Angular Unit Testing performance

Nikita Yakovenko Follow
Apr 23, 2018 · 6 min read

> *Click here to share this article on LinkedIn »*

There are plenty of testing frameworks and tools available for javascript applications nowadays: Jasmine, Mocha, Chai, Karma, Wallaby.Js you name it. Some of them are more hyped and popular than others, like Jest, but at the end of the day we all have to decide which testing stack we are gonna use for our project. This is not an easy question by itself but its getting especially complicated when we are talking about already existing enterprise applications.

In this article I will try to show you what we use @ZyLab to achieve a decent execution speed of our tests for our angular application by using just good old Jasmine and Karma (**default angular test setup**).

• • •

Common Practice

I won't go too deep into details about what kinds of tests exist and how are you suppose to test your components. Lots of articles highlight this theme, e.g. you might take a look at this one written by Viktor Savkin, if you haven't read it yet I highly recommend you to do so.

Let's get back to our tests performance. ZyLab Legal Review application makes a heavy usage of Component / Integration testing. Of course we do have isolated tests and unit tests of services layer, but we are trying the get the most out of Integration testing, to minimize the required amount of e2e tests.

So how do we write these kind of tests? We might take a look at the recommended way from angular.io documentation:

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

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(BannerComponent)
    component = fixture.componentInstance;
```

This is a very simple example and at the first glance it might look like everything is fine with this code, test bed is being used to compile our module and create components.

But imagine that we have a bigger and more complex component with dozens of tests in a suite. For every run it will recompile our components and this operation will take most of our test execution time. When this value gets high enough you will be spending ~75% of you time re-compiling components for your test, but not running tests themselves.

Can we address this issue somehow? Yeah, sure we can. But first we need to understand the root of the problem.

• • •

Angular TestBed and tests execution

First of all, Angular monkey patches our testing framework like this:

```
import {resetFakeAsyncZone} from './fake_async';
import {TestBed} from './test_bed';
declare var global: any;
const _global = <any>(typeof window === 'undefined' ?
// Reset the test providers and the fake async zone be
if (_global.beforeEach) {
  _global.beforeEach(() => {
    TestBed.resetTestingModule();
```

As we can see—Angular resets your testing module for you before you run each and every one of your tests.

But what does the *TestBed.resetTestingModule* function actually do?

```
resetTestingModule() {
    clearOverrides();
    this._aotSummaries = [];
    this._templateOverrides = [];
    this._compiler = null !;
    this._moduleOverrides = [];
    this._componentOverrides = [];
    this._directiveOverrides = [];
    this._pipeOverrides = [];

    this._isRoot = true;
    this._rootProviderOverrides = [];

    this._moduleRef = null !;
    this._moduleFactory = null !;
    this._compilerOptions = [];
    this._providers = [];
    this._declarations = [];
    this._imports = [];
    this._schemas = [];
    this._instantiated = false;
    this. activeFixtures.forEach((fixture) => {
```

It cleans up all your overrides, modules, module factories and disposes all active fixtures as well. If only we could keep the compiled factories, and just re-create components and services without re-compilation.

Now we are ready to define the problem—we would like to compile angular components only once per suite, however angular forces the policy of testing module reset which leads to eventual re-compilation. It would be great to **prevent angular from cleaning compilation results** and **just reuse them**, but **still clean up all the rest** to keep tests in isolation.

If we take a closer look at other TestBed methods we might notice the usage of *_instantiated* flag (or *_initIfNeeded* method which checks the flag eventually) in most public methods.

```
1  get(token: any, notFoundValue: any = Injector.THROW_IF
2      this._initIfNeeded();
3      if (token === TestBed) {
4        return this;
5      }
6      const result = this._moduleRef.injector.get(token,
7      return result === UNDEFINED ? this._compiler.injec
8    }
9
10 …
11
12 compileComponents(): Promise<any> {
13     if (this._moduleFactory || this._instantiated) {
14       return Promise.resolve(null);
```

If we imagine that we actually preserve our factories from the previous run, then *_initIfNeeded* seems to do exactly what we are missing. If the flag is false TestBed will re-create components required for the test, create a new zone and testing module, but it won't recompile anything if moduleFactory is in place.

```
private _initIfNeeded() {
    if (this._instantiated) {
      return;
    }
    if (!this._moduleFactory) {
      try {
        const moduleType = this._createCompilerAndModu
        this._moduleFactory =
this._compiler.compileModuleAndAllComponentsSync(modul
      } catch (e) {
        const errorCompType = this._compiler.getCompon
        if (errorCompType) {
          throw new Error(
              `This test module uses the component ${s
              `Please call "TestBed.compileComponents"
        } else {
          throw e;
        }
      }
    }
    for (const {component, templateOf} of this._templa
      const compFactory = this._compiler.getComponentF
      overrideComponentView(component, compFactory);
    }
```

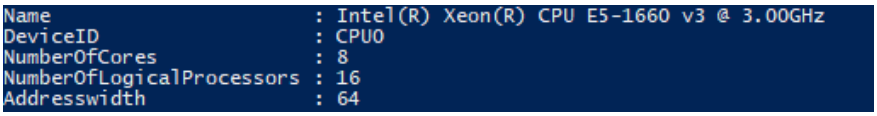
We considered all of the above with our frontend team and we were also inspired by this TestBed performance discussion, so at the end of the day we came up with the following “patch” for TestBed to meet our requirements:

```
/**
 * Reconfigures current test suit to prevent angular c
 * Forces angular test bed to re-create zone and all i
 * changing _instantiated to false after every test ru
 * Cleanups all the changes and reverts test bed confi
 */
export const configureTestSuite = () => {
  const testBedApi: any = getTestBed();
  const originReset = TestBed.resetTestingModule;

  beforeAll(() => {
    TestBed.resetTestingModule();
    TestBed.resetTestingModule = () => TestBed;
  });

  afterEach(() => {
      testBedApi. activeFixtures.forEach((fixture: Com
```

So let's do a small demo for comparison. I have a test suite prepared. It has 9 tests in it and its configuration section looks like this:

```
describe('#productions Production wizard bates setting

    beforeEach(done => (async () => {
        TestBed.configureTestingModule({
            imports: modules,
            providers: [defaultGlobalProvidersForTests
        });

        TestBed.overrideModule(ProductionsModule, over

        await TestBed.compileComponents();
    })().then(done).catch(done.fail));

    describe('number component ', () => {
        let ctx: TestCtx<BatesNumberHostComponent>;
        let page: BatesNumberPage;
```

My machine's specs are:



Machine specs

And it takes ~24 seconds to complete this test suite with the current setup:

tests execution time before patch

Now let's apply our patch finally and see how it performs. In 2 words: we need to call a *setupTestSuite* function inside our suite and replace *beforeEach* call with *beforeAll* for our suite configuration method. And after these changes code should look like this:

```
1  describe('#productions Production wizard bates setting
2
3      configureTestSuite();
4
5      beforeAll(done => (async () => {
6          TestBed.configureTestingModule({
7              imports: modules,
8              providers: [defaultGlobalProvidersForTests
9          });
10
11         TestBed.overrideModule(ProductionsModule, over
12
13         await TestBed.compileComponents();
14     })().then(done).catch(done.fail));
15
16     describe(' number component ', () => {
```

And now it takes ~8 seconds.

tests execution time after patch

It is at least **3 times faster** roughly speaking, but you should keep in mind that your results will highly depend on the number of tests in the suite. The **more tests you have per suite the more beneficial this technique will be** for you.

• • •

This is already a good step forward, our tests are running much faster now, but can we do even better?

Karma parallel tests execution

We could probably go even further if only karma supported parallel tests execution, but unfortunately for us, karma does not support it out of the box.

However, thanks to the community, we have a lot of different plugins and “karma-parallel” is one of them and exactly what we were looking for.

This package is really new at the moment, but don’t hesitate to give it a try https://www.npmjs.com/package/karma-parallel.

In short it spins up multiple instances of a browser from a single karma server. Each browser downloads all of the spec files, but when a describe block is encountered, the browsers deterministically decide if that block should run in the given browser. This leads to a way to split up unit tests across multiple browsers without changing any build processes.

We just need to tweak our karma configuration a bit, and we are ready to run.

```
    ...
    frameworks: [
        'parallel',
        'jasmine'
    ],
    plugins: [
        "karma-jasmine",
        "karma-chrome-launcher",
        "karma-junit-reporter",
        "karma-spec-reporter",
        "karma-parallel"
    ],
```

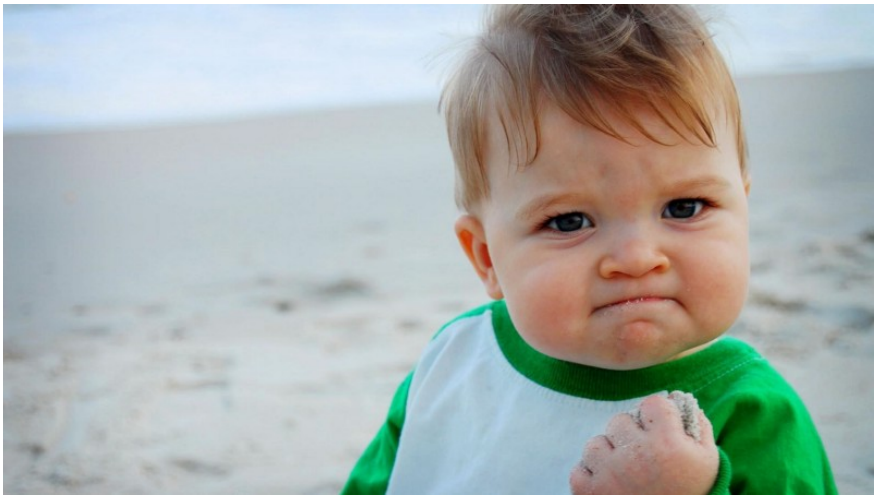
And i will just leave it here:

```
  Upload Component
Chrome 65.0.3325 (Windows 10.0.0): Executed 825 of 837 (skipped 12) SUCCESS (15 mins 36.695 secs / 14 min
s 59.337 secs)

Chrome 65.0.3325 (Windows 10.0.0): Executed 825 of 837 (skipped 12) SUCCESS (15 mins 36.695 secs / 14 min
s 59.337 secs)
TOTAL: 825 SUCCESS

Done in 942.45s.
all tests are finished
```

tests execution time before

```
  Chrome 65.0.3325 (Windows 10.0.0): Executed 110 of 110 SUCCESS (1 min 24.499 secs / 1 min 20.599 secs)
  Chrome 65.0.3325 (Windows 10.0.0): Executed 92 of 94 (skipped 2) SUCCESS (1 min 21.926 secs / 1 min 17.97
   secs)
  Chrome 65.0.3325 (Windows 10.0.0): Executed 93 of 94 (skipped 1) SUCCESS (1 min 26.773 secs / 1 min 24.12
  6 secs)
  TOTAL: 825 SUCCESS

  Done in 183.05s.
  all tests are finished
```

tests execution time after



Summary

In this article we examined two ways to improve unit tests performance for our components using just **default tools** provided by Angular—Jasmine, Karma + Angular TestBed.

- First one is a tricky **patch to angular's TestBed** which allows us to preserve our compilation results and re-use them for multiple tests per suite.
- Second one is a handy **karma-parallel plugin** for karma runner, which allows us (finally) to run our tests in parallel.

You can use them separate or together, both should provide you a good performance boost.

Thank you for reading!

Update 22.07.2018

I've released a library based on the results of this article called ng-bullet. It contains a more polished version of the technique described above + a couple of functions which should simplify your overall testing experience a bit more. I will really appreciate if you give it a try.

• • •

If you liked this, click the clap button below so other people will see this here on Medium.