## Build Process:

### SASS/CSS

For SASS and CSS, we would use node-sass which would be run from an NPM task:

npm run devsass

(for development, would include a 'watch' to continuously recompile when edits are saved)

node-sass would follow the imports of the main .scss file and compile all the partials down to app.css, or minified app.min.css

This process would be different from Typsecript/Javascript in that Webpack, as we would use for those files appears to handle CSS in a way different than a site-wide single stylesheet as we have now, 'bundled' within a deployment package that Webpack uses for everthing else. This process prescribed here would generate CSS in about the same way we currently do to avoid complications in introducing anything new that possibly drastically changes our current environment.

### Javascript libraries – “frameworks” vs Typescript for our own written code

For Typescript and Javascript, a process using Webpack is being developed. The process is two-part in that we need to keep existing Javascript libraries and plugins together as we do currently to separate them from our own written code. In the end we would generate two JS files: frameworks.min.js, for our frameworks - Foundation, JQuery, Slick, etc, and app.min.js, for our own house-written code (written in Typescript).

A sample of our module.exports object in webpack.config.js

```

module.exports = {

entry: {

'app.min': './src/typescript/index.ts',

'frameworks.min': './frameworks/frameworks-main.js'

},

output: {

path: path.resolve(\_\_dirname, './js'),

filename: '[name].js'

},

resolve: {

extensions: ['.ts','.tsx', '.js']

},

plugins: [

new webpack.ProvidePlugin({

$: "jquery",

jQuery: "jquery",

"window.jQuery": "jquery"

}),

new UglifyESPlugin({

minimize: true,

})

],

module: {

loaders: [

{

test: /\.tsx?$/,

loader: 'awesome-typescript-loader',

exclude: [/node\_modules/, /frameworks/],

query: {

declaration: false

}

}

]

}

};

```

We would have two 'entry points' or main code modules.

### Typescript

For our own in-house custom code, utilizing Typescript we can have type-control over defined variables, utilize ES6-type conventions such as Classes, Interfaces, imported/exported modules, etc. The most prevalent so far would be modules. These modules would allow us to break up our code into easier-to-maintain pieces of code that perhaps contain a few related functions, class declarations or the like that are all imported into parent modules, which are in turn imported/made available in the main Typescript file. Our "main" typescript file would contain all of our top-level imports, "global variables" (where needed) and our ```$(document).ready()``` function, which serves as our "main" code block, and runs when the page is loaded. When Webpack is run and the task to compile Typescript performs, it will follow the tree of import statements declared in each file to determine dependencies and render it all down to our single JS file, app.min.js, which like frameworks.min.js we include in our page template with a <script> tag.



### Issues

* Determining how to break up some modules and properly modularize them in a way that the Typescript compiler will be satisfied (as well as have the generated Javascript code fully functional) could be an ongoing process as new functionality is added.
* By design of the Webpack “bundle” system, variables defined within the code may not be available for testing inside the browser console, for example, except by exposing them to the Window object or using a console.log/console.info/table/etc. For instance, if you wanted to check the JQuery version or the current contents of the “shu” global object from the console, you may get a JS error returned stating that the function or variable is not defined. This seems to be a design of Webpack to prevent “leaking” to outside scopes.

### Advantages

* Our in-house generated code will be in smaller modules that themselves have dependent modules that would allow for better readability, maintenance and reusability.
* Our code would no longer be regular Javascript that partially uses some Typescript conventions (as TS is a superset of JS, so valid JS is valid TS but not the other way) run through a Typescript compiler. This would help to clean up the code and better follow its path of execution.
* Items such as GPA Calculator and Campus Directory Unit Filter could be made into standalone applications through frameworks such as React of Angular since they appear on only limited numbers of pages (GPA Calculator currently only one page in SHMS, Unit Filter only on Campus Directory page), this would decrease the complexity of our Typescript code base and reduce bloat in app.js, the overall site Javascript included on every page.

As far as relation to Foundation itself, the difference should be minimal as we still only need to include Jquery first then foundation.js. There would be an upgrade to at least JQuery 3 which would need to be tested to see what, if anything breaks. Items that come to mind that *may* have issues because of the version of JQuery they usually come with: Slick. We also shouldn’t need a dependency on Modernizr with this new version of Foundation, though it is included with the download if we were to need it.

### Renderhandlers/Custom Scripts/Templates

The only issues seen so far with custom scripts and render handlers would be if anything we are outputting to the page as far as Foundation widgets require an update to the HTML structure they’re rendered with. If there are JS changes they should be reflected in foundation.js for the new version.

For pure Foundation updates, the only changes to JS(TS) and ColdFusion code would be to support HTML changes (if any) needed for the rendered output, as seen upon looking at the new Foundation, there shouldn’t be any changes to those areas unless we decide to rewrite something to enhance it (a renderhandler or custom script with HTML or “Foundation-ized” output). The same would go for the page templates. The logic written in ColdFusion shouldn’t change, Javascript should only change for existing items to reflect any change in structure for a given widget (i.e., a certain widget’s structure perhaps used a <dl>/<dt>/<dd> structure but now recommends using a <ul>/<li>, or series of <div>s, etc. These would then be updated to reflect the new structure)\*

### Timeline

(?) 2 weeks to convert our existing Javascript house-written code to “real” Typescript, modularize, test, etc

2 weeks to investigate Renderhandlers, Custom Scripts and Page Templates for output markup changes

HTML/CSS updates\*

1 week to evaluate irrelevant code (some remnants of Monaco Lange) that we no longer need, examples:

-“FixedScrollBlock” module

- “TouchNav” Module

- “SameHeight” module

And remove extraneous dependencies where applicable

*\*Yielding to Obi for HTML/CSS perspective*

*\*\*assumes time focusing on other tasks simultaneously*