Node.js: (JSRE like JRE for Java)

- Node.js is JavaScript run-time environment that executes JavaScript code outside of a browser.

- open-source, cross-platform,

-There are thousands of open-source libraries for Node.js, most of them hosted on the npm website.

-Node.js allows the creation of Web servers and networking tools using JavaScript and helps them in running

-Node.js has an event-driven architecture capable of asynchronous I/O.

-Though .js is the standard filename extension for JavaScript code, the name "Node.js" does not refer to a particular file in this context and is merely the name of the product.

Initial release	May 27, 2009; 10 years ago
Written in	C, C++, JavaScript
Repository	github.com/nodejs/node

-V8 is the JavaScript execution engine which was initially built for Google Chrome. It was then open-sourced by Google in 2008. Written in C++, V8 compiles JavaScript source code to native machine code during runtime instead of interpreting it in ahead of time (AOT)

-As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications.

- download and install latest node version from (https://nodejs.org/download/release/latest/)

Commands:

node --version

jre =>node.js

selenium(uses webdriver to interact with web page) =>protractor(uses webdriverJS)

AngularJS :

AngularJS is a JavaScript-based open-source front-end web framework mainly maintained by Google.

Features:

-MVC – The framework is built on the famous concept of MVC (Model-View-Controller). This is a design pattern used in all modern day web applications. This

pattern is based on splitting the business logic layer, the data layer, and presentation layer into separate sections. The division into different sections is done so that each one could be managed more easily.

-Unit Testing ready – The designers at Google not only developed Angular but also developed a testing framework called "**Karma**" which helps in designing unit tests for AngularJS applications.

-Angular supports testing, both Unit Testing, and Integration Testing.

Visual Studio Code

-Visual Studio Code is a source-code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, embedded Git control and GitHub, syntax highlighting, intelligent code completion, snippets, and code refactoring.

-VS Code is free for private or commercial use.

Developer(s): Microsoft

Initial release: April 29, 2015; 4 years ago

Written in: TypeScript, JavaScript, Cascading Style Sheets

Protractor:

https://www.protractortest.org/#/toc

Protractor is an end-to-end test framework for AngularJS applications (also works for normal Web Applications). Protractor is a Node.js program that supports the Jasmine and Mocha test frameworks.

To run, you will need to have Node.js installed. You will download Protractor package using npm, which comes with Node.js.

By default, Protractor uses the Jasmine test framework for its testing interface.

Why can't we find Angular JS web elements using Normal Selenium Web driver?

Angular JS applications have some extra HTML attributes like ng-repeater, ng-controller, ng-model.., etc. which are not included in Selenium locators. Selenium is not able to identify those web elements using Selenium code. So, Protractor on the top of Selenium can handle and controls those attributes in Web Applications.

```
<input type="text" ng-model="person.name"/>
```

by.model, by.binding

Commands:

- npm install --proxy http://xyz.com:8080
  This will install all the packages specified in package.json

Or if there is no package.json then it can be installed individually

- npm install -g protractor
  This will install two command line tools, protractor and webdriver-manager.
- protractor --version
- webdriver-manager version
- webdriver-manager update

  we can use local protractor to run the script
- npm install protractor //to install locally
  when running webdriver manager locally we use below commands:
- node_modules/.bin/webdriver-manager status
- node_modules/.bin/webdriver-manager clean
- node_modules/.bin/webdriver-manager help
- node_modules/.bin/webdriver-manager update help
- node_modules/.bin/webdriver-manager update --proxy http://xyz.com:8080
  by default it installs/updates the latest version of chrome, gecko driver, and selenium standalone server
- node_modules/.bin/webdriver-manager update --ie --versions.standalone 3.4.0 --versions.ie 3.4.0
  --ie: install only ie driver (by default latest version of ie 32 bit),--ie32,--ie64
- webdriver-manager start
  (needed only when we want run via server otherwise directConnect:true (given in config file) is faster as script directly talks to chrome driver.)

  This will start up a Selenium Server and will output a bunch of info logs. Your Protractor test will send requests to this server to control a local browser. You can see information about the status of the server at http://localhost:4444/wd/hub.

  Protractor needs at least two files to run, a **spec**(test or script) file and a **configuration file**.

```
// spec.js
describe('Protractor Demo App', function() {
  it('should have a title', function() {
    browser.get('http://juliemr.github.io/protractor-demo/');
    element(by.model('second')).sendKeys(2);
    element(by.id('gobutton')).click();
    expect(browser.getTitle()).toEqual('Super Calculator');
  });
});
```

The `describe` and `it` syntax is from the Jasmine framework. `browser` ,element, by are globals created by Protractor, browser is used for browser-level commands such as navigation with `browser.get`.

Describe typically defines components of an application, which can be a class or function etc.

The `element` function is used for finding HTML elements on your webpage.

- element() returns ElementFinder object
- element.all) returns ElementArrayFinder
- element.all( ).count(), element.all( ).first(), element.all( ).last(),element.all( ).get(i)
- ElementArrayFinder also has methods `each`, `map`, `filter`, and `reduce` which are analogous to JavaScript Array methods
- `element` takes one parameter, a Locator, which describes how to find the element.
  The `by` object creates Locators
- The ElementFinder can be treated as a WebElement for most purposes, in particular, you may perform actions (i.e. click, getText) on them as you would a WebElement.
- Unlike a WebElement, an ElementFinder will wait for angular to settle before performing finds or actions.
- ElementFinder can be used to build a chain of locators that is used to find an element

  element(by.css('#abc')).element(by.css('#def')).isPresent()

- element().getWebElement() Returns the WebElement represented by this ElementFinder.
- Then() Retrieve the element(s) represented by the ElementFinder or ElementArrayFinder.

  element(by.binding('person.name')).getText().then(function(name) {
  expect(name).toBe('Foo');

  });

  Protractor uses `jasminewd2`, which wraps around jasmine's `expect` so that we can write:

expect(el.getText()).toBe('Hello, World!')

```
// conf.js
exports.config = {
  framework: 'jasmine',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['spec.js']
}
```

It requires min above 3 details. It will use the defaults for all other configuration which are not provided. Chrome is the default browser.

Config file: contains config details like browser & capabilities, framework type, specs (array of strings) details, beforeLaunch, onPrepare methods, `seleniumAddress` or directConnect:true, SELENIUM_PROMISE_MANAGER: false(so that we can use async and await) and other protractor related config details.

Test Framework types: jasmine, mocha , custom (Cucumber/SerenityJs)

Command to run :

- protractor conf.js

changing the configuration:

For a full list of config options, see the config file.

config.ts

```
// conf.js
exports.config = {
  framework: 'jasmine',
```

```
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['spec.js'],
  capabilities: {
    browserName: 'firefox'
  }
}
```

The `capabilities` object describes the browser to be tested against

You can also run tests on more than one browser at once. Change conf.js to:

```
// conf.js
exports.config = {
  framework: 'jasmine',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['spec.js'],
  multiCapabilities: [{
    browserName: 'firefox'
  }, {
    browserName: 'chrome'
  }]
}
```

```
element.all(by.css('.items li'))
```

// Or using the shortcut $$() notation instead of element.all(by.css()):

$$('.items li')

In css:

-> id

. ->class

WebDriver Syntax	**Protractor Syntax**
webdriver.By	by
browser.findElement(...)	element(...)

browser.findElements(...)	element.all(...)
browser.findElement(webdriver.By.css(...))	$(...)
browser.findElements(webdriver.By.css(...))	$$(...)

Protractor with Type script:

This package.json references the local protractor directory with `"protractor": "file: ../"`. For the type declarations to work, from the protractor directory run an `npm install` to generate the declarations file.

```
node_modules/       // contains downloaded node modules
tmp/                // compiled javascript output
package.json        // node dependencies for the project
```

Protractor typings

To use Protractor types, you'll need to import `protractor`. After this is imported, you should have autocompletion hints when typing.

```
import {browser, element, by, By, $, $$, ExpectedConditions} from 'protractor';
```

`async`/`await`

https://www.protractortest.org/#/async-await

- The Web Driver Control Flow is used to synchronize your commands so they reach the browser in the correct order (see /docs/control-flow.md for details). In the future, the control flow is being removed (see SeleniumHQ's github issue for details). Instead of the control flow, you can synchronize your commands with promise chaining or the upcoming ES7 feature `async`/`await`.
- Previously, we have Typescript support for `async`/`await`: Please see TypeScript examples which use `async`/`await`.
- The latest Node.js provides native async/await, which means we can get stable e2e test without using control flow in javascript test.

  Note: To write and run native async/await test, the node.js version should be greater than or equal to 8.0, and Jasmine version should be greater than or equal to 2.7

```
describe('angularjs homepage', function() {
  it('should greet the named user', async function() {
```

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

    await element(by.model('yourName')).sendKeys('Julie');

    var greeting = element(by.binding('yourName'));

    expect(await greeting.getText()).toEqual('Hello Julie!');
  });
```

Don't forget to turn off control_flow, you cannot use a mix of `async`/`await` and the control flow: `async`/`await` causes the control flow to become unreliable (see github issue). So if you `async`/`await` anywhere in a spec, you should use the `SELENIUM_PROMISE_MANAGER: false`

`Jasmine:`

Jasmine Matchers

```
to(Not)Be( null | true | false )
to(Not)Equal( value )
to(Not)Match( regex | string )
toBeDefined()
toBeUndefined()
toBeNull()
toBeTruthy()
toBeFalsy()
to(Not)Contain( string )
toBeLessThan( number )
toBeGreaterThan( number )
toBeNaN()
toBeCloseTo( number, precision )
toThrow()
```

https://www.protractortest.org/#/frameworks

Choosing a Framework

Protractor supports two behavior driven development (BDD) test frameworks out of the box: Jasmine and Mocha. These frameworks are based on JavaScript and Node.js and provide the syntax, scaffolding, and reporting tools you will use to write and manage your tests.

Using Jasmine

Currently, Jasmine Version 2.x is supported and the default test framework when you install Protractor. For more information about Jasmine, see the Jasmine GitHub site. For more information regarding how to upgrade to Jasmine 2.x from 1.3, see the Jasmine upgrade guide.

Using Mocha

Note: Limited support for Mocha is available as of December 2013. For more information, see the Mocha documentation site.

If you would like to use the Mocha test framework, you'll need to use the BDD interface and Chai assertions with Chai As Promised.

Using Cucumber

https://github.com/protractor-cucumber-framework/protractor-cucumber-framework

https://github.com/cucumber/cucumber-js/blob/master/features/rerun_formatter.feature

Note: Cucumber is no longer included by default as of version `3.0`. You can integrate Cucumber with Protractor with the `custom` framework option. For more information, see the Protractor Cucumber Framework site or the Cucumber GitHub site.

If you would like to use the Cucumber test framework, download the dependencies with npm. Cucumber should be installed in the same place as Protractor - so if protractor was installed globally, install Cucumber with -g.

```
npm install -g cucumber

npm install --save-dev protractor-cucumber-framework
```

Set the 'framework' property to custom by adding `framework: 'custom'` and `frameworkPath: 'protractor-cucumber-framework'` to the `config file(cucumberConf.js)`

Options for Cucumber such as 'format' can be given in the config file with cucumberOpts, A basic cucumberConf.js file has been provided below:

```
/*

Basic configuration to run your cucumber

feature files and step definitions with protractor.

**/

exports.config = {


  seleniumAddress: 'http://localhost:4444/wd/hub',
```

```
  baseUrl: 'https://angularjs.org/',


  capabilities: {
      browserName:'chrome'
  },


  framework: 'custom',  // set to "custom" instead of cucumber.


  frameworkPath: require.resolve('protractor-cucumber-framework'),  // path relative
to the current config file


  specs: [
    './cucumber/*.feature'     // Specs here are the cucumber feature files
  ],


  // cucumber command line options
  cucumberOpts: {
    require: ['./cucumber/*.js'],  // require step definition files before executing
features
    tags: [],                      // <string[]> (expression) only execute the featur
es or scenarios with tags matching the expression
    strict: true,                  // <boolean> fail if there are any undefined or pe
nding steps
    format: ["pretty"],            // <string[]> (type[:path]) specify the output for
mat, optionally supply PATH to redirect formatter output (repeatable)
    'dry-run': false,              // <boolean> invoke formatters without executing s
teps
    compiler: []                   // <string[]> ("extension:module") require files w
ith the given EXTENSION after requiring MODULE (repeatable)
  },


 onPrepare: function () {
    browser.manage().window().maximize(); // maximize the browser before executing th
e feature files
  }
```

```
};
```

The WebDriver Control Flow

https://www.protractortest.org/#/control-flow

The WebDriverJS API is based on promises, which are managed by a control flow and adapted for Jasmine. A short summary about how Protractor interacts with the control flow is presented below

Promises and the Control Flow

WebDriverJS (and thus, Protractor) APIs are entirely asynchronous. All functions return promises.

WebDriverJS maintains a queue of pending promises, called the control flow, to keep execution organized. For example, consider this test:

```
it('should find an element by text input model', function() {
  browser.get('app/index.html#/form');

  var username = element(by.model('username'));
  username.clear();
  username.sendKeys('Jane Doe');

  var name = element(by.binding('username'));

  expect(name.getText()).toEqual('Jane Doe');

  // Point A
});
```

At Point A, none of the tasks have executed yet. The `browser.get` call is at the front of the control flow queue, and the `name.getText()` call is at the back. The value of `name.getText()` at point A is an unresolved promise object.

Protractor Adaptations

Protractor adapts Jasmine so that each spec automatically waits until the control flow is empty before exiting.

Jasmine expectations are also adapted to understand promises. That's why this line works - the code actually adds an expectation task to the control flow, which will run after the other tasks:

```
  expect(name.getText()).toEqual('Jane Doe');
```

HTML Drag and Drop Simulator for E2E testing.

Now, **WebDriver** cannot handle HTML Drag and Drop. This module can simulate the HTML Drag and Drop by using the **Execute Script command**.

```
npm install --save-dev html-dnd
```

With Typescript

```
import {code as dragAndDrop} from 'html-dnd';

or

var dragAndDrop = require('html-dnd').code;



driver.executeScript(dragAndDrop, draggable, droppable);
```

Using Jasmine with node:

The Jasmine node package contains helper code for developing and running Jasmine tests for node-based projects.

Install

You can install Jasmine using npm, locally in your project and globally to use the CLI tool.

```
npm install jasmine

npm install -g jasmine
```

Init a Project

Initialize a project for Jasmine by creating a spec directory and configuration json for you.

```
jasmine init
```

Note that if you installed Jasmine locally you could still use the command line like this:

```
node node_modules/jasmine/bin/jasmine init
```

Generate examples

Generate example spec and source files

```
jasmine examples
```

GLOB:

https://github.com/isaacs/node-glob

* Matches 0 or more characters in a single path portion

** If a "globstar" is alone in a path portion, then it matches zero or more directories and subdirectories searching for matches. It does not crawl symlinked directories.

jasmine.json

```
{
  "spec_dir": "spec",
  "spec_files": [
    "**/*[sS]pec.js"
  ],
  "helpers": [
    "helpers/**/*.js"
  ],
  "stopSpecOnExpectationFailure": false,
  "random": true
}
```

Running tests

Once you have set up your `jasmine.json`, you can execute all your specs by running `jasmine` from the root of your project (or `node node_modules/jasmine/bin/jasmine.js` if you had installed it locally).
If you want to just run one spec or only those whom file names match a certain glob pattern you can do it like this:

```
jasmine spec/appSpec.js
jasmine "**/model/**/critical/**/*Spec.js"
```

https://jasmine.github.io/tutorials/your_first_suite.html

¶

Suites: `describe` Your Tests

The describe function is for grouping related specs, typically each test file has one at the top level. The string parameter is for naming the collection of specs, and will be concatenated with specs to make a spec's full name. This aids in finding specs in a large suite. If you name them well, your specs read as full sentences in traditional BDD style.

```
describe("A suite", function()
{
  it("contains spec with an
expectation", function() {
    expect(true).toBe(true);
  });
});
```

Specs

Specs are defined by calling the global Jasmine function it, which, like `describe` takes a string and a function. The string is the title of the spec and the function is the spec, or test. A spec contains one or more expectations that test the state of the code. An expectation in Jasmine is an assertion that is either true or false. A spec with all true expectations is a passing spec. A spec with one or more false expectations is a failing spec.

¶

It's Just Functions

Since `describe` and `it` blocks are functions, they can contain any executable code necessary to implement the test. JavaScript scoping rules apply, so

```
describe("A suite is just a
function", function() {
  var a;

  it("and so is a spec",
function() {
    a = true;

    expect(a).toBe(true);
  });
```

variables declared in a `describe` are available to any `it` block inside the suite.

¶

Expectations

Expectations are built with the function `expect` which takes a value, called the actual. It is chained with a Matcher function, which takes the expected value.

¶

Matchers

Each matcher implements a boolean comparison between the actual value and the expected value. It is responsible for reporting to Jasmine if the expectation is true or false. Jasmine will then pass or fail the spec.

¶

Any matcher can evaluate to a negative assertion by chaining the call to `expect` with a `not` before calling the matcher.

¶

Jasmine has a rich set of matchers included, you can find the full list in the API docs There is also the ability to write custom matchers for when a project's domain calls for specific assertions that are not included in Jasmine.

```
});



describe("The 'toBe' matcher
compares with ===", function()
{









  it("and has a positive
case", function() {
    expect(true).toBe(true);
  });









  it("and can have a negative
case", function() {

expect(false).not.toBe(true);
  });



});
```

¶

Setup and Teardown

To help a test suite DRY up any duplicated setup and teardown code, Jasmine provides the global beforeEach, afterEach, beforeAll, and afterAll functions.

¶

As the name implies, the `beforeEach` function is called once before each spec in the `describe` in which it is called

¶

and the `afterEach` function is called once after each spec.

¶

The `beforeAll` function is called only once before all the specs in `describe` are run

¶

and the `afterAll` function is called after all specs finish

¶

`beforeAll` and `afterAll` can be used to speed up test suites with expensive setup and teardown.
However, be careful using `beforeAll` and `afterAll`! Since they are not reset between specs, it is easy to accidentally leak state between your

```
describe("A suite with some shared setup", function() {
  var foo = 0;

  beforeEach(function() {
    foo += 1;
  });

  afterEach(function() {
    foo = 0;
  });

  beforeAll(function() {
    foo = 1;
  });

  afterAll(function() {
    foo = 0;
  });

});
```

specs so that they erroneously pass or fail.

¶

The `this` keyword

Another way to share variables between a `beforeEach`, `it`, and `afterEach` is through the `this` keyword. Each spec's `beforeEach`/`it`/`afterEach` has the `this` as the same empty object that is set back to empty for the next spec's `beforeEach`/`it`/`afterEach`.

```
describe("A spec", function() {
  beforeEach(function() {
    this.foo = 0;
  });

  it("can use the `this` to
share state", function() {

expect(this.foo).toEqual(0);
    this.bar = "test
pollution?";
  });

  it("prevents test pollution
by having an empty `this`
created for the next spec",
function() {

expect(this.foo).toEqual(0);

expect(this.bar).toBe(undefined);
  });
});
```

¶

Manually failing a spec with `fail`

The `fail` function causes a spec to fail. It can take a failure message or an Error object as a parameter.