Using jQuery and Callbacks to Call a .NET 5 Web API

Asynchronous **J**avaScript **A**nd **X**ML (Ajax) is the cornerstone of communication between your client-side and server-side code. Regardless of whether you use JavaScript, jQuery, Angular, React or any other client-side language, they all use Ajax under the hood to send and receive data from a web server. Using Ajax you can read data from, or send data to, a web server all without reloading the current web page. In other words, you can manipulate the DOM and the data for the web page without having to perform a post-back to the web server that hosts the web page. Ajax gives you a huge speed benefit because there is less data going back and forth across the internet. Once you learn how to interact with Ajax, you will find the concepts apply to whatever front-end language you use.

In this blog post you learn to make an Ajax call to a .NET 5 Web API server using jQuery's $.ajax() method and callbacks. jQuery's $.ajax() can work with either callbacks, or can work with promises. In this blog post you learn to use callbacks as you may still this approach used in code that you might work on. In the next blog post you learn to use the promise approach.

Download Starting Projects

Instead of creating a front-end web server and a .NET 5 Web API server in this blog post I have two sample projects you may download to get started quickly. If you are unfamiliar with building a front-end web server and a .NET 5 Web API server, you can build them from scratch step-by-step in my three blog posts listed below.

1. Create CRUD Web API in .NET 5
2. Create .NET 5 MVC Application for Ajax Communication
3. Create Node Web Server for Ajax Communication

You can find all three of these blog posts at https://www.pdsa.com/blog. Instructions for getting the samples that you can start with are contained in each blog post. You are going to need blog post #1, then choose the appropriate web server you wish to use for serving web pages; either .NET MVC (#2) or NodeJS (#3).

Start Both Projects

After you have reviewed the blog posts and downloaded the appropriate sample projects to your hard drive, start both projects running. The first project to load is the Web API project. Open the **WebAPI** folder in VS Code and click on the **Run | Start Debugging** menus to load and run the .NET Web API project.

Open the **AjaxSample** folder in VS Code.

If you are using node, open the **AjaxSample** folder in VS Code, open a Terminal window and type **npm install**. Then type **npm run dev** to start the web server running and to have it display the index page in your browser.

If you are using the .NET MVC application, open the **AjaxSample-NET** folder in VS Code and click on the **Run | Start Debugging** menus to load and run the .NET MVC project. The index.cshtml page should now be displayed in your browser window.

Try it Out

Go to your browser for the front-end web server (localhost:3000) and you should see a page that looks like Figure 1. If your browser looks like this, everything is working for your front-end web server.

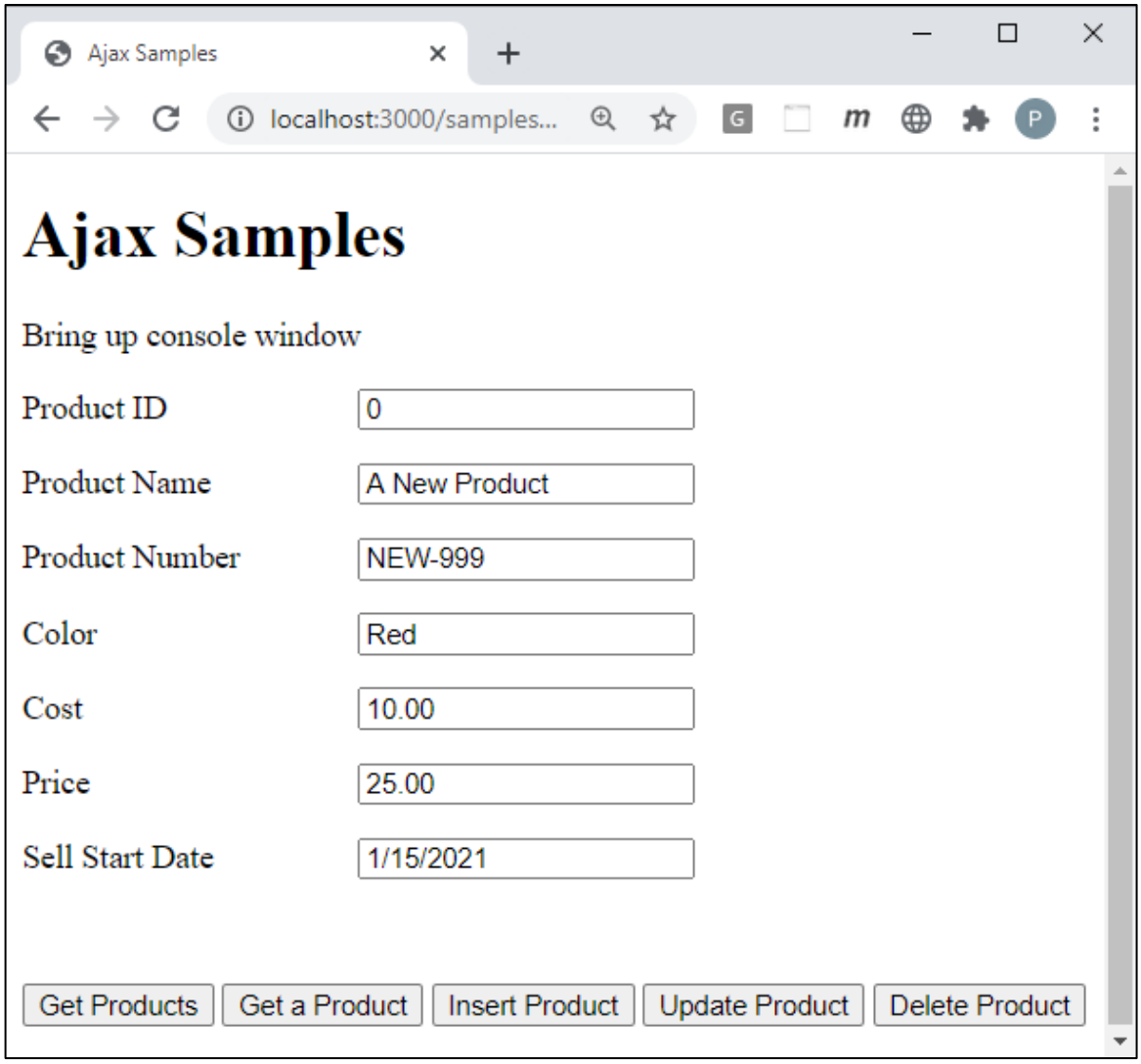

Figure 1: This is the starting project from which you are going to build your CRUD logic using Ajax and .NET 5.

Open the Browser Tools in your browser, usually accomplished by clicking the F12 key. Click the **Get Products** button and you should see the product data retrieved from the Product table in the AdventureWorksLT database and displayed in your console window.

Install jQuery

If you have not already done so, you need to install jQuery into your node server project. If you are using the MVC application, jQuery is already installed, so you can skip to the next section of this blog post. Open the Terminal window in VS Code in your node server project and type the following command.

```
npm install jquery
```

After jQuery has been installed, open you index page and add a new <script> tag to use jQuery before all other <script> tags on the page.

```
<script src="/node_modules/jquery/dist/jquery.min.js"></script>
```

Get all Products

Prior to jQuery 1.5, the implementation of Ajax used function callbacks to determine when an Ajax call was successful or not. Open the index page of the web server project you downloaded and write code in the empty get() function shown in Listing 1.

```
function get() {
  $.ajax({
    url: URL,
    type: 'GET',
    contentType: 'application/json',
    success: function (data) {
      displayMessage("Products Retrieved");
      console.log(data);
    }
  });
}
```

Listing 1: The jQuery $.ajax() function is a very compact way to make Ajax calls.

The $.ajax() accepts a JSON object with settings that tell the Ajax call what to do. The properties used in this example are shown in

Property	Description
url	The URL to call to retrieve the data.
type	The HTTP verb to use.
contentType	The type of content being sent. This property is optional when performing a GET.
success	A callback function that is called when the data is successfully retrieved.

Table 1: There are many options you may pass to $.ajax(), however these are the most popular.

There are many more properties you can pass in this settings object. Check out https://api.jquery.com/jquery.ajax/ for more information.

Try it Out

Save the changes you made to the index page and go to your browser. Open your developer tools on your browser and display the console window. Click on the Get Products button and you should see several objects appear in your console window.

Adding an Error Callback

There is no guarantee that the Ajax call you make will succeed, so besides supplying a success callback function it is recommended you always supply an error callback function as well. Modify the get() function you wrote previously to look like the following code.

```
function get() {
  $.ajax({
    url: URL,
    type: 'GET',
    contentType: "application/json",
    success: function (data) {
      displayMessage("Products Retrieved");
      console.log(data);
    },
    error: function (error) {
      handleAjaxError(error);
    }
  });
}
```

Listing 2: Add an error callback function to handle any exceptions.

The handleAjaxError() function is located in the ajax-common.js file and simply reports the error to the console window.

Adding a Complete Callback

Another callback you may specify in the settings object is called **complete** (Listing 3). This method executes regardless of whether the call was successful or not. It is not necessary to include this callback, but it is one of the many options available to you when using the $.ajax() method.

```
function get() {
  $.ajax({
    url: URL,
    type: 'GET',
    contentType: "application/json",
    success: function (data) {
      displayMessage("Products Retrieved");
      console.log(data);
    },
    error: function (error) {
      handleAjaxError(error);
    },
    complete: function () {
      console.log("In the complete function");
    }
  });
}
```

Listing 3: This version of the get() function has three callbacks to handle any situation of what might happen with an Ajax call.

Get a Single Product

There is no guarantee that the Ajax call you make will succeed, so besides supplying a success callback function it is recommended you always supply an error callback function as well. Modify the get() function you wrote previously to look like Listing 4.

```
function getProduct() {
  $.ajax({
    url: URL + "/" + getValue("productID"),
    type: 'GET',
    contentType: "application/json",
    success: function (data) {
      displayMessage("Product Retrieved");
      setInput(data);
    },
    error: function (error) {
      handleAjaxError(error);
    }
  });
}
```

Listing 4: Pass a unique identifier on the URL to retrieve a single record.

Try it Out

Save the changes you made to the index page and go to your browser. Click on the Get a Product button and you should the input fields update with the data from the database.

Insert a Product

Change the insertProduct() function, shown in Listing 5, to call $.ajax() to submit the data you gather from the inputs to the Post() method in the Web API project. When posting data to a Web API you set the *type* property as "POST" and set the *contentType* property to "application/json" to tell the server what kind of data is being sent. Set the *data* property to the stringified version of the product object created from the getFromInput() method.

```
function insertProduct() {
  // Build product object from user input
  let product = getFromInput();

  $.ajax({
    url: URL,
    type: "POST",
    contentType: "application/json",
    data: JSON.stringify(product),
    success: function (data) {
      displayMessage("Product Inserted");
      console.log(data);
      setInput(data);
    },
    error: function (error) {
      handleAjaxError(error);
    }
  });
}
```

Listing 5: When posting data, make sure you specify the contentType

Try it Out

Save the changes you made to the index page and go to your browser. Open your developer tools on your browser and display the console window. Click on the Insert Product button and you should a new object appear in your console window. The input fields on the screen should also update with the data from the Web API server.

Update a Product

Change the updateProduct() function, shown in Listing 6, to call $.ajax() to submit the data you gather from the inputs to the Put() method in the Web API project. Just like the insert function, change the *type* property to "PUT" and set the *contentType* property to "application/json" to tell the server what kind of data is being sent. Set the *data* property to the stringified version of the product object created from the getFromInput() method.

```
function updateProduct() {
  // Build product object from user input
  let product = getFromInput();

  $.ajax({
    url: URL + "/" + product.productID,
    type: "PUT",
    contentType: "application/json",
    data: JSON.stringify(product),
    success: function (data) {
      displayMessage("Product Updated");
      console.log(data);
      setInput(data);
    },
    error: function (error) {
      handleAjaxError(error);
    }
  });
}
```

Listing 6: When updating data, pass the product id on the URL line.

Try it Out

Save the changes you made to the index page and go to your browser. Open your developer tools on your browser and display the console window. Fill in the Product ID value you wish to update. Click on the Update Product button and you should a new object appear in your console window. The input fields on the screen should also update with the data from the Web API server.

Delete a Product

Change the deleteProduct() function, shown in Listing 7, to call $.ajax() to submit the product ID from the input field to the Delete() method in the Web API project.

```
function deleteProduct() {
  $.ajax({
    url: URL + "/" + getValue("productID"),
    type: "DELETE",
    success: function (data) {
      displayMessage("Product Deleted");
      console.log(data);
      clearInput();
    },
    error: function (error) {
      handleAjaxError(error);
    }
  });
}
```

Listing 7: After deleting a product, clear the input fields.

Try it Out

Save the changes you made to the index page and go to your browser. Open your developer tools on your browser and display the console window. Fill in the Product ID value you wish to delete. Click on the Delete Product button and you should a true value appear in your console window. The input fields on the screen should be reset to default values.

Summary

The jQuery $.ajax() is a nice alternative to the XMLHttpRequest object available in JavaScript. The syntax is much simpler, and can use callback as shown in this blog post, or can use a Promise-based approach as you will see in the next blog post.

Sample Code

You can download the complete sample code at my https://github.com/PaulDSheriff/BlogPosts page.