

Class ID: SD225085

Setup a Simple Local Fusion Lifecycle Development Environment with VS Code (Part 1)

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

Last Revised: 2018-11-03 Please see link below for possible newer versions



[Link to updated handout versions and all example code](#)

Learning Objectives

- Learn how to set up a simple local development project
- Learn how to write code locally for easy copy and paste into Fusion Lifecycle
- Learn how to utilize linting to keep code consistent
- Learn how to use simple debugging procedures

Description

Do you feel like you spend half your time in the online scripting reference? Do you have `println()` on almost every other line when you debug? Do you wish you could easily search all your scripts in a single step? Discover how you can develop, test, and maintain scripts on your local machine for your tenant. We will be utilizing a small but very powerful set of tools that let us mock and document objects, properties, methods, functions, and so on. We will construct the Fusion Lifecycle Item Object with IntelliSense to speed up the development process. We will dabble with code linting to help maintain clean and consistent code across all files. We will utilize powerful search-and-replace features across multiple files for when you need to rename that field ID but can't remember where you used them.

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

Terminology	3
Setup a local development project	4
Folder structure	4
Settings.....	4
Extensions.....	5
Getting the scripts in.....	6
Write code locally	7
Item object.....	7
JSDoc Commenting	10
Peek Definition and Go To Definition	13
Searching.....	14
Code Linting	16
Testing and Debugging	20
Setup.....	20
Using faked and mocked objects.....	21
Time to launch in 3... 2... 1... F5.....	24
Breakpoints	26
What's next.....	29
Resources	30

Terminology

Here are some basic terms and acronyms I'll be using throughout 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.

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.

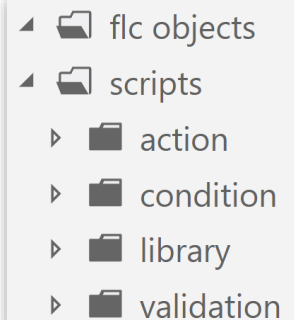
Setup a local development project

Before anything we need a good folder structure and setup the workspace in VS Code so that we work as efficiently as we can. Don't worry though. We can rearrange our folders but as you get more advanced, moving folders can break file links.

In this section, we'll create a workspace by building a folder structure, adjust settings as necessary, look at some extensions that I feel are must-haves, and finally we'll get some code populated in the folders.

Folder structure

There isn't anything too special about creating the folder structure. I like to create folders for specific types of files. I do this so I can reference the folder later for a wide variety of good reasons. Create a folder to be used as your root folder. Then create three subfolders and name them something meaningful like "scripts", "FLC objects"



Settings

Configuring your work environment for how you work is essential in any area of work. VS Code has an amazing amount of configurability. The amount of tweaking you can do can be a little overwhelming. But we only need to adjust a few settings to get this up and running. The settings are stored in a JSON file. There are three tiers of settings workspace, user, and system.

jsconfig.json

This configuration file is small but does a whole lot. Without this file setup correctly and in our root folder, VS Code doesn't know what files are supposed to work together. Here is the basic config file for our needs.

```
1 {  
2   "compilerOptions": {  
3     "target": "ES5"  
4   },  
5   "include": [  
6     "scripts/**/*.*",  
7     "object stubs/**/*.*"br/>8   ]  
9 }
```

Workspace settings

Workspace settings take precedence of user and system settings. This allows you to setup different workspaces for different projects. Such Python, C++, etc. Workspace settings should only be used to customize a specific workspace and you don't want those changes to effect other workspaces.

User Settings

User settings is the area where you make changes that will effect all the workspaces but not those workspaces that have settings that override. This is where you will do most, if not all, your customizations.

Default User Settings

The final set of settings are the settings at the core level. These cannot be changed. VS Code will add the change you want to the user or workspace settings.

Extensions

I'll be honest and say that extensions are not really a requirement. However, an environment configured for the way we work and how we work is a very important part of what we do.. There are thousands of extensions and I've tried many of them and found this meager list of extensions below to be my personal essentials. Here they are in order of precedence for me.

ESLint

From what I've experienced ESLint is pretty much the standard for linting JavaScript. We'll discuss linting a little more in the section called Linting.

Themes

Themes may seem frivolous but for me they can help me see the code easier. And being able to see certain parts of the code quickly is pretty helpful in my book.

GitLens

GitLens is a little outside the scope of this session but is amazing when you begin to use a source code control system. VS Code has git built right in to VS Code! I'll discuss this more in Part 2 of these sessions.

Icons

Themes change colors but they don't change icons in the browser window. So what? Well when you start to get a lot of files you can find what you need faster simply looking for the icon.

Extensions can be enabled and disabled at global level or at a workspace level. This is very helpful when you have multiple projects using different languages and you want to keep your project uncluttered with unnecessary extensions.

Getting the scripts in

So up until now we haven't done anything to do with coding but everything we did before really doesn't need to be done again. Now depending on how many scripts you have and what method you use, this could be easy or a real project. I'm going to go over a couple of methods that I know about and have done. There's probably other ways but the end goal is to get the code in. So here are the methods that I've used.

Copy and Paste

If you have a lot of scripts, this would be a nightmare. Create the JavaScript file on your machine and open it, switch over to your tenant, copy the code, switch back to the open file, and... paste. Rinse and repeat.

Via v3 API

If you know how to use the FLC API you could use that. It's a little faster than the previous method but can still be grueling.

FLC Toolkit (restricted access)

The FLC toolkit is what I currently use. It takes a bit of setup but in that toolkit is a script archival tool. What this tool does is gets every script you have in your tenant, creates and names the files, and organizes them by the type of script they are: Validation, Condition, Action, and Library. It's amazing.

Write code locally

If you skipped right to this section, then we are kindred spirits my friend. This is the good part. The next three sections will be a little more difficult than the previous sections. That is why this is an intermediate course. We're going to get into some tricky stuff and I'm going to stop rambling and get on with it.

So why write code locally instead of inside FLC? To start off, the FLC script environment is not bad. But for the way I work it just wasn't working for me. Here are my personal pros and cons of coding in VS Code.

Pros	Cons
Intellisense	Requires much more initial setup
Better debugging	No Access to the item record
Code Refactoring	Takes setup to access library functions
Searching across multiple files	No Mail object
JSDoc Commenting	No getPrintview()
Highly configurable	Getting the code out and back in

What did it for me was that most of those cons listed can be worked around. And that's what this part is all about.

So, we've handled most of the first con of having to setup VS Code for how we work so we'll move on to the second con, no access to the item record. Not having access to the actual items is a pretty big deal... sometimes. What I did to right this wrong was to build/mock my own item object. In JavaScript, other than [primitive types](#), everything is an object and is structured pretty much the same way.

Item object

The `item` object is one of the most important and one of the most complex FLC objects that we use in almost all our scripts...

When building our mock version of the item object we don't want to use anything that is specific to a workspace. We want to build the object as it is defined in the scripting reference.

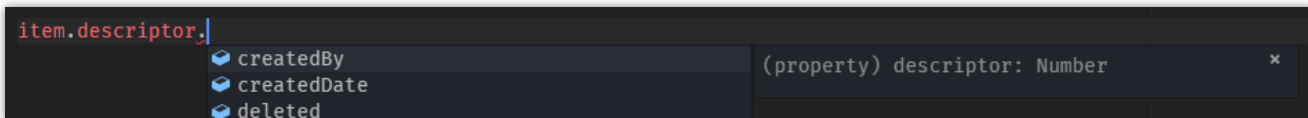
Properties:

```
1 | var item = {};
```

So now let's build upon this and add in some of the item descriptor properties while referencing the online scripting reference.

```
1 | var item = {
2 |     descriptor: {
3 |         createdBy: new String();
4 |     },
5 | };
```

What we've done here is created the descriptor property under item and then the createdBy property under that. Why have we done this? So that now when coding, VS Code will know how the item object is structured and provide us with Intellisense.



And for each additional property under descriptor, we just keep repeating that format.

```
1 | var item = {
2 |     descriptor: {
3 |         createdBy: new String(),
4 |         createdDate: new Date(),
5 |         deleted: new Boolean(),
6 |         // and on and on...
7 |     }
8 | };
```

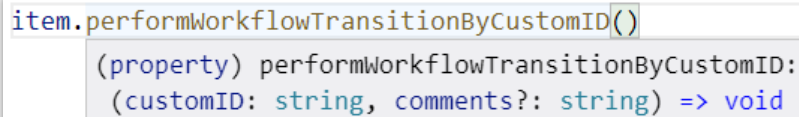
You may have noticed that I set each of the properties as new object types. You'll see why in a little bit.

Methods

Methods are pretty much the same as properties. Take notice of how the method is empty. This is because the mock objects are just to help us the same way as referencing the scripting reference.

```
performWorkflowTransitionByCustomID: function (customID: string, [comments]: string) {}
```


What we've accomplished here was to create the basic definition of the method. One thing you may have not noticed seen before, because it's not valid JavaScript, is that in the function arguments we've defined the data type for each of those arguments and identified the comments argument as optional. This is actually TypeScript syntax but we can use it to help in creating some very useful Intellisense. The following screenshot will show you why we've done this.



```
item.performWorkflowTransitionByCustomID()  
(property) performWorkflowTransitionByCustomID:  
(customID: string, comments?: string) => void
```

Now this function's Intellisense tells us that it requires the customID as a string, a comments parameter is optional but should be a string, and returns nothing (void).

A more complex example

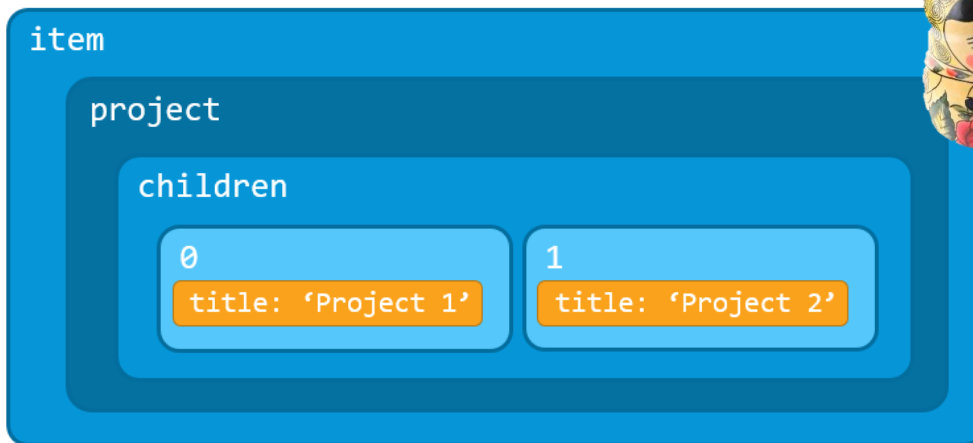
Let's turn it up a notch and create the item.project property. The value of this property is an object. And inside that object is the children property which is an array of objects. And within those objects are more properties and methods. Let's look at an example of a project child's title. We would access this title property like so:

```
1 | item.project.children[0].title
```

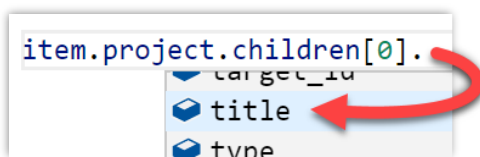
And the definition looks like this.

```
1 | item = {  
2 |     project: {  
3 |         children: [  
4 |             {  
5 |                 title: 'Project 1',  
6 |             },  
7 |             {  
8 |                 title: 'Project 2',  
9 |             }  
10 |         ],  
11 |     },  
12 | };
```

Here's a more graphical depiction of the above block of code. It's simply a series of nested objects.



And now our Intellisense will provide us with



We no longer have to struggle to remember the objects and their nested objects and methods.

Sometimes we need a little more help. Maybe some information on what a method does or maybe some examples. Well luckily a lot of smart people thought of this too and implemented.

A full item object can be found at <https://github.com/dennerj/au2018>

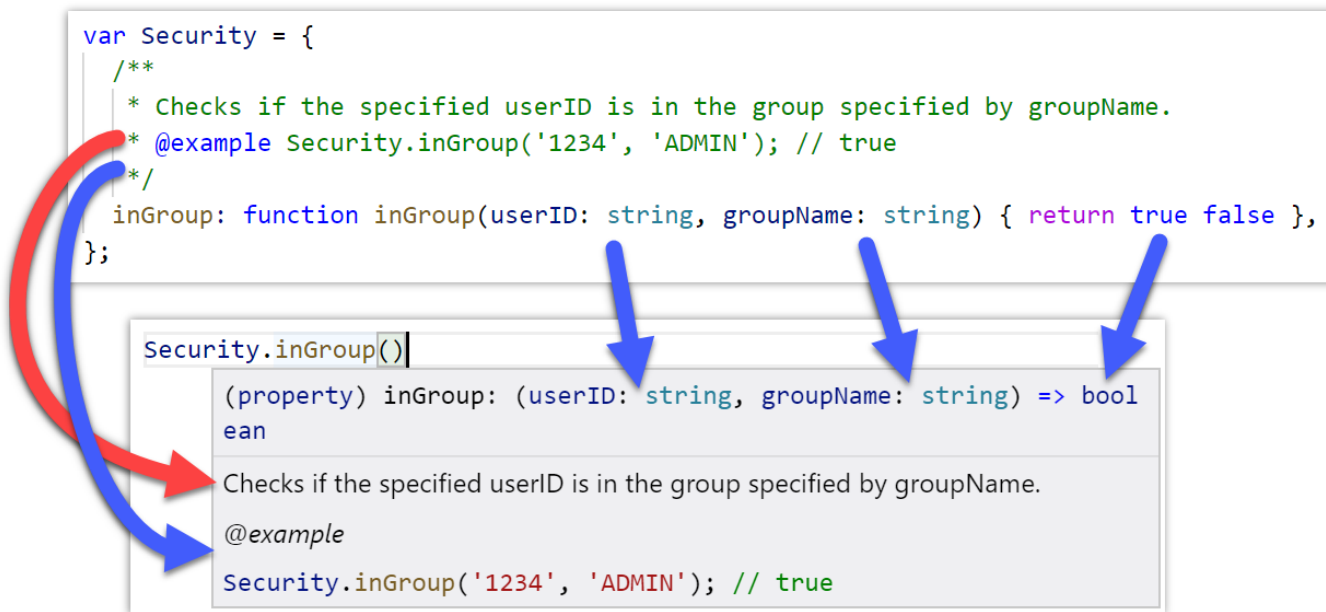
JSDoc Commenting

JSDoc is a markup language used to annotate JavaScript source code files. Using comments containing JSDoc, programmers can add documentation describing... ..the code they're creating. - [Wikipedia entry for JSDoc](#)

The FLC IDE and VS Code both recognize the JSDoc syntax. This doesn't do much in the FLC IDE. But in VS Code it allows us to document code even further and provide more detailed Intellisense text where we need it. The previous function is pretty straight forward. But what about something like `Security.inGroup()`? Knowing what we know now, let's create the Security object, add the method `inGroup` and add in some additional documentation and some examples.

```
var Security = {
  /**
   * Checks if specified userID is in the user group specified by groupName.
   * @example Security.inGroup('1234', 'ADMIN'); // true
   */
  inGroup: function (userID: string, groupName: string) { return true }
};
```

So there are a few new pieces of syntax in this example. There is the JSDoc text, an interesting line with @example on it, the function itself, a return statement of true. The JSDoc text will not only provide us with some in-document commenting but also provides us with additional Intellisense help. The @example tag tells VS Code to display the line as code. The return true will give us some helpful information on what the function/method will return, if anything.



Now we have Intellisense that tells us what the parameters are, what type of data type should be passed, what to expect in return, some additional explanation of the method, and an example. Bam!

This functionality is not restricted to our local definitions of FLC objects. It can also provide us the same information for our library functions but when using JSDoc with working functions, we need to use valid JavaScript syntax. Let's look at the ubiquitous vanilla `getUserName()` library function and see if we can't slightly improve it.

```
function getUserName() {
  var usr = Security.loadUser(userID);
  return usr.lastName + ", " + usr.firstName;
}
```

Which will provide us with...

```
getUserName()  
function getUserName(): string
```

If we relied solely on the Intellisense we wouldn't fully know if it needed a parameter(s), what data type the parameter(s) should be was returned (except for that it's a string) and what format the returned result would be in. We'd need to either consult the online scripting reference or peek at the function inside the library. This is a simple example, but as our function library grows, becomes more complex and abstract (as they should) we can start to become more and more inefficient in many ways. Let's add some JSDoc documentation and see if we can't abstract this function a little bit.

function which can do a bit more in terms of functionality and with inline help.

We've:

- Added a description
- Used @param to tell us what to pass the function
- Used @returns to tell us what is returned and in what format
- Provided examples on its usage
- Added the ability to omit the userID so as to return the current user's ID

```
/**  
 * Returns the users name given the passed userID.  
 * - If the userID is omitted then the current user's ID is used.  
 * @param {string} [lookupID] The `userID` to lookup.  
 * @returns {string} `Lastname, Firstname`  
 * @example  
 * getUserName(); // 'Dent, Arthur'  
 * getUserName('IS42'); // 'Prefect, Ford'  
 */  
function getUserName(lookupID) {  
    lookupID = (typeof lookupID === 'undefined') ? userID : lookupID;  
    var usr = Security.loadUser(lookupID);  
    return usr.lastName + ', ' + usr.firstName;  
}
```

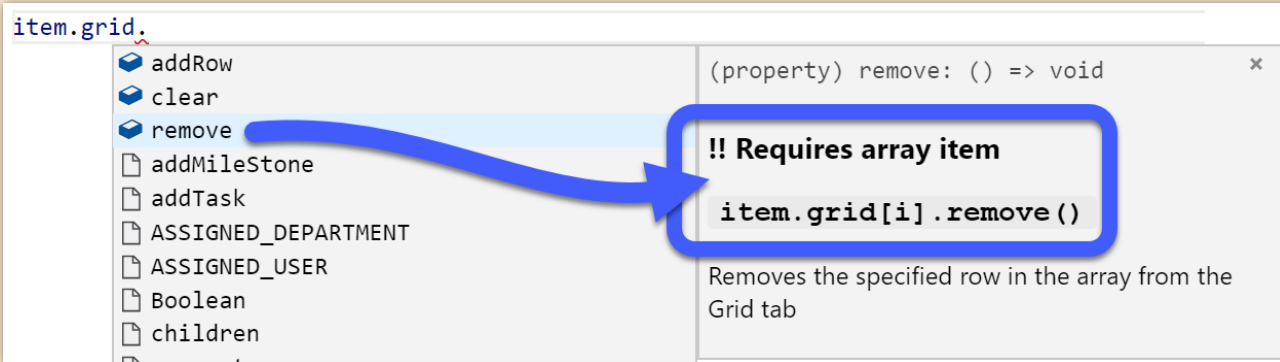
```
getUserName()  
function getUserName(lookupID?: string): string  
Returns the users name given the passed userID.  
• If the userID is omitted then the current user's ID is used.  
@param lookupID  
The `userID` to lookup.  
• @returns {string} 'Lastname, Firstname'  
• @example  
• getUserName(); // 'Dent, Arthur'  
• getUserName('IS42'); // 'Prefect, Ford' *
```

These examples were still slightly simple but JSDoc has many more keywords. Most do not help with adding information to the code since they are intended for generating HTML documentation using extensions. More information can be found on the [official JSDoc site](https://www.typescriptlang.org/docs/3.0/jsdoc-comment-syntax.html).

Note: I've found it difficult (and maybe just not possible with JavaScript) to document methods on array items.

As an example, I've not yet found a way to provide Intellisense for the `remove()` method on an array item. However, we can provide note text in the JSDoc comment that can help us later.

Notice how the Intellisense for `remove()` is on the `grid` property itself and not on an item in the array.



Peek Definition and Go To Definition

The next couple of features we'll explore are the ability to take a peek at the definition of a variable, object, method or function or open the file and jump to the location of the definition.

Peek Definition

Provides a windowed view into the file containing the definition. If the definition is outside the current file, that file will not be opened. Only a view into the file will occur without opening the file. In both cases the code is editable. In the next image you'll see how this works. I've invoked the command to peek at the definition of `getUserName()`. Notice how the definition of `getUserName` is shown immediately below the line and the filename is shown with the location of the file.

```

1
2  getUsername()
getUserName.js C:\Users\jdenner\Box\Personal Box Folder - Denner\AU 2018\Part 1\Code\scripts\library
1
2
3  /**
4   * Returns the users name given the passed userID.
5   * - If the userID is ommited then the current user's ID is used.
6   * @param {string} [lookupID] The `userID to lookup.
7   * @returns {string} `Lastname, Firstname`
8   * @example
9   * getUsername(); // 'Dent, Arthur'
10  * getUsername('IS42'); // 'Prefect, Ford'
11  */
12  function getUsername(lookupID) {
13    lookupID = (typeof lookupID === 'undefined') ? userID : lookupID;
14    var usr = Security.loadUser(lookupID);
15    return usr.lastName + ', ' + usr.firstName;
16  }

```

3 // Line 3
4 // Notice how the function definition
5 // is shown between line 2 and 3

Go To Definition

There's not much to say about this functionality other than it opens the file containing the definition and jumps to the line where the definition is located.

We've covered some areas that are pretty simple to implement once you get the hang of it. But I've discovered some areas that seem to be tricky or not possible to do locally.

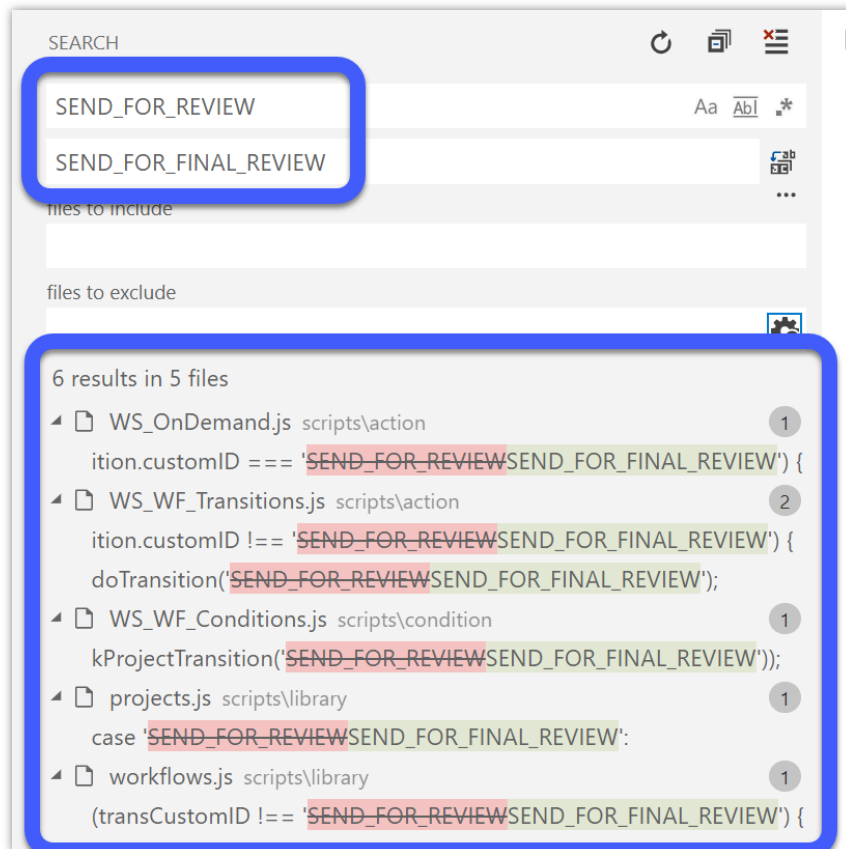
Searching

One of the more difficult tasks to perform in the FLC script editor is to locate where a specific variable, function, field name, or more typically a workflow transition's custom ID is located across multiple scripts.

As a quick example let's say that we have a workflow custom ID of SEND_FOR_REVIEW as the complexity of our workflow increases we would like to change the ID to SEND_FOR_FINAL_REVIEW. This ID is used in an Action, Condition, Validation scripts. Unfortunately, we also hardcoded this ID into some library scripts but forgot we did. It's an

annoying but not too difficult of a task to fix the ID in the main scripts. But once we test inside of FLC things break. Now we have to investigate why. So what usually happens after something like this is that the next time we need to change an ID we might just live with the ID the way it is.

A better option would be to use VS Codes powerful search and or search and replace functionality to identify all instances of the old custom ID and replace them with the new one. For the full documentation [read the VS Code help file for file searching](#).



Code Linting

Linting could probably be better described as de-linting. Which is to say removing/fixing undesirable pieces of code that do not conform to a standardized style. Since JavaScript is such a flexible language, there are numerous ways to do the same task but some methods are better than others. This could be analogous to AutoCAD. There are more than a few ways to draw the same object but only a handful are considered acceptable methods.

There are a few different code linting extensions available but I will be discussing ESLint. ESLint one of the most widely used linters for JavaScript. A linter is such an amazing tool to help you write consistent and maintainable code.

It's better to use more specific rules than to arbitrarily disable an entire rule. While this may be convenient, you may miss issues that should have been caught.

Installing ESLint is a no-brainer when doing so through the extensions panel. Configuring ESLint is a whole other story. This section will not attempt to guide you through the installation and configuration but in the additional resources section we'll look at that. For this section we'll be using the basic ESLint recommended style rules. Using this style guide coincides with the online documentation for ESLint.

Linting can help with catching potential bugs but is not a debugger.

Potential issues with your code will be displayed in the problems panel.

Some issues are not actually issues but false positives. These can be defined in the `.eslintrc` file. Full documentation on this can be found at their [rules page](#):

In the next two images, we see `getUserName` function without ESLint enabled and then after. The `getUserName` function is completely legal JavaScript but has some bad practices inside of it. After we enable ESLint, we can now see that we have some problems to look at. We have some unidentified variables and an unused function.


```

1  /**
2   * Returns the users name given the passed userID.
3   * - If the userID is omitted then the current user's ID is used.
4   * @param {string} [lookupID] The `userID` to lookup.
5   * @returns {string} `Lastname, Firstname`
6   * @example
7   * getUsername(); // 'Dent, Arthur'
8   * getUsername('IS42'); // 'Prefect, Ford'
9   */
10 function getUsername(lookupID) {
11     lookupID = (typeof lookupID === 'undefined') ? userID : lookupID;
12     usr = Security.loadUser(lookupID);
13     return usr.lastName + ', ' + usr.firstName;
14 }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Filter. Eg: text, **/*

No problems have been detected in the workspace so far.

```

1  /**
2   * Returns the users name given the passed userID.
3   * - If the userID is omitted then the current user's ID is used.
4   * @param {string} [lookupID] The `userID` to lookup.
5   * @returns {string} `Lastname, Firstname`
6   * @example
7   * getUsername(); // 'Dent, Arthur'
8   * getUsername('IS42'); // 'Prefect, Ford'
9   */
10 function getUsername(lookupID) {
11     lookupID = (typeof lookupID === 'undefined') ? userID : lookupID;
12     usr = Security.loadUser(lookupID);
13     return usr.lastName + ', ' + usr.firstName;
14 }

```

[eslint] 'userID' is not defined. (no-undef)

any

PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL Filter. Eg: text, **/*.ts, !**/n...

getUsername.js scripts\library 6

- ✖ [eslint] 'getUsername' is defined but never used. (no-unused-vars) (10, 10)
- ✖ [eslint] 'userID' is not defined. (no-undef) (11, 50)
- ✖ [eslint] 'usr' is not defined. (no-undef) (12, 3)
- ✖ [eslint] 'Security' is not defined. (no-undef) (12, 9)
- ✖ [eslint] 'usr' is not defined. (no-undef) (13, 10)
- ✖ [eslint] 'usr' is not defined. (no-undef) (13, 32)

If we now modify the ESLint configuration file to ignore issues that we do not care about or pertain to newer versions of JavaScript we can reduce our number of problems to four.

To the right is our .eslintrc configuration file. The two sections to focus on now are globals and rules.

Names listed in the globals section will not be identified as undefined in the problems panel. In the previous image, userID was identified but used but not defined. We know this is a name that should always be ignored in any file in our code.

Rules tell ESLint which style rules to ignore.

We should do our best to adhere to the defined style rules but sometimes they just do not pertain to what we are doing or are not even possible in ES5.

One of the problems is that we declared the function but never used it. This is extremely helpful in identifying unused variables or mistyped variable and function names. In this case the function is in a library so it's perfectly fine it's not being used. Notice that we didn't identify it as a global variable in the configuration file. We could have but in this case, in the next image, I used an inline comment on line 10 to disable the no-unused-vars rule on the next line.

The next three errors are identify that we are using the variable usr but we never defined it. Not defining a variable before we use it is legal in JavaScript but is very poor practice. So in the next image on line 13, I improved our code by declaring the variable usr using the keyword var.

```
1  {
2    "env": {
3      "browser": true,
4      "node": true
5    },
6    "extends": "eslint:recommended",
7    "globals": {
8      "Security": true,
9      "userID": true
10   },
11   "rules": {}
12 }
```

```
1  /**
2   * Returns the users name given the passed userID.
3   * - If the userID is omitted then the current user's ID is used.
4   * @param {string} [lookupID] The `userID` to lookup.
5   * @returns {string} `Lastname, Firstname`
6   * @example
7   * getUsername(); // 'Dent, Arthur'
8   * getUsername('IS42'); // 'Prefect, Ford'
9   */
10 // eslint-disable-next-line no-unused-vars
11 function getUsername(lookupID) {
12   lookupID = (typeof lookupID === 'undefined') ? userID : lookupID;
13   var usr = Security.loadUser(lookupID);
14   return usr.lastName + ', ' + usr.firstName;
15 }
```

ESLint can also automatically fix all problems that can be safely resolved. Such as missing spaces, replacing double-quotes, with single, fixing bad indentation, etc.

Testing and Debugging

Contrary to popular belief, our jobs within FLC is not to write lines of code. What we do is provide a complete product. Many times events happen under-the-hood that's never seen by the end user. In this section, we are going to add A LOT of additional lines to our files. You can try to eliminate where you can but understand that if you trim too much you will be sacrificing the completeness of testing.

The FLC IDE has a competent debugger. Many times, the only way to completely test and debug the code is through that interface. However, for all of the other times the VSC is hard to be beat.

Typically, testing functions is the simplest task due to the simplicity of the inputs and outputs. Testing condition, validation, workflow action and other similar scripts can be much more complex to setup. For this reason, the benefit of setting up a testing and debugging session should be evaluated.

Setup

There are a few different components to setting up a testing and debugging structure. One is the `launch.json` file that defines what is to be debugged, what's to be ignored, and how it should be done. The other components would be faking the item record data and using mock functions.

`launch.json`

Not only is this simple file required but it also defines how VS Code should handle the debugging of our files. As with anything in VSC we have a lot of control. From this one file we can define multiple launch configurations. For this handout, we'll focus on the most simple which is to run the active file. There are only three sections to worry about. The name of the launch configuration, what is the file that should be launched, and what to not include in the debugging environment. The first two are simple. You'll quickly notice an issue if you forget the excludes. If we don't exclude the core NodeJS modules then we'll be forced to step through them. The config that I've been using is shown below.

```
{
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Run Active File",
      "program": "${file}",
      "skipFiles": [
        "${workspaceRoot}/node_modules/**/*.js",
        "<node_internals>/**/*.js"
      ]
    },
  ]
}
```

Faking item data

Faking the data of an item record can be one of the more tedious tasks but critical for testing your scripts completely. The item fields and other properties relevant for your testing need to be created and they need to mirror how they work in FLC. If FLC returns an array-like object rather than an array, your faked data should be the same.

Mocking objects and functions

Mocking the FLC objects and functions so that we can use them outside of FLC is the hardest part of all this. However, once it's done, it's done. These mocked objects and functions should behave the same regardless of who is using them. Because of this, we can share these scripts and improve them as a community.

Using faked and mocked objects

In FLC we have the luxury of having item records available, top-level objects created, and library files imported. In VS Code, the pure JavaScript we are running can only work with one file at a time. In part 2 we'll discuss how to work around this. But for now because of only being able to work in one file at a time we need to include everything at the top of the script we want to test. This does not mean to include everything, just what's needed to test.

Depending on what type of script it is and how detailed your testing is we'll need things like:

- Faked item data
- Faked Security data
- Mocked returnValue
- Relevant library scripts

```
function returnValue(msgs) {
  if (typeof msgs === 'undefined') {
    console.log('returnValue argument is undefined!');
  } else if (typeof msgs === 'boolean') {
    console.log(msgs);
  } else if (!Array.isArray(msgs)) {
    console.log('returnValue argument is not an Array!');
  } else if (Array.isArray(msgs)) {
    if (msgs.length > 0) {
      msgs.forEach(element => { console.log(element); });
    } else {
      console.log('Passed all validations');
    }
  }
}

var Security = {
  inGroup: function (userID, groupName) {
    return testUsers[userID].groups.indexOf(groupName) >= 0;
  },
  inRole: function (userID, roleName) {
    return testUsers[userID].roles.indexOf(roleName) >= 0;
  },
};

var testUsers = {
  goat: {
    groups: ['Admin', 'Engineer', 'Drafter'],
    roles: ['ER', 'IR', 'QC']
  },
};

var customTransID = 'SEND_FOR_APPROVAL';
```

Additionally... to test multiple scenarios at once we would need:

- An array with each faked item
- And a for loop

```
// Array of test cases
var testCase = [
  {
    // Null field MAKE
    customTransID: 'BEGIN',
    MAKE: null,
    // Unsure if not having attachments makes the
    // property undefined or if it is an empty array.
    // Need to test in tenant.
  },
  {
    // Valid test case
    customTransID: 'BEGIN',
    MAKE: 'Ford',
    attachments: [{
      fileStatus: 'Checked IN'
    }],
  },
  {
    // Attachment is checked out
    customTransID: 'BEGIN',
    MAKE: 'Ford',
    attachments: [{
      fileStatus: 'Checked OUT'
    }],
  }
];

for (var i = 0; i < testCase.length; i++) {
  console.log('\r\n' + 'Test case ' + i);
  var messages = [];
  item = testCase[i];
  customTransID = testCase[i].customTransID;
  // Begin standard code
```

Testing multiple scenarios at once can be very helpful when testing out new functionality. But if you only need to debug a single scenario then it might be overkill.

Full examples can be found on GitHub at <https://github.com/dennerj/au2018>

Time to launch in 3... 2... 1... F5

We have everything ready to go and we feel mildly confident to launch the script. I'm only going to describe and explain the parts of debugging in VS Code that I feel are relevant to our goal. The full documentation can be found at <https://code.visualstudio.com/docs/editor/debugging>.

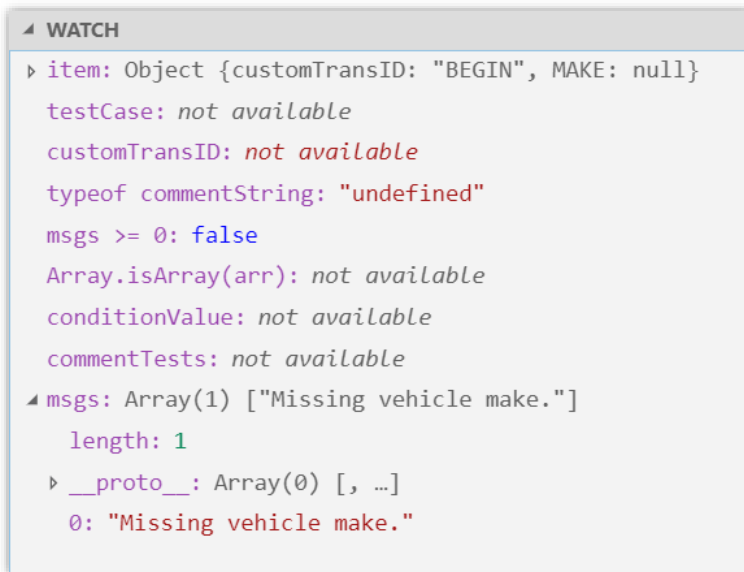
Variables Panel

The variables panel is similar to the Global and Local panels in FLC but can show many more variables than you need.



Watch panel

Similar to the Expressions panel in FLC but does not reset after each run of the debug session.

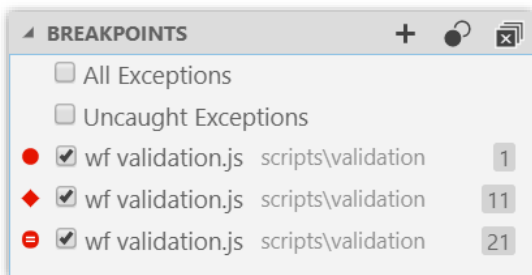


```

WATCH
  item: Object {customTransID: "BEGIN", MAKE: null}
    testCase: not available
    customTransID: not available
    typeof commentString: "undefined"
    msgs >= 0: false
    Array.isArray(arr): not available
    conditionValue: not available
    commentTests: not available
  msgs: Array(1) ["Missing vehicle make."]
    length: 1
  __proto__: Array(0) [, ...]
    0: "Missing vehicle make."
  
```

Breakpoints panel

This panel lists existing breakpoints, allows us to create more, and enable or disable specific breakpoints.



Debug console

This is where any run-time errors will display. Run-time errors will almost always show you the line and the location on the line where the error was encountered. Text from our console.log, mocked println/Logger.log, or from Logpoints will also display here. I'll explain Logpoints more in the [Breakpoints](#) section.

Launching the script

The debug session without any breakpoints in VS Code is the same as when we click "Test" in FLC. In FLC if we want to step through our code we choose "Debug" and then apply breakpoints if we want to jump ahead. In VS Code we set our breakpoints first and then start the script. But in VS Code our breakpoints are on steroids.

Breakpoints

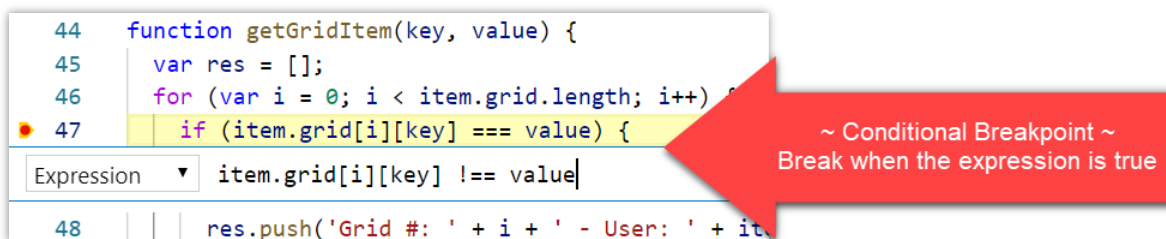
Breakpoints are a critical tool in tracking down bugs or unexpected results in our code. Within the FLC IDE we can insert breakpoints only after running the code in debug mode. This is fine but we have to redefine the breakpoint on every time. Breakpoints are so critical that in VS Code we are given not just one, but a handful of different types of breakpoints.

Expression (conditional)

Expression breakpoints work by breaking on the line only when the expression resolves to true. Think of it as putting in an additional if statement for the sole purpose of checking an issue while debugging.

Some examples:

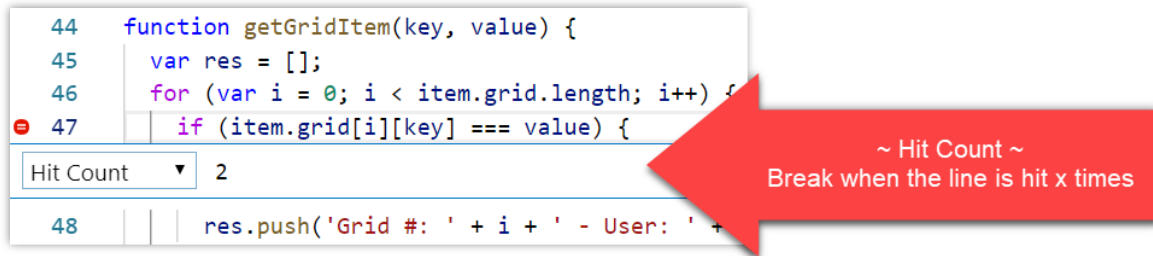
- When a field has a certain value
- When a variable becomes empty
- When a variable goes below zero
- When a for loop has ran x amount of times. The Hit Count breakpoint is better suited for this but an Expression breakpoint also works.



The image above is an example of an Expression breakpoint. Our code says to do the code in our if block when the grid key equals the variable value. Our Expression breakpoint says to halt execution whenever they don't equal each other.

Hit Count

The Hit Count breakpoint is simple, break whenever the line is hit x number of times. This could be helpful in determine if a certain block of code is being ran more times than we are expecting. This can be very helpful when trying to optimize our code. Or when we want to start stepping through a loop only after it has looped 500 times.



Logpoints

Logpoints are a special breed in the breakpoint family. In fact they don't really seem to be breakpoints at all in the sense that they don't halt the execution of code. What logpoints do is send text to the debug console whenever the line is hit. Essentially what this helps to do is remove any need to insert `console.log` or `println` lines into our code.

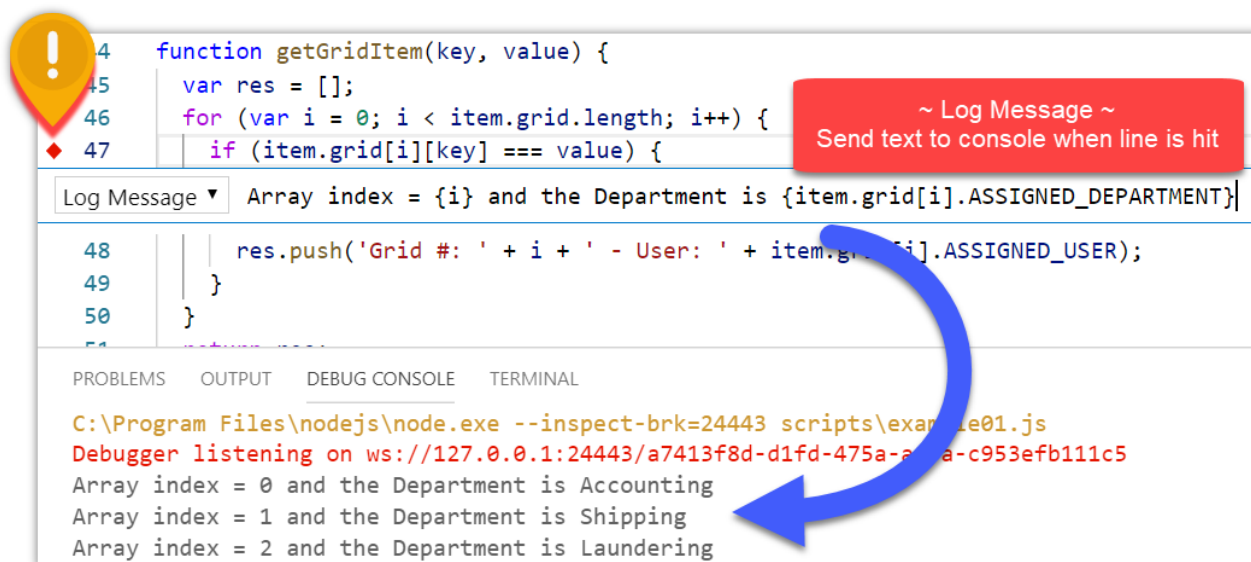
Special note: Similar to normal breakpoints the logpoint triggers BEFORE the line executes. Meaning that if you want to log the result of a variable reassignment, you'll need to put the logpoint AFTER the variable declaration line.

If we wanted to log the text "The user Denner, John cannot transition to CLOSE" It might look like this using `println`:

```
println('The user ' + nameLastFirst + ' cannot transition to ' + customTransID);
```

If using Logpoints then we can eliminate an entire line of code and write a simpler line in the logpoint like:

```
The user {nameLastFirst} cannot transition to {customTransID}
```



This functionality within VS Code can help by:

- Removing the need to insert additional lines of code just for debugging purposes
- Simpler statements for creating the string
- Can be combined with other types of breakpoints

I highly encourage you to [read the official VS Code debugging documentation](#).

What's next

We've put a dent in understanding what VSC can help us do outside of FLC but there is still a lot we can do to create truly robust and easy to test code.

In part 2 we'll discover how we can include external script files

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. RegExp builder and Tester
11. MDN
12. Stack Overflow
13. FLC Forums