



Javascript Testing Frameworks

an overview

MochaJS

Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun.

- Provides compatibility for both frontend and backend testing
- NodeJS debugger is supported which makes error tracing easier
- Accurate reporting
- Provides support for all browsers including the headless Chrome library
- Very convenient framework for the developers to write test cases

<https://mochajs.org/>

JEST

JEST is arguably the most popular JavaScript testing framework used and maintained by Facebook.

- Compatible with NodeJS, React, Angular, VueJS, and other Babel based projects
- Standard syntax with documentation support
- Very fast and highly performant
- Managing tests with larger objects is possible using Live Snapshots

<https://jestjs.io/>

Jasmine

Introduced in 2010, Jasmine is an open-source JavaScript testing framework. It is capable of testing all types of JavaScript applications. This framework supports Behavioral Driven Development (BDD).

Using Jasmine, one can perform test cases similar to user behavior on a website.

- Provides small, clean and straightforward syntax for easy testing
- Does not require any Document Object Model (DOM)
- Provides support for both frontend and backend tests
- Ease in coding as the syntax used is very similar to a natural language
- Strong documentation and community support

<https://jasmine.github.io/>

Karma

Karma is another popular open-source productive testing environment. It allows a QA to perform tests for an application in different environments. Karma allows the application script to be executed on real browsers and devices like phones and tablets.

- Supports integrations with top CI/CD tools like Jenkins, Travis, and Semaphore
- Tests on real devices and browsers are possible. Want to run a quick automated test on a real device cloud? Try now.
- Provides support for headless environments like PhantomJS
- Supports remote testing directly from a terminal or IDE
- Is framework agnostic, which means one can describe tests with popular frameworks like Mocha, Jasmine. One can also write a simple adapter for a specific framework.

<https://karma-runner.github.io/3.0/>

Cypress

Fast, easy and reliable testing for anything that runs in a browser.

A complete end-to-end testing framework.

- open source and popular
- Supports integrations with top CI/CD tools like Jenkins, Travis, and Semaphore
- great dashboard

<https://www.cypress.io/>

Jest into an Angular application and library

a demo

Why JEST

- Sensible faster; parallelized test runs
- Snapshot testing; to make sure your UI does not change unexpectedly
- Rich CLI options; only run failed tests, filter on filename and/or test name, only run related tests since the latest commit
- Readable and useful tests reports
- Sandboxed tests; which means automatic global state resets
- Built in code coverage

Creating an angular application

Setup a demo app

```
ng new my-awesome-app
```

Start the app

```
cd my-awesome-app  
npm start
```

Run the tests

```
npm test
```

```
npm install -D jest jest-preset-angular @types/jest
```

Configuration 1/3

In your project root, create `setup-jest.ts` file with following contents:

```
import 'jest-preset-angular/setup-jest';

Object.defineProperty(window, 'CSS', { value: null });
Object.defineProperty(window, 'getComputedStyle', {
  value: () => {
    return {
      display: 'none',
      appearance: ['-webkit-appearance'],
    };
  },
});
Object.defineProperty(document, 'doctype', {
  value: '<!DOCTYPE html>',
});
Object.defineProperty(document.body.style, 'transform', {
  value: () => {
    return {
      enumerable: true,
      configurable: true,
    };
  },
});
```

Configuration 2/3

Add the following section to your root `package.json`

```
{
  "jest": {
    "preset": "jest-preset-angular",
    "setupFilesAfterEnv": ["<rootDir>/setup-jest.ts"],
    "globalSetup": "jest-preset-angular/global-setup"
  }
}
```

Adjust your `tsconfig.spec.json` to be:

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/spec",
    "module": "CommonJs",
    "types": ["jest"]
  },
  "include": ["src/**/*.spec.ts", "src/**/*.d.ts"]
}
```


Configuration 3/3

To run our tests with Jest we can now modify the test script to use Jest instead of ng test withing the package.json.

```
"test": "jest",
```

Clean your folder

```
npm uninstall karma karma-chrome-launcher karma-coverage-istanbul-reporter karma-jasmine karma-jasmine-html-reporter
```

Adapt your test in component.spec

```
test(`the title is 'my-awesome-app'`, async(() => {  
  const fixture = TestBed.createComponent(AppComponent)  
  const app = fixture.debugElement.componentInstance  
  expect(app.title).toEqual('my-awesome-app')  
}))
```

Remove src/test.ts and karma.conf.js

```
rm src/test.ts  
rm karma.conf.js
```

Remove the target "test" of the "angular.json" file

Run the test

run npm test,

```
npm test
```


Write more complex test

```
import { ComponentFixture, fakeAsync, TestBed } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  let component: AppComponent;
  let fixture: ComponentFixture<AppComponent>;

  beforeEach(fakeAsync(() => {
    TestBed.configureTestingModule({
      imports: [RouterTestingModule],
      declarations: [ AppComponent ]
    });
    fixture = TestBed.createComponent(AppComponent);
    component = fixture.debugElement.componentInstance;
  }));

  test('should exist', () => {
    expect(component).toBeDefined();
  });
  test(`the title is 'my-awesome-app'`, fakeAsync(() => {
    expect(component.title).toEqual('my-awesome-app')
  }))
});
```

Unit-Test d'un Service 1/2

```
describe('PickyWeatherStation', () => {  
  
  it('should give temperature', fakeAsync(() => {  
    const weatherStation: PickyWeatherStation = TestBed.get(PickyWeatherStation);  
    let temperature;  
    weatherStation.getTemperature('Lyon')  
      .subscribe(_temperature => temperature = _temperature);  
    expect(temperature).toBe(42);  
  }));  
  
});
```

La méthode statique **TestBed.get** permet d'injecter les services dans les tests unitaires.

Unit-Test d'un Service 2/2

Pour éviter de récupérer l'instance dans chaque "spec", pensez à utiliser la fonction `beforeEach`

```
let weatherStation: PickyWeatherStation;  
beforeEach(() => weatherStation = TestBed.get(PickyWeatherStation))
```


Unit-Test d'un Service 2/2

```
describe('PickyWeatherStation', () => {  
  let weatherStation: PickyWeatherStation;  
  beforeEach(inject([PickyWeatherStation], _weatherStation => weatherStation = _weatherStation));  
  it('should give temperature', fakeAsync(() => {  
    let temperature;  
  
    weatherStation.getTemperature('Lyon')  
      .subscribe(_temperature => temperature = _temperature);  
    expect(temperature).toBe(42);  
  }));  
});
```

La fonction inject peut être utilisée directement autour de la fonction de "spec" mais il est généralement plus simple d'ajouter un beforeEach pour chaque service afin d'éviter les problèmes liés à l'ordre des tokens d'injection et le refactoring en général.

Unit-Test d'un Composant 1/2

Pour déclencher des événements sur le DOM (e.g. : changement d'un input de formulaire), il faut utiliser la méthode native `dispatchEvent`.

```
const input: HTMLInputElement = debugElement.nativeElement.querySelector('input');
input.value = 'test';
input.dispatchEvent(new Event('input'));
```

N'oubliez pas d'appeler la méthode `detectChanges` dès l'instanciation du composant pour initialiser le formulaire et permettre à Angular d'ajouter les bons listeners etc...

Unit-Test d'un Composant 2/2

```
describe('AppComponent', () => {  
  let component: AppComponent;  
  let fixture: ComponentFixture<AppComponent>;  
  let debugElement: DebugElement;  
  beforeEach(fakeAsync(() => {  
    TestBed.configureTestingModule({  
      imports: [RouterTestingModule],  
      declarations: [ AppComponent ]  
    });  
    fixture = TestBed.createComponent(AppComponent);  
    component = fixture.debugElement.componentInstance;  
    debugElement = fixture.debugElement  
    fixture.detectChanges();  
  }));  
  
  test('should exist', () => {  
    expect(component).toBeDefined();  
  });  
  
  test(`the title is 'my-awesome-app'`, fakeAsync(() => {  
    expect(component.title).toEqual('my-awesome-app')  
  }))  
});
```


Tips

- Difference between *it* and *test* => **The Jest docs state it is an alias of test.**
- Should I use both Cypress and Jest together? => It's very common to use Jest and Cypress in the same codebase. Jest for **Unit, Integration tests** and Cypress for **E2E tests**

Cypress for E2E testing

The screenshot displays the Cypress test runner interface. On the left, the 'Tests' panel shows a test suite named 'looks good on mobile' with a duration of 3.40. The test suite includes the following steps:

- 1. VISIT `http://bootstrap-themes.github.io/a...`
- 2. - WAIT 100
- 3. GET `img[src="assets/img/instagram_3.jpg"]`
- 4. - SCROLLINTOVIEW(duration: 1000)
- 5. - CLICK
- 6. - WAIT 100
- 7. GET `input[placeholder=Message]`
- SCROLLINTOVIEW(duration: 100)

The application being tested is a social media profile for Dave Gamache. The profile includes a profile picture, a bio, and a post with a video and images. The application is running at `http://bootstrap-themes.github.io/application/` with a resolution of 1000 x 660 (86%).

At the bottom of the screen, there is a video player interface with a play button, a progress bar, and a timestamp of 01:48. The video player also includes a volume icon, a settings gear, and a full-screen icon.

References

- <https://medium.com/@nerdic.coder/how-to-use-jest-unit-tests-with-angular-87509b500158>
- <https://semaphoreci.com/community/tutorials/testing-angular-2-and-continuous-integration-with-jest>

Learn More

[Documentations](#) · [GitHub](#) · [Showcases](#)