Working with jQuery, Promises and Error Handling

In this blog post you learn to make an Ajax call to a .NET 5 Web API server using jQuery's $.ajax() method and promises. jQuery's $.ajax() can work with either callbacks, or can work with promises. In this blog post you learn to use promises as this is the preferred approach today.

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.

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.

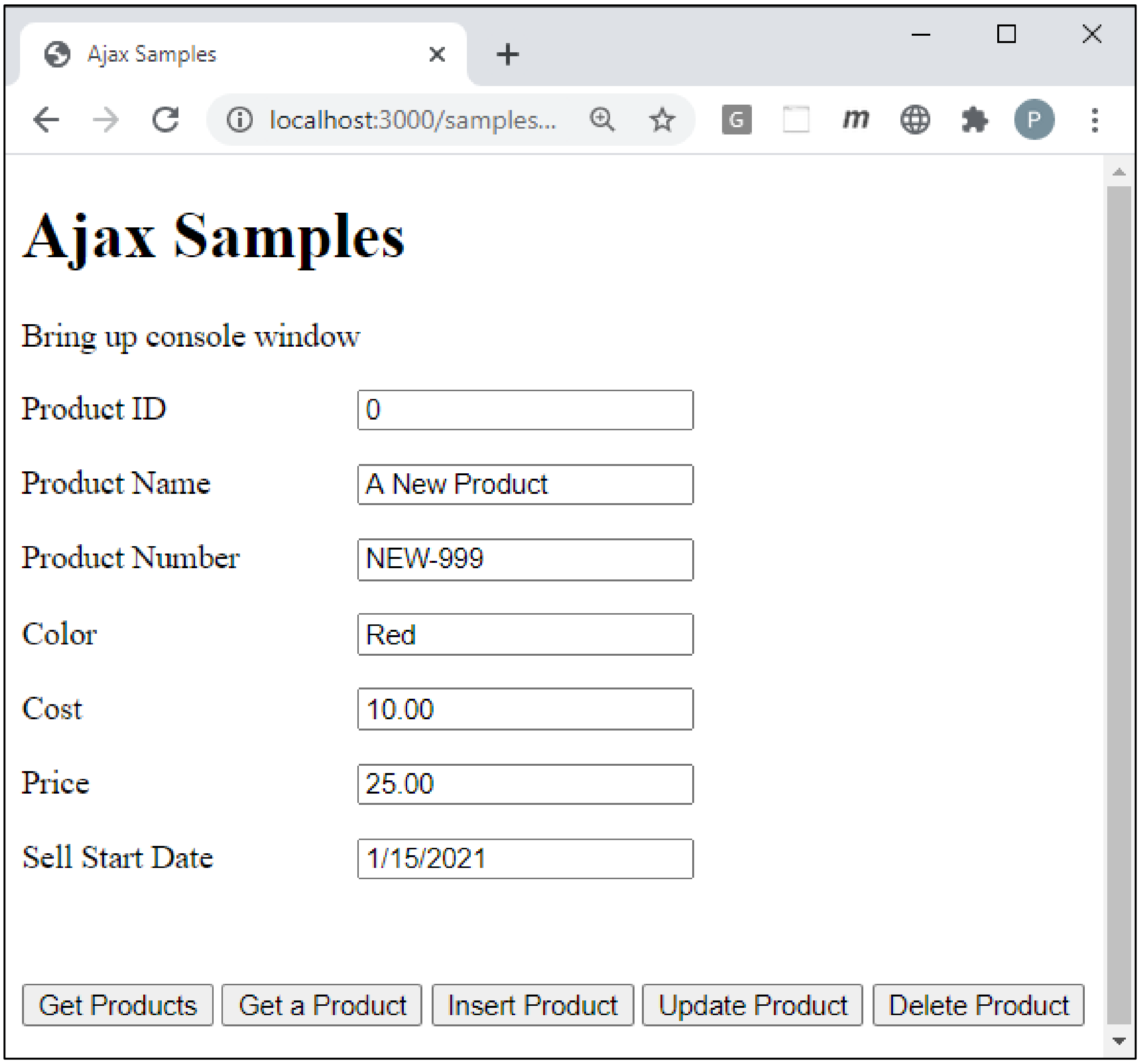
Open the **AjaxSample** folder in VS Code.

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.



Ajax Samples

Bring up console window

Product ID	0
Product Name	A New Product
Product Number	NEW-999
Color	Red
Cost	10.00
Price	25.00
Sell Start Date	1/15/2021

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

Using callbacks is an older method of getting notification that something has succeeded or failed. Promises have been around for many years now and is how you should be checking for success or failure of an Ajax call. In this next section of this blog post you are going to duplicate the two functions you wrote above for getting and inserting product data using Promises.

The complete get() function is shown in Listing 1. This code uses the .done() method to be notified the jQuery Ajax call has completed successfully. It uses the .fail() method to be notified of any error that occurred. The .always() method is run regardless of whether the call succeeded or not. The settings object passed to the $.ajax() method is the same as what you used previously, you just omit the success, error and complete callback functions.

```
function get() {
  $.ajax(URL)
    .done(function (data) {
      displayMessage("Products Retrieved");
      console.log(data);
    })
    .fail(function (error) {
      handleAjaxError(error);
    })
    .always(function () {
      console.log("In the always() method");
    });
}
```

Listing 1: Retrieve data using an Ajax promise.

Try it Out

Copy the get() function from Listing 1 into the index.html page and save your changes. Go to the browser and click on the Get Products button and you should see all the same 200+ product objects appear in the console window.

Passing in Settings Object

When performing a GET to retrieve data from a Web API server you can just pass in the URL. However, the $.ajax() method can also take a literal settings object where you can set various properties to tell how $.ajax() to make the call. For example, the code shown in Listing 1 is equivalent to the code shown in Listing 2. The literal object you pass to $.ajax() can have one or several properties set. You can see the complete list of properties at https://api.jquery.com/jquery.ajax/.

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

Listing 2: Pass in a settings object to inform $.ajax() how to process the Ajax call.

Get a Single Product

To retrieve a single product from the Web API you include the unique product identifier on the URL as shown in Listing 3. Once you get the data back from the server, display the product in the input fields by calling the setInput() function.

```
function getProduct() {
  $.ajax(URL + "/" + getValue("productID"))
    .done(function (data) {
      displayMessage("Product Retrieved");
      console.log(data);
      setInput(data);
    })
    .fail(function (error) {
      handleAjaxError(error);
    })
    .always(function () {
      console.log("In the always() method");
    });
}
```

Listing 3: Retrieve a single product by passing the product id on the URL.

Try it Out

Save the changes on the index page and go back to your browser. Fill in a product id in the input field and click on the Get a Product button. You should see the data for that product appear in the various input fields on the page.

Insert a Product

Change the insertProduct() function to look like Listing 4. Notice that you set the *data* property of the literal object to the stringified version of the literal product object returned from the getFromInput() function.

```
function insertProduct() {
  // Gather data from user input
  let product = getFromInput();

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

Listing 4: Insert a product using Ajax promises.

Try it Out

Save your changes on the index page. Go to the browser and click on the Insert Product button and you should see the new product data appear in the input fields with the new product id filled in.

Update a Product

When you want to update a product, pass the product id on the URL, stringify the literal product object and place it into the *data* property, and set the *type* property to "PUT" as shown in Listing 5.

```
function updateProduct() {
  // Gather data from user input
  let product = getFromInput();

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

Listing 5: Update data using the PUT verb and passing the product id on the URL.

Try it Out

Save your changes on the index page. Go to the browser and put the id of the product to update in the Product ID input field. Click on the Update Product button and you should see the updated product data appear in the input fields.

Delete a Product

Modify the deleteProduct() function on the index page to look like Listing 6. Pass the product id to delete on the URL and set the *type* property to "DELETE".

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

Listing 6: Delete a product by passing the product id on the URL and setting the type to DELETE.

Try it Out

Save the changes to the index page. Go to the browser and fill in the product id to delete in the input field. Click on the Delete Product button and you should see a true value displayed in the console window. The input fields will all be set to a default value.

Using Try…Catch with Ajax

Error checking is important in all applications. When using asynchronous calls, using try…catch may not always be the correct approach for error handling, however. It's not that you can't use a try…catch, but you need to be aware of where you can use it and where you can't.

Incorrect Method of Catching Errors

Do not wrap a try...catch around the $.ajax() method call because it is an asynchronous call. The try...catch goes out of scope immediately after the $.ajax() method has been executed. So, once the promise is fulfilled, the try...catch is out of scope. The code in Listing 7 is NOT the correct approach. I have sprinkled some console.log() calls within this code so you can see the order of execution. If you notice that before the .done() method is called, the finally block has already executed. This tells you quite clearly that the try…catch is out of scope prior to the promise being fulfilled.

```
function incorrectTryCatch() {
  try {
    console.log("In the try block");
    $.ajax({
      url: URL,
      type: 'GET',
      contentType: "application/json"
    })
      .done(function (data) {
        console.log("Retrieved the data");
        console.log(data);
      })
      .fail(function (error) {
        handleAjaxError(error);
      })
      .always(function () {
        console.log("In the always() method");
      });
  } catch (error) {
    console.log("In the catch block");
    console.error(error);
  }
  finally {
    console.log("In the finally block");
  }
  console.log("Ending this function");
}
```

Listing 7: Don't wrap a try…catch around an asynchronous call as the catch block will be out of scope when the call is completed.

Try it Out

Open the index page and add a new button somewhere by the other buttons on the form.

```
<button type="button" onclick="incorrectTryCatch();">
  Incorrect try...catch Sample
</button>
```

Add the code in Listing 7 anywhere within the <script> tags where your other code is located. Save all your changes and go back to the browser. Click on the Incorrect try…catch Sample button and check the results in the console window.

Correct Method of using Try…Catch

Within the .done(), .fail() or .always() methods is where it is ok to place try…catch blocks. Remember, the .fail() method will be invoked if there is a problem with the Ajax call so really there is no need for a try…catch around the entire $.ajax() call.

```
function correctTryCatch() {
  $.ajax({
    url: URL,
    type: 'GET',
    contentType: "application/json"
  })
    .done(function (data) {
      // NOTE: try...catch belongs within each promise method
      try {
        console.log(data);
      } catch (error) {
        console.error(error);
      }
    })
    .fail(function (error) {
      handleAjaxError(error);
    })
    .always(function () {
      console.log("In the always() method");
    });
}
```

Listing 8: It is fine to use a try…catch within any individual method of the promise.

Try it Out

Open the index page and add a new button somewhere by the other buttons on the form.

```
<button type="button" onclick="correctTryCatch();">
  Correct try...catch Sample
</button>
```

Add the code in Listing 8 anywhere within the <script> tags where your other code is located. Save all your changes and go back to the browser. Click on the Correct try…catch Sample button and check the results in the console window.

Summary

The jQuery $.ajax() is a nice alternative to the XMLHttpRequest object available in JavaScript. The syntax is much simpler, and it provides a Promise-based approach to development which can greatly simplify complicated code. Be careful of your use of try…catch blocks as incorrect usage can provide results you may not be expecting.

Sample Code

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