# ASP.NET Web API Student Service CRUD Demo with XML Datastore & Client "Tester" Console Application

## About

This project presents an ASP.NET Web API application that demos CRUD (create, read, update, and delete) operates on data in the web application while serializing/deserializing in an XML Datastore. Also, a client “tester” console application tests the CRUD operations with the HTTP protocol.

## Architecture

The demo project consists of these component topics:

* ASP.NET Web API Application
  + Update-to-Date Web API Components
  + XML Datastore
  + Data Models with XML Serializable Attributes
  + Web API Controller with CRUD Methods/Actions
* Web API Client “Tester to Service” Console Application
  + Client Local Data Models
  + Helper Classes
  + CRUD Operations Testing

## ASP.NET Web API Application

This project demos the ASP.NET Web API portion only, not the MVC. I did not build a fancy front-end webpage for the user interface. However, the project did build help documentation for the Web API service from the XML comments in the controller code. The API help document page can be access from the ASP.NET Web API main page and clicking on “help” from the main menu. See the entire ASP.NET Web API Application project code directory in the GitHub repo here [].

### Update-to-Date Web API Components

Packages from *Microsoft.AspNet.WebApi* were installed since the Web API updates are released out of band from the .Net framework. The Web API application should be deployed with the most recent and complete Web API package so that the project is not dependent on the localized .Net framework code of Web API, wherever it is hosted. *Note: Use the package manager to update the installed packages before a deployment.* See the project package code directory in the GitHub repo here [].

### XML Datastore

This data for this demo project is serialized/deserialized with the simple file “*students.xml*” stored in the *App\_Data* directory of the web application project. To see it in the solution explorer you may need to click on the “*Show All Files*”. The file’s *Build Action* property, is set as “Content”. See the section “*Web API Controller with CRUD methods/actions*” below, to learn how the controller serialized/deserialized to this XML file. See the demo students.xml file in the GitHub repo here [].

### Data Models with XML Serializable Attributes

The data class models used for this service are: Student and a StudentList (List<Student>) of Student objects.

#### Student

The student class data model with XML attributes for serialization is shown below. The properties are self-explanatory by names. Note: The student “id” property is not a child XML element but an attribute on the Student parent object. Also, it is not required to specify XML serialization attributes for enum types as they automatically serialize (only if you wanted to exclude a particular enum value, would you then specify the XML serialization). The Student class data model file is available in the GitHub repo here [].

/// <summary>

/// Student Data Model

/// </summary>

[XmlType(TypeName = "student")]

public class Student

{

/// <summary>

/// Enumeration for Student Grade level

/// </summary>

public enum GradeEnum

{

PreSchool = -1,

Kindergarten = 0,

First = 1,

Second,

Third,

Fourth,

Fifth,

Sixth,

Seventh,

Eighth,

Freshmen,

Sophomore,

Junior,

Senior,

College

} // end of enum

[XmlAttribute(AttributeName = "id")]

public int ID { get; set; }

[XmlElement(ElementName = "lastName")]

public string LastName { get; set; }

[XmlElement(ElementName = "firstName")]

public string FirstName { get; set; }

[XmlElement(ElementName = "dob")]

public DateTime DOB { get; set; }

[XmlElement(ElementName = "gpa")]

public float GPA { get; set; }

[XmlElement(ElementName = "grade")]

public GradeEnum Grade { get; set; }

#### StudentList

The StudentList class data model with XML attributes is shown below. The *Load()* method deserializes from the xml file into memory all the Student objects into a StudentList object which inherits from List<Student> and returns the entire Datastore memory object to the caller. *Note: The Deserialization method is static and called on the class level.*

Serialization is done by the *Save()* method, where the caller is an *instance* object of this class type StudentList containing the entire in memory Datastore of Student objects. The caller will be a StudentList object that contains a List of Student objects per the class definition. Serialization writes its own in-memory Datastore of the instance object to a specified file location.

The StudentList class data model file is available in the GitHub repo here [].

/// <summary>

/// StudentList

/// Serializes and Deserializes data to/from an

/// xml file

/// </summary>

[XmlRoot(ElementName = "students")]

public class StudentList : List<Student>

{

/// <summary>

/// Load

/// Loads data fom an xml file into memory

/// </summary>

/// <param name="filename">(string) name of file to load xml</param>

/// <returns></returns>

public static StudentList Load(string filename)

{

StudentList list = new StudentList();

XmlSerializer ser = new XmlSerializer(typeof(Models.StudentList));

using (StreamReader reader = new StreamReader(filename))

{

if (reader != null)

{

list = ser.Deserialize(reader) as Models.StudentList;

}

}

return list;

} // end of Load method

/// <summary>

/// Save

/// Saves and writes the data to XML file

/// </summary>

/// <param name="filename">(string) name of file to write data</param>

public void Save(string filename)

{

XmlSerializer ser = new XmlSerializer(typeof(Models.StudentList));

using (StreamWriter writer = new StreamWriter(filename))

{

ser.Serialize(writer, this);

}

} // end of Save method

} // end of class

### Web API Controller with CRUD Methods/Actions

The *StudentsController* class file is available in the GitHub repo here []. It is specifically an ApiController and very specific to the Web API engine and implements the API methods or actions that a client requested.

#### FilePath

The FilePath property stores the file name and location of the Datastore xml file where data is to be serialized/deserialized. For this demo project, it is stored in the *App\_Data* directory of the web application project. The Controller calls the *StudentList* static class method to deserialize into an instance object and also calls upon an instance object method to serialize to file (*see the StudentList data model above*).

/// <summary>

/// Gets the full path to the XML document containing the student data

/// </summary>

private string FilePath

{

get

{

return HttpContext.Current.Server.MapPath("~/App\_Data/student.xml");

}

} // end of method

#### Post “Create”

The Post() method in the controller creates a record. It simply takes the HTTP Request’s content body and attempts to add the Student object to the XML Datastore. It will first try and see if there is already an existing record of the Student object and throw an exception to prevent adding duplicates. Otherwise, it will first load the XML Datastore to temporary memory “StudentList”, add the HTTP Request’s Student object, then call the Save() method on the StudentList object to serialize and save to the XML file Datastore.

/// <summary>

/// Post

/// The Post method will represent Add. This is a bit

/// more complicated than Get as we have to check for duplicate

/// Students and then save any changes. If there is a duplicate

/// we must throw an HttpException.

/// </summary>

/// <param name="value">(Student) object to create</param>

public void Post([FromBody]Student value)

{

var data = StudentList.Load(FilePath);

Student existing = data.Where(s => s.ID == value.ID).FirstOrDefault();

if (existing == null)

{

data.Add(value);

data.Save(FilePath);

}

else

{

throw new HttpResponseException(HttpStatusCode.Conflict);

}

} // end of method

#### Get “Read”

The Web API supports two types of Get (Read) requests: Get by single entity “Student id” and Get by multiple entities “List<Student> by paging”.

##### Get Student by Id

This method returns a Student object that matches the requested id property in the Http Request. First the entire Datastore is loaded into memory and then LINQ is used to query the list of Student objects to find one that matches the specific id. There should be no duplicate student id’s but it ensures the selection of a single object by use of the extension method FirstOrDefault().

/// <summary>

/// Get by id

/// Get method that returns a specific Student.

/// This method will also use Linq to filter the

/// results, but the IQueryable interface is not required

/// </summary>

/// <param name="id">(int) id of the student to retrieve</param>

/// <returns>(Student) object</returns>

public Student Get(int id)

{

var data = StudentList.Load(FilePath);

return data.Where(s => s.ID == id).FirstOrDefault();

} // end of method

##### Get List<Student> by Paging

This method returns a List of Student objects but maybe not all of them. If the page number and count parameters are not specified, it will return only the first 50 Student object records. The IQueryable interface provides paging support after the XML Datastore is loaded into temporary memory as a StudentList object (AsQuerable()). LINQ is used to query on the StudentList (List<Student>) object the requested page and number of records.

/// <summary>

/// Get using paging

/// Collection retrieval version of Get. Because our

/// dataset can be quite large we will need to limit

/// the results.This can be done with optional URL

/// parameters and the IQueryable interface

/// to provide paging support

/// </summary>

/// <param name="page">(int) page to retrieve</param>

/// <param name="count">(int) number of items to retrieve</param>

/// <returns>List of type student enuemerable list of students</returns>

public List<Student> Get(int page = 0, int count = 50)

{

var data = StudentList.Load(FilePath).AsQueryable();

return data.Skip(page \* count).Take(count).ToList();

} // end of method

#### Put “Update”

The Put() method updates a record by an identifier, in this case it is the Student id. It will attempt to replace the existing Student object that has an id specified in the HTTP Request, with the entire new Student specified in the Request content body. Instead of throwing an exception if the record is non-existing, it just treats the Request as Adding a new Student record to the Datastore. Otherwise, it removes all Student objects with a specific id (there can be more than one record returned in the LINQ query, but should not be), and just Adds the new Student object. Basically, to replace a Student, it first deletes the old record, then adds the new one. Finally, it calls the temporary Datastore memory object of type StudentList to *Save* and *Serialize* its data to the XML file.

/// <summary>

/// Put

/// Put represents the Update method. Just like the

/// Post action, the data is contained in the body

/// of the request. Unlike Post, however, if the

/// item exists then it is replaced. If the item

/// doesn’t exist then it is added.

/// </summary>

/// <param name="id">(int) id of the student to update</param>

/// <param name="value">(Student) value to update</param>

public void Put(int id, [FromBody]Student value)

{

var data = StudentList.Load(FilePath);

Student existing = data.Where(s => s.ID == id).FirstOrDefault();

if (existing == null)

{

data.Add(value);

}

else

{

data.RemoveAll(s => s.ID == id);

data.Add(value);

}

data.Save(FilePath);

} // end of Put method

#### Delete “Delete”

The Delete method simply loads the Datastore into temporary memory StudentList and queries the list to find all the Student objects that match the HTTP Request Student id. If there are no matching records to delete, it does not pass an exception or do anything other than return. Otherwise, the method removes all occurrences of Student objects in the List that match the predicate of having the Student id record match the HTTP Request parameter.

/// <summary>

/// Delete

/// Delete will remove a Student record that has the matching

/// ID but it will not fail if the ID is non-existent

/// </summary>

/// <param name="id">(integer) id of the student</param>

public void Delete(int id)

{

var data = StudentList.Load(FilePath);

Student existing = data.Where(s => s.ID == id).FirstOrDefault();

if (existing != null)

{

data.RemoveAll(s => s.ID == id);

}

data.Save(FilePath);

} // end of method

## Web API Client “Tester to Service” Console Application

A simple console application was created to test the Student ASP.NET Web API service capabilities. The Student ASP.NET Web API publishes a help page in the project build settings that make public the available API methods and instructions on how to consume it (see Demo section). The purpose of the client was to test consumption of all the available methods in the Web API. The client console “tester” application project code is available on GitHub here [].

### Client Local Data Models

The client “tester” application has local data models, very similar to the server side, to know how a Student class data model is structured. In addition, several other “helper” classes are discussed.

#### Student

The Student class data model is available on the client side so it knows how to consume the Student objects it sends/receives to the service. This class also has overridden its ToString() to return all the object properties in one formatted string. Note: XML serialization/deserialization was not done on the client site so the attributes are removed. The code for the Student class is available in the client project on GitHub here [].

public class Student

{

// Formatted for brevity...

public enum GradeEnum

{

PreSchool = -1, Kindergarten = 0, First = 1, Second,

Third, Fourth, Fifth, Sixth,

Seventh, Eighth, Freshmen, Sophomore,

Junior, Senior, College

}

public int ID { get; set; }

public string LastName { get; set; }

public string FirstName { get; set; }

public DateTime DOB { get; set; }

public float GPA { get; set; }

public GradeEnum Grade { get; set; }

public override string ToString()

{

return string.Format(

"{0:000-00-0000} {1,-20} {2,-15} {3:yyyy-MM-dd} {4,-12} {5:0.000}",

ID, LastName, FirstName, DOB, Grade, GPA);

}

} // end of class

#### Enums

The Enums data model class just lists the different types of serialization modes this project supports (xml and json) and helps to encapsulate and simplify the Web API Request/Response processing as explained under the Helper Classes\* section. The code for the Enums class is available in the client project on GitHub here [].

// For a list of common http application content types:

// http://en.wikipedia.org/wiki/Internet\_media\_type#List\_of\_common\_media\_types

public enum SerializationModesEnum

{

Undefined = 0,

Xml = 1,

Json = 2

}

#### HttpResults

The HttpResults data model class helps to encapsulate and simplify the Web API Response processing as explained under the Helper Classes\* section. The code for the HttpResults class is available in the client project on GitHub here [].

public class HttpResults<OutputType>

{

public HttpStatusCode StatusCode { get; set; }

public string Error { get; set; }

public OutputType Result { get; set; }

public string RawData { get; set; }

public HttpResults(HttpStatusCode statusCode, OutputType result)

{

StatusCode = statusCode;

Result = result;

}

public HttpResults(HttpStatusCode statusCode, string error)

{

StatusCode = statusCode;

Error = error;

}

} // end of class

### Helper Classes\*

#### ClientHelper

The ClientHelper class is a generic framework that simplifies the HTTP Request/Response processing and operations required by the client “tester” application to formally build request objects and process the responses from a service. The code for the ClientHelper class is available in the client project on GitHub here [].

A summary of the capabilities and methods are shown in the list below:

* Gets an HttpClient object to connect to the given service URL
* Gets the media type for the given serialization mode
* Modifies the headers to support the given serialization mode
* Determines the contents of the http response message
* Parses an Http response message
* Executes a GET on the given Url
* Executes an HTTP method on the given Url
* Executes a POST on the given Url
* Executes a PUT on the given Url
* Executes a DELETE on the given Url
* Executes a DELETE on the given Url without header inputs or outputs

*Note: I did not develop this ClientHelper class nor the supporting data model classes (Enums and HttpResults) for this class lab project. This class is very lengthy and complicated. We would not had been capable to code this in a week’s deadline for the class assignment. Instead, we were expected to learn about and how to utilize this framework and integrate into our client application to test the Web API service.*

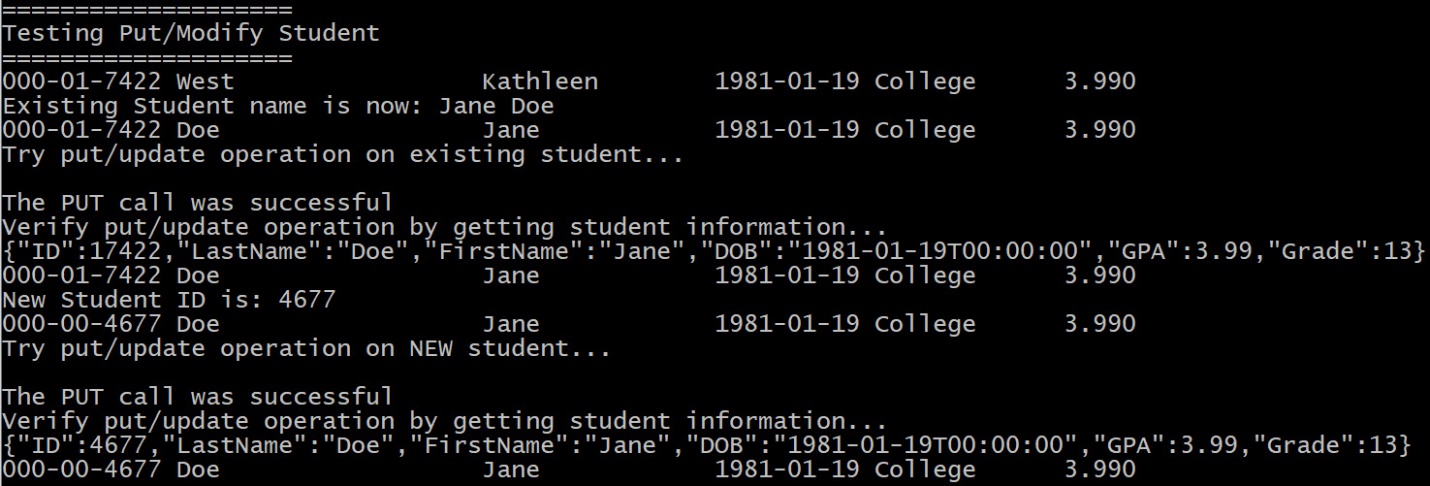
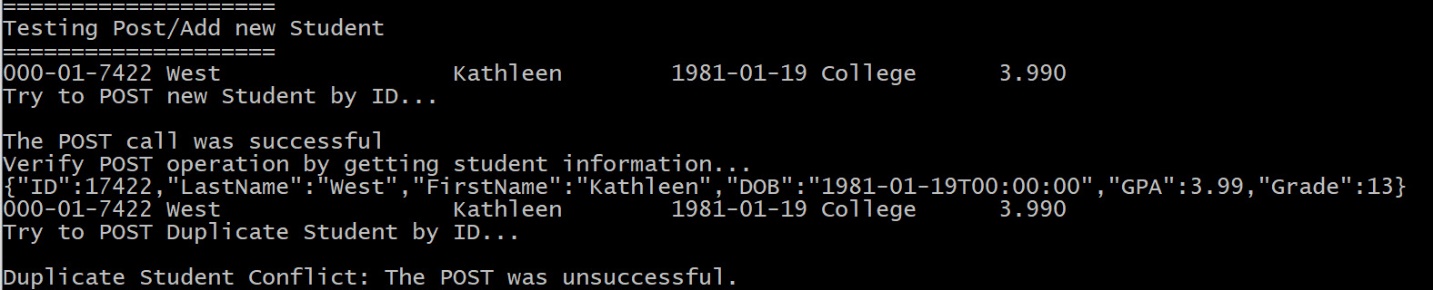
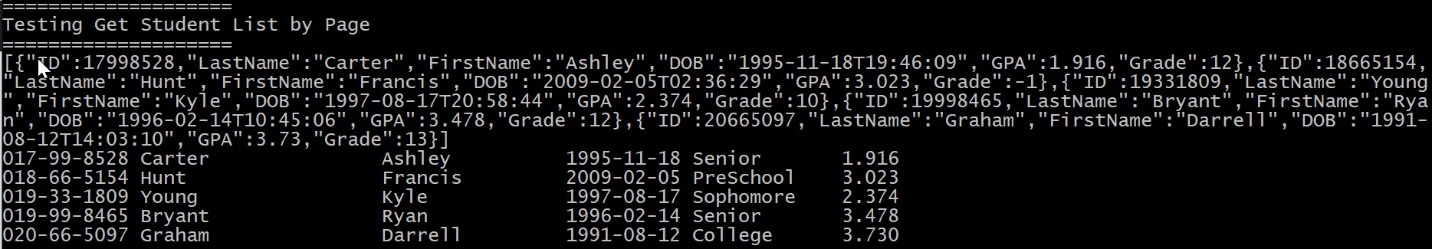
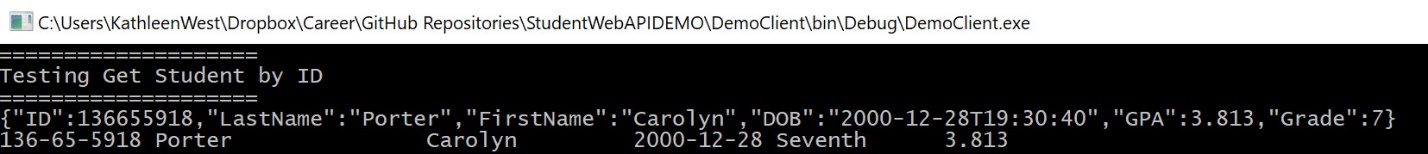
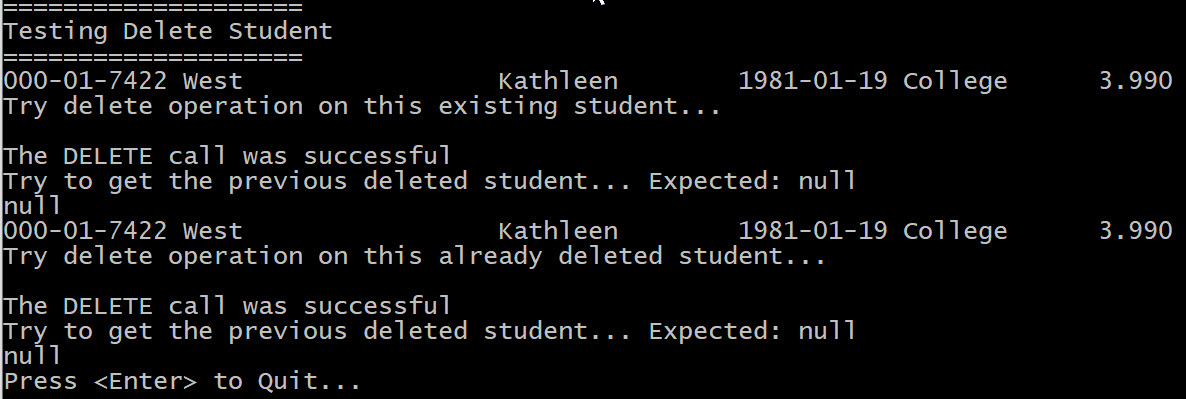
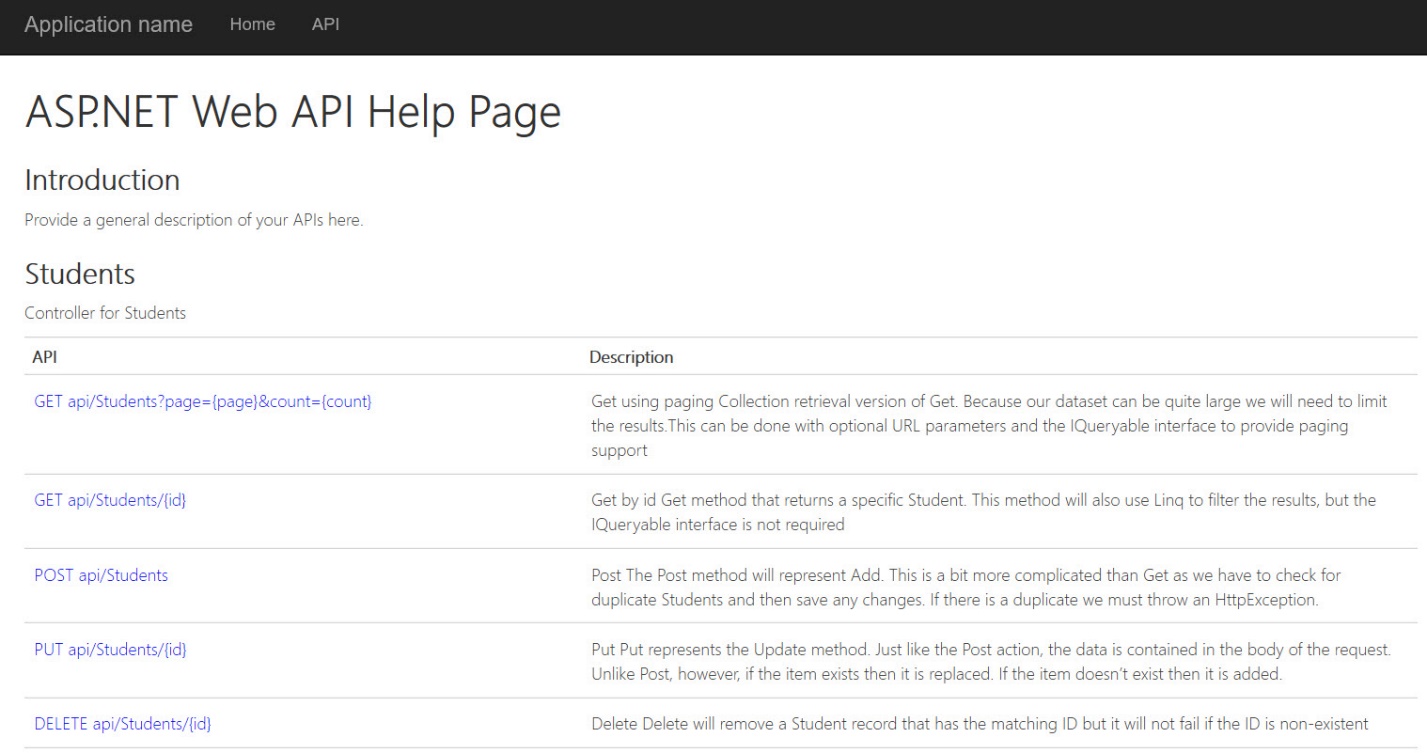
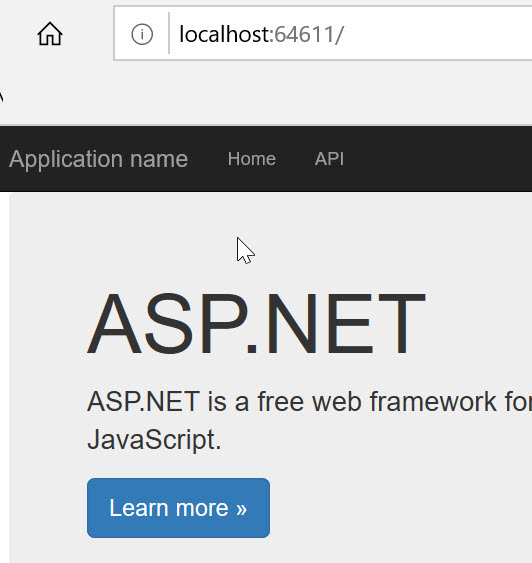
### CRUD Operations Testing

The main Program class in the Client Console Application “Tester” executes CRUD (create, read, update, and delete) operations on the Student Web API Service. It utilizes the ClientHelper class (see above) to execute commands with appropriate parameters (which may include Student model data) on the service. The code for the Program class is available in the client project on GitHub here []. Note: It specifies in the code the Student Web API Service address as a constant for use in testing the API. The Demo section (next) shows screen captures of the client console application testing the service and the results printed to the console.

// Base Address for the API and controller

const string BASE\_ADDR = "http://localhost:64611/api/students/";

## Demo



## Code

The entire project code repository is available on GitHub here.