Debugging Protractor Tests

End-to-end tests can be difficult to debug because they depend on an entire system, may depend on prior actions (such as log-in), and may change the state of the application they're testing. WebDriver tests in particular can be difficult to debug because of long error messages and the separation between the browser and the process running the test.

Types of Failure

Protractor comes with examples of failing tests (failure_spec.js). To run, start up the test application and a Selenium Server, and run the command below. Then look at all the stack traces.

```
protractor debugging/failureConf.js
```

This test suite shows various types of failure:

- WebDriver throws an error - When a command cannot be completed, for example an element is not found.
- Protractor will fail when it cannot find the Angular library on a page. If your test needs to interact with a non-angular page, access the WebDriver instance directly with `browser.driver`.
- Expectation Failure - Shows what a normal expectation failure looks like.

Timeouts

There are several ways that Protractor can time out. See the Timeouts reference for full documentation.

Pausing to Debug

Protractor supports two methods for pausing to debug - `browser.pause()` and `browser.debugger()`. You probably want to use `browser.pause()`, unless you would like precise control over the node debugger.

Using pause

Insert `browser.pause()` into your test where you want to pause.

```
it('should fail to find a non-existent element', function() {
  browser.get('app/index.html#/form');

  browser.pause();

  // This element doesn't exist, so this fails.
  var nonExistent = element(by.binding('nopenopenope')).getText();
});
```

Run your tests normally.

```
protractor failureConf.js
```

The test will pause execution after the scheduled navigation to `app/index.html#/form` but before trying to get text from the nonnexistant element. The terminal will print instructions for continuing or inspecting the application and a list of the currently pending tasks on the WebDriver control flow.

```
-- WebDriver control flow schedule
 |- waiting for debugger to attach
 |---    at [object Object].<anonymous> (failure_spec.js:13:13)
 |- Protractor.waitForAngular()
 |---    at [object Object].<anonymous> (failure_spec.js:16:59)
wd-debug>
```

Enter `c` to move the test forward by one task.
While the test is paused you may also interact with the browser. Note that if you open the Chrome Dev Tools, you must close them before continuing the test because ChromeDriver cannot operate when the Dev Tools are open.

When you finish debugging, exit by pressing `Ctrl-C`. Your tests will continue where they left off, using the same browser.
You can also use `browser.explore()` in your test script to pause and enter an interactive repl loop. In this interactive mode, you can send WebDriver commands to your browser. The resulting value or error will be reported to the terminal.

```
> element(by.binding('nopenopenope')).getText()
NoSuchElementError: No element found using locator: by.binding("nopenopenope")
>
> element(by.binding('user')).getText()
'Anon'
```

Note: Since these are asynchronous tasks, you would have to increase the default timeout of your specs else default timeout exception would be thrown!

Using debugger

Insert `browser.debugger();` into your test where you want to break:

```
it('should fail to find a non-existent element', function() {
  browser.get('app/index.html#/form');

  // Run this statement before the line which fails. If protractor is run
  // with the debugger (protractor debug <...>), the test
  // will pause after loading the webpage but before trying to find the
  // element.
  browser.debugger();

  // This element doesn't exist, so this fails.
  var nonExistent = element(by.binding('nopenopenope')).getText();
});
```

Then run the test *in debug mode*:

```
protractor debug debugging/failureConf.js
```

This uses the node debugger. Enter `c` to start execution and continue after the breakpoint or enter `next` command.The next command steps to the next line in control flow.
`browser.debugger();` is different from node's `debugger;` statement because it adds a breakpoint task asynchronously in queue. This means the example above will pause after the `get` statement has been executed, whereas `debugger;` pauses the test after the get command is scheduled but has not yet been executed.
Protractor's `debugger()` method works by scheduling a node debug breakpoint on the control flow.
When `debugger()` is called, it also inserts all the client side scripts from Protractor into the browser as `window.clientSideScripts`. They can be used from the browser's console.

```
// In the browser console (e.g. from Chrome Dev Tools)
> window.clientSideScripts.findInputs('username');
// Should return the input element with model 'username'.

// You can also limit the scope of the locator
> window.clientSideScripts.findInputs('username', document.getElementById('#myEl'));
```

Setting Up VSCode for Debugging

VS Code has built-in debugging support for the Node.js runtime and can debug JavaScript, TypeScript, and any other language that gets transpiled to JavaScript.

To set up VSCode for Protractor, follow the below steps:

1. Click on the Debugging icon in the View Bar on the side of VS Code.

2. Click on the Configure gear icon on the Debug view top bar and choose nodejs environment.
3. It will generate a `launch.json` file under your workspace's `.vscode` folder.
4. Setup your launch.json file by configuring below two commands:

```
 "program": "${workspaceRoot}/node_modules/protractor/bin/protractor",
  "args": ["${workspaceRoot}/protractorConfig.js"],
```

5. Save your launch.json, put some breakpoints and start debugging.

Setting Up WebStorm for Debugging

To set up WebStorm for Protractor, do the following:

1. Open the Run/Debug Configurations dialog
2. Add new Node.js configuration.
3. On the Configuration tab set:

- **Node Interpreter**: path to node executable
- **Working directory**: your project base path
- **JavaScript file**: path to Protractor cli.js file (e.g. *node_modules\protractor\lib\cli.js*)
- **Application parameters**: path to your Protractor configuration file (e.g. *protractorConfig.js*)

4. Click OK, place some breakpoints, and start debugging.

Testing Out Protractor Interactively

When debugging or first writing test suites, you may find it helpful to try out Protractor commands without starting up the entire test suite. You can do this with the element explorer.

To run element explorer, simply run protractor as you normally would, but pass in the flag --elementExplorer:

```
protractor --elementExplorer
```

This will load up the URL on WebDriver and put the terminal into a REPL loop. You will see a > prompt.
The `browser`, `element` and `protractor` variables will be available. Enter a command such as:

```
> browser.get('http://www.angularjs.org')
```

or

```
> element(by.id('foobar')).getText()
```

Typing tab at a blank prompt will fill in a suggestion for finding elements. You can also use the `list(locator)` command to list all elements matching a locator.
Element explorer will start chrome by default. However, you can specify another browser, change browser settings, or specify any other config that you normally would with your protractor test. To do this, pass configs to protractor like you normally would, but with the `--elementExplorer` flag set:

```
protractor [configFile] [options] --elementExplorer
```

For example, to connect to ChromeDriver directly, use

```
protractor --directConnect --elementExplorer
```

Element explore will ignore your specs, not set up your framework (e.g. jasmine, mocha, cucumber), and only allow you to pass in 1 capability, but will honor every other parameter in your config.

Note `baseUrl` is used here as the initial page, i.e. element explorer will try to navigate to `baseUrl` automatically on start.

Taking Screenshots

WebDriver can snap a screenshot with `browser.takeScreenshot()`. This can be a good way to help debug tests, especially for tests that run on a continuous integration server. The method returns a promise which will resolve to the screenshot as a base-64 encoded PNG.
Sample usage:

```
// at the top of the test spec:
var fs = require('fs');

// ... other code

// abstract writing screen shot to a file
function writeScreenShot(data, filename) {
    var stream = fs.createWriteStream(filename);

    stream.write(new Buffer(data, 'base64'));
    stream.end();
}

// ...

// within a test:
browser.takeScreenshot().then(function (png) {
    writeScreenShot(png, 'exception.png');
});
```