FPS Vendor Call

Instructional supplement

Sweitzer Ryan (SHX8RKD)

2018

Contents

[Introduction 2](#_Toc521055393)

[General Overview 2](#_Toc521055394)

[Obtain and Upload Vendor Script 3](#_Toc521055395)

[ASP.NET Core Web Api Overview 6](#_Toc521055396)

[Setting up the front-end to accept the client code 8](#_Toc521055397)

[Return Data as JSON to the Back-end 10](#_Toc521055398)

[Back-end: Parse the JSON and Insert to DB 12](#_Toc521055399)

# Introduction

The FPS Vendor Call is a bit of a complicated program. The purpose of this document is to help guide others who have taken over the project and need a reference to its functionality from beginning to end. This program is broken down into a few steps:

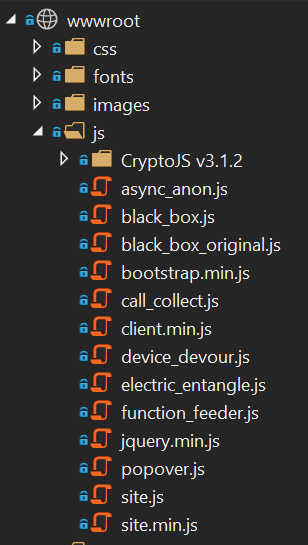
1. Overview of general project setup.
2. Obtain vendor script and insert it into the database.
3. ASP.NET Core Web Api Overview.
4. Setting up the front-end to accept the client code.
5. Transforming that data from script to JSON to be pushed to the back-end.
6. Strip down the back-end JSON object and parse it to the database.

The objective of this instruction is to guide the user from beginning to end of the process. There are a lot of moving parts within this machine and hope it helps to supplement the process.

# General Overview

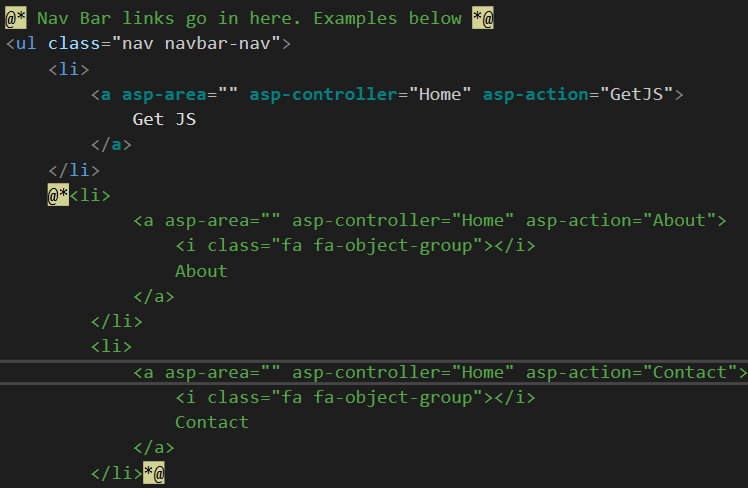
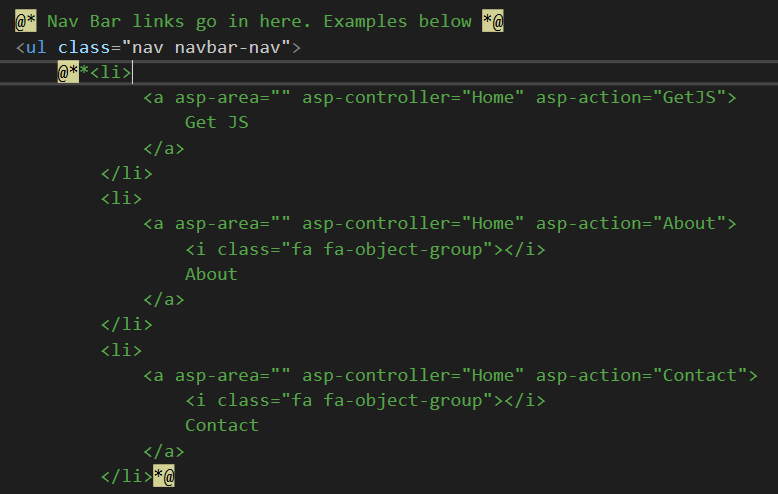
The program was created using Visual Studio’s ASP.NET Core MVC model version 2.1 using the entity framework. It involves C#, SQL, HTML/CSS, JavaScript with jQuery and AJAX. The project consists of two components: The Web Api and Web Client. The Web Api is very small and only makes one call out to the database to return data that is equal true in a JSON format. The Web Client uses a series of models, views and controllers to call out to the Web Api for the client code and processes that into usable data to be submitted into a database. Next I will go through the process of how this works from a fresh start perspective.

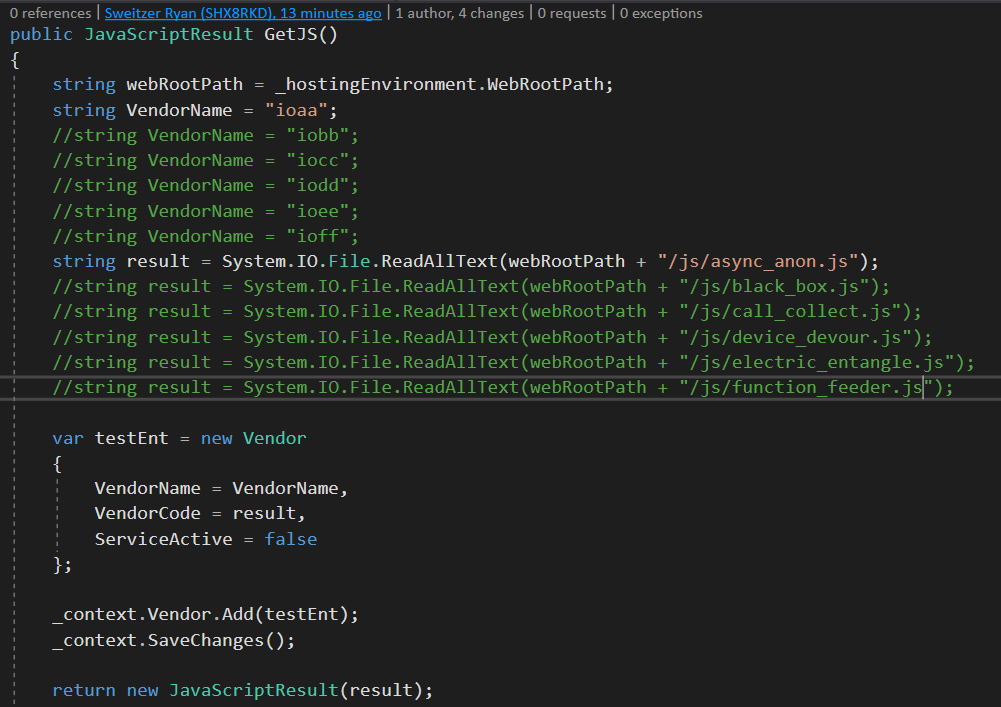
# Obtain and Upload Vendor Script

When you receive this program, it’ll be attached to a remote DB but you’ll probably want to tinker with it in your own environment or you might be trying to add a new vendor call to the process. This part of the guide should help you out in that regard. If you look in wwwroot > js, there is a long list of JS files but the ones you are concerned with are the ones that are holding the vendor code. Specifically, these are:

1. async\_anon
2. black\_box
3. call\_collect
4. device\_devour
5. electric\_entangle
6. function\_feeder

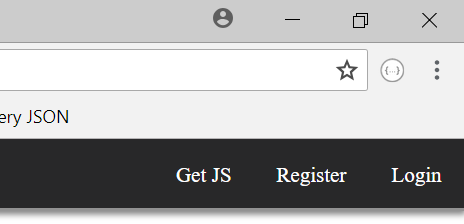
These all contain the code that is directly up loaded into the database. All of them with the exception of black\_box all generate generic static data. 1,3,4 and 5 are all hard-coded strings and 6 returns a JSON object. Black\_box however, is an actual call out to iOvations using their test server. The black\_box\_original is the orginial code that I received from CAB that I’ve built this whole project around while the black\_box file is more or less the dumbed down version used to make the same call for what I needed it for. Refer to the black\_box\_orginal if you plan to use the live calls since the address is stored in there. Now for storing the JS calls in the database:

1. Go to the “\_Layout.cshtml” file and uncomment the link for “GetJS”.
2. Go to the HomeController and find the GetJS method

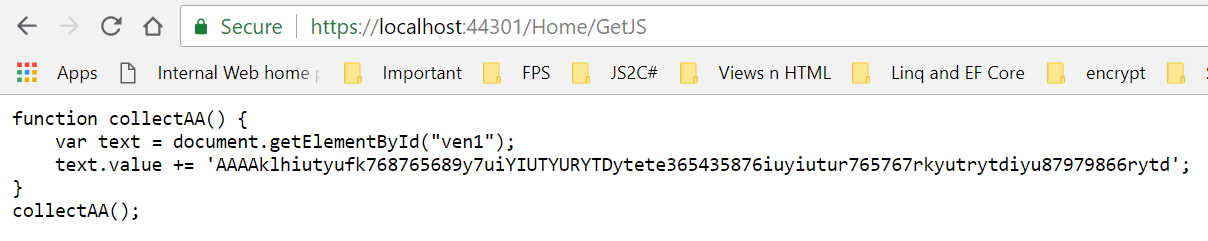


The “VendorName” is the name that is added to name column in the DB and the “result” is the path to the JS file and anything that is written in there will be inserted into the vendercode column. This all happens in one shot in the next step.

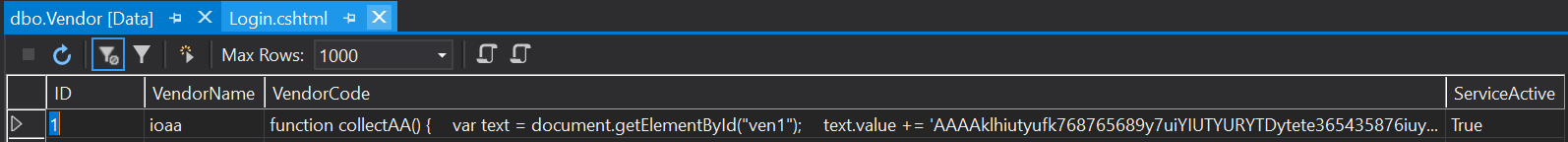
1. Start the program up and click on the Get JS link in the navbar



Once you click on the link the next screen appears:



This shows you what is in the JS file and what has been uploaded to the database. Since VendorName ioaa and the result was equal to async\_anon.js, that was what was uploaded to the database table “Vendor”:

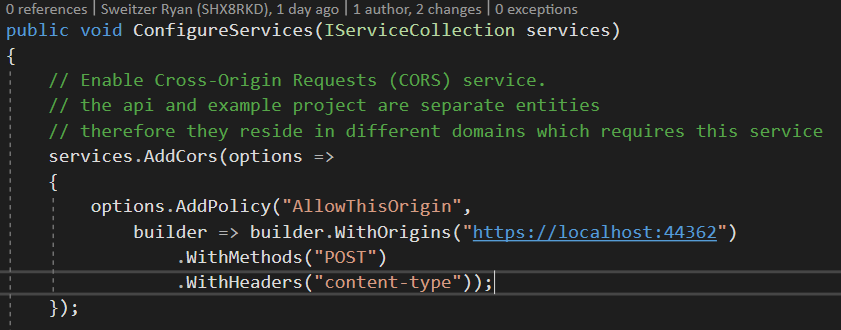


You repeat these steps as necessary to upload all the necessary vendors to the database.

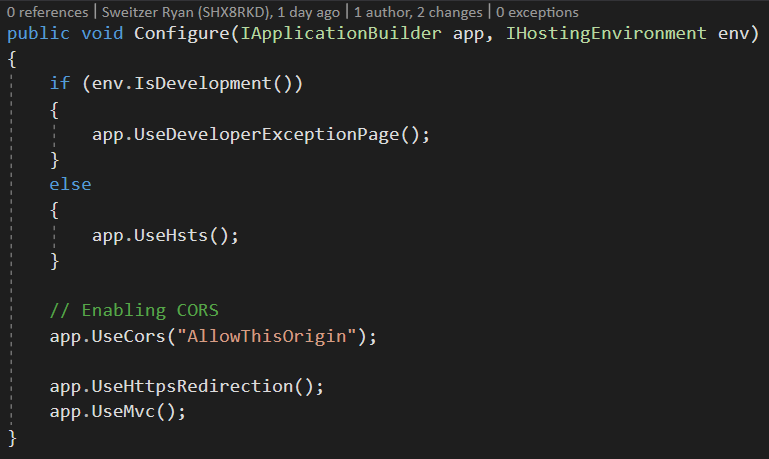
# ASP.NET Core Web Api Overview

As mentioned before, the Web Api is pretty simple and straight forward. There’s only one return that delivers all vendor code that is marked true on the address api/object. The one thing I want to mention in this section is the fact that the solution is split into two separate projects. Since this is the case, you will have to have multiple startup projects so both will run at the same time and work in tandem. I mention this because they will be using different port numbers hence they will be located in two different domains which adds a complication in making an AJAX request. To get around this, CORS has to be applied to the Web Api. Here’s how I set that up:

In the startup.cs I have the following set up:

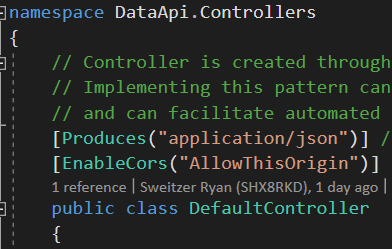


This points to the port number that is used by the ProjectFps project and specifically looking for the POST method and header with content-type. If you don’t want it to be so specific you can remove the methods you see here and replace them with .AllowAnyOrigin/Method/Header. If you are having any difficulty getting the AJAX request to come through, this is a good place to start.

To get this added to the project, it’s simply stated as:

🡺

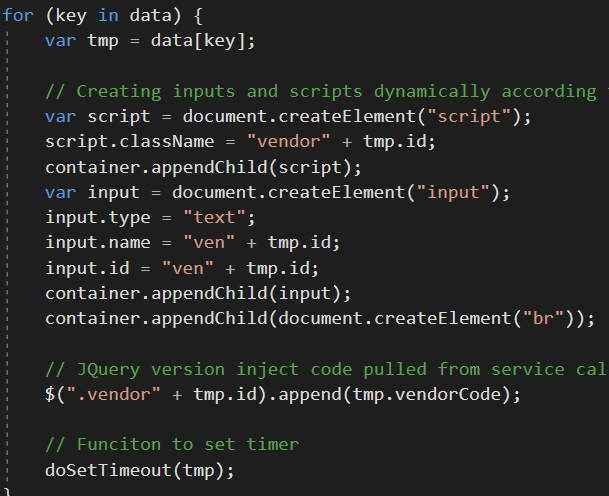
Last, you just add the reference in the controller:



🡺

See the project to look at how the output is returned.

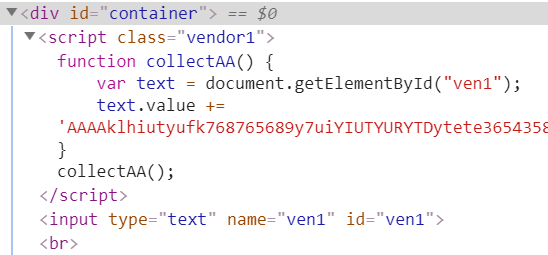
# Setting up the front-end to accept the client code

In this section, I’m going to break down the first half of the JavaScript to show you exactly what is happening in the background. The AJAX request is sent out immediately as the Login client loads to fetch the Vendor rows that are marked true. Based on this number of objects that are returned, the JS function creates that amount of scripts and inputs and inserts them into the div with the id of “container”:

Create the scripts 🡺

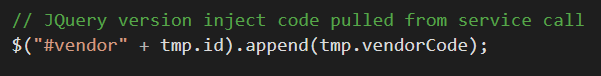
Create the inputs 🡺

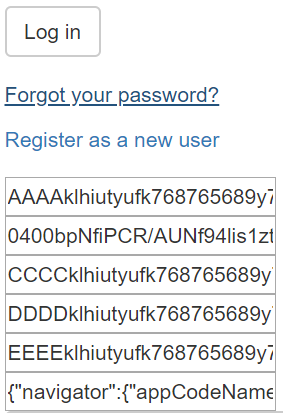
Add inputs & scripts to div 🡺



Example of executed code 🡺

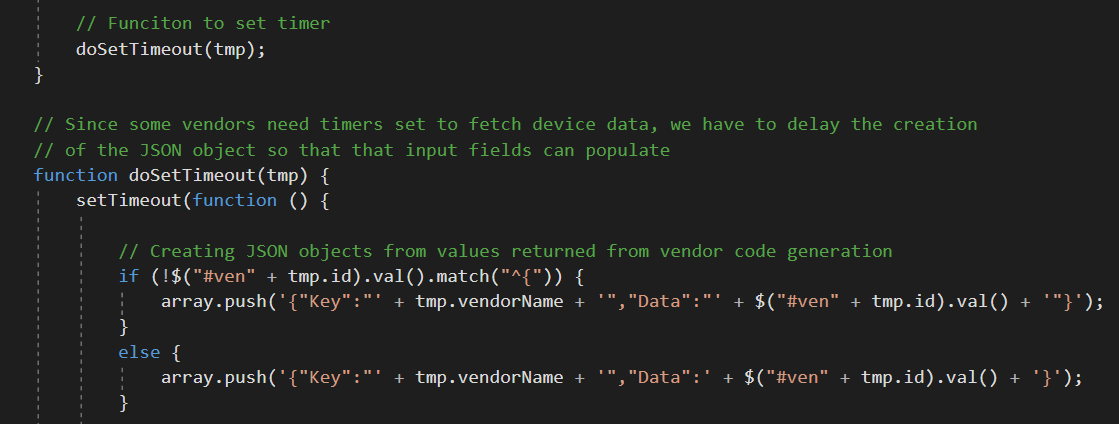
The IDs that are generated from the Api call serve as the ID values for the scripts and inputs. For example: the ID of ioaa is 1 and iobb is 2. Using the returned AJAX Data as an object, we call the ID property and append that to a hard-coded “vendor” for the scripts and “ven” for the inputs. It would be a good time to mention that the vendor scripts stored in the DB are hard-coded to call these inputs as they are generated to store their results with in the inputs to further processed in the code. The I use this part of the code to actually call the scripts from the DB and insert them into the generated script tags:



Then, as a result, I will show you how the webpage will look while the inputs are not hidden:

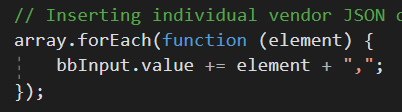
At this point. The scripts have run and have returned their results into the inputs. For the next part, we are going to take these results and create a JSON object that can be returned to the back-end as well as encrypting the object before it is sent.

# Return Data as JSON to the Back-end

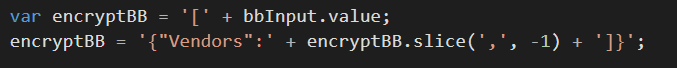
Now that we have our returned data, we need to reformat it into a JSON object in order to be sent to the back-end. It would be wise to note here that the rest of the JS is going to be set on a timer for one second. The reason this is done is because like the Iobb call, some vendor returns are set on a timer to ensure that the user information that is gathered is complete since their functions (I’m assuming at this point) are complicated. So here is how the function works:

JSON objects are simple in themselves but are much harder to create especially in a programmatic way. There are a lot of things to check for such as what is being returned (i.e. object, string, int) and to ensure whatever is being returned that it is being returned in the right format (i.e. strings are surrounded with quotes, objects retain their original form). Since we are expecting a mix of either strings or objects as a return from a vendor, we have first check and see if the return is a string, if so then the returns need to be surrounded with quotation marks since this is not done for us beforehand. The vendor name is taken from the returned “data” object and inserted into the “Key” and the values that are within the generated inputs are set the “Data” creating the key/value pair. Once the objects are created, they are then pushed into an array that was initialized at the beginning of the script. The function is then executed within the for in loop to ensure that all the objects are created and pushed into the array.

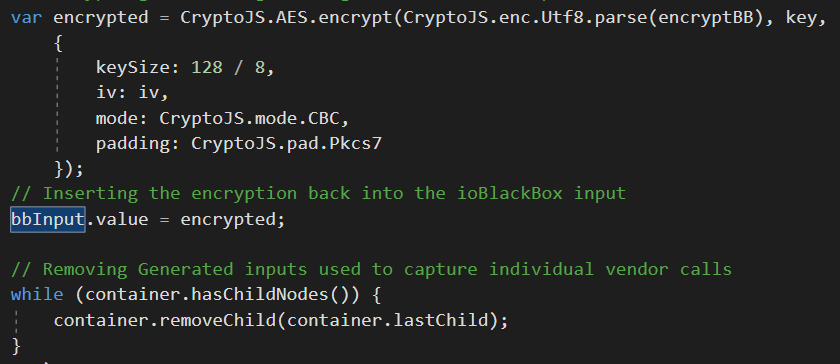
Now that we have all of our separate objects within an array, we can work outside of the for in loop but we still need to be within the timer so that code will not execute until the returns are complete. We will now take our separate objects and form them into one JSON object to be encrypted and sent to the backend. First, we will take the array and insert it into our “ioBlackBox” input that is coupled with the div “container” inside the form:



Each element is appended by a comma to ensure the correct JSON format which is needed to separate the separate objects. Next, we are going to create a variable that is going to do two things at once: Create the JSON object and hold the object to be encrypted:



We are going to put our objects within an array within an object. The code is also there to trim off the trailing comma so that were still ensuring the correct JSON format. Finally, we are going to encrypt the object and have it ready to be accepted from the back-end:



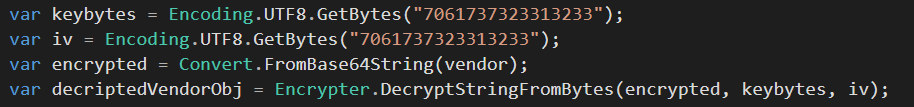
It’s pretty straight forward. The variable is passed to the encryption method and then passed back to the input value and the key and initialization vector are created at the beginning of the script. The while statement is there to ensure the clean-up of the generated inputs in regards to formatting and obfuscation.

# Back-end: Parse the JSON and Insert to DB

Since we need to execute all of this code as the user submits their form, all of this code need to be ran in the post method. In order to grab the encrypted JSON object from the front-end we can use the Request.Form:



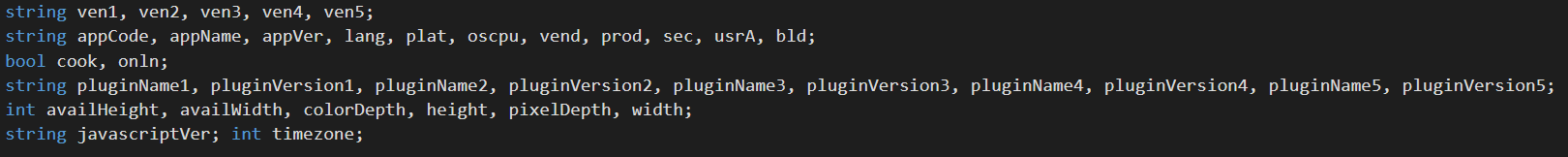
This will look for the value within the “name” attribute from the front-end. Next we set up our decryption environment:



The key and the initialization vector is the same as the front end and it uses a class “Encrypter” that is located in the Classes folder. Next, we take the decrypted string and create a JSON object for the C# code to work with:



The C# code uses the JSON.Net framework to work with the JSON object. This is really powerful framework that makes working with JSON in C# pretty much a breeze and it is also found to be highly efficient in processing JSON objects as well. The next step involves actually parsing out the JSON object. First the code Instantiates all the fields needed to submit the data to the DB:



This includes all the fields we will gather from the returned strings from ioaa to ioee as well as every separate value field we are expected to return from the ioff JSON object. Next is the creation of JToken collection of each individual return. Since all of the vendors can be turned on or off we first have to check to see if we even need to create the collection in the first place:



If this proves to be false we can skip over the creation of the collection and move on to the next object but if it is true, then we move on with the creation:



The collection is created by taking the whole JSON object and using the SelectToken method which uses the JSONPath expression to find the correct object. For this example, the JSONPath expression takes the “Vendors” array and does a search for the “Key” value of ‘ioaa’ then sets the collection to the value of that object. So now, the C# ioaa variable is now holding the “Key” ioaa and the “Data”, the returned string. In the next line. We set the C# variable “ven1” to value that is held in “Data” which is the returned string using the “ElementAt” method. For the rest of the code, if the vendor is turned off, the else statement sets a null or zero value to the global variables. The code pretty much works the same for the rest of the objects until you get to the ‘ioff’ vendor. Since this is a JSON object within a JSON object, we have to break it down in separate collections. The separate collections work just as the same as the previous other than having to take the JSONPath a little further into overall object:



Like before, we do a search for the ‘ioff’ object. Within that object, we want to set the collection to the “Data” of the “navigator” object. The program will use the “ElementAt” method to attach all the returns to global variables that will be used to insert the data into the DB. For more information about JSONPath, please refer to the websites <http://goessner.net/articles/JsonPath/index.html> and <http://jsonpath.com/>. For the rest of the code the stored procedure is ran to ensure that the user has the right DeviceId and then the user login is collected in the LoginRecord as well as the vendor returns in their respected tables and columns. That’s it for the guide, I hope you find this useful.