## Developing Custom RESTful Services with Web API

**Lab Time**: 60 minutes

**Lab Folder**: [c:\Student\Modules\WebAPI\Labs

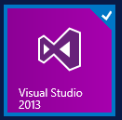
**Lab Overview**: In this lab you will create an OData service using Web API. After investigating the service with Fiddler, you will enable Cross Origin Resource Sharing (CORS), and create a SharePoint-Hosted app to consume the service using the data.js JavaScript library.

**Lab Setup**: Ensure you have a development site available for this lab

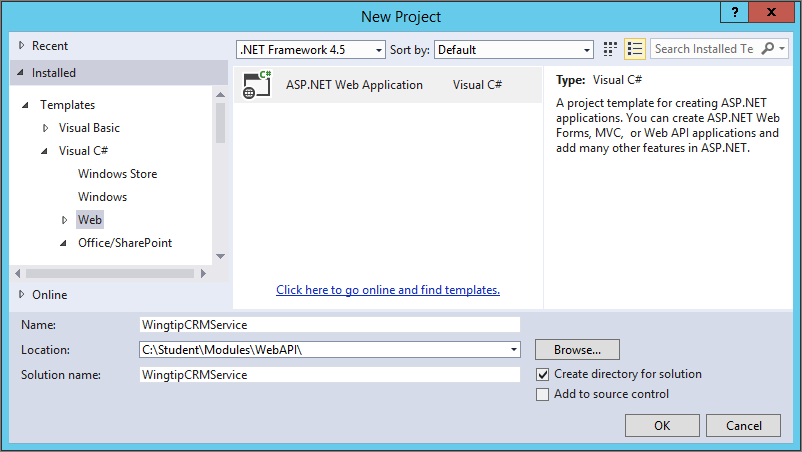
### Exercise 1: Create a RESTful Service

In this exercise you create a new OData service using Web API scaffolding and Entity Framework 6.0.

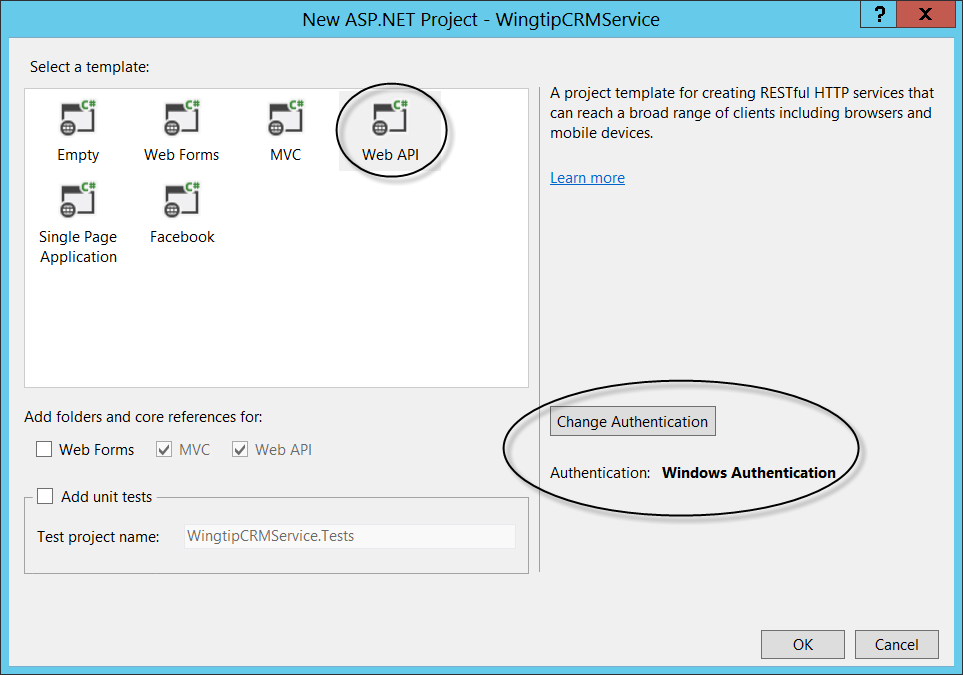
1. Launch **Visual Studio 2013** as administrator:
   1. **Windows** Keyboard Key 🡪 Right click on the Visual Studio 2013 tile and select **Run as administrator**.



1. Create the new solution in Visual Studio 2013:
   1. In Visual Studio select **File 🡪 New 🡪 Project**.
   2. In the **New Project** dialog:
      1. Select **Templates 🡪 Visual C# 🡪Web**.
      2. Click **ASP.NET Web Application**.
      3. Name the new project **WingtipCRMService**
      4. Set the Location to **C:\Student\Modules\WebAPI\**
      5. Click **OK**.

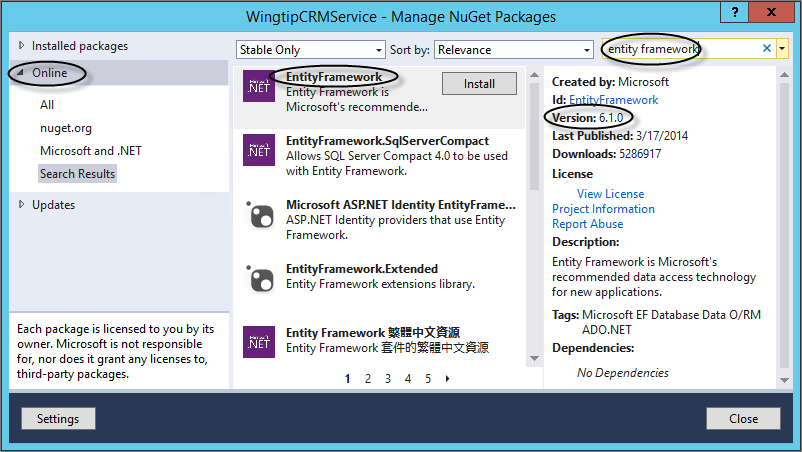


* 1. In the New ASP.NET Project dialog:
     1. Select the **Web API** template.
     2. Click **Change Authentication**.
     3. Select **Windows Authentication** and then click **OK**.

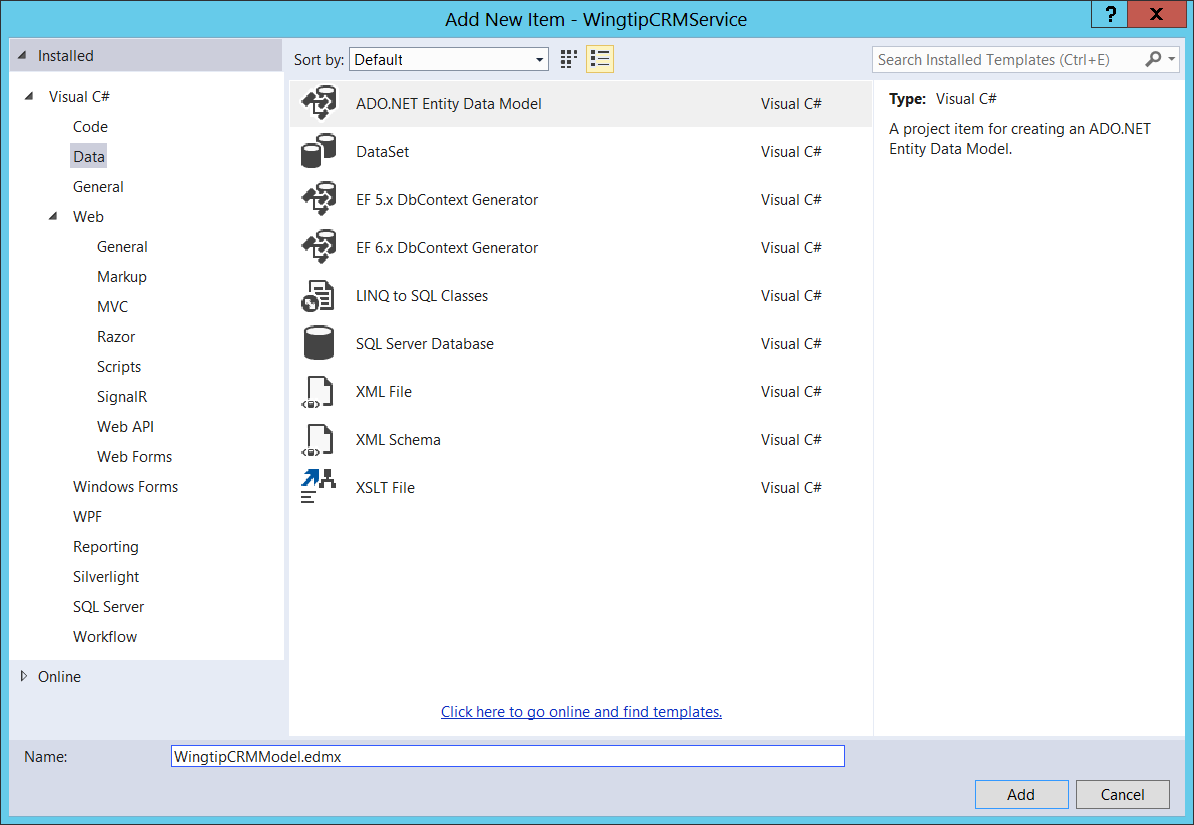


* + 1. Click **OK**.

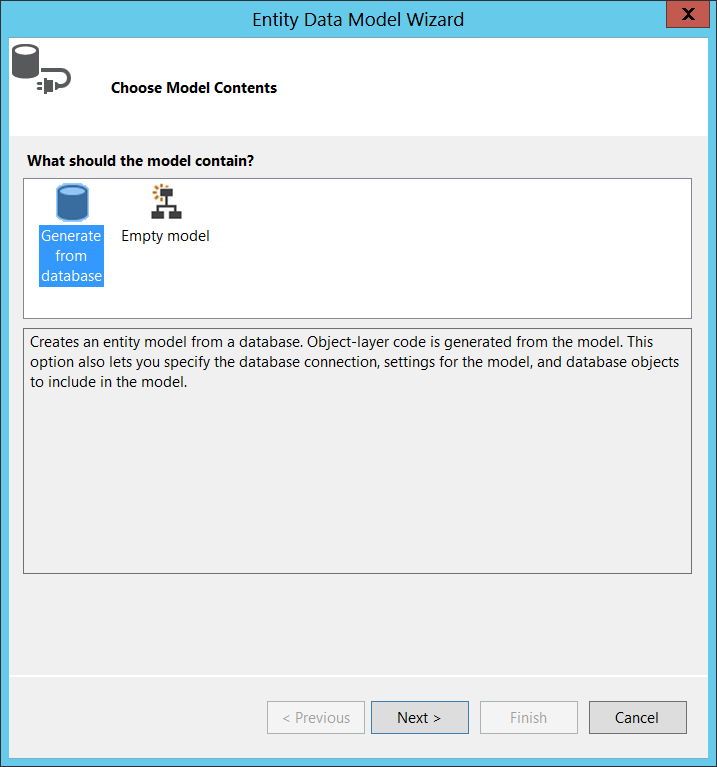
1. Add Entity Framework 6.0 Support
   1. In the **Solution Explorer**, right click the **WingtipCRMService** project node.
   2. Select **Manage NuGet Packages** from the context menu.
   3. On the Manage NuGet Packages page, select **Online** from the left side menu area
   4. Type **entity framework** in the search box.
   5. Locate the **Entity Framework 6.x** package and click **Install**.
      1. If prompted click **I** **Accept** to accept the license.
      2. Click Close to close the **Manage NuGet Packages** page



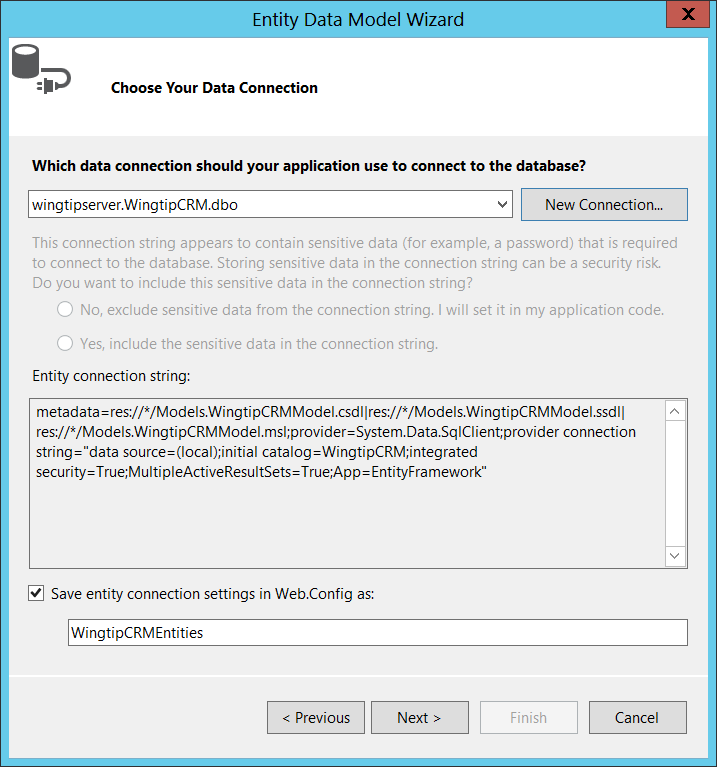
1. Add an Entity Framework Model
   1. In the **Solution Explorer**, right-click the **Models** folder.
   2. Select **Add🡪New** **Item** from the context menu.
   3. In the Add New item dialog:
      1. Click **Visual C#🡪Data**.
      2. Select **ADO.NET Entity Data Model**
      3. Name the model **WingtipCRMModel**.



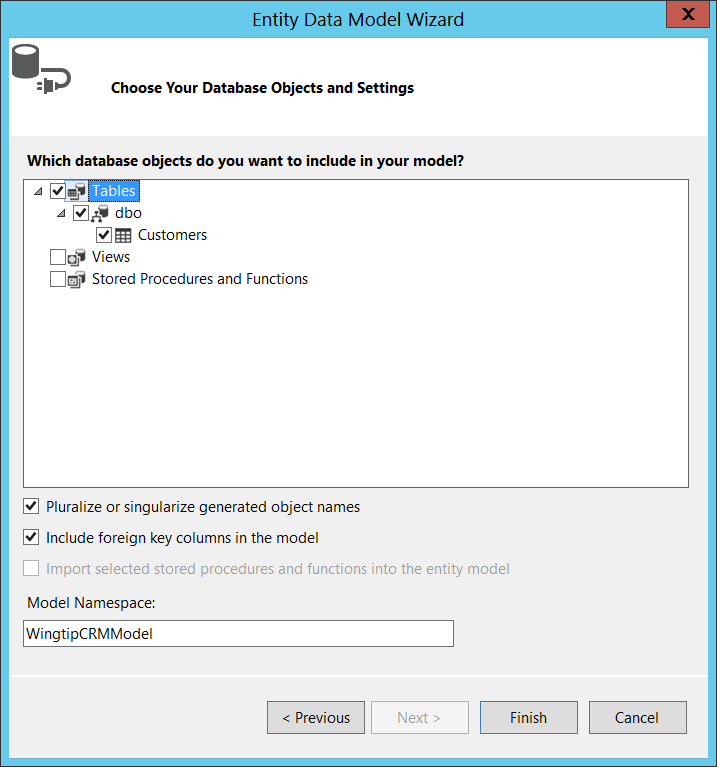
* + 1. Click **Add**.
  1. In the first screen of the **Entity Data Model Wizard**:
     1. Select **Generate from Database**.
     2. Click **Next**.



* 1. In the first second screen of the **Entity Data Model Wizard**:
     1. Click **New Connection**.
     2. Enter **(local)** in the Server Name field.
     3. Select **WingtipCRM** as the database name.
     4. Click **Test Connection**.
     5. Click **OK**. Click **OK** again
     6. Click **Next**.



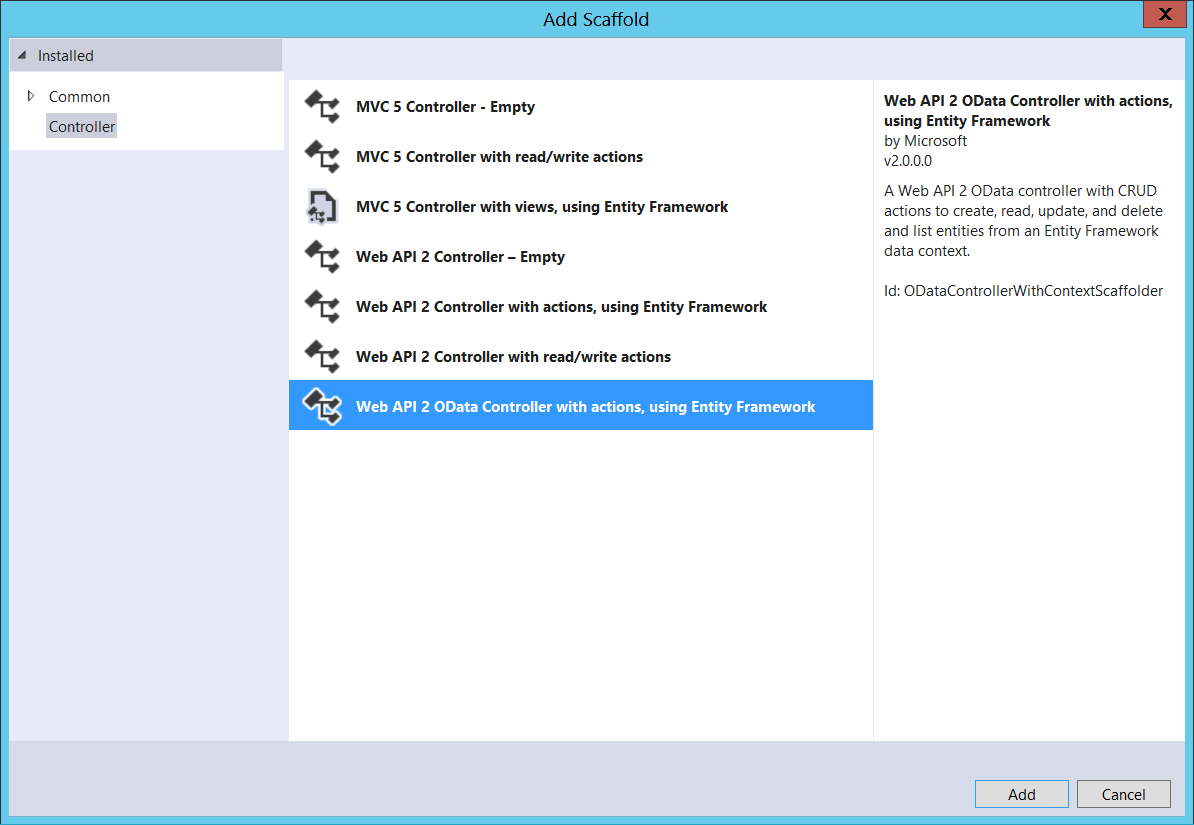
* 1. In the first third screen of the **Entity Data Model Wizard**:
     1. Expand **Tables**.
     2. Expand **dbo.**
     3. Select **Customers**.
     4. Click **Finish**.



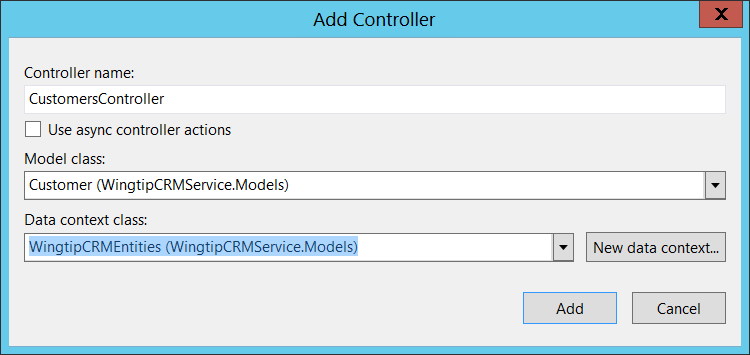
1. Build the project.
   1. Select **Build 🡪 Build Solution** from the main menu in Visual Studio
   2. Select **Window 🡪 Close All Documents** to clean up your workspace.

The project must be built before proceeding. Otherwise, Visual Studio will not be able to properly utilize the Entity Framework model to make a new controller.

1. Add an OData Controller
   1. In the **Solution Explorer**, right click the **Controllers** folder.
   2. Select **Add🡪Controller** from the context menu.
   3. In the Add Scaffold dialog:
      1. Select **Web API 2 OData Controller, with actions using Entity Framework.**
      2. Click **Add**.



* 1. In the Add Controller dialog:
     1. Name the new controller **CustomersController.**
     2. Select **Customer (WingtipCRMService.Models)** in the Model Class drop-down list.
     3. Select **WingtipCRMEntities (WingtipCRMService.Models)** in the Data Context Class drop-down list.
     4. Click **Add**.



1. Define the controller routes.
   1. Open the **CustomersController.cs** file that was just generated.
   2. Locate the following commented code and **Copy** it to the clipboard:

/\*

To add a route for this controller, merge these statements into the Register method of the WebApiConfig class. Note that OData URLs are case sensitive.

using System.Web.Http.OData.Builder;

using WingtipCRMService.Models;

ODataConventionModelBuilder builder = new ODataConventionModelBuilder();

builder.EntitySet<Customer>("Customers");

config.Routes.MapODataRoute("odata", "odata", builder.GetEdmModel());

\*/

* 1. Expand the **App\_Start** folder.
  2. Open the **WebApiConfig.cs** file.
  3. **Paste** the following using statements from the copied code:

using System.Web.Http.OData.Builder;

using WingtipCRMService.Models;

* 1. **Paste** the following code from the copied code as the **top** of the **Register** method just below the **//Web API configuration and services** comment:

//Web API configuration and services

ODataConventionModelBuilder builder = new ODataConventionModelBuilder();

builder.EntitySet<Customer>("Customers");

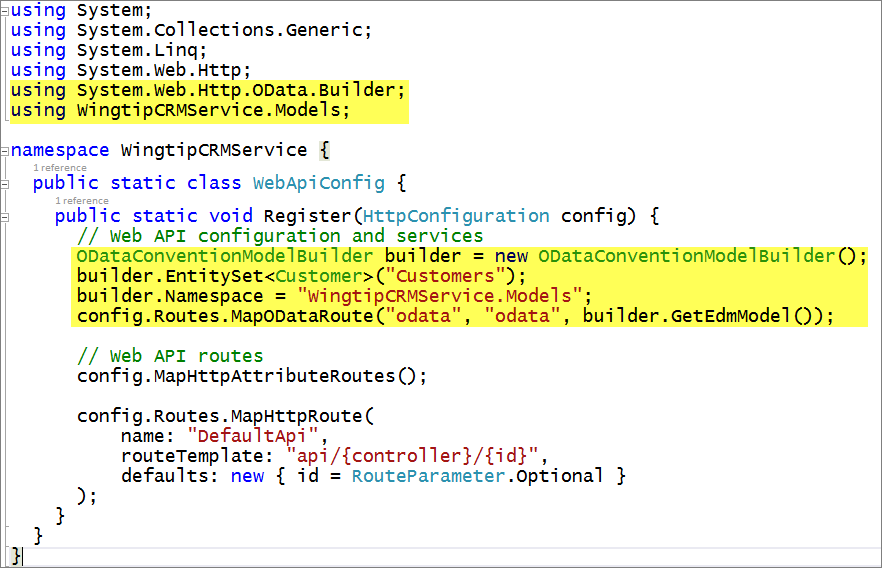
config.Routes.MapODataRoute("odata", "odata", builder.GetEdmModel());

The routes for the OData controller are created by the code in the Register method. For OData controllers, the ODataConventionModelBuilder class will build out routes that have an “odata” prefix and a “Customers” endpoint (e.g., http://localhost:60982/odata/Customers). The order in which routes are defined is important. If routes conflict, the first definition will be used.

* 1. **Add** the following line of code after the **builder.EntitySet<Customer>(“Customers”)** line to include namespace information in the OData model:

builder.Namespace = "WingtipCRMService.Models";

The ODataConventionModelBuilder class will also build a schema that supports the $metadata endpoint. The class normally generates a schema without namespace information. So, you must manually add the namespace, which is normally the same as the namespace used in the Entity Framework model.



1. Build the project.
   1. Select **Build🡪Rebuild Solution** from the main menu in Visual Studio
   2. Select **Window🡪Close All Documents** to clean up your workspace.

### Exercise 2: Testing the OData Service with Fiddler

In this exercise, you will use Fiddler to manually test the service.

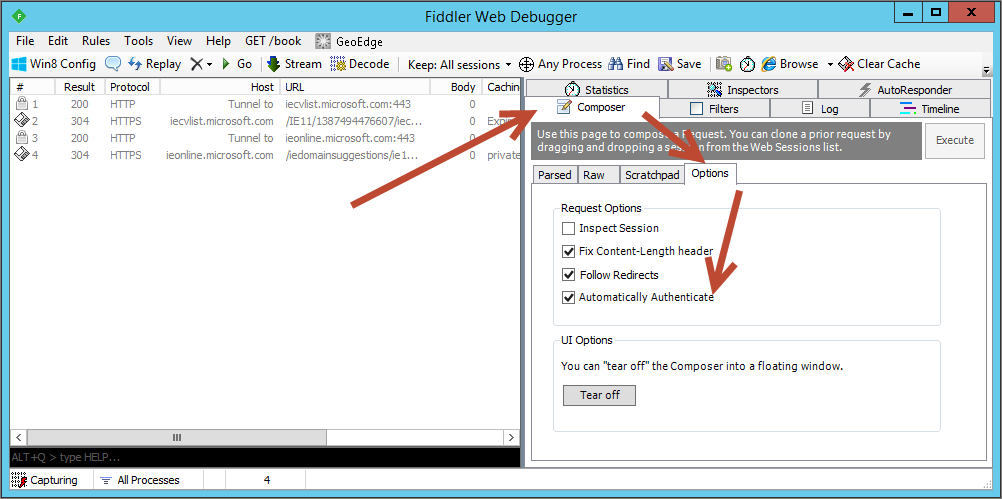
1. Start the service for testing.
   1. Launch **Fiddler2**. (press your **Windows** key and then type “**fiddler**” and click on the **Fiddler2** tile).
   2. In Visual Studio 2013, open **CustomersController.cs**
      1. Place a breakpoint in the following methods: **GetCustomers**, **GetCustomer**, **Put**, **Post**, **Patch**, and **Delete**.
   3. In Visual Studio 2013, using the main menu, select **Debug🡪Start Debugging**.
      1. If prompted, log in with your Windows credentials.
      2. Verify that the default page of the project appears in the browser.

The default Web API project includes a visual home page that you can use to provide information about the service. Remember that it’s easy to mix MVC controllers with Web API controllers in the same project.

1. Testing methods from Fiddler
   1. In the browser, note of the root URI for the service. This should reference the localhost along with a port number (e.g., <http://localhost:60982>).

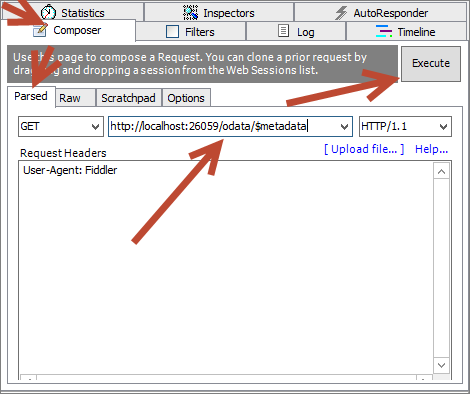
Be certain to write your port number down you will need this later

* 1. In Fiddler, select **Edit🡪Remove🡪All Sessions** to clear the display.
  2. In the **Composer** tab:
     1. Click the **Options** tab.
     2. Check **Automatically Authenticate**.



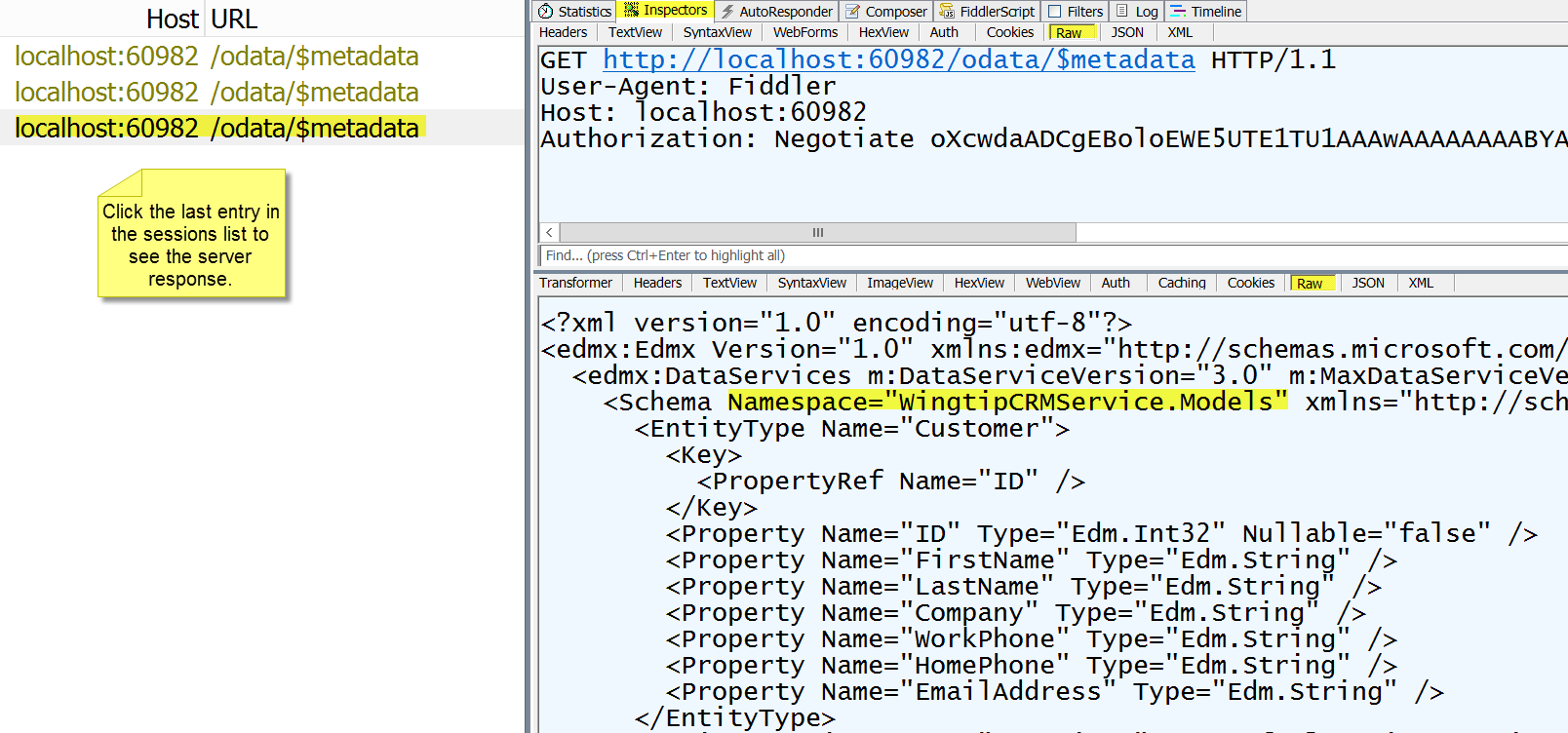
The option to “Automatically Authenticate” allows Fiddler to respond to Windows Authentication challenges from the OData service.

* + 1. On the **Composer** tab, click the **Parsed** tab.
    2. Enter the root URI plus **/odata/$metadata** (e.g., http://localhost:60982/odata/$metadata).
    3. Click **Execute**.



The Composer tab allows you to manually construct HTTP requests that Fiddler will send to the OData service.

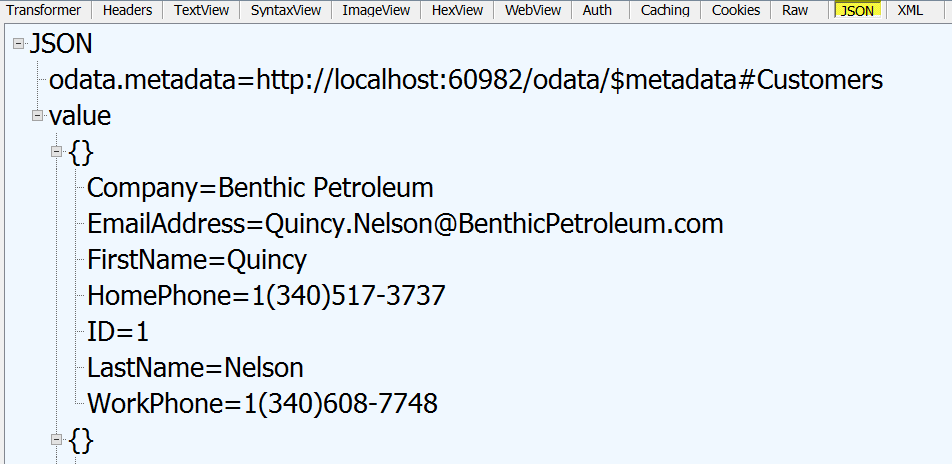
* 1. After the service responds, double click the last session line in Fiddler, which represents the response from the server.
  2. In the **Inspectors** tab:
     1. Click the **Raw** tab in the upper half.
     2. Click the **Raw** tab in the lower half.



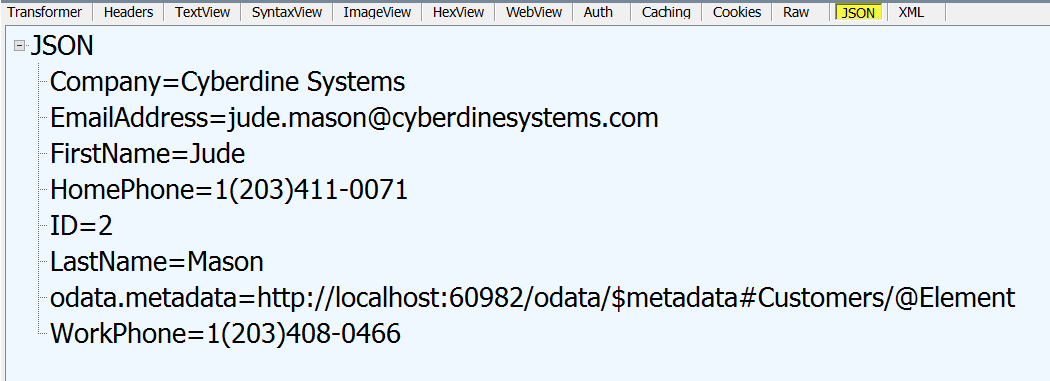
* + 1. View the schema results and note the namespace you defined in the OData service.

The Inspectors tab allows you to see the request in the top-half and the response in the lower-half. Different renderings of the data (e.g., “Headers”, “Raw”, “XML”, “JSON”) allow you to see different parts of the request and response formatted appropriately.

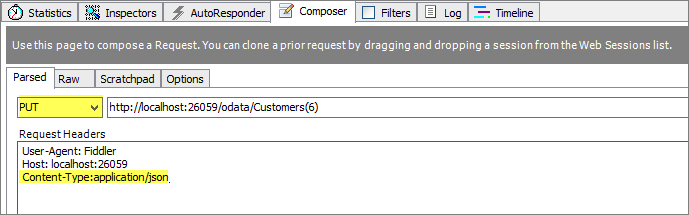
* 1. In the **Composer** tab:
     1. Click the **Parsed** tab.
     2. Enter the root URI plus **/odata/Customers** (e.g., http://localhost:60982/odata/Customers).
     3. Click **Execute**.
     4. On the breakpoint, note the method that was called and then press **F5** to continue.
  2. After the service responds, double click the last session line in Fiddler, which represents the response from the server.
  3. In the **Inspectors** tab:
     1. Click the **Headers** tab in the upper half.
     2. Click the **JSON** tab in the lower half.
     3. View the JSON results from the initial call.



* 1. In the **Composer** tab:
     1. Change the URI to **/odata/Customers(2)** (e.g., <http://localhost:60982/odata/Customers(2))>.
     2. Click **Execute**.  
        Note: if you receive a 404 error… pick another number and try again (i.e. you may have deleted that record in a prior lab).
     3. On the breakpoint, note the method that was called and then press **F5** to continue.
  2. After the service responds, click the last session line in Fiddler, which represents the response from the server.
  3. In the **Inspectors** tab:
     1. Click the **Headers** tab in the upper half.
     2. Click the **JSON** tab in the lower half and verify a single record was returned.



* 1. In the **Composer** tab:
     1. Change the HTTP verb to **PUT**.
     2. Add a new request header **Content-Type:application/json**.



The Content-Type header specifies the form of the request data you will send to the server.

* + 1. Add the following request body (in Fiddler2 **Request Body** is the next text box below the Request Headers) to provide updated information:  
       Note: if ID 2 did not work for you above (in step 2.i) please pick the same number that you used above

{

"ID":2,

"Company":"Critical Path Training",

"EmailAddress":"jude.mason@criticalpathtraining.com",

"FirstName":"Jude",

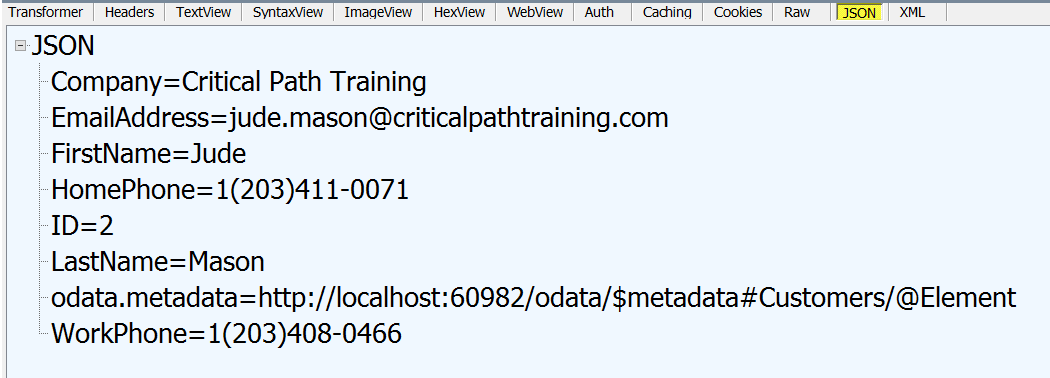
"HomePhone":"1(203)411-0071",

"LastName":"Mason",

"WorkPhone":"1(203)408-0466"

}

* + 1. Click **Execute**.
    2. On the breakpoint, hit **F5** to continue.
    3. Back on the Composers tab, change the HTTP verb from PUT to **GET**.
    4. Delete the **Request Body** text.
    5. Delete the **Content-Type:application/json** Request Header.
    6. Click **Execute**.
    7. On the breakpoint, hit **F5** to continue.
  1. After the service responds, click the last session line in Fiddler, which represents the response from the server.
  2. In the **Inspectors** tab:
     1. Click the **Headers** tab in the upper half.
     2. Click the **JSON** tab in the lower half and verify the data changes were made.

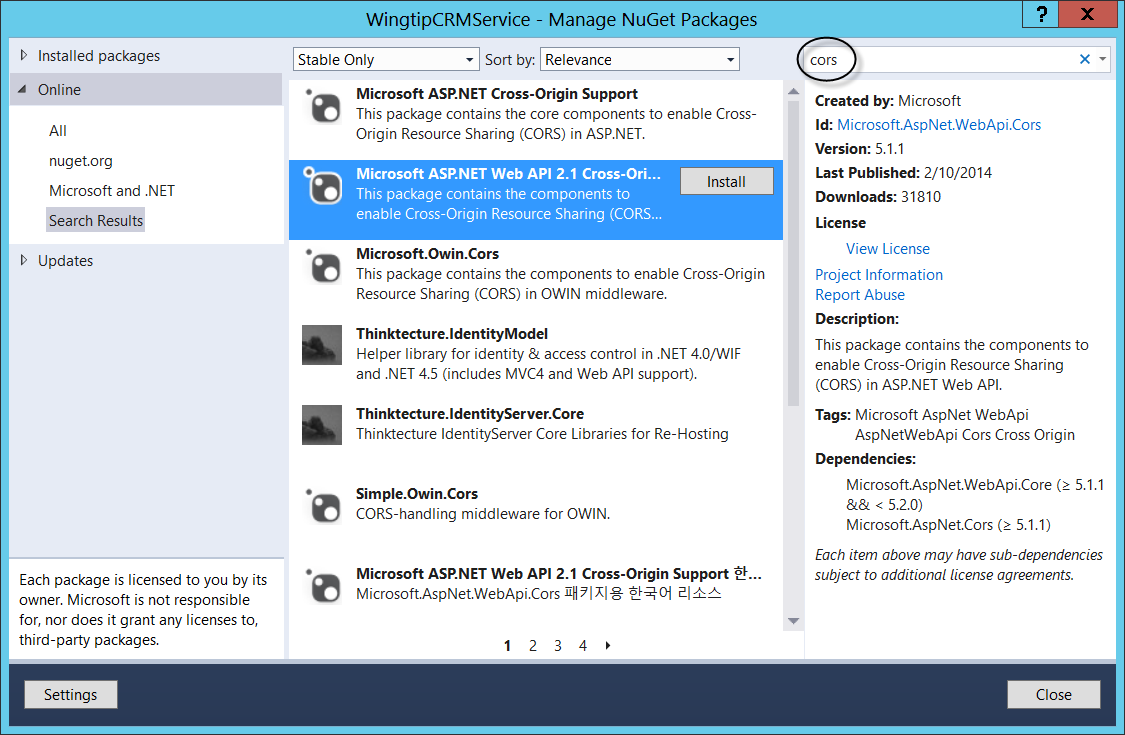


* 1. **Close** the browser to stop debugging.
  2. **Close** Fiddler.

### Exercise 3: Create a SharePoint-Hosted App

In this exercise, you will enable Cross Origin Resource Sharing (CORS) for the OData service and create a SharePoint-Hosted app to consume the service.

1. Enable Cross Origin Resource Sharing:
   1. In the **Solution Explorer**, right click the **WingtipCRMService** project node.
   2. Select **Manage NuGet Packages** from the context menu.
      1. Type **cors** in the search box.
      2. Locate the **Microsoft ASP.Net Web API 2 Cross Origin Support** package and click Install.
         1. If prompted click **I Accept** to accept the licenses.
      3. **Close** the NuGet Package Manager after the installation is complete.



* 1. Expand the **App\_Start** folder.
  2. Open the file **WebApiConfig.cs**.
  3. Add the following code at the bottom of the Register method to enable CORS for the entire service as shown below:

...

// Web API routes

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

config.EnableCors();

}

}

} ();

* 1. Expand the **Controllers** folder.
  2. Open the file **CustomersController.cs**.
  3. Add the following using statement.

using System.Web.Http.Cors;

* 1. Add the following attribute above the class definition (i.e. just above public class **CustomersController : ODataController**).

[EnableCors("\*","\*","\*","DataServiceVersion,MaxDataServiceVersion")]

You may enable CORS for specific origins, methods, or custom “author” headers. You may also specify what additional headers to return. In the code above, all origins, methods, and headers are accepted. Additionally, the service returns the DataServiceVersion and MaxDataServiceVersion headers, which are expected by the Chrome browser. In a production application, you would always limit the accepted origins to those that you trust.

1. Add the client app to your solution
   1. In the Starter Files folder for this lab (C:/Student/Modules/WebAPI/StarterFiles), locate the folder **WingtipCRMClient**.
   2. Copy this entire folder into the working directory you are using for this lab (i.e. C:/Student/Modules/WebAPI/WingtipCRMService/).
   3. In Visual Studio 2013, select **File🡪Add🡪Exiting Project**.
   4. Browse to the Project folder you just copied (C:\Student\Modules\WebAPI\WingtipCRMService\WingtipCRMClient)
   5. Add the **WingtipCRMClient** project to your solution.
      1. Select the **WingtipCRMClient.csproj** file and click **Open**

Other labs in the class will deal explicitly with creating JavaScript-based apps and binding data to web pages. For this lab, you will focus on the operations necessary to read and write to the OData service.

* 1. In the Solution explorer, expand the **scripts** folder and note the **datajs-1.1.2** library.

Datajs is a library specifically intended to be used with OData services. It supports data caching to make it easy to handle large data sets efficiently.

* 1. In the scripts folder, open the **wingtip.crm.viewmodel.js** library for editing.

The wingtip.crm.viewmodel library is used to wrap the datajs operations and generate a view model that can be bound to the web page. In this exercise, you’ll edit the parts of the library that deal with datajs.

* 1. Locate the comment **//TO DO: Update the Service Root** and edit the service root to point to your OData endpoint.

Note: this is the port number you were asked to write down in the earlier exercise.

Warning: Be sure that the service root ends with a forward slash.

* 1. Locate the comment **//TO DO: Edit refreshCache function** and add the following code as shown below:

var refreshCache = function () {

//TO DO: Edit refreshCache function

if (cache !== null) cache.clear();

cache = datajs.createDataCache({

name: "contacts",

source: serviceRoot + "odata/Customers?$orderby=LastName,FirstName&$top=100",

prefetchSize: 100,

pageSize: 100

});

loadCustomers();

};

The refreshCache function loads the cache with data from the OData service. Read operations for the app can then be performed against the cache instead of the OData service.

* 1. Locate the comment **//TO DO: Edit loadCustomers function** and add the following code as shown below:

var loadCustomers = function () {

//TO DO: Edit loadCustomers function

return cache.readRange(0, 100).then(

function (data) {

onLoadCustomersComplete(data);

},

function (err) {

onError(err);

}

);

};

The loadCustomers function reads all of the data from the cache. Once read, it calls the onLoadCustomersComplete function, which will bind the data to a table for viewing.

* 1. Locate the comment **//TO DO: Edit getCustomer function** and add the following code as shown below:

var getCustomer = function (Id) {

//TO DO: Edit getCustomer function

var deferred = jQuery.Deferred();

cache.readRange(0, 100).always(function (data) {

deferred.resolve(

\_.chain(data)

.filter(function (contact) { return (contact.ID === Id); })

.rest(0)

.first()

.value()

);

});

return deferred.promise();

}

The getCustomer function returns a single customer record from the cache. This is accomplished using an additional third-party JavaScript library named underscore.js. Underscore provides many useful utility functions for manipulating data sets.

* 1. Locate the comment **//TO DO: Edit addCustomer function** and add the following code as shown below:

var addCustomer = function (FirstName, LastName, Company, EmailAddress, WorkPhone, HomePhone) {

//TO DO: Edit addCustomer function

OData.request(

{

requestUri: serviceRoot + "odata/Customers",

method: "POST",

contentType: "application/json",

data: {

'LastName': LastName,

'FirstName': FirstName,

'Company': Company,

'EmailAddress': EmailAddress,

'WorkPhone': WorkPhone,

'HomePhone': HomePhone

},

headers: {

"accept": "application/json"

}

}, function (data) {

refreshCache();

}, function (err) {

refreshCache();

}

);

};

The addCustomer function makes a POST to the OData service to add a new customer then calls refreshCache to display the data.

* 1. Locate the comment **//TO DO: Edit updateCustomer function** and add the following code as shown below:

var updateCustomer = function (Id, FirstName, LastName, Company, EmailAddress, WorkPhone, HomePhone) {

//TO DO: Edit updateCustomer function

OData.request(

{

requestUri: serviceRoot + "odata/Customers(" + Id + ")",

method: "PUT",

contentType: "application/json",

data: {

'ID': Id,

'LastName': LastName,

'FirstName': FirstName,

'Company': Company,

'EmailAddress': EmailAddress,

'WorkPhone': WorkPhone,

'HomePhone': HomePhone

},

headers: {

"accept": "application/json"

}

}, function (data) {

refreshCache();

}, function (err) {

refreshCache();

}

);

};

* 1. Locate the comment **//TO DO: Edit deleteCustomer function** and add the following code as shown below:

var deleteCustomer = function (Id) {

//TO DO: Edit deleteCustomer function

OData.request(

{

requestUri: serviceRoot + "odata/Customers(" + Id + ")",

method: "DELETE",

headers: {

"accept": "application/json"

}

}, function (data) {

refreshCache();

}, function (err) {

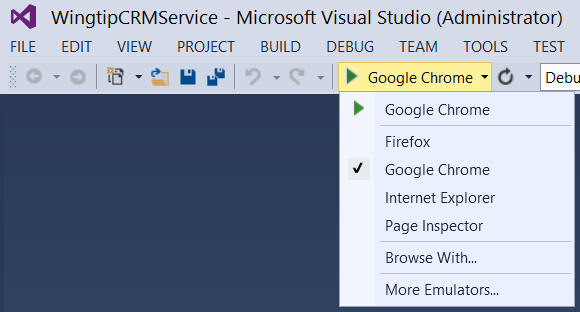
refreshCache();

}

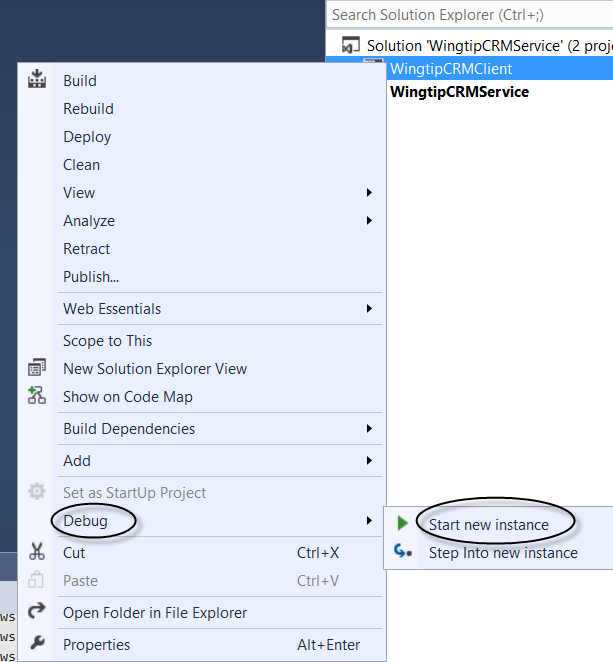
);

};

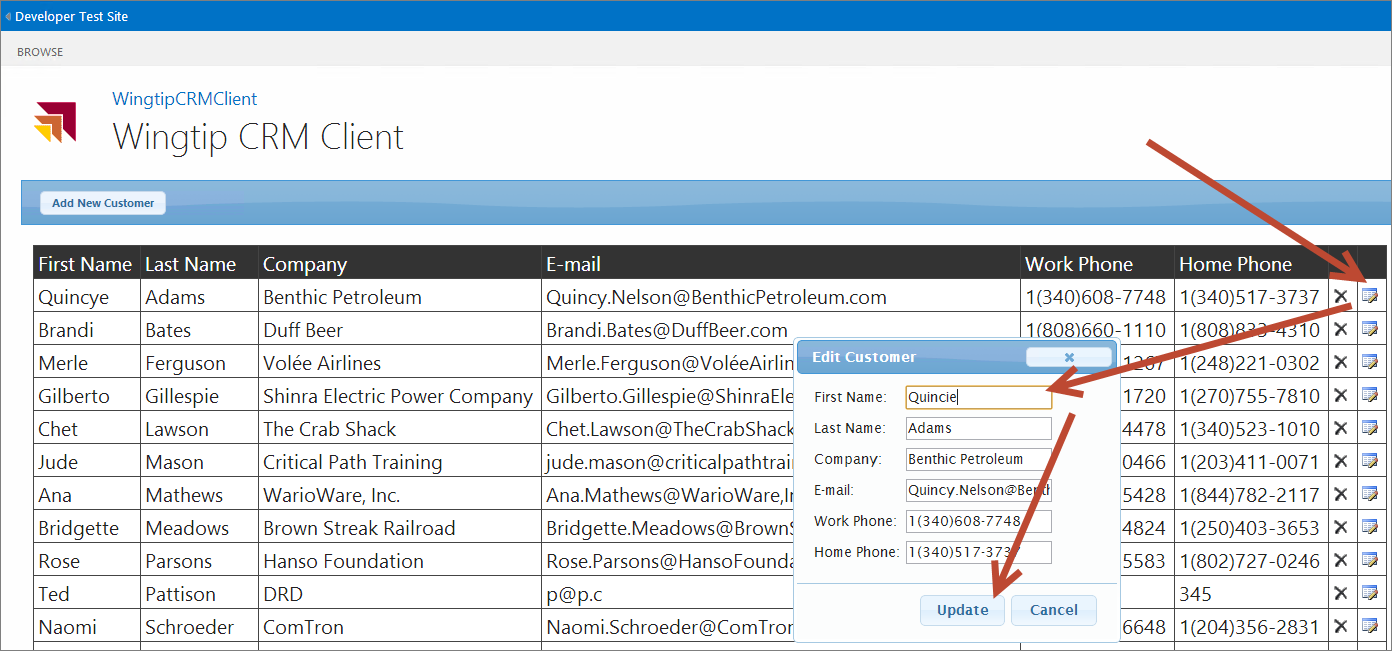
1. Test the solution
   1. In **Visual Studio Solution Explorer** select **WingtipCRMClient** and in the **Properties** window below set the **Start Action** to **Google Chrome**.
   2. Now, in Visual Studio, click on your **CustomersController.cs** tab (or if needed re-open this file from the WingtipCRMService project Controllers folder). (Note: we must be inside a file that renders in a web browser for this next feature of Visual Studio to work correctly)
   3. In Visual Studio, change the browser that will run the application to Google Chrome; using the Run drop-down arrow as show below, select Google Chrome as your browser choice.



* 1. Press **F5** to start debugging the solution.
  2. If prompted, log in using your Windows credentials.
  3. After the OData service starts, launch **Fiddler**.
     1. Press your **windows** key and type “**fiddler**” then click on the Fiddler2 tile to start Fiddler.
  4. In Visual Studio 2013, return to your **Solution Explorer** tab (Note: you may need to click on the Solution Explorer tab on the right hand side of the page) right click the WingtipCRMClient project node and select **Debug🡪Start New Instance** to run the associated app.

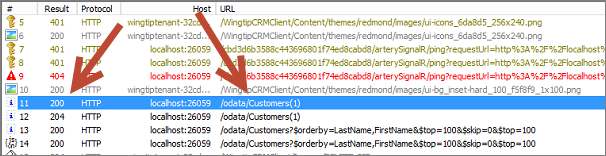


* 1. After the app starts, (This may take a while… be patient until a new tab opens in Chrome)
  2. If you are prompted by Microsoft Office for permission to run at the top of this new tab in Chrome click **Always run on this site**.
  3. Back in Fiddler, using the main menu, select **Edit🡪Remove🡪All Sessions** to clear the display.
  4. Back in Chrome, Edit a record in the **Wingtip CRM Client** grid, by clicking on the Edit button on the far right hand side of the row, and edit part of the information as shown below:

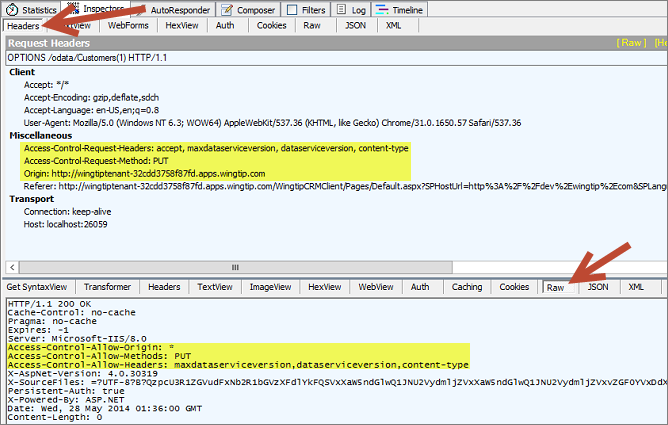


* 1. After saving the changes, by clicking the **Update** button, examine the PUT operation in Fiddler by clicking on the correct operation in the results area on the left side of the application and note the Cross Origin Resource Headers that were sent between the client and server.

(Note: the operation you are looking for is the one that is the PUT operation; that is, the first /**odata/Customers(1)** operation with a Result code of 200 as shown in the sample image below)



When this is opened in Fiddler you should see text similar to the text below:



1. Feel free to use this same process to try and investigate other operations on the data (Add a new Customer or Delete a Customer (be sure to just delete the customer you created in the Add step to keep the Dataset in good shape))
2. When you are finished, you should close Chrome and Visual Studio 2013.

Congratulations! You have just created an OData service using Web API. You have learned how to use Fiddler to investigate the inner workings of this API. Next you successfully enabled Cross Origin Resource Sharing (CORS), and then created and tested, using Fiddler) a SharePoint-Hosted app to consume the service using the data.js JavaScript library.