Class ID: SD225096

Extending Your Fusion Lifecycle Development Environment with VS Code (Part 2)

John Denner | Oldcastle Infrastructure| Systems Administrator (PLM, PDM, CAD, LMS)

Last Revised: 2018-10-29 Please see link below for possible newer versions

Link to updated handout versions and all example code

Learning Objectives

- Learn how to fine-tune your environment from part 1
- Learn how to build and use code snippets
- Learn how to utilize Node.js require and export functions
- Learn how to use a source control system within Visual Studio Code

Description

Tired of typing the same pattern of code or just struggling to remember the syntax? Well, let's build some snippets! Wondering how to include other files into your code like import in Fusion Lifecycle software? We'll look at how to use Node.js require and export to further modularize our code. We'll cover how to use source control from within Visual Studio Code. We'll also dive deeper into the areas from Part 1 that we glossed over, such as configuration files.

Speaker Bio

John is a full-time systems administrator at Oldcastle Infrastructure. His career path wasn't very straight forward. He was an air traffic controller and airfield manager in the US Air Force, a roofer, steel detailer with Tekla Structures, AutoCAD drafter, Inventor Certified Professional, learning and development developer, and now an engineering systems administrator with a focus on Fusion Lifecycle. John has enjoyed programming off-and-on as a hobby since the early 90s. He's been blessed with having been able to travel all over the world in the US Air Force and in 2003 served in Iraq. John currently lives in Fresno, CA with his wife and three kids.

Table of Contents

Disclaimer

Software and software editors are evolving on a seemingly daily basis. I feel it would be bad form in attempting to give detailed instructions on the usage of the software when that documentation is already maintained by the Visual Studio Code (VSC) developers. What this handout *will* attempt to do is show how to use the tools inside of VSC to work more efficiently in creating and maintaining your Fusion Lifecycle codebase.

Whenever appropriate, I will give details on how I implemented a particular feature that took me a while to figure out, required multiple searches across multiple sites, and a lot of trial and error.

I am not a professional developer but I discovered some ways that I could code cleaner, safer, and more efficiently for Fusion Lifecycle and I want to pass that on to you.

John Denner

Terminology

Here are some basic terms and acronyms I'll be using through this handout.

FLC – Autodesk Fusion Lifecycle

IDE – The coding environment (Integrated Development Environment)

VSC – Microsoft Visual Studio Code

Top level object – Global objects within Fusion Lifecycle

Top level function – Global functions within Fusion Lifecycle

Module – Self-contained object hiding its state and implementation

Linting – The concept of cleaning code to conform to best practices. Removing the fluff.

Workspace – Depending on context it could be th……..

JSON – JavaScript Object Notation, A data structure based upon JavaScript objects.

Root – The very base folder of your workspace in VS Code

Fake(ed) – Data being substituted for actual values within Fusion Lifecycle

Mock(ed) – Typically top-level functions and methods to mimic the functionality in FLC.

Intellisense – Functionality that provides code completion and inline documentation of code.

Finetune

In part 1 of this series we introduced a handful of configuration files used in various areas of the software. In this section we'll dig a little deeper into using these files.

launch.json

This file configures how running our code should operate inside our current workspace. Think of it as a set of recipes for launching the debug session.

```
1  {
2      "configurations": [
3          {
4              "type": "node",
5              "request": "launch",
6              "name": "Run Active File",
7              "program": "${file}",
8              "skipFiles": [
9                  "${workspaceRoot}/node_modules/**/*.js",
10                 "<node_internals>/**/*.js"
11             ]
12         },
13         {
14             "type": "node",
15             "request": "launch",
16             "name": "Run On Edit",
17             "program": "${workspaceRoot}/action/On Create.js",
18             "skipFiles": [
19                 "${workspaceRoot}/node_modules/**/*.js",
20                 "<node_internals>/**/*.js"
21             ]
22         },
23
24     ]
25 }
```

Lines 4 to 11 is our original launch config from part 1. Lines 14 to 21 is a nother config we added. The main difference is that the first config will launch the active vault and the second part will specifically launch `On Create.js`

Multiple jsconfig.json

This file configures how the folder structure in the current workspace should work. In part 1 we learned how important this file is to our workspace. But something we didn't get into was having multiple

User and Workspace Settings

In part 1 we took a superficial look into what these two sets of settings do. Let's look a little more into how we can leverage this.

Folder vs Workspace

I've been using the term workspace to describe the current root folder we are working in. In VSC the term workspace has a slightly different bit of functionality. Opening a folder in VSC makes that folder the root and you only have access to that folder and it's subdirectories. When we use a workspace, we can include folders outside of the current root. I found two good benefits for myself when using a workspace.

1. Having a common library to share across other root folders
2. Managing the other root folders in their own repositories

Using a common tenant library

eslint.rc

This file configures how the rules in the extension ESLint should behave.

Snippets

Code snippets allow us to quickly insert predefined chunks of code into our scripts. This can be an enormous time saver. VSC comes with some predefined snippets specific to different languages.

Below is a very basic look into snippets please read the VS Code help file on Snippets for the full details.

Using and Behavior

Snippets are typically invoked with Intellisense by using a prefix. As we cycle through the options that Intellisense provides us, we are shown what the snippet will be inserting. Snippets don't necessarily need to be blocks of code, they can also be standard comment blocks.

Structure

Snippets are stored in JSON files and have a few basic components:

- Name
- Intellisense prefix (how we can trigger it)
- The body (the part to be inserted)
- And a description that is shown in the Intellisense

Here is an example of a standard for loop that steps over each element in an array.

```
{
    "For_Loop": {
        "prefix": "for",
        "body": [
          "for (const ${2:element} of ${1:array}) {",
          "\t$0",
          "}"
        ],
        "description": "For Loop"
    }
}
```

How to create

For me, the easiest way for me to create a snippet is from a bit of existing code I have. Then from there I can more easily see where the tabstops, placeholders, variables and any indentation should go.

Idea for demo: create snippet that will be like mad libs. Ask people at the beginning for words. Have the snippet use methods that type at the same time, jump around, use dropdown options, etc.

I am a **pilot.** I'm really good at **eating** on the worst possible days, like **TODAYS DAY NAME**. Being a **pilot** means that my **feet** are really smelly. Thank you for attending AU**2018**!
~ Your **pilot**

Node Require and Exports

In part 1 of this series we needed to include all of the requisite functions and data at the top of each file. The problem with this is that the code can get long and out of sync with each other. In a way, the part 1 method is easier but much harder to maintain. In this section, we'll look at a slightly more confusing process but enormously more maintainable.

DRY Code

In software development, don't repeat yourself (DRY) code is an idea around creating reusable code in an effort to reduce repetitive code and more errors. In FLC we have a concept of library functions that can be "imported". This is immensely powerful functionality and allows us to modularize our code. Under the hood in VSC we are using Node.js. Therefore, to import external code files into our scripts we'll need to learn the methods that allow importing files. But first we'll need to export.

Exporting

In Node.js we must explicitly export the chunks of code we want to make available to other files. This is done through a special Node.js object `module.exports` and the require `require`. At first it may seem that having to explicitly export is frustrating but exporting allows us the flexibility to only expose specific parts of code. Some library code has functions that would never be used anywhere except to support a main function. Using `module.exports` allows us to do this.

There are a few ways to implement this export/require functionality. We are obviously not going to go into the details of how this works since there is enough authoritative information available online. The following example shows what I believe is the easiest way to code locally but also be able to cut and paste back into FLC with minimal work.

In example 1 below, we define our objects and functions as usual. But at the bottom of the file we add them as properties and methods `module.exports`. These objects and functions are now available to any JavaScript file in our project. This functionality works similarly to importing in FLC. lightly different in the fact that everything will be made accessible unless you employ closures.

```
// Example 1: Export at the end of the file.
function println(output) {
    console.log(output);
}
var Logger = {
    log: function (output) {
        console.log(output);
    }
};
module.exports.getUserName = println;
module.exports.getUserName = Logger;
```

Another method is simply wrap objects and functions within `module.exports`. This is completely valid. However, for scripts destined for FLC, this requires a good deal of rework in order to simply copy and paste back into FLC. You would need to reformat almost In the example below I'm mocking some FLC functions and exposing them to other files by using module exports. For the rest of the examples we'll be using the first example.

```
// Example 2: Export at the end of the file.
module.exports = {
    println: function (output) {
        console.log(output);
    },
    Logger: {
        log: function (output) {
            console.log(output);
        }
    },
};
```

If the code is intended to be moved into FLC you will need to, at a minimum, comment ot or remove the above highlighted lines.

Importing

Now that we've identified code that should be exposed to other files, it's time to include/import those files into our typical scripts. Like exporting, we'll need to add some additional lines of code that will not be included in our final code to be pasted into FLC.

While `module.exports` says what to expose to other files, `require()` identifies which files should be included on our other files.

Importing is somewhat easier than exporting. To import we use the require function. In this example we'll be importing two functions that I created to mimic the println() and Logger.log functionality. Both functions are contained in a file called topLevelFunctions.js so we need to be more specific when assigning the variables. To do this we simply need to reference the function.

```
var println = require('../../flcFunctions/topLevelFunctions.js').println;
var Logger = require('../../flcFunctions/topLevelFunctions.js').Logger;

println('Hello ');
Logger.log('world!');

> Hello
> world!
```

When requiring a non-node module, we have to use `./` or `../` for the system to find the file. We can also omit the file extension .js or .json. Here are a few more require() examples:

```
var example1 = require('./functions); // loads functions.js
var example2 = require('./functions.js'); // loads functions.js
var example3 = require('./data'); // loads data.json
var example4 = require('./data.json'); // loads data.json
var example5 = require('./library.js').function341;
var example6 = require('./library.js').object42;
var example7 = require('../root.js');
var example8 = require('../../deepData.json').function1;
9



var println = require('../../flcFunctions/topLevelFunctions.js').println;
var Logger = require('../../flcFunctions/topLevelFunctions.js').Logger;

println('Hello ');
Logger.log('world!);

> Hello
> world!
```

Some issues I've encountered is the relative pathing structure. Blah blah blah

Source Code Control

If you aren't familiar yet with source code control (SCC) you should familiarize yourself ASAP. For the purposes of this handout we'll be talking specifically about the "git" system for SCC.

> **Note:** Git is simply a word the creator used to name the software. In British slang, git is a insult used to describe incompetent, foolish, or annoying person. Linus Torvalds, the creator of git, named the software after himself.

In FLC we can manage versions of items and utilize working versions so as not to inadvertently change the released item. This is pretty much exactly the goal behind SCC. There are entire books written just on this subject. I'll only touch briefly on how as FLC developers can take advantage of this system. But as always… please read VS Code help on source control..

Integrated

VSC has a git built right in. This is fantastic for FLC developers! This helps us folks that don't think they need or cannot use services like Azure DevOps Services or GitHub (both owned by Microsoft) to manage code in the cloud.

Configuration

There are two primary files and a special folder used in all git systems. These are the .gitattributes file, .gitignore file and .git folder. When you initialize your repository these files and folder will be created.

.gitattributes (file)

This file tells the git system how to handle certain file types. I personally have never needed to modify this file for any of my FLC tenants.

.gitignore (file)

This file is very important in excluding files that should not be versioned control. Typically I need to modify this file so as to ignore other configuration files or entire directories.

.git (folder)

This folder holds all the version information of your repository from the beginning of time.

GitLens Extension

If using SCC I would strongly suggest installing the extension called GitLens. In the git system all changes are tracked with notes telling who, when, and why a change was made. In a nutshell, GitLens will provide you with in-editor change information about the specific line of code you are on.

Repositories

Local Git

GitHub

Azure DevOps Services

Resources

1. Private Message John Denner (Autodesk Forums)
2. FLC Objects Project on GitHub
3. Official VS Code Help Files
4. Official ESLint Documentation
5. Official JSDoc Documentation (This can be a little hard to read)
6. JSON Validator
7. JSON Formatter
8. Postman
9. RegEx Railroad Diagrams
10. RegEx builder and Tester
11. MDN
12. Stack Overflow
13. FLC Forums