

AppWorks Developer Lab: Clean code - Practical

Using appworks-js safely

Session Outline

In this practical session we will be using appworks-js to write some example code in an Angular application. We will be looking at an example of use of appworks-js that does not meet **Clean Code** standard and will refactor this into something more testable.

appworks-js can be used to build compelling mobile experiences as demonstrated by the OpenText applications that make use of its native plugins. In this example we will look at how we can make use of these plugins while maintaining testability and general **Clean Code**.

Session Steps

1. Open the Angular project in Visual Studio Code
2. Review the use of appworks-js in an example component
3. Create a service
4. Use the service to eliminate our issue
5. Review testability

Session Outcome

After this session we expect the participant should

- be familiar with the appworks-js libraries place within an AppWorks hybrid-app
- be aware of structural coding mistakes that affect testability
- gain some exposure to general app development (Angular, NPM, Visual Studio Code)

The development environment

There are a few basic things we need to get started in a typical web development environment. Hybrid-app development **IS** web programming as we are developing JavaScript HTML and CSS just like a single page application. The major difference being our code can reach mobile devices as well as full desktop applications via AppWorks.

The Developer Lab workstations are already configured with our intended development environment, so we can begin coding.

Visual Studio Code

Visual Studio Code (<https://code.visualstudio.com>) is an open source light-weight code editor. It is not just a text editor but can be classed as an IDE as it supports:

- Intelligent code completion
- Debugging
- Git integration
- Plus, many plugins that support a wide variety of languages and tasks

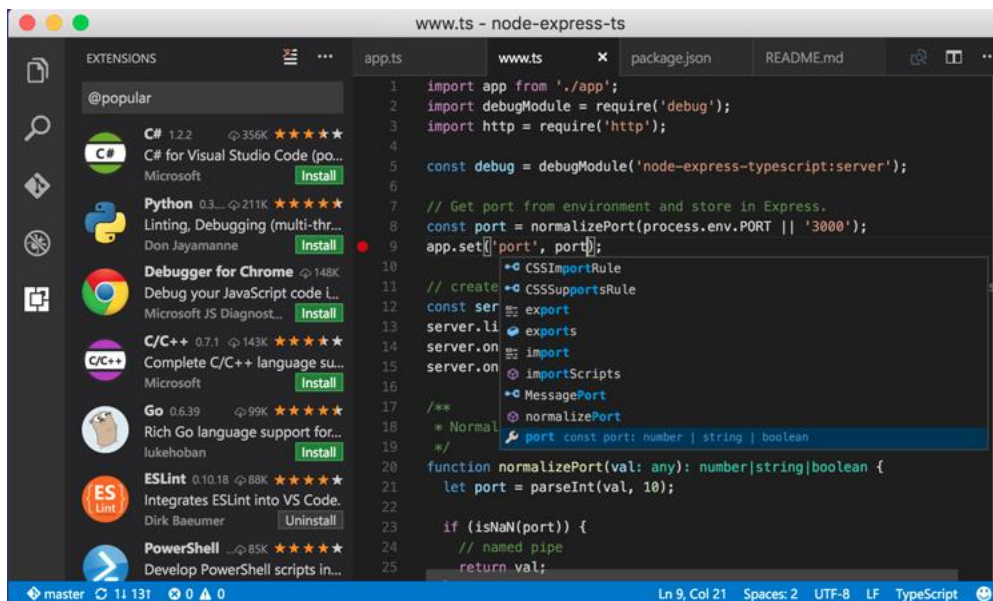


Figure 1 – Visual Studio Code

opentext™

Enterprise World 2018

npm

npm is **the** packager manager for *JavaScript*. It is a massive public repository of code and as well as a set of tools for working with and sharing these packages. Angular and AppWorks are made available to the world via *npm*, plus **all** of the other leading frameworks past and present (React, Bootstrap, Ember, jQuery, ...).

The *package.json* file (<https://docs.npmjs.com/files/package.json>) forms that basis of most sensible endeavors with appworks-js and is where *npm* based projects start. There are many features made available via this file, but the following should be noted:

- the local `/node_modules` containing our projects dependencies is created during *npm*'s inspection of *package.json* when we run the *npm install* command
- a *scripts* section allows developers to introduce build processes into their projects build lifecycle, this is used heavily by frameworks such as Angular

npm is usually installed with Node.js.

Angular

Angular (<https://angular.io>) is one of the leading web development frameworks. It has had a focus on testing since its inception and allows users to write clean code by providing a dependency injection mechanism. It has a rich set of test libraries and debugging tools and has evolved to meet the needs of modern developers operating across a diverse set of platforms.

angular-cli

The Angular CLI is a useful productivity tool that lets us bootstrap projects, and generate components, services, pipes, and many other useful types. We need to install it using *npm* and then can use it any project.

Installing the CLI globally on your developer machine from the terminal:

```
npm install -g @angular/cli
```

We can then use its features from the terminal, the CLI is aware of the directory it is working in, so we want to be at the projects root directory before running commands.

- Creating a new app project: `ng new my-app`
- Creating a new component in my app: `ng generate component mycomponent`

For more details please see the Angular CLI [README](#).

appworks-js

appworks-js is the AppWorks hybrid app library (<https://github.com/opentext/appworks-js>). It provides a bridge between the AppWorks application and supporting native container (the AppWorks mobile client).

It provides almost 30 plugins, many are cross platform, but there are some specialized AppWorks desktop features that differ from mobile.

Using the *appworks-js* library we can implement an AppWorks app that runs on Android, iOS, Windows and OSX with a single code base.

It is actually implemented as a TypeScript library at this time but can be used with vanilla JavaScript. Angular makes use of TypeScript as its primary language so this suits us well.

Many examples of apps and individual plugin examples exist on the OpenText GitHub page which can be accessed here <https://github.com/opentext>, including plain JavaScript examples.

<https://github.com/opentext/appworks-js-example-offlinemanager>

<https://github.com/opentext/appworks-js-example-cache>

<https://github.com/opentext/appworks-js-example-contacts>

<https://github.com/opentext/appworks-js-example-media-capture>

Clean coding with AppWorks

Introduction

We will be working with an example Angular project that makes use appworks-js. The author has created a component that calls an AppWorks plugin to open a file selection dialog, and then performs some business logic with the selected file. This project has been made publicly available via a Git repository the AppWorks team have created. This can be found at <https://github.com/opentext/ew2018-cleanCodeWithAppworks>. This Git repository has three branches:

- master – to be ignored
- practical-session-1 – the initial state of the code to be worked on
- practical-session-1.5 – an intermediary state where we focus on test code
- practical-session-2 – the completed state, testable clean code

Step 1: Open the Angular project in Visual Studio

Open the Visual Studio Code editor and using the header bar menu we need to use the File menu to open the root directory of our project.

File > Open Folder > Find Location of /ew2018-cleanCodeWithAppworks

The project should now load and we should see our files including package.json in the project explorer on the left.

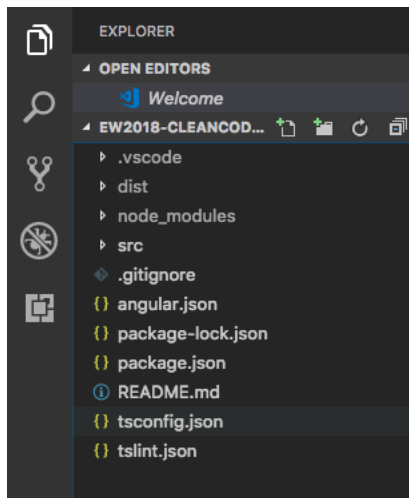


Figure 2 – Explorer tab shows the project files

Ensure we are on the correct Git branch by selecting 'practical-session-1' via the branch selector bottom left. If there are changes to the source code the editor will complain so, please ensure any files that were changed are reset. Once on the correct Git branch we can review the app code.

opentext™

Enterprise World 2018



Figure 3 – Git branch quick selector

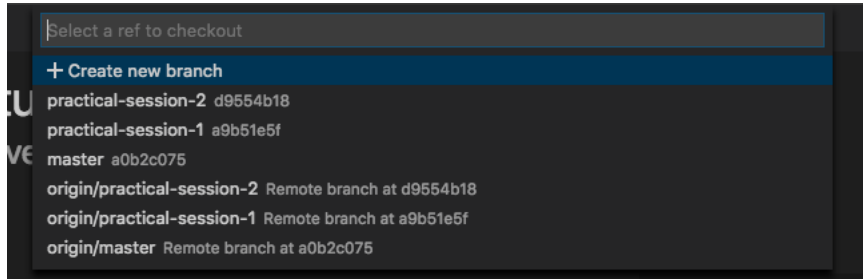


Figure 4 – Select the Git branch we want

Building the Project

We now want to make sure the project builds. Open the terminal made available with Visual Studio Code

View > Integrated Terminal

In the terminal window type the following and hit return

```
npm install
```

This will inspect the package.json file for dependencies and download and place them in a new directory it creates, /node_modules.

node_modules is a local copy of all of the dependencies our project requires.

In the terminal we will now invoke Angular's build pipeline. Angular is part of our project as we can see in the package.json file. It exposes a suite of tools to npm that allow to compile, serve and test our project. At the terminal type the following and hit return.

```
npm run ng build
```

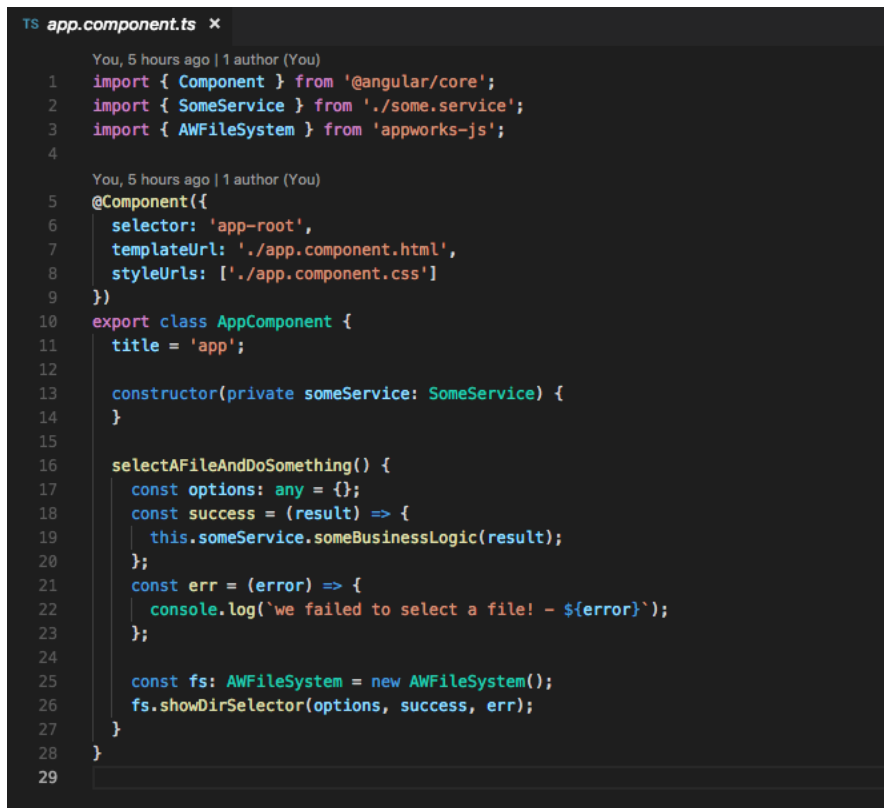
The build process will run the TypeScript compiler and produce the built app inside the /dist directory. Under the hood Webpack is being run, but Angular hides us from this fact.

Step 2: Review the use of appworks-js

We will now take a look at the example component, App Component. Please open the following file using the Explorer panel on the left of the Visual Studio Code editor, `/src/app/app.component.ts`.

Here we see a basic component that Angular created for me when I bootstrapped the project.

- It has one method `selectAFileAndDoSomething` (line 16)
- It has one collaborator service injected via its constructor, `SomeService` (line 13)
- It makes use of the `AWFileSystem` appworks-js plugin to open a directory selector (lines 25, 26)
- It makes use of its collaborator to perform some business logic, calling `SomeService`'s `someBusinessLogic` method after a directory was successfully selected (line 19)
- It will print a message to the console if the plugin fails to select a directory (line 26)



```
TS app.component.ts x
You, 5 hours ago | 1 author (You)
1 import { Component } from '@angular/core';
2 import { SomeService } from './some.service';
3 import { AWFileSystem } from 'appworks-js';
4
You, 5 hours ago | 1 author (You)
5 @Component({
6   selector: 'app-root',
7   templateUrl: './app.component.html',
8   styleUrls: ['./app.component.css']
9 })
10 export class AppComponent {
11   title = 'app';
12
13   constructor(private someService: SomeService) {
14   }
15
16   selectAFileAndDoSomething() {
17     const options: any = {};
18     const success = (result) => {
19       this.someService.someBusinessLogic(result);
20     };
21     const err = (error) => {
22       console.log('we failed to select a file! - ${error}');
23     };
24
25     const fs: AWFileSystem = new AWFileSystem();
26     fs.showDirSelector(options, success, err);
27   }
28 }
29
```

Figure 5 – Our first attempt at AppComponent

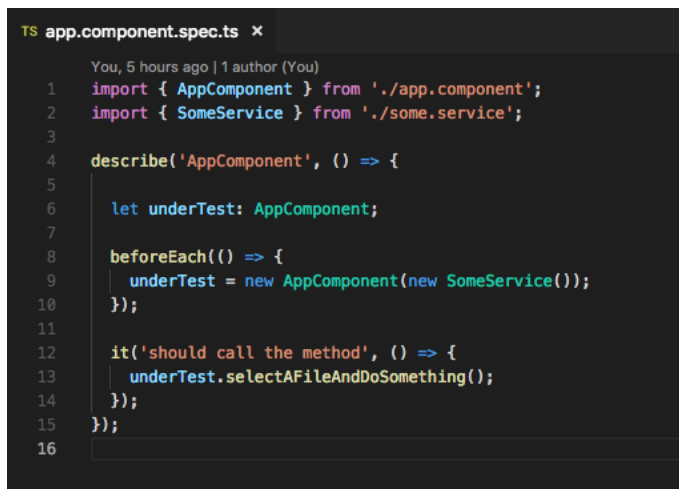
Running a unit test

Please open the following file using the Explorer panel on the left of the Visual Studio Code editor, `/src/app/app.component.spec.ts`.

Here we see a basic unit test for our component. Angular added this for me via the Angular CLI *generate* command but I have made it very basic removing the extra dependency injection tooling it added.

jasmine is the basic toolkit used for testing, it provides the tests structure, and mocking capabilities.

- The **describe** key word defines the suite, it can be run individually, multiple **describe** blocks can be used per file (line 4)
- The **beforeEach** method provided by the jasmine is run to setup our test environment before each test, we create the instance of AppComponent to test in ours (line 8)
- The **it** method lets us define a single test case, naming it and then providing a function which is the test (line 12)



```
TS app.component.spec.ts x
You, 5 hours ago | 1 author (You)
1 import { AppComponent } from './app.component';
2 import { SomeService } from './some.service';
3
4 describe('AppComponent', () => {
5
6     let underTest: AppComponent;
7
8     beforeEach(() => {
9         underTest = new AppComponent(new SomeService());
10    });
11
12    it('should call the method', () => {
13        underTest.selectAFileAndDoSomething();
14    });
15 });
16
```

Figure 6 – Select the Git branch we want

We don't really test anything here, and we can see why when try to call our basic method. At the terminal window type the following, we are going to run Angulars test build step. This uses **karma** to run **jasmine** tests.

```
ng test
```

opentext™

Enterprise World 2018

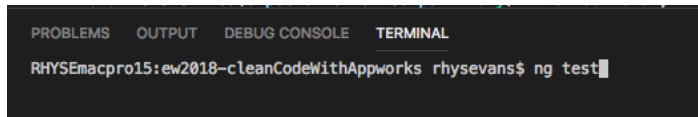


Figure 7 – Running the tests

The **karma** runner will then launch a browser window, giving our test an environment to run in. This environment however is not an AppWorks runtime environment, it is not the AppWorks mobile or desktop environment, so we see an error.

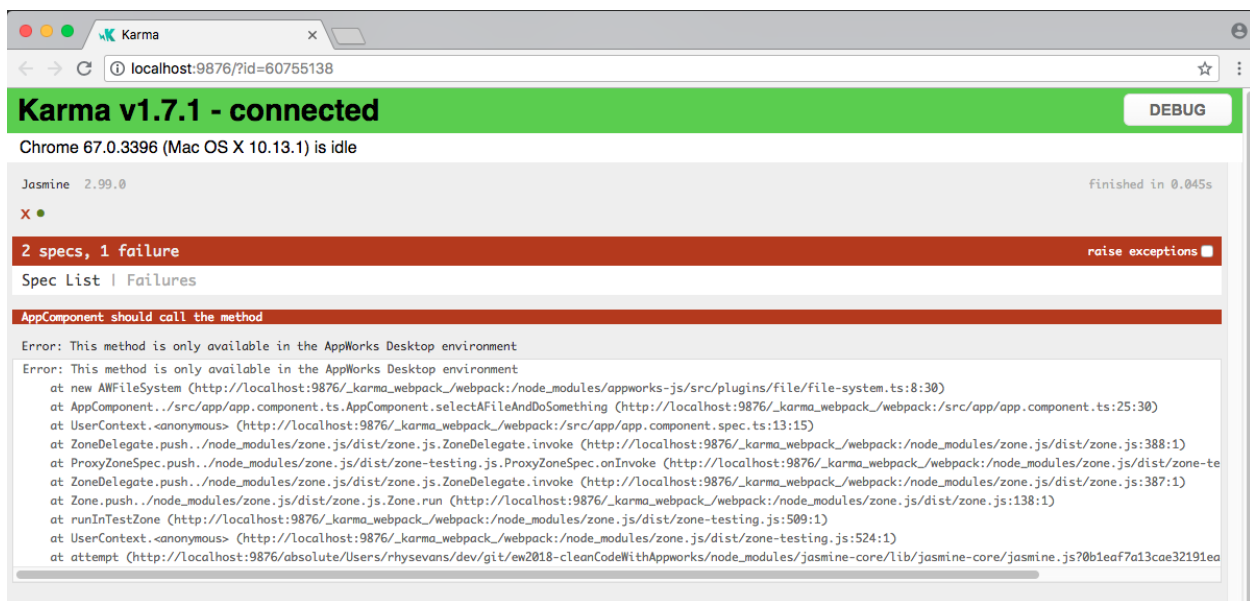


Figure 8 – The unit test has failed

This failure is preventing us from unit testing AppComponent, this is not acceptable!

- The problem is that we are constructing a real AWFileSystem instance
- This is an external third-party library in this usage context
- When the AWFileSystem is asked to perform tasks, it will do so attempting to use the hybrid-app environment which is not available
- This test is about ensuring our business logic works **NOT** that appworks-js can select a directory
- We need to separate code that constructs objects from business logic

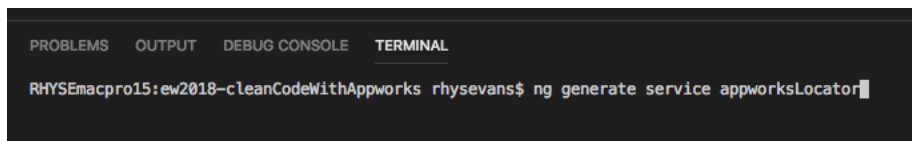
Step 3: Create a service

We want to fix our issue by not mixing creation and business logic. Angular offers a powerful dependency injection mechanism that we can leverage to inject a collaborator that can construct the AWFileSystem on our behalf.

Angular CLI offers utilities to create *services* that can be injected into AppComponent.

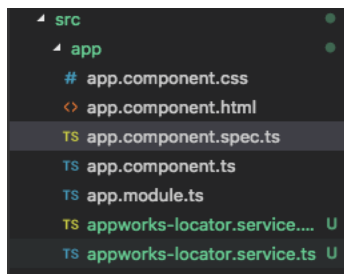
At the terminal please run the following command and then hit return. A pair of files, a service class and unit test (spec) will be created.

```
ng generate service appworksLocator
```



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
RHYSEmacpro15:ew2018-cleanCodeWithAppworks rhysevans$ ng generate service appworksLocator
```

Figure 9 – Generating a service



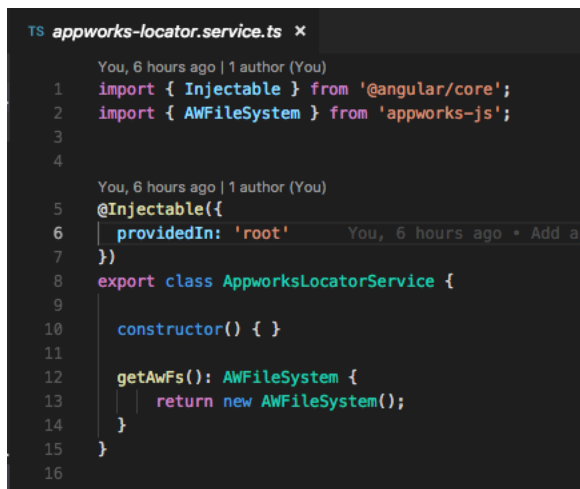
```
src
└─ app
   # app.component.css
   <> app.component.html
   TS app.component.spec.ts
   TS app.component.ts
   TS app.module.ts
   TS appworks-locator.service.... U
   TS appworks-locator.service.ts U
```

Figure 10 – Service files added to the project by Angular CLI

Implementing the AppworksLocator

We can now implement a method that provides an instance of the appworks-js plugin. Here we see the service:

- @Injectable marks the class as available to Angulars DI framework as something that can be used in constructor injection (line 5)
- There is one method, getAwFs, it simply constructs the AWFileSystem instance and returns it (line 12)



```
TS appworks-locator.service.ts x
You, 6 hours ago | 1 author (You)
1 import { Injectable } from '@angular/core';
2 import { AWFileSystem } from 'appworks-js';
3
4
5 @Injectable({
6   providedIn: 'root'
7 })
8 export class AppworksLocatorService {
9
10   constructor() { }
11
12   getAwFs(): AWFileSystem {
13     return new AWFileSystem();
14   }
15 }
16
```

Figure 11 – Appworks Locator implementation

Coding Steps

The following coding steps are required:

- a) Add the import for the AWFileSystem class

```
import { AWFileSystem } from 'appworks-js' ;
```

- b) Add the new method, getAwFs

```
getAwFs(): AWFileSystem {  
  }  
}
```

- c) Simply return a new instance of AWFileSystem

```
return new AWFileSystem()
```

Step 4: Use the service to eliminate the issue

In this step we are going to make use of the AppworksLocatorService to provide an instance our plugin.

We need to make the following updates:

- The locator service should now be injected into the constructor, Angular will do this for us at run time (line 15)
- We get the instance of AWFileSystem using the locator instance (line 27)



```
TS app.component.ts x
You, 7 hours ago | 1 author (You)
1 import { Component } from '@angular/core';
2 import { SomeService } from './some.service';
3 import { AppworksLocatorService } from './appworks-locator.service';
4 import { AWFileSystem } from 'appworks-js';
5
You, 7 hours ago | 1 author (You)
6 @Component({
7   selector: 'app-root',
8   templateUrl: './app.component.html',
9   styleUrls: ['./app.component.css']
10 })
11 export class AppComponent {
12   title = 'app';
13
14   constructor(private someService: SomeService,
15               private awLocator: AppworksLocatorService) {
16   }
17
18   selectAFileAndDoSomething() {
19     const options: any = {};
20     const success = (result) => {
21       this.someService.someBusinessLogic(result);
22     };
23     const err = (error) => {
24       console.log(`we failed to select a file! - ${error}`);
25     };
26
27     const fs: AWFileSystem = this.awLocator.getAwFs();
28     fs.showDirSelector(options, success, err);
29   }
30 }
31
```

Figure 12 – Updates to our AppComponent

Coding Steps

The following coding steps are required:

- a) Add the import for the service we created

```
import { AppWorksLocatorService } from 'appworks-locator.service' ;
```

- b) Inject the new service into the AppComponent as a dependency

```
constructor(private someService: SomeService,  
private awLocator: AppworksLocatorService) {}
```

- a) Add the new method, getAwFs

```
const fs: AWFileSystem = this.awLocator.getAwFs();
```

Step 5: Review testability

Now we have our AppComponent in a better shape we can write a real unit test and make an assertion about whether we managed to call our business logic.

We can utilize some of **jasmynes** mocking capabilities to perform the test safely without have to call the real `AWFileSystem#showDirSelector` method. We will discuss this in more detail next before reviewing the code.

Mocking

In unit tests we strive to test behavior instead of implementation. Consider the following;

- we are writing a calculator class that has an add method that adds two numbers
- we have a test case, $2 + 2$
- assert result is 4

We have tested the implementation here unfortunately.

Instead, imagine each number is an object, this is the test that we really want to see in the assert step

- assert that the add method of `arg1` was passed `arg2`

Spy objects

Spy objects allow us to spy on the objects that are being used in our test, this allows us to see which methods are called, how many times and with what arguments. As we are now injecting the AppworksLocatorService into the we can *spyOn* it via **jasmine**. **Intercepting** the useless native calls and allowing the business logic to flow.

In our test we intercept:

- the call to return the AWFileSystem instance

```
spyOn(awLocator, 'getAwFs').and.returnValue(awFileSystem);
```

- the call to AWFileSystem#showDirSelector
 - o In our example we allow the call to pass through the mocked out showDirSelector call, and we call the success handler passed to the method.
 - o We could instead:
 - call the *error* handler
 - pass a directory path to the *success* handler if we were doing this for real

```
spyOn(awFileSystem, 'showDirSelector').and.callFake(
```

```
function(options, success, err) {
```

```
    success();
```

```
});
```

opentext™

Enterprise World 2018

The following shows the revised, heavily annotated version of the AppComponent unit test.

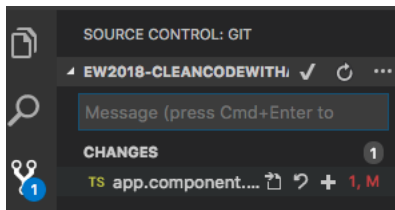
```
TS app.component.spec.ts x
You, 7 hours ago | 1 author (You)
1 import { AppComponent } from './app.component';
2 import { SomeService } from './some.service';
3 import { AppworksLocatorService } from './appworks-locator.service';
4 import { AWFileSystem } from '../../node_modules/appworks-js';
5 import { create } from 'domain';
6
7 describe('AppComponent', () => {
8
9     // the component we are testing
10    let underTest: AppComponent;
11    // the business type service we want to observe calls to
12    let someServiceMock;
13    // our new service that is responsible for construction
14    let awLocator;
15    // the AppWork plugin
16    let awFileSystem;
17
18    // setup method that runs before each test
19    beforeEach(() => {
20        createComponentUnderTest();
21    });
22
23    // tests
24
25    it('should call the business logic method', () => {
26        underTest.selectAFileAndDoSomething();
27        expect(someServiceMock.someBusinessLogic).toHaveBeenCalled();
28    });
29
30    // test helpers
31
32    function createComponentUnderTest() {
33        // we dont need the real obbject here so we can use a spy
34        someServiceMock = jasmine.createSpyObj('someService', ['someBusinessLogic']);
35        awLocator = new AppworksLocatorService();
36        awFileSystem = new AWFileSystem();
37        mockAppWorksFs();
38
39        underTest = new AppComponent(someServiceMock, awLocator);
40    }
41
42    function mockAppWorksFs() {
43        // mock the return of the AppWorks plugin from our new component
44        spyOn(awLocator, 'getAwFs').and.returnValue(awFileSystem);
45
46        // mock out the actual AppWorks call
47        spyOn(awFileSystem, 'showDirSelector').and.callFake(function(options, success, err) {
48            // we simply call success here, this IS the real success handler we implemented, we could pass back a file array
49            success();
50        });
51    }
52
53    });
```

Figure 13 – Our revised test

Coding Steps

Please switch to the practical-sessopm-1.5 branch at this point

If you have existing code changes use the Git view of Visual Studio Code to revert the files. Each changed file appears in this list, there is 1 change in the image below. Use the arrow item per file to revert the changes.



Implement the mockAppWorksFs method

- a) Ensure our AWFileSystem instance is returned when the AppWorksLocator service is asked for it

```
spyOn(awLocator, 'getAWFs').and.returnValue(awFileSystem);
```

- b) Ensure that the AWFileSystem#showDirSelector method simply calls the success handler

```
spyOn(awFileSystem, 'showDirSelector').and.callFake(
```

```
function(options, success, err) {
```

```
    success();
```

```
});
```

- c) Assert that we did actually call the business logic method

```
expect(someServiceMock.someBusinessLogic).toHaveBeenCalled();
```

Conclusion

Hopefully we have seen:

- That simple changes, that are actually supported as first-class members of frameworks, can be used to produce testable code even when working in the hybrid-app environment
- There isn't any reason that we should have low test coverage in an AppWorks app.
- AppWorks can be integrated with modern frameworks to produced testable, structured and understandable code

The Law of Demeter

Astute engineers would have noticed that we implemented what could be described as a service locator. The AppworksLocatorService is acting as a middle-man in our solution. It would be better to inject the plugin directly if possible. ng-appworks exists for this reason, supplying @Injectable versions of the plugin that integrate with Angular's DI system.

About OpenText

OpenText enables the digital world, creating a better way for organizations to work with information, on-premises or in the cloud. For more information about OpenText (NASDAQ: OTEX, TSX: OTEX), visit opentext.com.

Connect with us:

[OpenText CEO Mark Barrenechea's blog](#)

[Twitter](#) | [LinkedIn](#) | [Facebook](#)