

How-To Create CRUD in HTML, JavaScript, and jQuery Using the Web API

In my last article, I showed you how to manipulate data in an HTML table using only JavaScript and jQuery. There were no post-backs, so the data didn't go anywhere. In this article, you'll use the same HTML and jQuery, but add calls to a Web API to retrieve and modify product data. It isn't necessary to go back and read the previous article; this article presents all of the HTML and the calls to work client-side and add the server-side code as well. I'll be using Visual Studio and .NET to build the Web API service, but the client-side coding is generic and can call a Web API built in any language or platform.

This article focuses on the four standard HTTP verbs that you use to work with the Web API: `GET`, `POST`, `PUT`, and `DELETE`. The `GET` verb retrieves a list of data, or a single item of data. `POST` sends new data to the server. The `PUT` verb updates an existing row of data. `DELETE` sends a request to remove a row of data. These verbs are used to map to a method you write in your Web API controller class. It's up to you to perform the retrieval of data, adding new rows, and updating and deleting of rows of data. Let's see how all of this works by building a project step-by-step.

Create a Product Information Page

If you're using Visual Studio, create a new ASP.NET Web Application project. Select "Empty" for the project template as you don't want any MVC, Web Forms, or even the Web API at this point. Add a new HTML page and name it `Default.html`. Open the Manage NuGet Packages dialog to add Bootstrap to your project. Bootstrap isn't necessary for the demo, but it does make your page look nicer.






Open up `Default.html` and drag the `bootstrap.min.css` file, the [jQuery-1.9.1.min.js](#) file, and the [bootstrap.min.js](#) files into the `<head>` area of the page, as shown in the following code snippet.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml";>
<head>
```

```
<title></title>
<link href="Content/bootstrap.min.css" rel="stylesheet" />
<script src="Scripts/jquery-1.9.1.min.js"></script>
<script src="Scripts/bootstrap.min.js"></script>
</head>
<body>
</body>
</html>
```

In the `<body>` tag of this page, build the Web page that looks like **Figure 1**. Add a Bootstrap container, and a row and column within the `<body>` element.

Paul's Training Company

Edit	Product Name	Introduction Date	URL	Delete
	Extending Bootstrap with CSS, JavaScript and jQuery	2015-06-11T00:00:00	http://bit.ly/1SNzc0i	
	Build your own Bootstrap Business Application Template in MVC	2015-01-29T00:00:00	http://bit.ly/1l8ZqZg	
	Building Mobile Web Sites Using Web Forms, Bootstrap, and HTML5	2014-08-28T00:00:00	http://bit.ly/1J2dcrl	

Add Product

Product Information

Product Name

Introduction Date

URL

Add

Figure 1: Use a product information page to list, add, edit, and delete data.

Add an `<h2>` element with the words Paul's Training Company (or substitute your name).

```
<body>
  <div class="container">
    <div class="row">
      <div class="col-sm-6">
        <h2>Paul's Training Company</h2>
      </div>
    </div>
  </div>
</body>
```

Immediately below the row you just added, create another row, and within that row, build the skeleton of an HTML table. Just build the header for the table, as you'll build the body dynamically in JavaScript using data retrieved from your Web API. To learn more about building a table dynamically in JavaScript, please read my last article entitled "[CRUD in HTML, JavaScript, and jQuery](#)".

```
<div class="row">
  <div class="col-sm-6">
    <table id="productTable"
      class="table table-bordered
        table-condensed table-striped">
      <thead>
        <tr>
          <th>Product Name</th>
          <th>Introduction Date</th>
          <th>URL</th>
        </tr>
      </thead>
    </table>
  </div>
</div>
```

In **Figure 1**, you can see an "Add Product" button immediately below the table. This button is used to clear the input fields of any previous data so that the user can add new product data. After entering data into the input fields, the user clicks on the "Add" button to send the data to the Web API. Build this "Add Product" button by adding another Bootstrap row and column below the previous row. In the `onClick` event for this button, call a function named `addClick`. You haven't created this function yet, but you will later in this article.

```
<div class="row">
  <div class="col-sm-6">
```

```

        <button type="button"
            id="addButton"
            class="btn btn-primary"
            onclick="addClick();" >Add Product</button>
    </div>
</div>

```

To create the product input area you see in **Figure 1**, the individual fields are placed inside a Bootstrap panel class. The panel classes are ideal to make the input area stand out on the screen separate from all other buttons and tables. A Bootstrap panel consists of the panel wrapper, a heading, a body, and a footer area made out of `<div>` tags.

```

<div class="row">
  <div class="col-sm-6">
    <div class="panel panel-primary">
      <div class="panel-heading">
        Product Information
      </div>
      <div class="panel-body">
      </div>
      <div class="panel-footer">
      </div>
    </div>
  </div>
</div>

```

All label and input fields are placed within the “panel-body” `<div>` tag. To achieve the input form look, use the Bootstrap `<div class="form-group">` around the label and input fields. Use the “for” attribute on the `<label>` and include `class="form-control"` on each of your input types to achieve the correct styling.

```

<div class="form-group">
  <label for="productname">Product Name</label>
  <input type="text" id="productname" class="form-control" />
</div>
<div class="form-group">
  <label for="introdate">Introduction Date</label>
  <input type="date" id="introdate" class="form-control" />
</div>
<div class="form-group">
  <label for="url">URL</label>
  <input type="url" id="url" class="form-control" />
</div>

```

The final piece of the input form is the **Add** button that you place into the panel-footer `<div>` tag area. This input button's text changes based on whether or not you're

doing an add or edit of the data in the input fields. The JavaScript code you write checks the text value of this button to determine whether to POST the data to the Web API or to PUT the data. A POST is used for adding data and the PUT is used to update data. Again, there's a function name in the `onClick` event that you haven't written yet, but you will.

```
<div class="row">
  <div class="col-xs-12">
    <button type="button"
      id="updateButton"
      class="btn btn-primary"
      onclick="updateClick();">
      Add
    </button>
  </div>
</div>
```

If you run the `Default.html` page now, it should look similar to **Figure 1**, except there will be no listing of product data.

Add the Web API to Your Project

Now that you have your HTML page built, it's time to add on the appropriate components within Visual Studio so you can build a Web API to retrieve and modify product data. In Visual Studio, this is easily accomplished using the Manage NuGet Packages dialog, as shown in **Figure 2**. Search for the "Microsoft ASP.NET Web API 2.2" package and click the Install button to add the appropriate components to your Web project.

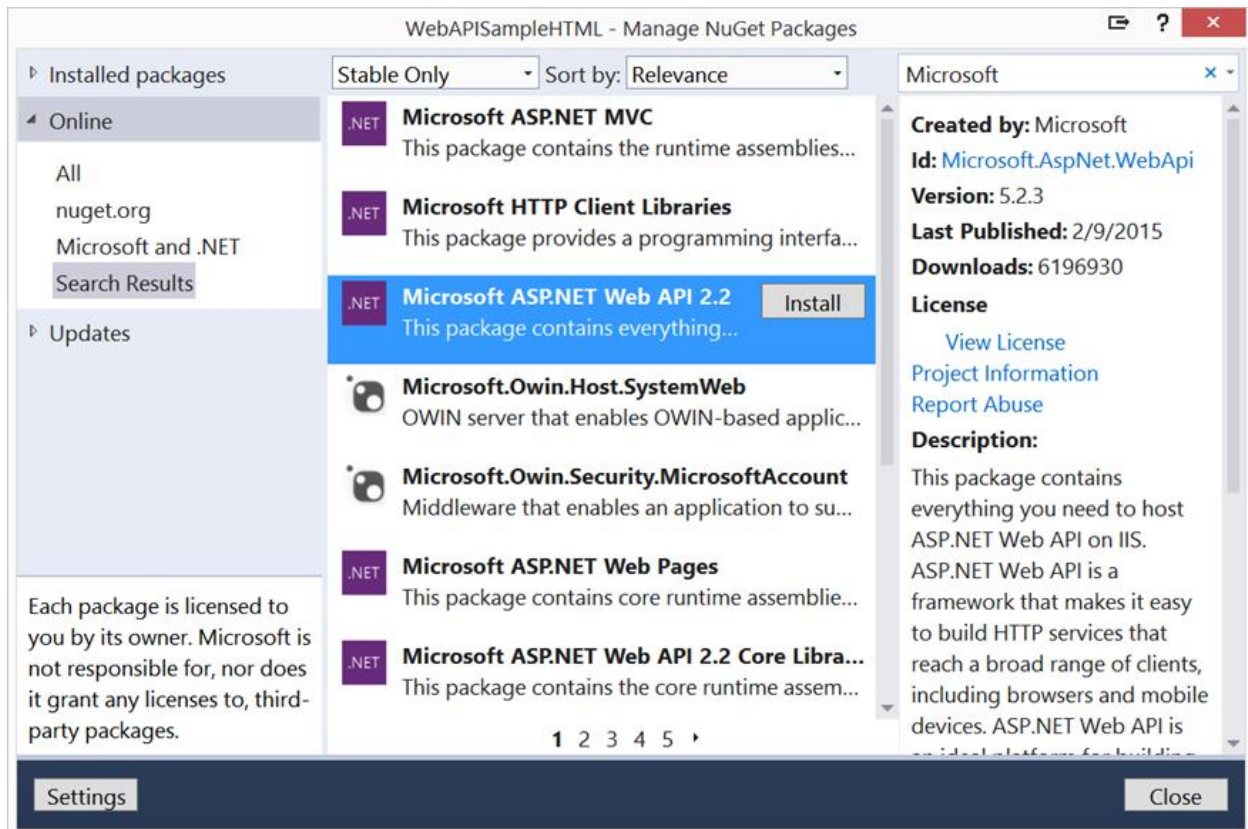


Figure 2: Use the Manage NuGet Packages screen to add the Web API to your project.

To create an endpoint for your Web API, create a controller class. Create a new folder called **\Controllers** in your project. Right-click on the **Controllers** folder and select **Add | Web API Controller Class (v2.1)**. Set the name of your new controller to `ProductController`.

Next, you need to specify the routing for your Web API. Create a new folder called **\App_Start** in your project. Add a new class and name it `WebApiConfig`. Add a `Using` statement at the top of this file to bring in the namespace `System.Web.Http`. Add a `Register` method in your `WebApiConfig` class.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.Clear();
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional });
    }
}
```

Clear the Routes collection in case there are any default routes created by .NET. The route, `api/{controller}/{id}`, that is specified in the `MapHttpRoute` method is very standard for Web API projects. However, feel free to change this to whatever you want.

For example, you could just use `{controller}/{id}` if you don't want to have to specify **api** in front of all your calls. This is just a matter of preference. However, as most developers have an existing Web project that they're adding API calls to, it makes sense to have a different route for your API calls. This keeps them separate from the route of your Web pages, which is typically "views."

The last thing you need to do before you can start calling your Web API is to register this route so that your Web application running on your Web server will recognize any call made to your API. Right-click on your project and select **Add | New Item | Global Application Class**. This adds a `Global.asax` file to your project. At the top of the global application class, add the following using statement.

```
using System.Web.Http;
```

Within the `Application_Start` event, add a call to the `Register` method of the `WebApiConfig` class that you created earlier. ASP.NET creates an instance of an `HttpConfiguration` class and attaches it as a static property of the `GlobalConfiguration` class. It is to this configuration object that you'll add to the routes you wish to support in your Web application.

```
protected void Application_Start(object sender, EventArgs e)
{
    WebApiConfig.Register(GlobalConfiguration.Configuration);
}
```

Create Controller and Mock Data

You now have a Web page and all the plumbing for the Web API ready to go. Let's start building your first API call to return a list of product objects to display in the table on the HTML page. Create a class called `Product` in your project and add the following properties.

```
public class Product
{
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public DateTime IntroductionDate { get; set; }
}
```



```
    public string Url { get; set; }  
}
```

Instead of worrying about any database stuff, you can just create some mock data to learn how to work with the Web API. Open the `ProductController` class and add a private method to create mock data, as shown in **Listing 1**.

Listing 1: Create some mock data for your Web API

```
private List<Product> CreateMockData()  
{  
    List<Product> ret = new List<Product>();  
    ret.Add(new Product()  
    {  
        ProductId = 1,  
        ProductName = "Extending Bootstrap with CSS, JavaScript and  
jQuery",  
        IntroductionDate = Convert.ToDateTime("6/11/2015"),  
        Url = "http://bit.ly/1SNzc0i"  
    });  
  
    ret.Add(new Product()  
    {  
        ProductId = 2,  
        ProductName = "Build your own Bootstrap Business Application  
Template in MVC",  
        IntroductionDate = Convert.ToDateTime("1/29/2015"),  
        Url = "http://bit.ly/1I8ZqZg"  
    });  
  
    ret.Add(new Product()  
    {  
        ProductId = 3,  
        ProductName = "Building a Web API with ASP.NET MVC 5",  
        IntroductionDate = Convert.ToDateTime("1/29/2015"),  
        Url = "http://bit.ly/1I8ZqZg"  
    });  
}
```

```

    {
        ProductId = 3,
        ProductName = "Building Mobile Web Sites Using Web Forms,
        Bootstrap, and HTML5",
        IntroductionDate = Convert.ToDateTime("8/28/2014"),
        Url = "http://bit.ly/1J2dcrj"
    });
}

return ret;
}

```

Return Values from API Calls

If you look at the `ProductController`, you'll see methods that look like **Listing 2**. The problem with these methods is that each one returns a different type of value or no value at all. This means that if you want to return HTTP status codes like a `200`, `201`, `404`, etc. you have to write extra code. If you want to return error messages back to the client, you have to change the return value on each of these.

Listing 2: The default methods in the controller need to be modified.

```

// GET api/<controller>
public IEnumerable<string> Get ()
{
    return new string[] { "value1", "value2" };
}

```

```

// GET api/<controller>/5
public string Get(int id)
{
    return "value";
}

```

```
// POST api/<controller>
```

```
public void Post ([FromBody] string value)
```

```
{
```

```
}
```

```
// PUT api/<controller>/5
```

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

```
{
```

```
}
```

```
// DELETE api/<controller>/5
```

```
public void Delete(int id)
```

```
{
```

```
}
```

Introduced in the Web API 2 is a new interface called `IHttpActionResult`. This interface is built into the `ApiController` class (from which your `ProductController` class inherits) and defines Helper methods to return the most common HTTP status codes such as a `202`, `201`, `400`, `404`, etc. The methods you'll use in this article are `Ok`, `Created<T>`, and `NotFound`. These methods return `200`, `201`, and `404` respectively.

The `Ok` and `Created` methods allow you to pass data back so you can include things like collections of products or a new product object.

Get (GET) All Products

Let's start by modifying the `GET` method to return all products created in the mock data collection. Locate the GET method that has no parameters in the `ProductController` class, and modify the code to look like the following.

```
[HttpGet ()]  
public IHttpActionResult Get ()
```

```

{
    IHttpActionResult ret = null;
    List<Product> list = new List<Product> ();
    list = CreateMockData();
    ret = Ok(list);
    return ret;
}

```

Modify the return value of the `GET` method to use the new `IHttpActionResult` interface. Although it's not necessary, I like adding the attribute `[HttpGet()]` in front of the method to be very explicit about which HTTP verb this method supports. Declare a variable named `ret`, of the type `IHttpActionResult`. Declare a variable named `list`, to hold a collection of product objects. Build the list of data by calling the `CreateMockData` method that you defined earlier. Set the `ret` variable to the `Ok` method built into the `ApiController` class, passing in the list of product objects. The `Ok` method does a couple of things; it sets the HTTP status code to `200`, and it includes the list of products in the `HttpResponseMessage` sent back from this API call.

Call the GET Method

With the `GET` method created to return a list of products, you can now call it from your HTML page. Open the `Default.html` page and add a `<script>` tag at the bottom of the page just above the `</body>` tag. You know that you have to create at least two functions right away because they were the ones you called from the buttons you defined in the HTML. Add these two function stubs now.

```

<script>
    // Handle click event on Update button
    function updateClick() {
    }
    // Handle click event on Add button
    function addClick() {
    }
</script>

```

Add a new function called `productList` to make the Ajax call to the `GET` method that you created.

```

function productList() {
    // Call Web API to get a list of Product
    $.ajax({
        url: '/api/Product/',
        type: 'GET',
    });
}

```

```

dataType: 'json',
success: function (products) {
    productListSuccess (products);
},
error: function (request, message, error) {
    handleException(request, message, error);
}
});
}

```

this Ajax call, there are two additional functions that you need to write.

The `productListSuccess` function processes the collection of products returned when you successfully retrieve the data. The `handleException` function takes the error information and does something with it. The `productListSuccess` function is very simple and uses the jQuery `$.each()` iterator to loop over the collection of product objects.

```

function productListSuccess (products) {
    // Iterate over the collection of data
    $.each(products, function (index, product) {
        // Add a row to the Product table
        productAddRow (product);
    });
}

```

The `productAddRow` function called from within the iterator is responsible for building a new row to add to the HTML table.

```

function productAddRow (product) {
    // Check if <tbody> tag exists, add one if not
    if ($("#productTable tbody").length == 0) {
        $("#productTable").append("<tbody></tbody>");
    }
    // Append row to <table>
    $("#productTable tbody").append(
        productBuildTableRow (product));
}

```

Notice that you first check to ensure that the `<tbody>` tag exists on the table. This ensures that any `<tr>` elements you add go into the correct location in the DOM for the table. The function to build the actual `<tr>` is in a function called `productBuildTableRow`. This is in a separate function because you'll use this later in this article to build a row for editing a row in a table.

```
function productBuildTableRow(product) {
    var ret =
        "<tr>" +
        "<td>" + product.ProductName + "</td>" +
        "<td>" + product.IntroductionDate + "</td>"
        + "<td>" + product.Url + "</td>" +
        "</tr>";
    return ret;
}
```

The last function to add is `handleException`. If an error occurs, display the error message information in an alert dialog. You can figure out how you want to display error messages later, but for now, you only want to see the error details.

```
function handleException(request, message, error) {
    var msg = "";
    msg += "Code: " + request.status + "\n";
    msg += "Text: " + request.statusText + "\n";
    if (request.responseJSON != null) {
        msg += "Message" + request.responseJSON.Message + "\n";
    }
    alert(msg);
}
```

Call the `productList` function after the Web page loads using the jQuery `$(document).ready()` function. Add the following code within your `<script>` tag.

```
$(document).ready(function () {
    productList();
});
```

Run the HTML page, and if you've done everything correctly, you should see your mock product data displayed in the HTML table. What's really neat about this is that you didn't have to use MVC, Web Forms, PHP, or any other Web development system. You simply use an HTML page to call a Web API service.

Modify the GET Method

The `GET` method you wrote earlier assumes that you successfully retrieved a collection of data from your data store. However, when you retrieve data from a database table, you may have an empty table. In this case, you need to respond back to the front-end

client that no data was found. In this case, the list variable would be an empty list. If no data is returned, you should send back a 404 status using the `NotFound` method. Modify the `GET` method to look like the following.

```
[HttpGet()]
public IActionResult Get()
{
    IActionResult ret = null;
    List<Product> list = new List<Product>();
    list = CreateMockData();
    if (list.Count > 0)
    {
        ret = Ok(list);
    }
    else
    {
        ret = NotFound();
    }
    return ret;
}
```

Get a Single Product

When you wish to edit a product, you call the Web API to retrieve a single product object to ensure that you have the latest data from the database, and then display that data in input fields to the user. The user then modifies that data and posts it back. You'll learn how update data later in this article, but for now, let's see how to get a single product into the HTML input fields.

Open the `ProductController` and locate the second `Get` method, the one that accepts a single parameter named `id`. Modify that function to look like **Listing 3**. Because you're using mock data, go ahead and build the complete collection of products. Locate the product id using LINQ to search for the ID passed into the method. If the product is found, return an `Ok` and pass the product object to the `Ok` method. If the product is not found, return a `NotFound`.

Listing 3: Get a single product

```
[HttpGet()]
public IActionResult Get(int id)
{
    IActionResult ret;
```

```
List<Product> list = new List<Product>();
```

```
Product prod = new Product();
```

```
list = CreateMockData();
```

```
prod = list.Find(p => p.ProductId == id);
```

```
if (prod == null)
```

```
{
```

```
    ret = NotFound();
```

```
}
```

```
else
```

```
{
```

```
    ret = Ok(prod);
```

```
}
```

```
return ret;
```

```
}
```

Add an Edit Button to Each Row of the Table

Each row of data in the HTML table should have an edit button, as shown in **Figure 1**. The raw HTML of the button looks like the following, but of course you have to build this code dynamically so you can get the `data-` attribute assigned to the correct product id in each row.

```
<button class="btn btn-default"
        onclick="productGet(this);"
        type="button"
        data-id="1">
    <span class="glyphicon glyphicon-edit"></span>
</button>
```

To add this button into each row, add a new `<th>` element in the `<thead>` of the table.

```
<th>Edit</th>
```


Modify the `productAddRow` function and insert the code below before the other `<td>` elements.

```
"<td>" +
  "<button type='button' " +
    "onclick='productGet(this);' " +
    "class='btn btn-default' " +
    "data-id='" + product.ProductId + "'" +
    "<span class='glyphicon glyphicon-edit' />"
  + "</button>" +
"</td>" +
```

Within the `<td>`, build a button control. Add an `onClick` to call a function named `productGet`. Pass in `this` to the `productGet` function so that you have a reference to the button itself. You're going to need the reference so you can retrieve the value of the product ID you stored into `data-id` attribute.

To simplify the code for this article, I concatenated the HTML values together with the data. You could also use a template library, such as Underscore or Handlebars, to separate the HTML markup from the data.

Advertisement

Make an Ajax Call to Get a Single Product

In **Listing 4**, you can see the `productGet` function that you need to add to your `<script>` on your HTML page. In this function, you retrieve the product ID from the `data-id` attribute you stored in the edit button. This value needs to be passed to the `Get` method in your controller and it needs to be kept around for when you submit the data back to update. The best way to do this is to create a hidden field in your HTML body.

```
<input type="hidden" id="productid" value="0" />
```

Listing 4: Get the product ID and use Ajax to get a single product object

```
function productGet(ct1) {
    // Get product id from data- attribute
    var id = $(ct1).data("id");
```

```

// Store product id in hidden field
$("#productid").val(id);

// Call Web API to get a list of Products
$.ajax({
    url: "/api/Product/" + id,
    type: 'GET',
    dataType: 'json',
    success: function (product) {
        productToFields(product);
    },
    error: function (request, message, error) {
        handleException(request, message, error);
    }
});
}

```

You're now ready to call the `Get(id)` method in your `ProductController`. Add the function `productGet` within your `<script>` tags, as shown in **Listing 4**. The Ajax call is very similar to the previous call you made, but the product ID is included on the URL line. This extra ID is what maps this call to the `Get(id)` method in your controller. If the Get method succeeds in returning a product object, call a function named `productAddToFields` and pass in the product object. Change the update button's text from "Add" to "Update." This text value will be used later when you're ready to add or update data.

The `productToFields` function uses jQuery to set the value of each input field with the appropriate property of the product object retrieved from the API call.

```
function productToFields (product) {
    $("#productName").val(product.ProductName);
    $("#introddate").val(product.IntroductionDate);
    $("#url").val(product.Url);
}
```

Run this sample and ensure that you can retrieve a specific product from your API.

Add (POST) a New Product

You used the `GET` verb to retrieve product data, so let's now learn to use the `POST` verb to add a new product. In the `ProductController` class, to call the `Post` method. Modify the `Post` method by adding an `[HttpPost()]` attribute and changing the return value from `void` to `IHttpActionResult`. Change the parameter to the `Post` method to accept a `Product` object. You don't need the `[FromBody]` attribute, so go ahead and delete that. The `[FromBody]` attribute is only needed for simple data types, such as string and int.

```
[HttpPost()]
public IHttpActionResult Post(Product product)
{
    IHttpActionResult ret = null;
    if (Add(product))
    {
        ret = Created<Product>(Request.RequestUri +
            product.ProductId.ToString(), product);
    }
    else
    {
        ret = NotFound();
    }
    return ret;
}
```

The `Add` method is a mock method to simulate adding a new product. This method calculates the next product ID for the new product and returns a `Product` object back to the Web page with the new ID set in the `ProductId` property.

```
private bool Add(Product product)
{
    int newId = 0;
    List<Product> list = new List<Product>();

    list = CreateMockData();
}
```

```

newId = list.Max(p => p.ProductId);
newId++;
product.ProductId = newId;
list.Add(product);

// TODO: Change to false to test NotFound()
return true;
}

```

Add a New Product in HTML

With the `Post` Web API method written, you can write the necessary JavaScript in `Default.html` to call this method. Define a new JavaScript object called `Product` with the following name/value pairs.

```

var Product = {
  ProductId: 0,
  ProductName: "",
  IntroductionDate: "",
  Url: ""
}

```

Each of the names in this JavaScript object needs to be spelled exactly the same as the properties in the `Product` class you created in C#. This allows the Web API engine to map the values from the JavaScript object into the corresponding properties in the C# object.

Each of the names in this JavaScript object needs to be spelled exactly the same as the properties in the `Product` class you created in C#.

The user fills in the blank fields on the screen, then click the Add button to call the `updateClick` function. Modify this function to create a new `Product` object and retrieve the values from each input field on the page and set the appropriate values in the `Product` object. Call a function named `productAdd` to submit this JavaScript object to the `Post` method in your API.

```

function updateClick() {
  // Build product object from inputs
  Product = new Object();
  Product.ProductName = $("#productname").val();
  Product.IntroductionDate = $("#introddate").val();
  Product.Url = $("#url").val();
  if ($("#updateButton").text().trim() == "Add") {
    productAdd(Product);
  }
}

```

```
}
```

The `productAdd` function uses Ajax to call the Post method in the Web API. There are a couple of changes to this Ajax call compared to the `GET` calls you made earlier. First, the `type` property is set to `POST`. Second, you add the `contentType` property to specify that you're passing JSON to the API. The last change adds a `data` property where you take the JavaScript `Product` object you created and serialize it using the `JSON.stringify` method.

```
function productAdd(product) {
  $.ajax({
    url: "/api/Product",
    type: 'POST',
    contentType:
      "application/json;charset=utf-8",
    data: JSON.stringify(product),
    success: function (product) {
      productAddSuccess(product);
    },
    error: function (request, message, error) {
      handleException(request, message, error);
    }
  });
}
```

Add a New Product to the HTML Table

If the call to the `Post` method is successful, a new `product` object is returned from the Web API. In the success part of the Ajax call, pass this object to a function called `productAddSuccess`. Add the newly created product data to the HTML table by calling the `productAddRow` function you created earlier. Finally, clear the input fields so that they're ready to add a new product.

```
function productAddSuccess (product) {
  productAddRow(product);
  formClear();
}
```

The `formClear` function uses jQuery to clear each input field. You should also change the `addClick` function to clear the fields when the user clicks on the Add button.

```
function formClear () {
  $("#productname").val("");
  $("#introdote").val("");
  $("#url").val("");
}
```

```
}  
function addClick() {  
    formClear();  
}
```

Update (PUT) a Product

At some point, a user is going to want to change the information about a product. Earlier in this article, you added an Edit button to retrieve product data and display that data in the input fields. The function also changes the updateButton's text property to **Update**. When the user clicks on the Update button, take the data from the input fields and use the HTML verb `PUT` to call the `Put` method in your Web API. Modify the `Put` method in your `ProductController` to look like the following.

```
[HttpPut ()]  
public IActionResult Put (int id, Product product)  
{  
    IActionResult ret = null;  
    if (Update (product))  
    {  
        ret = Ok (product);  
    }  
    else  
    {  
        ret = NotFound ();  
    }  
    return ret;  
}
```

The call to the `Update` method from the `Put` method is a mock to simulate modifying data in a product table. To see the `NotFound` returned from this method, change the `true` value to a `false` value in the `Update` method shown below.

```
private bool Update (Product product)  
{  
    return true;  
}
```

Edit and Submit the Product Data

With your `Put` method in place, it's now time to modify the `updateClick()` function in your JavaScript. Locate the `updateClick()` function and add an else condition to call a function named `productUpdate`.

```
function updateClick() {
    ...
    ...
    if ($("#updateButton").text().trim() == "Add") {
        productAdd(Product);
    }
    else {
        productUpdate(Product);
    }
}
```

The `productUpdate` function is passed the product object, and you'll send that via an Ajax call using the verb `PUT`. This maps to the Put method you created in the controller. Just like you did in the POST Ajax call, change the `type` property to `PUT`, set the `contentType` to use JSON and serialize the Product object using `JSON.stringify`.

```
function productUpdate(product) {
    $.ajax({
        url: "/api/Product",
        type: 'PUT',
        contentType:
            "application/json;charset=utf-8",
        data: JSON.stringify(product),
        success: function (product) {
            productUpdateSuccess(product);
        },
        error: function (request, message, error) {
            handleException(request, message, error);
        }
    });
}
```

If the update is successful, a function called `productUpdateSuccess` is called and passed the updated product object. The `productUpdateSuccess` function passes the product object on to another function named `productUpdateInTable`. I know that you could just call the `productUpdateInTable` function directly from the Ajax call, but I like to keep my pattern of naming functions consistent. Besides, in the future, you may want to do some additional coding within the `productUpdateSuccess` function.

```
function productUpdateSuccess(product) {
    productUpdateInTable(product);
}
```

Update the Modified Data in the HTML Table

The `productUpdateInTable` function locates the row in the HTML table just updated. Once the row is located, a new table row is built using the `productBuildTableRow` function that you created earlier. This newly created row is inserted immediately after the row of the original data. The original row is then removed from the table. All of this happens so fast that the user doesn't even realize when this add and delete occur. Another option is to clear the whole table and reload all of the data by calling the `productList` function.

```
function productUpdateInTable(product) {
    // Find Product in <table>
    var row = $("#productTable button[data-id='" +
        product.ProductId + "']").parents("tr")[0];

    // Add changed product to table
    $(row).after(productBuildTableRow(product));

    // Remove original product
    $(row).remove();
    formClear(); // Clear form fields

    // Change Update Button Text
    $("#updateButton").text("Add");
}
```

Delete (DELETE) a Product

The last HTTP verb you need to learn is `DELETE`. Once again, you need to modify the `ProductController`. Locate the `Delete` method and modify it by adding the `[HttpDelete]` attribute, and changing the return value to `IHttpActionResult`.

```
[HttpDelete()]
public IHttpActionResult Delete(int id)
{
    IHttpActionResult ret = null;
    if (DeleteProduct(id))
    {
        ret = Ok(true);
    }
    else
    {
        ret = NotFound();
    }
}
```



```
    }  
    return ret;  
}
```

The `DeleteProduct` method is a mock to simulate deleting a product from a table. Just create a dummy method that returns `true` for now. You can switch this method to return `false` to test what happens if the delete fails.

```
private bool DeleteProduct(int id)  
{  
    return true;  
}
```

Add a Delete Button

Each row in your table should have a Delete button (**Figure 1**). Add that button now by adding a new `<th>` within the `<thead>` tag.

```
<th>Delete</th>
```

Modify the `productBuildTableRow` function and add the following code immediately before the closing table row tag (`</tr>`).

```
"<td>" +  
    "<button type='button' " +  
        "onclick='productDelete(this);' " +  
        "class='btn btn-default' " +  
        "data-id='" + product.ProductId + "'>" +  
        "<span class='glyphicon glyphicon-remove' />" +  
    "</button>" +  
"</td>" +
```

Delete a Product Using Ajax

In the code that builds the delete button, the `onClick` event calls a function named `productDelete`. Pass `this` to the `productDelete` function so that it can use this reference to retrieve the value of the product ID contained in the `data-id` attribute. The Ajax call sets the URL to include the ID of product to delete and sets the type property

to `DELETE`. Upon successfully deleting a product, remove the complete row from the HTML by finding the `<tr>` tag that contains the current delete button and calling the `remove()` method on that table row.

```
function productDelete(ctl) {
    var id = $(ctl).data("id");

    $.ajax({
        url: "/api/Product/" + id,
        type: 'DELETE',
        success: function(product) {
            $(ctl).parents("tr").remove();
        },
        error: function(request, message, error) {
            handleException(request, message, error);
        }
    });
}
```

Summary

In this article, you learned the various HTTP verbs `GET`, `POST`, `PUT`, and `DELETE` and how to use them to create add, edit, delete, and list Web pages. What's nice about combining the Web API with pure HTML is that you're not performing full postbacks to the server, rebuilding the whole page, and then resending the whole page back down to the client browser. This comes in very handy on mobile devices where your user may be on a limited data connection through a service provider. The less data you send across, the fewer minutes and gigs of data you use on mobile phone plans. In the next article, you'll see some methods for handling validation and error messages.

Courtesy: <https://www.codemag.com/article/1601031/CRUD-in-HTML-JavaScript-and-jQuery-Using-the-Web-API>

Modified: 2021.10.06.8.10.PM

Dököll Solutions, Inc