Guidelines Document –

Asset Management Development Challenge

The purpose of this document is to provide some pointers in how to approach this technical challenge.

These are meant as pointers only and should not be viewed as a single approach but act as a guideline document to ensure that all the below points are covered properly by the candidate

Note: The standards required for this challenge are very high.

Use Classes

Do not simply write all code in the index.js page. You will be marked down for doing so. You should separate your code into classes and each class should be contained within its own file. These should live under a subdirectory of site.

It is recommended to have a class for the grid. This is a view class and should receive updates from the model. A model class should also be scripted that maintains state. The grid should listen to updates from the model and render itself in response.

My assignment had three files in addition to the index.js. One for the model, one for the view and one that implemented the observable pattern. There are other valid approaches but this one allows for a natural discussion of the MV* pattern(s) in a resulting interview.

ES6

The assignment uses WebPack which is a Javascript bundler (concatenates all JS files to a single file) and also a build script and live reloader. One of the loaders configured is Babel which transpiles ES6 syntax to ES5 which runs in modern browsers.

It is not a stated requirement but the candidate would be expected to look at the WebPack configuration and see that the Babel transpiler is being employed. As such it will be beneficial to your chances to use ES6 syntax for classes.

Please refer the below URL for more details.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

Architecture

Updates are streamed into the application via sockets on Stomp. These updates should update the data store in the model. In order for the view (the grid) to listen to these updates a design pattern such as Observer should be employed.

You will be asked to justify your choice of design pattern employed by which the view responds to updates from the model. The observer pattern is probably the easiest to implement and most understood.

Please refer the below URL for more details.
http://www.dofactory.com/javascript/observer-design-pattern

Commenting

Comment your code liberally. The marker will check that your code contains comments that are clear and easy to understand. Use JSDoc

Please refer the URL for more details - https://en.wikipedia.org/wiki/JSDoc

Methods

Use methods that are logically named and perform small logical units of work.

These methods should return their output – even if that output is not required to be returned. This is because it demonstrates that the code is testable. This feedback has been made by the markers on several occasions.

Consistent variable naming

The variable and method names should be consistent. Use capitals when defining classes. Method names and variable names should all start with lower case (camel case)

DOM efficiency

The markers are not expecting the DOM manipulations to be efficient so don't go overboard trying to perform as few DOM manipulations as possible. It will make your life harder in the interview when further requirements are added for a pair programming exercise.

Instead have the grid completely re render itself on data updates. As the model has the full state of all known prices this should be trivial.

Testing

It is critical that unit tests are provided as without demonstrating at least a single working unit test the assignment will be rejected outright.

The assignment marker will navigate to the root directory of your project and type

```
npm test
```

It will then expect to see a browser loading, unit test(s) running and passing. Before you submit your assignment check that this works. If unit tests do not successfully run on this command there is zero point submitting the assignment.

Setting up unit testing

This part requires the addition of a test framework to the project. I opted for Karma as the test runner and Jasmine as the BDD framework. But you should use what you are familiar with for client side unit testing.
You will then need to add these node modules to the project and ensure that the package.json file is updated. This is so that the marker can simply perform an "nom update" to download the modules you have selected.

For my choice then the following commands were executed

```
npm install --save-dev jasmine
npm install --save-dev karma
npm install --save-dev karma-webpack
```
Double check that this has updated the package.json file with the modules you have added!!

Webpack needs to be informed where the tests live. It does this by using a file called tests.webpack.js which you should create in the root directory. The content should look like this

```
var context = require.context('./test', true, /.spec.js$/);
context.keys().forEach(context);
```

You will then need to configure the test runner to work properly with WebPack. This is no mean feat and it can take a lot of tinkering.

In my case using karma I created a file called karma.conf.js in the root directory. I include the content of my file below to serve as a guide in the hope it might save valuable time. You can see that WebPack is defined as a preprocessor and that it has a various configuration blocks.

```
var webpack = require('webpack');

module.exports = function(config) {
    config.set({
        frameworks: ['jasmine'],
        files: [
            'tests.webpack.js' //just load this file
        ],
        preprocessors: {
            'tests.webpack.js': [ 'webpack' ]
        },
        webpack: {
            // webpack configuration
            target: "web",
            debug: true,

            module: {
                loaders: [
                    {
                        test: /\.spec.js$/,
                        loader: 'babel-loader'
                    }
                ]
            }
        },
        webpackServer: {
            noInfo: true
        },
        webpackPort: 1234,
        webpackMiddleware: {
            noInfo: true
        },
        reporters: ['progress'],
        colors: true,
        logLevel: config.LOG_INFO,
        singleRun: true, // run the tests once and exit.
        autoWatch: true,
        browsers: ['Chrome']
    })
}
```

Implementing tests

It is doubtful that you will have time to implement full test coverage. Test 2-3 methods on which you application depends to ensure correctness. This should be suffice to demonstrate that you know what you are doing.

Add to the readme.md file

This is a courtesy. If the marker is having trouble running your application for any reason he will go to this file to see your notes. It will be appreciated that you have added some notes here for their benefit.

Provide a paragraph explaining your architectural design decisions. And what testing node modules you installed.

Interview stage extension

Once you are selected for an interview, you will be required to conduct a pair programming exercise – over Webex or similar. You will have an hour to add a new feature to the code.

It is likely however that you will be required to add a visual transformation to the grid so be please go through CSS3 transformations and transitions so that you are properly prepared.

Other Key Points to be considered before submitting the Test are mentioned below.

- **Meaningful naming of variables and functions**. Try to use camel casing while naming variable and functions. Name should be such that it fully justifies its functionality.
- **Consistent, appropriate use of js features and data structures** (implementing your own bubble sort or standard data structure is bad). Try to use as much js methods as you can.
- **Code split** into appropriate modules with low coupling and high cohesion. So divide you code into module and import it from different files wherever necessary.
- Testing commensurate with the riskiness of the code. Code in such a way that it should be testable. Please do write test cases and run to ensure your code is fully testable and is working as expected.
- **Careful treatment of state, especially global state**. Try to use less global variable. If at all you are using global variable make sure their value should not be overridden.
- Source files that read well – nothing fancy, just show a basic concern for the reader.
- Complexity must always be justified. So try to put comments wherever you feel that the end user will not be able to understand it. It is better to put comments to explain code complexity wherever necessary.
- DRY code except where justified**.** If possible, try not to repeat your code.
- **Proper indentation** should be there to make code looks clean.

All the Best