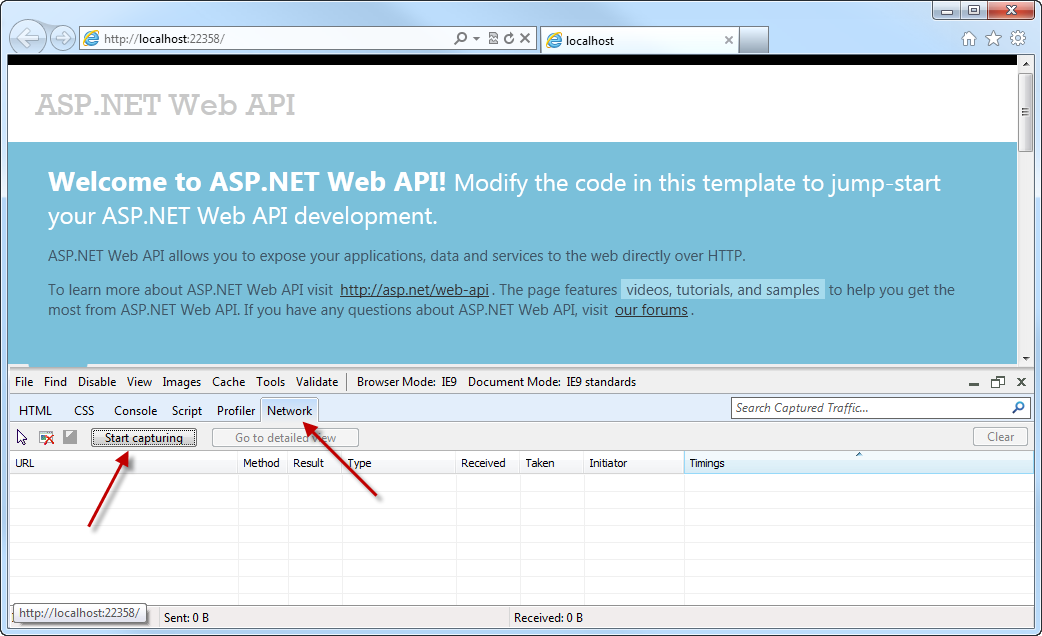
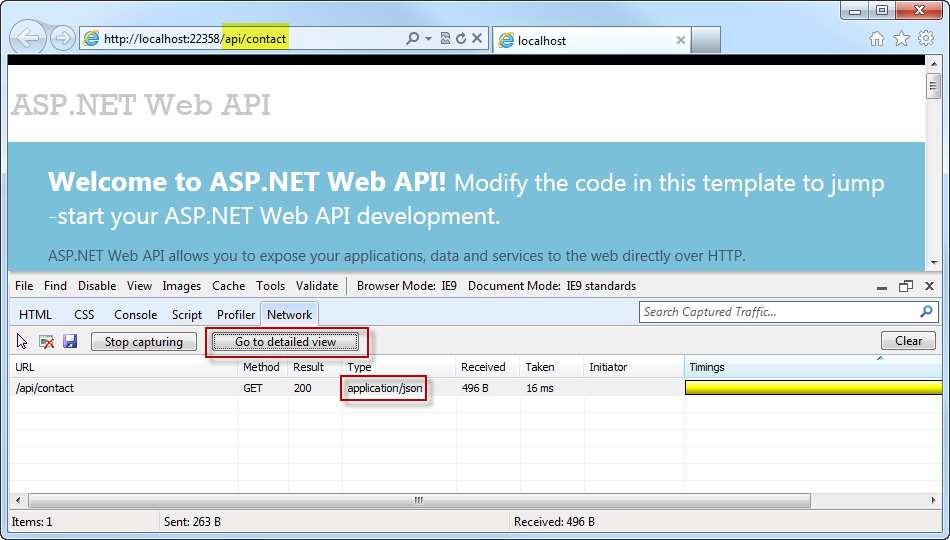
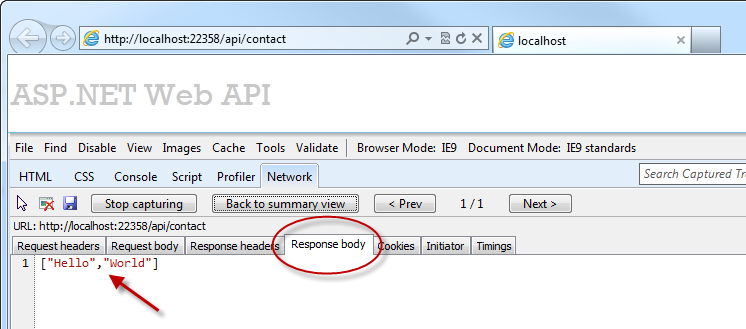
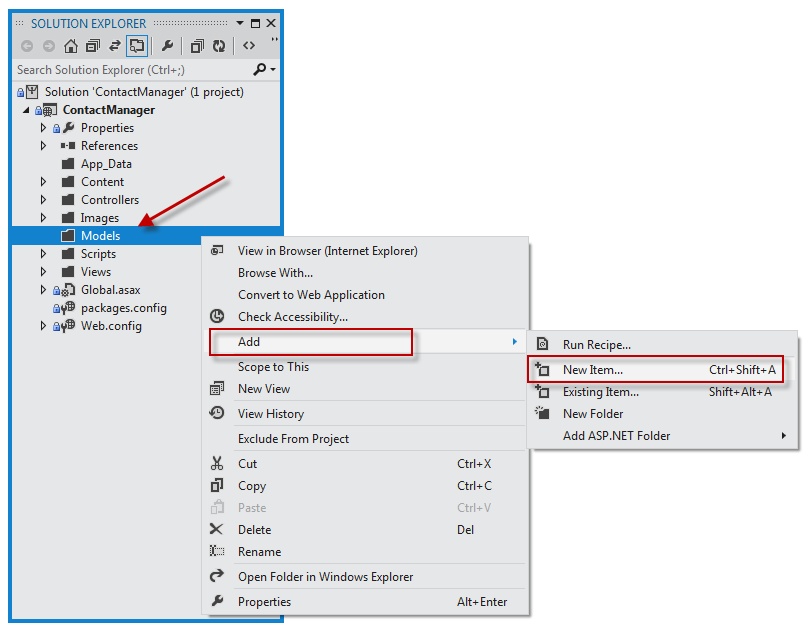
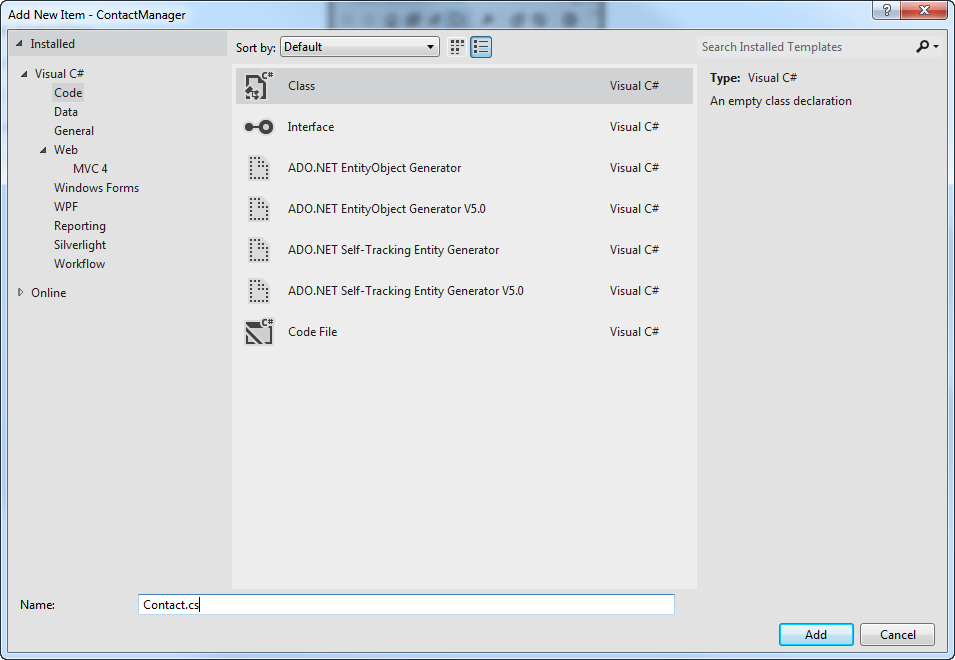
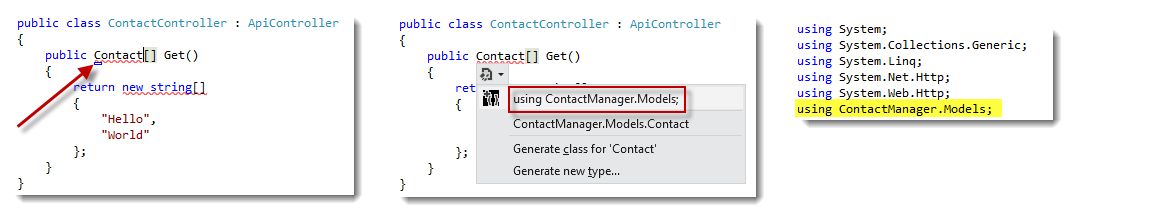
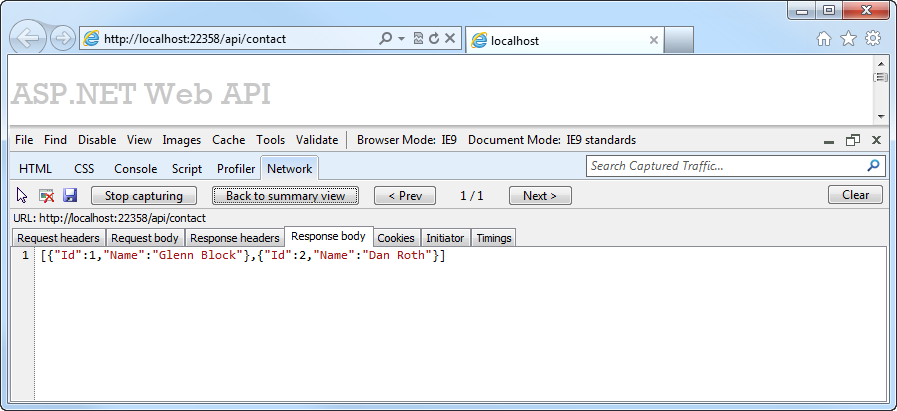
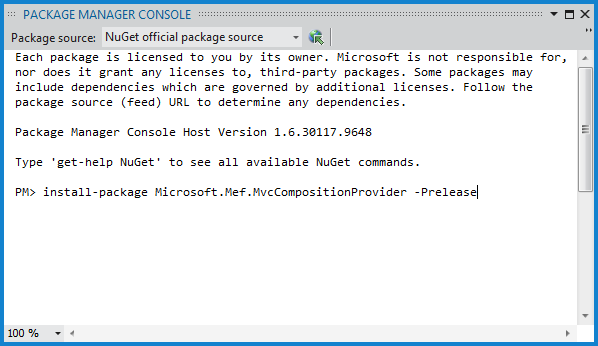
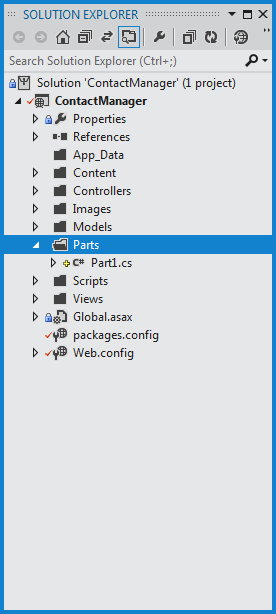
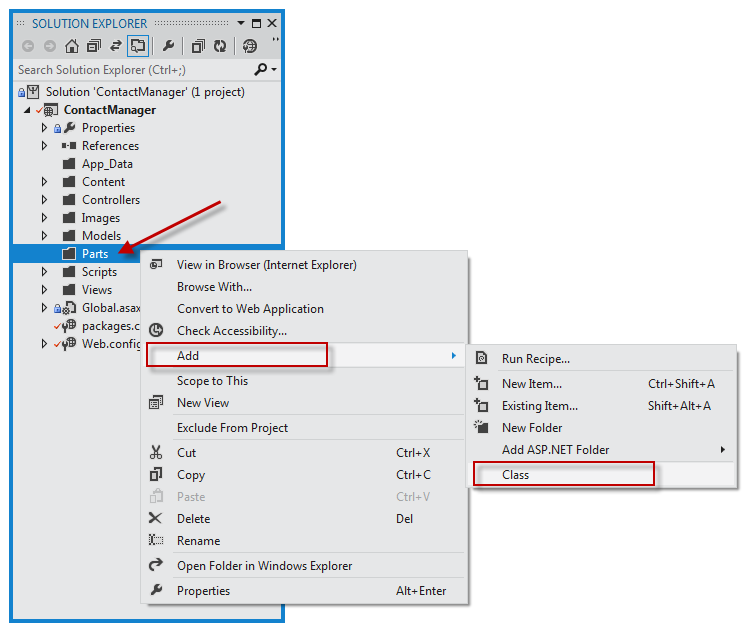
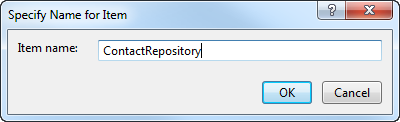
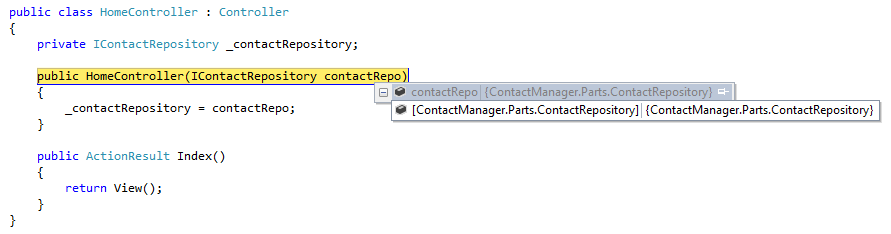
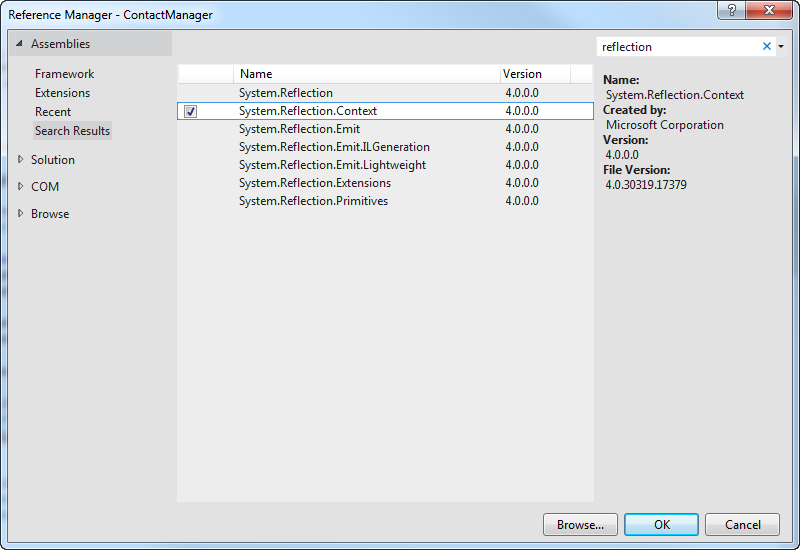
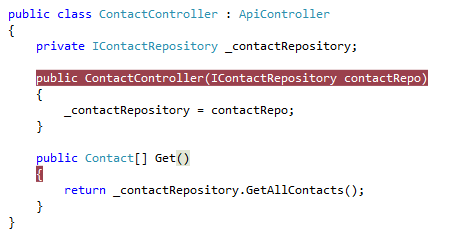
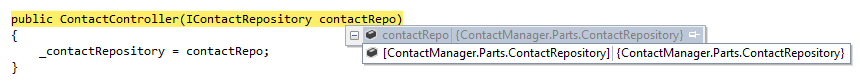
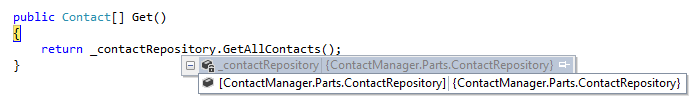
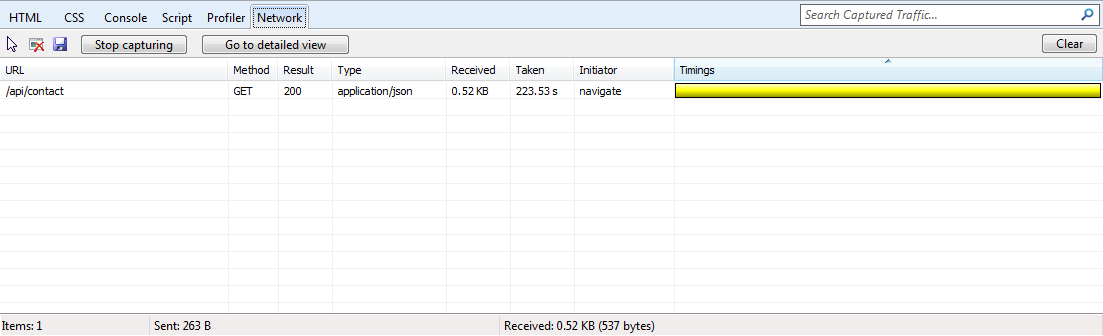
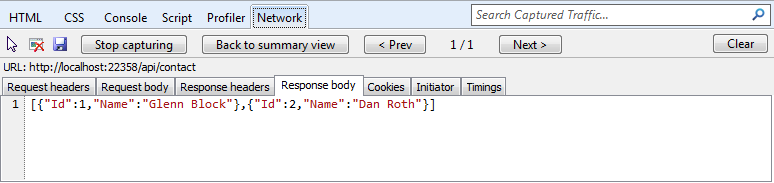
Exercise 1: Create a Read-Only Web API

In this exercise, you will implement the read-only GET methods for the contact manager.

1. Task 1 – Creating the API Project

In this task, you will use the new ASP.NET web project templates to create a Web API web application.

* 1. Run Visual Studio 11.
  2. From the **File** menu, select **New Project**. Select the **Visual C# | Web** project type from the project type tree view, then select the **ASP.NET MVC 4 Web Application** project type. Name the project **ContactManager** and then click OK.
     1. 
     2. Figure
     3. Creating a new MVC 4.0 Web Application Project
  3. In the ASP.NET MVC 4 project type dialog, select the **Web API** project type. Click OK.
     1. 
     2. Figure
     3. Specifying the Web API project type

1. Task 2 – Creating the Contact Manager API Controllers
   1. In this task, you will create the controller classes in which API methods will reside.
   2. Delete the file named **ValuesController.cs** from the project.
   3. Right-click the **Controllers** folder in the project and select **Add | Controller** from the context menu.
      1. 
      2. Figure
      3. Adding a new controller to the project
   4. In the **Add Controller** dialog that appears, select **Empty API Controller** from the Template menu. Name the controller class **ContactController**. Then, click **Add.**
      1. 
      2. Figure
      3. Using the Add Controller dialog to create a new Web API controller
   5. Add the code below to the **ContactController.**
      1. (Code Snippet – *Web API Lab - Ex01 – Get API Method*)
      2. C#
      3. **public string[] Get()**
      4. **{**
      5. **return new string[]**
      6. **{**
      7. **"Hello",**
      8. **"World"**
      9. **};**
      10. **}**
   6. Press **F5** to debug the application. The default home page for a Web API project should appear.
      1. 
      2. Figure
      3. The default home page of an ASP.NET Web API application
   7. Press the F12 key to open the Developer Tools window. Click the **Network** tab, and then click the **Start Capturing** button to begin capturing network traffic into the window.
      1. 
      2. Figure
      3. Opening the network tab and initiating network capture
   8. Append the URL in the browser’s address bar with **/api/contact** and press enter. The transmission details will appear in the network capture window. Note that the response’s MIME type is **application/json**. This demonstrates how the default output format is JSON.
      1. 
      2. Figure
      3. Viewing the output of the Web API request in the Network view
      4. **Note:** Internet Explorer 9’s default behavior at this point will be to ask if the user would like to save or open the stream resulting from the Web API call. The output will be a text file containing the JSON result of the Web API URL call. Press the cancel button unless you want to see the text contents of the output.
   9. Click the **Go to detailed view** button to see more details about the response of this API call, then click the **Response body** tab to view the actual JSON response text.
      1. 
      2. Figure
      3. Viewing the JSON output text in the network monitor
2. Task 3 – Creating the Contact Models and Augment the Contact Controller
   1. In this task, you will create the controller classes in which API methods will reside.
   2. Right-click the **Models** folder and select **Add | New Item** from the context menu.
      1. 
      2. Figure
      3. Adding a new model to the web application
   3. In the **Add New Item** dialog, name the new file **Contact.cs** and click **Add.**
      1. 
      2. Figure
      3. Creating the new Contact class file
   4. Add the following code to the Contact class.
      1. (Code Snippet – *Web API Lab - Ex01 – Get API Method*)
      2. C#
      3. **public class Contact**
      4. **{**
      5. **public int Id { get; set; }**
      6. **public string Name { get; set; }**
      7. **}**
   5. In the **ContactController** class, select the word **string** in method definition of the Get method, and type the word **Contact**. Once the word is typed in, an indicator will appear at the beginning of the word *Contact*. Either hold down the Ctrl key and press the period (.) key or click the icon using your mouse to open up the assistance dialog in the code editor, to automatically fill in the **using** directive for the Models namespace.
      1. 
      2. Figure
      3. Using Intellisense assistance for namespace declarations
   6. Modify the code for the **Get** method so that it returns an array of Contact model instances.
      1. (Code Snippet – *Web API Lab - Ex01 – Returning a list of contacts*)
      2. C#
      3. **public Contact[] Get()**
      4. **{**
      5. **return new Contact[]**
      6. **{**
      7. **new Contact**
      8. **{**
      9. **Id = 1,**
      10. **Name = "Glenn Block"**
      11. **},**
      12. **new Contact**
      13. **{**
      14. **Id = 2,**
      15. **Name = "Dan Roth"**
      16. **}**
      17. **};**
      18. **}**
   7. Press **F5** to debug the web application in the browser. To view the changes made to the response output of the API, perform the following steps.
      1. Once the browser opens, press F12 to re-open the developer tools.
      2. Click the **Network** tab.
      3. Press the **Start Capturing** button.
      4. Add the URL suffix **/api/contact** to the URL in the address bar and press the Enter key.
      5. Press the **Go to detailed view** button.
      6. Select the **Response body** tab. You should see a JSON string representing the serialized form of an array of Contact instances.
         1. 
         2. Figure
         3. JSON serialized output of a complex Web API method call
3. Task 4 – Extracting Functionality into a Service Layer and Injecting it Using MEF 2.0
   1. This task will demonstrate how the Microsoft Managed Extensibility Framework (MEF) can provide more separation between controller logic and the services that offer functionality to Web API. MEF 2.0, which ships with the Microsoft .NET 4.5 Framework provides Dependency Injection features and makes it easy for developers to separate their service functionality from the controller layer, thereby allowing reusability of the services that actually do the work. MEF 2.0 is available via NuGet as a free component and is accessible via the Package Manager Console in Visual Studio 11.
   2. Click on the **Tools** menu and select **Library Package Manager | Package Manager Console** from the menu. The Package Manager Console window will open.
   3. In the Package Manager Window, type the command **Install-Package Microsoft.Mef.MvcCompositionProvider –Prerelease** in order to pull the package from NuGet and add it to the Web API project.
      1. 
      2. Figure
      3. Adding a reference to MEF 2.0 using NuGet
   4. A new folder will be added to the project during the MEF package install process. The **Parts** folder’s purpose is to contain any custom service interfaces and implementations required by the Web API application to function. Within the folder will be a new file by default, named **Part1.cs.** Delete this file, as it won’t be used by the Web API application.
      1. 
      2. Figure
      3. The MEF 2.0 Parts folder, which should contain service interfaces and implementations
   5. Right-click the **Parts** folder and select **Add | Class** from the context menu.
      1. 
      2. Figure
      3. Adding a new class to the Parts folder
   6. When the **Specify Name for Item** dialog appears, name the new class **ContactRepository** and click **OK**.
      1. 
      2. Figure
      3. Creating a class file to contain the code for the Contact Repository service layer
   7. Add a using directive to the **ContactRepository.cs** file to include the models namespace.
      1. C#
      2. **using ContactManager.Models;**
   8. Add the following code to the **ContactRepository.cs** file to implement the contact repository interface and implementation class.
      1. (Code Snippet – *Web API Lab - Ex01 – Contact Repository*)
      2. C#
      3. **public interface IContactRepository**
      4. **{**
      5. **Contact[] GetAllContacts();**
      6. **}**
      8. **public class ContactRepository : IContactRepository**
      9. **{**
      10. **public Contact[] GetAllContacts()**
      11. **{**
      12. **return new Contact[]**
      13. **{**
      14. **new Contact**
      15. **{**
      16. **Id = 1,**
      17. **Name = "Glenn Block"**
      18. **},**
      19. **new Contact**
      20. **{**
      21. **Id = 2,**
      22. **Name = "Dan Roth"**
      23. **}**
      24. **};**
      25. **}**
      26. **}**
      27. **Note:** The interface and class could be separated into separate files if preferred.
   9. Change the **HomeController.cs** code so that it supports constructor injection and add a private field to represent the instance supplied at construction, so that the rest of the class members can make use of the service implementation.
      1. (Code Snippet – *Web API Lab - Ex01 – Contact Repository Implementation*)
      2. C#
      3. **private IContactRepository \_contactRepository;**
      5. **public HomeController(IContactRepository contactRepo)**
      6. **{**
      7. **\_contactRepository = contactRepo;**
      8. **}**
   10. Put a breakpoint on the constructor of the **HomeController** class. Press F5 to begin a debugging session and note that the breakpoint is hit and an instance of the **ContactRepository** class has been passed to it.
       1. 
       2. Figure
       3. The HomeController when passed a service instance via Dependency Injection
   11. Add a reference to **System.Reflection.Context** to the project.
       1. 
       2. Figure
       3. Adding a reference
   12. Open the **Global.asax.cs** file in Visual Studio 11 if it is not already open in the IDE.
   13. Add the following namespaces to the list of namespaces already included in the file.
       1. (Code Snippet – *Web API Lab - Ex01 – Namespaces for MEF*)
       2. C#
       3. **using System.ComponentModel.Composition.Lightweight.Hosting;**
       4. **using System.ComponentModel.Composition.Registration;**
       5. **using System.ComponentModel.Composition.Web.Mvc;**
   14. Replace the **Application\_Start** method with the following code. This code configures the MEF 2.0 runtime so that all controller classes – both standard MVC controllers *and* Web API controllers – can use the same MEF service injection functionality.
       1. (Code Snippet – *Web API Lab - Ex01 – Global MEF Wireup*)
       2. C#
       3. **protected void Application\_Start()**
       4. **{**
       5. **AreaRegistration.RegisterAllAreas();**
       7. **RegisterGlobalFilters(GlobalFilters.Filters);**
       8. **RegisterRoutes(RouteTable.Routes);**
       10. **var conventions = new RegistrationBuilder ();**
       11. **conventions.ForTypesDerivedFrom<IController>().Export();**
       12. **conventions.ForTypesDerivedFrom<ApiController>().Export();**
       14. **conventions**
       15. **.ForTypesMatching((t) =>**
       16. **t.Namespace != null &&**
       17. **(t.Namespace.Contains(".Parts.") ||**
       18. **t.Namespace.EndsWith("Parts")))**
       19. **.Export()**
       20. **.ExportInterfaces();**
       22. **CompositionProvider.SetConfiguration(**
       23. **new ContainerConfiguration()**
       24. **.WithAssembly(typeof(WebApiApplication).Assembly, conventions)**
       25. **);**
       27. **GlobalConfiguration.Configuration.ServiceResolver.SetResolver(**
       28. **t => DependencyResolver.Current.GetService(t),**
       29. **t => DependencyResolver.Current.GetServices(t)**
       30. **);**
       31. **}**
       32. **Note:** This step is necessary at the time of the Visual Studio 11 beta release. Future releases will probably enhance automatic support for Web API and reduce the code required to wire MEF 2.0 up to those controllers.
   15. Open the **ContactController.cs** file if it is not already open.
   16. Add the following using statement to the namespace declaration section of the file.
       1. C#
       2. using ContactManager.Parts;
   17. Change the **ContactController.cs** code so that it supports constructor injection and add a private field to represent the instance supplied at construction, so that the rest of the class members can make use of the service implementation already in use by the HomeController class.
       1. (Code Snippet – *Web API Lab - Ex01 – Contact Controller*)
       2. C#
       3. **public class ContactController : ApiController**
       4. **{**
       5. **private IContactRepository \_contactRepository;**
       7. **public ContactController(IContactRepository contactRepo)**
       8. **{**
       9. **\_contactRepository = contactRepo;**
       10. **}**
       11. **}**
   18. Change the **Get** method so that it makes use of the contact repository service.
       1. (Code Snippet – *Web API Lab - Ex01 – Returning a list of contacts via the repository*)
       2. C#
       3. **public Contact[] Get()**
       4. **{**
       5. **return \_contactRepository.GetAllContacts();**
       6. **}**
   19. Put a breakpoint on the **ContactController’s** constructor, and on the **Get** method definition.
       1. 
       2. Figure
       3. Adding breakpoints to the contact controller
   20. Press F5 to run the application.
   21. When the browser opens, press F12 to open the developer tools.
   22. Click the **Network** tab.
   23. Click the **Start Capturing** button.
   24. Append the URL in the address bar with the suffix **/api/contact** and press the Enter key to load the API controller.
   25. Visual Studio 11 should break at the constructor.
       1. 
       2. Figure
       3. Breaking at the constructor of the contact controller during debug
   26. Press F5 to continue execution. Visual Studio 11 should break again, this time as the Get controller method begins execution.
       1. 
       2. Figure
       3. Breaking within the Get method
   27. Press F5 to continue.
   28. Go back to Internet Explorer if it is not already in focus. Note the network capture window.
       1. 
       2. Figure
       3. Network view in Internet Explorer showing results of the Web API call
   29. Click the Go to detailed view button.
   30. Click the **Response body** tab. Note the JSON output of the API call, and how it represents the two contacts retrieved by the service layer.
       1. 
       2. Figure
       3. Viewing the JSON output from the Web API in the developer tools window

Exercise 2: Create a Read/Write Web API

* 1. In this exercise, you will implement POST, PUT, and DELETE methods for the contact manager to enable it with data-editing features.

Task 1 – Opening the Web API Project

In this task, you will prepare to enhance the Web API project created in Exercise 1 so that it can accept user input.

* 1. Open Visual Studio 11 if it is not already running.
  2. Open the project created in Exercise 1, or open the project in the **Source/CS/Ex02/Begin/ContactManager** folder that accompanied this lab.
  3. Open the **Parts/ContactRepository.cs** file.

Task 2 – Adding Data-Persistence Features to the Contact Repository Implementation

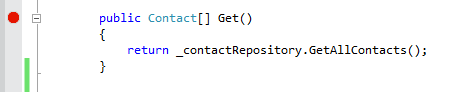
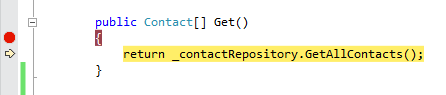
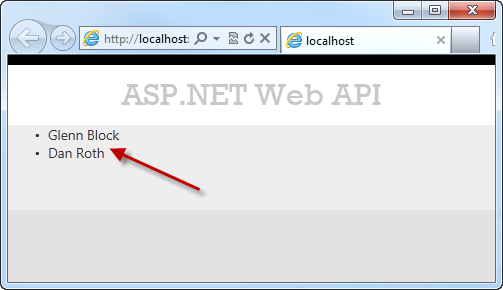
In this task, you will augment the ContactRepository class of the Web API project created in Exercise 1 so that it can persist and accept user input and new Contact instances.

* 1. Add the following constant to the class to represent the name of the web server cache item key name later in this exercise.
     1. C#
     2. const string cacheKey = "ContactStore";
  2. Add a constructor to the ContactRepository containing the following code.
     1. (Code Snippet – *Web API Lab - Ex02 – Contact Repository Constructor*)
     2. C#
     3. **public ContactRepository()**
     4. **{**
     5. **var ctx = HttpContext.Current;**
     7. **if (ctx != null)**
     8. **{**
     9. **if (ctx.Cache[cacheKey] == null)**
     10. **{**
     11. **var contacts = new Contact[]**
     12. **{**
     13. **new Contact**
     14. **{**
     15. **Id = 1, Name = "Glenn Block"**
     16. **},**
     17. **new Contact**
     18. **{**
     19. **Id = 2, Name = "Dan Roth"**
     20. **}**
     21. **};**
     23. **ctx.Cache[cacheKey] = contacts;**
     24. **}**
     25. **}**
     26. **}**
  3. Modify the code for the GetAllContacts method as demonstrated below.
     1. (Code Snippet – *Web API Lab - Ex02 – Get All Contacts*)
     2. C#
     3. **public Contact[] GetAllContacts()**
     4. **{**
     5. **var ctx = HttpContext.Current;**
     7. **if (ctx != null)**
     8. **{**
     9. **return (Contact[])ctx.Cache[cacheKey];**
     10. **}**
     12. **return new Contact[]**
     13. **{**
     14. **new Contact**
     15. **{**
     16. **Id = 0,**
     17. **Name = "Placeholder"**
     18. **}**
     19. **};**
     20. **}**
     21. **Note:** This example is for demonstration purposes and will use the web server’s cache as a storage medium, so that the values will be available to multiple clients simultaneously, rather than use a Session storage mechanism or a Request storage lifetime. One could use Entity Framework, XML storage, or any other variety in place of the web server cache.
  4. Add a new method named **SaveContact** to the **IContactRepository** interface to do the work of saving a contact. The **SaveContact** method should take a single **Contact** parameter and return a Boolean value indicating success or failure.
     1. (Code Snippet – *Web API Lab - Ex02 – Adding the SaveContact Method*)
     2. C#
     3. **public interface IContactRepository**
     4. **{**
     5. **Contact[] GetAllContacts();**
     6. **bool SaveContact(Contact contact);**
     7. **}**
  5. Implement the **SaveContact** method in the **ContactRepository** class by adding the following code.
     1. (Code Snippet – *Web API Lab - Ex02 – Implementing the SaveContact Method*)
     2. C#
     3. **public bool SaveContact(Contact contact)**
     4. **{**
     5. **var ctx = HttpContext.Current;**
     7. **if (ctx != null)**
     8. **{**
     9. **try**
     10. **{**
     11. **((Contact[])ctx.Cache[cacheKey]).ToList().Add(contact);**
     13. **return true;**
     14. **}**
     15. **catch (Exception ex)**
     16. **{**
     17. **Console.WriteLine(ex.ToString());**
     18. **return false;**
     19. **}**
     20. **}**
     22. **return false;**
     23. **}**

Exercise 3: Consume the Web API from an HTML Client

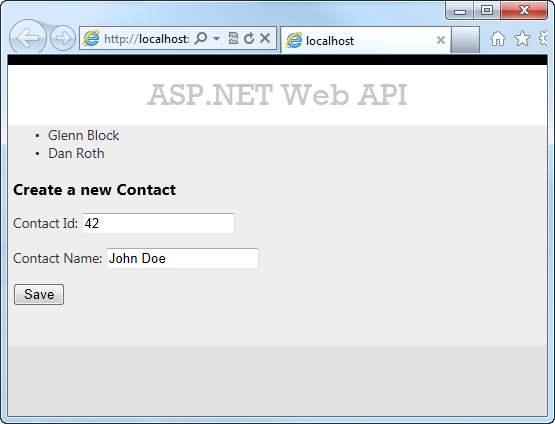
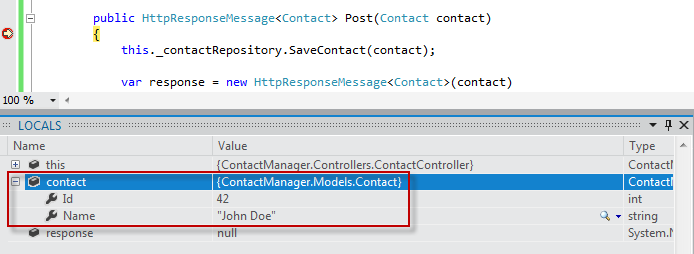
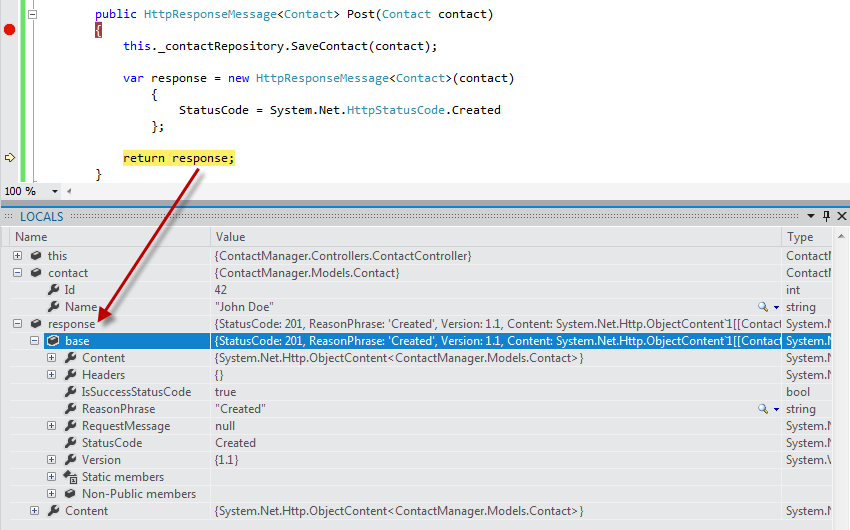
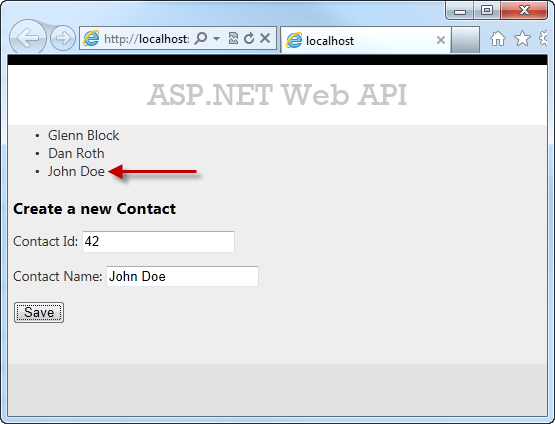
1. In this exercise, you will create an HTML client to call the Web API. This client will facilitate data exchange with the Web API using JavaScript and will display the results in a web browser using HTML markup.

Task 1 – Modifying the Index View to Provide a GUI for Displaying Contacts

* 1. In this task, you will modify the default Index view of the web application to support the requirement of displaying the list of existing contacts in an HTML browser.
  2. Open Visual Studio 11 if it is not already open.
  3. Open the **ContactManager** solution located in the **CS/Ex03/Begin/ContactManager** folder accompanying this lab.
  4. Open the **Views/Home/Index.cshtml** file.
  5. Add a script reference to load the jQuery JavaScript library at the top of the Index view.
     1. HTML
     2. **<script src="@Url.Content("~/Scripts/modernizr-2.0.6-development-only.js")" type="text/javascript"></script>**
  6. Remove the HTML code within the **body** div element so that the page is relatively bare.
     1. HTML
     2. ~~<body>~~
     3. ~~<header>~~
     4. ~~<div class="content-wrapper">~~
     5. ~~<div class="float-left">~~
     6. ~~<p class="site-title"><a href="/">ASP.NET Web API</a></p>~~
     7. ~~</div>~~
     8. ~~</div>~~
     9. ~~</header>~~
     10. ~~<div id="body">~~
     12. ~~</div>~~
     13. ~~</body>~~
  7. Add an unordered list (an HTML ul element) as a child element of the div tag. Give the unordered list an id attribute value of **contacts**.
     1. HTML
     2. **<div id="body">**
     3. **<ul id="contacts"></ul>**
     4. **</div>**
  8. Add a new JavaScript code block just before the closing **head** tag of the **Index.cshtml** view. This code will perform the HTTP request to the Web API.
     1. HTML
     2. **<script type="text/javascript">**
     3. **$(function()**
     4. **{**
     5. **$.getJSON('/api/contact', function(contactsJsonPayload)**
     6. **{**
     7. **$(contactsJsonPayload).each(function(i, item)**
     8. **{**
     9. **$('#contacts').append('<li>' + item.Name + '</li>');**
     10. **});**
     11. **});**
     12. **});**
     13. **</script>**
  9. Open the **ContactController.cs** file if it is not already open.
  10. Place a breakpoint on the **Get** method of the **ContactController** class.
      1. 
      2. Figure
      3. Placing a breakpoint on the Get method of the API controller
  11. Press the F5 key to run the project. The browser will load the HTML document.
  12. Once the page loads and the JavaScript executes, the breakpoint will be hit and the code execution will pause in the controller.
      1. 
      2. Figure
      3. Debugging into the Web API call using Visual Studio 11
  13. Press F5 or the debugging toolbar’s **Continue** button to continue loading the view in the browser. Once the Web API call completes you should see the contacts returned from the Web API call displayed as list items in the browser.
      1. 
      2. Figure
      3. Results of the API call displayed in the browser as list items

Task 2 – Modifying the Index View to Provide a GUI for Creating Contacts

In this task, you will continue to modify the Index view of the MVC application. A form will be added to the HTML page that will capture user input and send it to the Web API to create a new Contact, and a new Web API controller method will be created to collect date from the GUI.

* 1. Open the **ContactController.cs** file in Visual Studio 11 if it is not already open.
  2. Add a new method to the controller class named **Post** as demonstrated in the following code.
     1. (Code Snippet – *Web API Lab - Ex03 – Post Method*)
     2. C#
     3. **public HttpResponseMessage<Contact> Post(Contact contact)**
     4. **{**
     5. **this.\_contactRepository.SaveContact(contact);**
     7. **var response = new HttpResponseMessage<Contact>(contact)**
     8. **{**
     9. **StatusCode = System.Net.HttpStatusCode.Created**
     10. **};**
     12. **return response;**
     13. **}**
  3. Open the **Index.cshtml** file in Visual Studio if it is not already open.
  4. Add the HTML code below to the file just after the unordered list you added in the previous task.
     1. HTML
     2. **<form id="saveContactForm" method="post">**
     3. **<h3>Create a new Contact</h3>**
     4. **<p>**
     5. **<label for="contactId">Contact Id:</label>**
     6. **<input type="text" name="Id" />**
     7. **</p>**
     8. **<p>**
     9. **<label for="contactName">Contact Name:</label>**
     10. **<input type="text" name="Name" />**
     11. **</p>**
     12. **<input type="button" id="saveContact" value="Save" />**
     13. **</form>**
  5. Within the script element in the document’s head, add the following code to handle button-click events, which will post the data to the Web API using an HTTP POST call.
     1. HTML
     2. **$('#saveContact').click(function()**
     3. **{**
     4. **var nm = $('#contactName').val();**
     5. **var id = $('#contactId').val();**
     7. **$.post("api/contact",**
     8. **$("#saveContactForm").serialize(),**
     9. **function(value)**
     10. **{**
     11. **$('#contacts').append('<li>' + value.Name + '</li>');**
     12. **},**
     13. **"json"**
     14. **);**
     15. **});**
  6. In **ContactController.cs**, place a breakpoint on the **Post** method.
  7. Press the F5 key to run the application in the browser.
  8. Once the page is loaded in the browser, type in a new contact name and Id and click the **Save** button.
     1. 
     2. Figure
     3. The client HTML document loaded in the browser
  9. When the debugger window breaks in the **Post** method, take a look at the properties of the **contact** parameter. The values should match the data you entered in the form.
     1. 
     2. Figure
     3. The Contact object being sent to the Web API from the client
  10. Step through the method in the debugger until the **response** variable has been created. Upon inspection in the **Locals** window in the debugger, you’ll see that all the properties have been set.
      1. 
      2. Figure
      3. The response following creation in the debugger
  11. If you press **F5** or click **Continue** in the debugger the request will complete. Once you switch back to the browser, the new contact has been added to the list of contacts stored by the **ContactRepository** implementation.
      1. 
      2. Figure
      3. The browser reflects successful creation of the new contact instance

Summary

* 1. This lab has introduced you to the new ASP.NET Web API framework and to the implementation of RESTful Web API’s using the framework. From here, you could create a new repository that facilitates data persistence using any number of mechanisms and wire that service up rather than the simple one provided as an example in this lab. Web API supports a number of additional features, such as enabling communication from non-HTML clients written in any language that supports HTTP and JSON or XML. The ability to host a Web API outside of a typical web application is also possible, as well as is the ability to create your own serialization formats.
  2. The ASP.NET Web site has an area dedicated to the ASP.NET Web API framework at <http://asp.net/web-api>. This site will continue to provide late-breaking information, samples, and news related to Web API, so check it frequently if you’d like to delve deeper into the art of creating custom Web API’s available to virtually any device or development framework.