Angular Testing and a Weird Trick for Faster Unit Tests (2019)

July 15, 2018 By Christian — 7 Comments

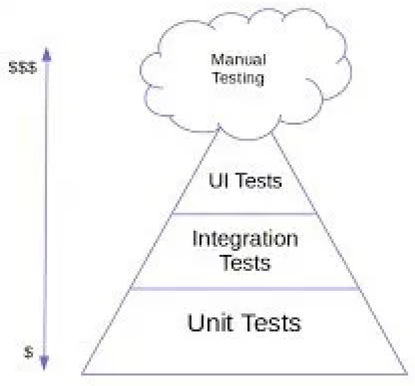
—

When doing practices like continuous delivery and continuous integration, testing becomes a more important part of the work because it's unreasonable to regression test all possible effects of a release manually. When doing five releases on a workday, you don't get to do a full manual regression testing on every release as that would easily take up one person's time for a day. Instead, you want your code to be nicely covered with **the right tests for ensuring the application work as expected.**

What are the different kinds of tests?

There are different ways to test in your application and you should always ask yourself: "What kind of test will cover this

integration tests and **5 % is UI tests.** These percentages might vary depending on who you ask but the ratio should be around the same. Note also, that we get static typing in Angular apps, which can be seen as part of the unit tests coverage, as I don't recommend we unit test every component with 100% code coverage, but instead focus on the essential areas: services, pure functions, and container components.



When going up the testing pyramid, tests become more expensive because; they are harder to create, they are harder to maintain/more flakey because they test on fragile interfaces like the UI. On the top of the pyramid is manual testing, which still has its place, even in the modern ages of automatization and DevOps, because validating a new feature for the first time requires a human to decide if it is passing the **user acceptance criteria (UAT)**. In Angular context, there are the following

Isolated tests are used to test a unit in isolation. The unit might contain some business logic that needs to be tested in isolation. When writing isolated unit tests all external dependencies should be mocked out, eg using **Spies**.

Here is an example of a service with an isolated test:

```
import { TestBed, inject } from '@angular/core/testing';
import { TodoListService } from './todo-list.service';
import { HttpClient } from '@angular/common/http';
import { TODOItem } from '../shared/models/todo-item';
import { of } from 'rxjs/observable/of';

describe('Service: TodoList', () => {
  let todoListService: TodoListService;
  const httpClientSpy: jasmine.SpyObj<HttpClient> = jasm
  httpClientSpy.get.and.returnValue(of([new TODOItem('Bu
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [TodoListService,
        {
          provide: HttpClient,
          useValue: httpClientSpy
        }]
    });
  });


  beforeEach(inject([TodoListService], (service: TodoLis
    todoListService = service;
  }));

```

todo-list.service.spec.ts hosted with ❤ by GitHub view raw

Shallow unit testing

Shallow testing is when you test a component with a template but you don't render child component by setting schema to **NO_ERRORS_SCHEMA.** This will ignore unknown tags in the template, making us not needing to import the child components. An example of a shallow tested component:

```
import { async, ComponentFixture, TestBed } from '@angula
import { BrowserModule } from '@angular/platform-browser
import { NO_ERRORS_SCHEMA } from '@angular/core';
import { TodoListComponent } from './todo-list.component
import { TodoListService } from '../core/todo-list.servi
import { TODOItem } from '../shared/models/todo-item';

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

  beforeEach(async(() => {

    const todo1 = new TODOItem('Buy milk', 'Remember to 
    todo1.completed = true;
    const todoList = [
      todo1,
      new TODOItem('Buy flowers', 'Remember to buy flowe
    ];

    TestBed.configureTestingModule({
        useValue: {todoList: todoList}
      }
    ],
    schemas: [NO_ERRORS_SCHEMA]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(TodoListComponent)
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should have two completed TODO item', () => {
    expect(component.todoList.length).toBe(2);
  });
});
```

todo-list.component.spec.ts hosted with ❤ by **GitHub** **view raw**

Alternatively, you can use the method **overrideTemplate** on the **configureTestingModule** to override the template to one that fits the test.

Integration testing

Integration testing is when you test two or more components together. This makes sense for parts of the applications where

```
import { async, ComponentFixture, TestBed } from '@angul
import { By, BrowserModule } from '@angular/platform-bro
import { TodoListCompletedComponent } from './todo-list-
import { AppComponent } from '../app.component';
import { NavbarComponent } from '../navbar/navbar.compon
import { TodoListComponent } from '../todo-list/todo-lis
import { TodoItemComponent } from '../todo-item/todo-ite
import { FooterComponent } from '../footer/footer.compon
import { AddTodoComponent } from '../add-todo/add-todo.c
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';
import { FormsModule } from '@angular/forms';
import { appRouterModule, completedTodoPath } from '../a
import { APP_BASE_HREF } from '@angular/common';
import { TodoListService } from '../core/todo-list.servi
import { TODOItem } from '../shared/models/todo-item';

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

  beforeEach(async(() => {

    const todo1 = new TODOItem('Buy milk', 'Remember to 
    todo1.completed = true;
    const todoList = [
      todo1,
      new TODOItem('Buy flowers', 'Remember to buy flowe
    ];

    TestBed.configureTestingModule({
      declarations: [
        AppComponent,
        NavbarComponent,
        TodoListComponent,
        TodoItemComponent,
      ],
      providers: [
        {provide: APP_BASE_HREF, useValue : completedTod
        {
          provide: TodoListService,
          useValue: {
            todoList: todoList
          }
        }
      ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(TodoListCompletedC
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should have one completed TODO item', () => {
    expect(component.todoList.length).toBe(1);
  });
});
```

todo-list-completed.component.integration.spec.ts hosted with ❤ by **GitHub** **view raw**

End to end testing

End to end testing means that you run the complete application together, including the backend, using a browser automatization

Out of the box a new Angular CLI project ships with the ***page object pattern***, which is a method of **separating the *stable and the fragile* part** of the end to end testing into a .po file (fragile part) and the .e2e-spec file (stable part).

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

export class AppPage {
  navigateTo() {
    return browser.get('/');
  }

  getAmountOfTODOs() {
    return element(by.css('.todo-item')).count;
  }
}
```

app.po.ts hosted with ❤ by GitHub view raw

End to end test spec

The test specification should use the page object for accessing dom elements. Protractor comes with built-in integrations for Angular such as waitForAngular, which will wait for an Angular page to be ready before doing something.

```
import { browser } from 'protractor';
import { AppPage } from './app.po';

```

```
      browser.waitForAngular().then(() => {
        expect(page.getAmountOfTODOs()).toBe(2);
        done();
      });
    });
  });
```

app.e2e-spec.ts hosted with ❤ by GitHub view raw

A word about fragile E2E tests

If you have had any experience with automatic E2E testing you know that they are fragile by nature and breaks easily.

Here are some techniques to make them more stable:

- Retry expects with a reasonable fallback and retry count, eg. 1 second retry of up to 5 retries
- Automatically rerun tests that have a fragile dependency, such as a fragile dev API
- Make page objects select using id attributes since classes are more prone to changes

A pragmatic standpoint on writing Angular tests

For this reason, **I recommend NOT to have a 100% percent code coverage** as not all tests give an equal return of investment and we want to **focus our effort on the code that is closest to the use cases and business logic** which is going to yield the highest ROI.

I recommend prioritizing your tests effort like this:

- Services – Sandboxes, business logic, NgRx effects (100% coverage)
- Pure functions – Pipes, NgRx reducers, helpers (100% coverage)
- Container component – Integration tests for happy paths (mock out services but render all components)

Having these tests in place will yield you the highest ROI. When starting to write unit tests for presentation components, the ROI is getting less and less because the tests are likely not going to save you if your component looks wrong.

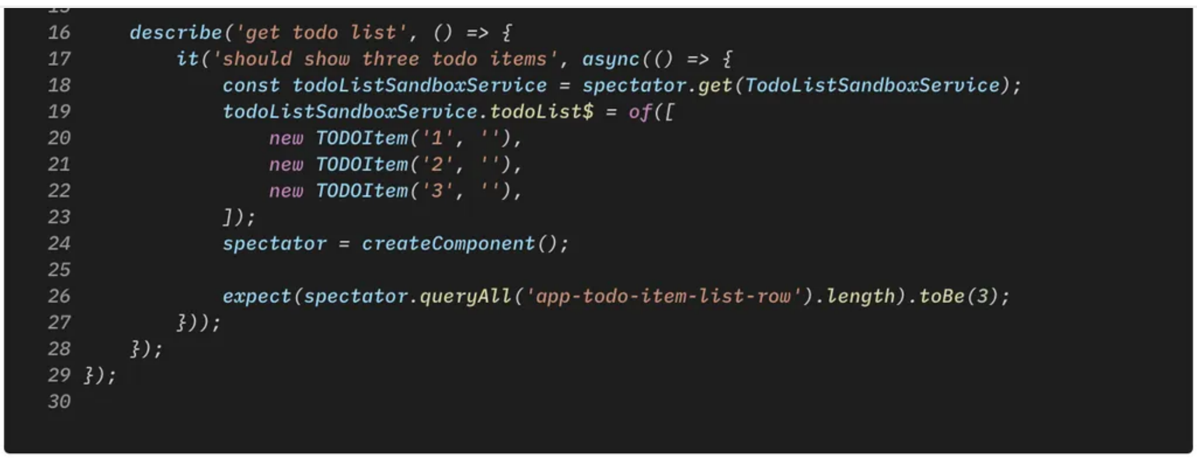
What about presentation component?

Using Jest's snapshot tests might be the best option here to test presentation components, but still, manual testing will be needed to make sure everything looks and works correctly.

Using Spectator for less boilerplate

We can get rid of a lot of the boilerplate when writing tests and also get some mock helpers by using Spectator. Spectator wraps the Angular testing methods and provides easy configuration for setting up tests and interacting with the testing API

Look how simple it can be done to create this integration test:



```
    describe('get todo list', () => {
        it('should show three todo items', async(() => {
            const todoListSandboxService = spectator.get(TodoListSandboxService);
            todoListSandboxService.todoList$ = of([
                new TODOItem('1', ''),
                new TODOItem('2', ''),
                new TODOItem('3', ''),
            ]);
            spectator = createComponent();

            expect(spectator.queryAll('app-todo-item-list-row').length).toBe(3);
        }));
    });
});
```

Pretty neat, huh?

Keeping unit tests fast by running them separately

Running unit tests should be fast, so you get the fastest feedback cycle possible when doing TDD style development. The bottleneck for Angular test executing is usually component template rendering. For this reason, I recommend postfixing integration test files with **integration.spec.ts** for distinguishing between unit tests (.spec.ts postfix). I use this for

For creating an easy facade for executing the test scripts, these are written as **npm scripts** so they can be executed with **npm run test, npm run test:unit** and **npm run test:integration** with additional variations for running in watchmode and without sourcemaps**.**

```
"scripts": {
  "ng": "ng",
  "start": "nodemon server | ng serve",
  "build": "ng build --prod",
  "test": "ng test --prod",
  "test:watch": "ng test --source-map=false -c all-wat
  "test:unit": "ng test -c unit",
  "test:unit:watch": "ng test -c unit-watch",
  "test:unit:watch:no-sm": "ng test --source-map=false
  "test:integration": "ng test -c integration",
  "test:integration:watch": "ng test -c integration-wa
  "test:integration:watch:no-sm": "ng test --source-ma
  "lint": "ng lint",
  "e2e": "ng e2e"
},
```

package.json hosted with ❤ by GitHub view raw

Setup config for unit tests

For running unit test you want to run all test that ends with **.spec** and exclude all **.integration.spec.ts:**

```
declare const require: any;

// Prevent Karma from running prematurely.
__karma__.loaded = function() {};

// First, initialize the Angular testing environment.
getTestBed().initTestEnvironment(BrowserDynamicTestingMo

// Then we find all the tests.
const context = require.context('./', true, /\.spec\.ts/

// And load the modules.
context
  .keys()
  .filter(function(element, index) {
    // The regex in require.context didn't work for filt
    return !element.endsWith('.integration.spec.ts');
  })
  .map(context);
// Finally, start Karma to run the tests.
__karma__.start();
```

test.unit.ts hosted with ❤ by GitHub view raw

Setup config for integration tests

For running integration test I like to use the convention ***.integration.spec.ts**. Ensuring that only files ending with integration.spec.ts will be run with this configuration:

```
// This file is required by karma.conf.js and loads recu

// Prevent Karma from running prematurely.
__karma__.loaded = function() {};

// First, initialize the Angular testing environment.
getTestBed().initTestEnvironment(BrowserDynamicTestingMo
// Then we find all the tests.
const context = require.context('./', true, /\.integrati
// And load the modules.
context.keys().map(context);
// Finally, start Karma to run the tests.
__karma__.start();
```

test.integration.ts hosted with ❤ by GitHub view raw

Setting up test configs in Angular.json

As of Angular 6's Angular.json file, you can specify test configurations using the **test.configurations** property. We create one for **all, unit** and **integration tests** including variations for watch mode:

```
        "test": {
          "builder": "@angular-devkit/build-angular:karm
          "options": {
            "main": "src/test.ts",
            "karmaConfig": "./karma.conf.js",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "src/tsconfig.spec.json",
            "scripts": [
            ],
            "watch": false,
            "codeCoverage": true
          },
          "configurations": {
            "unit": {
              "main": "src/test.unit.ts",
              "tsConfig": "src/tsconfig.unit.spec.json"
            },
            "unit-watch": {
              "main": "src/test.unit.ts",
              "tsConfig": "src/tsconfig.unit.spec.json",
              "watch": true
            },
            "integration": {
              "main": "src/test.integration.ts",
              "tsConfig": "src/tsconfig.spec.json"
            },
            "integration-watch": {
              "main": "src/test.integration.ts",
              "tsConfig": "src/tsconfig.spec.json",
              "watch": true
            },
            "all-watch": {
              "main": "src/test.ts",
              "tsConfig": "src/tsconfig.spec.json",
              "watch": true
            },
            "production": {
              "karmaConfig": "./karma-ci.conf.js",
              "progress": false
            }
          }
        },
        "lint": {
          "builder": "@angular-devkit/build-angular:tsli
```

```
                }
            }
        },
```

angular.json hosted with ❤ by GitHub view raw

Discovering slow tests

You can use Karma Verbose Reporter to measure the execution speed for each test and postfix them with integration.spec.ts if they are too slow so they won't impact unit test, that always should be running fast.

Wrapping up

In this post, we discussed the different ways of testing an Angular app: **isolated, shallow, integration, end to end and visual regression testing**.
We looked at a pragmatic approach to testing to **focus on services, pure functions, and container components** to ensure the highest ROI when testing.
We also saw how to keep unit tests fast by using a separate postfix for integration test spec files, setting up NPM scripts for running different types of tests and how to discover slow tests that should be moved to integration tests.

Full Name*

Type your name

Email*

Type your email

SUBMIT

Christian

Hi there!

I'm Christian, a freelance software developer helping people with Angular development. If you like my posts, make sure to follow me on Twitter.

Also published on Medium.

Share this:

Like this:

Like

Be the first to like this.

Related

In "Angular" In "Angular"

Filed Under: Angular

Tagged With: Angular, E2E, Protractor, Testing personal data Disqus collects and your choices about how it is used. All users of our service are also subject to our Terms of Service.

Proceed