//Prerequisites:

//open new file – updateCSV.js

//node -v .... make sure the node is installed on your system to run java script code

//npm -v

//terminal – node updateCSV.js to run the java script and update the file

//npm install papaparse –save .... in order to install Papaparse dependency – it’s a third-party library called Papaparse – it will allow you to both parse a csv file and update the package as well

//https://www.papaparse.com/ .... Papaparse Documentation

//npm init - create a package.json file using npm init .... this will create a package where you can store the dependencies/versions – just go with the default

//npm install ..... install all the dependencies required for continues delivery of the project .... the node\_module will hold/store these dependencies

//npm install newman -save .... To install newman runner dependencies. This will also update your package.json file

const fs = require('fs');

//import library for file system. It helps to read and write file

const Papa = require('papaparse');

//to update the file itself – import java script library to parse a csv file and as well to generate a csv file from java script object ..

const newman = require('newman');

//import newman runner dependency

newman.run({

//newman runner .... use newman as a library

//https://learning.postman.com/docs/running-collections/using-newman-cli/command-line-integration-with-newman/#library

collection: require('./collection.json'),

//collection ... postman collection exported from postman, saved on desktop and imported to VS Code

reporters: 'cli',

//reporter ... just showing the data

iterationData: './data.csv'

//data.cvs .... its the cvs file where the data is stored and runner will iterate thru all the lines/rows

}, (error) => {

//error handling method

if (error) {

throw error;

}

console.log('Collection run complete.');

//if no error, print collection run complete!

}).on('beforeDone', (error, data) => {

//beforeDone .... need to capture the event that will trigger before completing the run. It is because to assert whether the API request is successful or failure

//.on or .once methods .... newman.run returns an event emitter instance where you can catch listeners using the .on or .once methods

//callback function (error, data) ... every event handler (callback function) is called with two arguments:

//error .... is an Error instance, if an error occured, otherwise it's always null

//data .... the second argument and its an object having event-specific properties

if (error) {

throw error;

}

const findFailures = (a, c) => {

//define findFailures as variable

return a && (c.error === null || c.error === undefined);

//return accumulator and the current value from debugger (variables.data.summary.run.executions.cursor.assertions.error) and the error sometimes can be null or undefined

//so if the previous result was true and there is no error, then true (a) and true (c.error === null || c.error === undefined) = true (logic operator)

}

const result = data.summary.run.executions.reduce((a, c) => {

//run the Debugger

//After identifying the relevant object you need to access and get data, right-click on the object and Copy it as an Expression

//result is the variable and data.summary.run.executions is what we get from debugger and thats where the data is hidding (summary.run.executions - copy as expression)

//reduce .... When the Collection is running, to capture the results from the execution and mark each iteration PASSED or FAILED, you can use .reduce() function with a callback function

//reduce .... its a function to reduce the data to Passed and Failed in result and not interested in anything else

//a, c .... the callback function:

//a .... accumulator: the return value of the callback function is the accumulated result, and is provided as an argument in the next call to the callback function

//c .... current value

if (a[c.cursor.iteration] !== 'FAILED') {

//as soon something is FAILED/any of the assertions, leave it Failed

a[c.cursor.iteration] = c.assertions.reduce(findFailures, true) ? 'PASSED' : 'FAILED';

//go thru each assertion in the test section of the collection and if any of the asssertion failed, throw error/failed even if the statusCode is 200

//in order to get index ... using debugger (variables.data.summary.run.executions.cursor.iteration) ... copy as expression

//c.cursor.iteration .... c is the current value/ current index from the iteration is added

//findFailures: in order to add Passed and Failed conditions

}

return a;

}, []);

//accumulator is an array, defined here as an initial value

//the test result will be an array so we define it as an array

updateCsvFile(result);

//use debugger to see the error = undefined message (variables.data.summary.run.error:null)

//use debugger to see the error = null messages (variables.data.summary.run.executions.cursor.assertions.error:undefined)

//update the data.updated.csv file with test results

});

function updateCsvFile(result) {

//in order to put everything in java script function and update the file with result after the run

fs.readFile('./data.csv', 'utf8', (error, data) => {

//fs.readFile .... to read from csv file

//./data .... reference to local file called data.csv

//utf8 .... this is the encoding to handle special characters

//callback function (//error: to handle the error ....AND.... data: handling/containing the data itself from csv file) .... Unicode Transformation Format - 8 bits

if (error) {

//error handling in case the file is not found or anything wrong with it

throw error;

}

const jsonData = Papa.parse(data, { header: true });

//Papa.parse .... Papa Parse to parse CSV String

//data .... is the csvString in the csv file

//header .... to handle CSV files having headers, add header: true inside the Parser. Then it will avoid taking header values as actual data

jsonData.data.map((item, index) => item.result = result[index]);

//mapping the data by index wise .... go thru each line/row and run all the assertions and results back in same order

//to update every item in the file based on the response data from Newman, we use .map() function.

//for every line/item in this array we use map function. We get the item iself and for every line in our file, we want to update the result and from the result you get the correspondance key [index]

const updatedCsv = Papa.unparse(jsonData.data);

//generate a new data file with results called "updatedCsv" .... the content we want to use

//Papa.unparse .... don't read from this new file but just update it with updated results each time the runner is triggered

console.log(updatedCsv); //to see the Passed/Failed results in terminal as well

fs.writeFile('./data-updated.csv', updatedCsv, (error) => {

//write the Passed/Failed results back into the updatedCsv file

//updatedCsv .... is the content we want to use

//error .... callback object

if (error) {

//error handling

throw error;

}

console.log('Updated CSV file.'); //if no error, show the print the results in console

});

});

}