

Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

code coverage



Manivel Arjunan Follow

Dec 9, 2018 · 7 min read



1. What is unit testing?
2. Why unit testing is important?
3. Angular 7 application configuration for unit testing, debug and code coverage.
4. Basic Component unit testing
5. What is a destructuring assignment and why not to use beforeEach.
6. Synchronous and Async service(Observables implementation)unit testing using TestBed, Async, fakeAsync and tick utilities.
7. What is isolation testing?
8. How to test HTTP service?
9. What are fdescribe and fit (Skip the test(s) execution).
10. Why put spec file next to the file it tests?



1. What is unit testing?

Unit testing is a **software** development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. **Unit testing** can be done manually but it is often automated.

2. Why unit testing is important?

1. Unit testing Analyze the code behavior (expected and unexpected changes).
2. It behaves as a safeguard against breaking changes. One of the best ways to keep your project bug free is through a test suite.
3. Makes developers feel guilty free on their code changes.
4. It also reveals the design mistakes.

3. Angular 7 application configuration for unit testing debug and code coverage.

3.1 configuration

The Angular CLI downloads and install everything you need to test an Angular application with the [Jasmine test framework](#).

When you create a component, directive, pipe etc through [Angular CLI](#), spec file will be created automatically with the default test suite.

Once you created a project through Angular CLI/ [cloning from a remote repository](#), go to project root directory and run the below command to run the test suite.

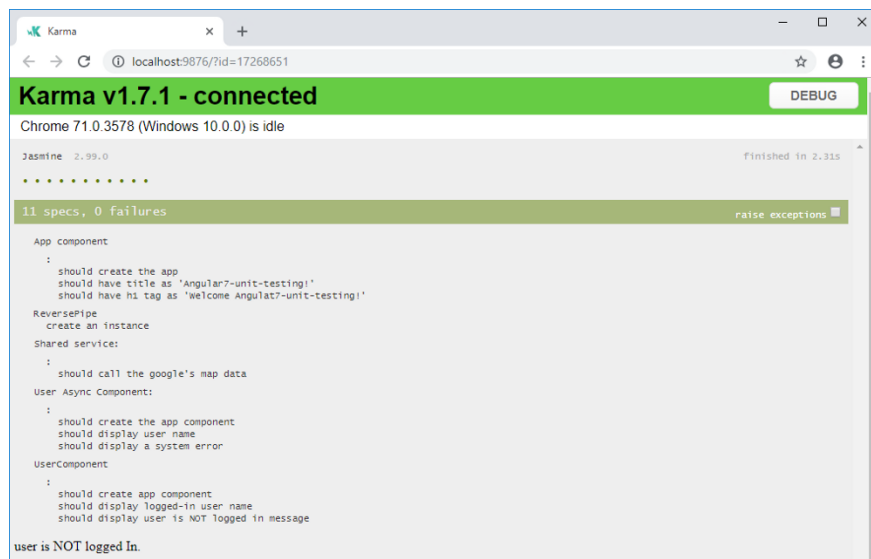
```
ng test
```

and you will see the below logs in the console.

```
10% building modules 1/1 modules 0 active
...INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/
...INFO [launcher]: Launching browser Chrome ...
...INFO [launcher]: Starting browser Chrome
...INFO [Chrome ...]: Connected on socket ...
Chrome ...: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)
```

ng test logs

Also, **ng test** opens up chrome window to view the success and failed test cases.



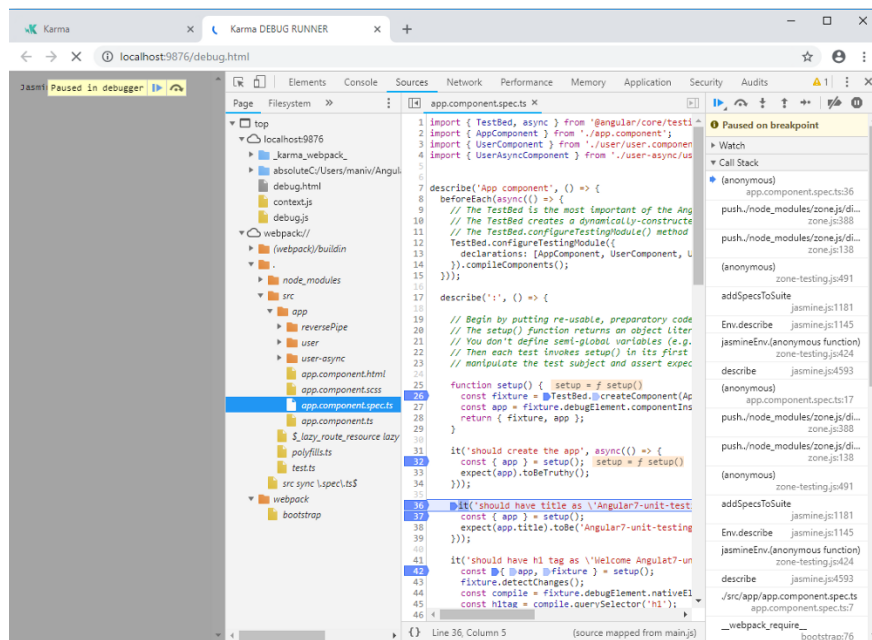
3.2 debug

Debug specs in the browser in the same way that you debug an application.

1. Reveal the karma browser window (hidden earlier).
2. Click the DEBUG button; it opens a new browser tab and re-runs the tests.
3. Open the browser's "Developer Tools" (Ctrl+Shift+I on windows; command + option + I in OSX).
4. Pick the "sources" section.

5. Open the app.component.spec.ts test file (Control/Command-P, then start typing the name of the file).
6. Set a breakpoint in the test.
7. Refresh the browser, and it stops at the breakpoint.

Below diagram depicts how to debug.



Debug the spec file

3.3 code coverage

If you want to create code-coverage reports every time you run the test suite, you can set the following option in the CLI configuration file,

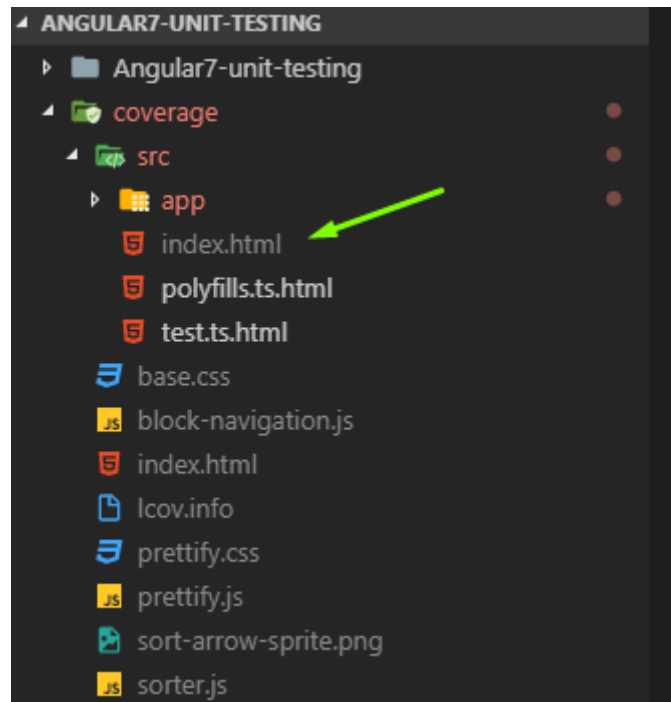
angular.json :

```
"test":{
  "options":{
    "codeCoverage": true
  }
}
```

(or) run the below command in the project root directory.

```
ng test --code-coverage
```

Whenever test suite executed successfully, you will see a “**coverage**” folder in project root directory.



coverage folder

Open the index.html file in your favorite browser to view the test case coverage for all the files in the project.

All files

100% Statements 82/82 100% Branches 2/2 100% Functions 23/23 100% Lines 67/67

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements	Branches	Functions	Lines
src	100% 82/82	100% 2/2	100% 23/23	100% 67/67
src/app	100% 7/7	100% 0/0	100% 3/3	100% 5/5
src/app/reversePipe	100% 7/7	100% 0/0	100% 2/2	100% 5/5
src/app/shared	100% 10/10	100% 0/0	100% 3/3	100% 8/8
src/app/user	100% 22/22	100% 2/2	100% 6/6	100% 18/18
src/app/user-async	100% 28/28	100% 0/0	100% 9/9	100% 23/23

istanbul code coverage report

4. Component unit testing with angular testing utilities.

What is TestBed?

The **TestBed** is the most important of the Angular testing utilities. It creates a dynamically-constructed Angular test module that emulates an Angular `@NgModule`. **TestBed.configureTestingModule()** method

takes a metadata object that can have most of the properties of an **@NgModule**.

The **TestBed** API consists of static class methods that either update or reference a *global* instance of the **TestBed**.

Internally, all static methods cover methods of the current runtime **TestBed** instance, which is also returned by the **getTestBed()** function.

Call **TestBed** methods *within* a `beforeEach()` to ensure a fresh start before each individual test.

```
beforeEach(async(() => {  
  
  TestBed.configureTestingModule({  
  
    declarations: [AppComponent, UserComponent,  
                  UserAsyncComponent]  
  
  }).compileComponents();  
  
}));
```

What is a component fixture?

The **TestBed.createComponent<T>** creates an instance of the component **T** and returns a strongly typed for that component.

The **ComponentFixture** properties and methods provide access to the component, its DOM representation, and aspects of its Angular environment.

The *fixture* methods cause Angular to perform certain tasks on the component tree. Call these methods to trigger Angular behavior in response to simulated user action.

What is DebugElement?

The **DebugElement** provides crucial insights into the component's DOM representation.

From the test root component's **DebugElement** returned by **fixture.debugElement**, you can walk (and query) the fixture's entire element and component subtrees.

Basic component testing

A component, unlike all other parts of an Angular application, combines an HTML template and a TypeScript class. The component truly is the template and the class *working together* and to adequately test a component, you should test that they work together as intended.

Such tests require creating the component's host element in the browser DOM, as Angular does, and investigating the component class's interaction with the DOM as described by its template.

Here is the example of Basis component testing using Angular TestBed utility.

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    templateUrl: './app.component.html',
6    styleUrls: ['./app.component.scss']
7  })
8  export class AppComponent implements OnInit {
9    title = 'Angular7-unit-testing!';
```

app.component.ts

```
1  <!--The content below is only a placeholder and can be repla
2  <div style="text-align:center">
3    <h1>
4      Welcome to {{ title }}!
5    </h1>
6  <app-user></app-user>
7  </div>
```

app.component.html

```

1  import { TestBed, async } from '@angular/core/testing';
2  import { AppComponent } from './app.component';
3  import { UserComponent } from './user/user.component';
4  import { UserAsyncComponent } from './user-async/user-async
5
6
7  describe('App component', () => {
8    beforeEach(async(() => {
9      // The TestBed is the most important of the Angular tes
10     // The TestBed creates a dynamically-constructed Angula
11     // The TestBed.configureTestingModule() method takes a
12     TestBed.configureTestingModule({
13       declarations: [AppComponent, UserComponent, UserAsync
14     }).compileComponents();
15   }));
16
17   describe(':', () => {
18
19     // Begin by putting re-usable, preparatory code in a se
20     // The setup() function returns an object literal with
21     // You don't define semi-global variables (e.g., let ap
22     // Then each test invokes setup() in its first line, be
23     // manipulate the test subject and assert expectations.
24
25     function setup() {
26       const fixture = TestBed.createComponent(AppComponent)
27       const app = fixture.debugElement.componentInstance;
28       return { fixture, app };
29     }
30
31     it('should create the app', async(() => {
32       const { app } = setup();
33
34       app.component.spec.ts

```

5. What is a destructuring assignment(setup method) and why is it better than using beforeEach()

Note that, in above spec file I used destructuring assignment instead of `BeforeEach`. Most test suites call ***beforeEach()*** to set the preconditions for each ***it()*** test and rely on the ***TestBed*** to create classes and inject services.

There's another school of testing that never calls ***beforeEach()*** and prefers to create classes explicitly rather than use the ***TestBed***. The

setup() function returns an object literal with the variables, such as `app`, that a test might reference. You don't define semi-global variables (e.g., `let app, fixture`) in the body of the **describe()**. Then each test invokes **setup()** in its first line, before continuing with steps that manipulate the test subject and assert expectations.

*Angular team recommends to use **setup** function instead of traditional **beforeEach()***

Here is the spec file using **traditional** `beforeEach()`.

```

1  import { TestBed, async } from '@angular/core/testing';
2  import { AppComponent } from './app.component';
3  import { UserComponent } from './user/user.component';
4  import { UserAsyncComponent } from './user-async/user-async
5
6  describe('App component', () => {
7    beforeEach(async(() => {
8      // The TestBed is the most important of the Angular tes
9      // The TestBed creates a dynamically-constructed Angula
10     // The TestBed.configureTestingModule() method takes a
11     TestBed.configureTestingModule({
12       declarations: [AppComponent, UserComponent, UserAsync
13     }).compileComponents();
14   }));
15   describe(':', () => {
16     let fixture, app;
17     // Most test suites in this guide call beforeEach() to
18     // test and rely on the TestBed to create classes and i
19     beforeEach(() => {
20       fixture = TestBed.createComponent(AppComponent);
21       app = fixture.debugElement.componentInstance;
22     });
23     it('should create the app', async(() => {
24       expect(app).toBeTruthy();

```

app.component.spec.ts

Synchronous service unit test

Testing synchronous service which is injected in a component. Injecting the real service rarely works well as most dependent services are difficult to create and control.

```

1  import { Component, OnInit } from '@angular/core';
2  import { UserService } from './user.service';
3
4  @Component({
5    selector: 'app-user',
6    templateUrl: './user.component.html',
7    styleUrls: ['./user.component.scss'],
8    providers: [UserService]
9  })
10 export class UserComponent implements OnInit {
11   user: { name: string };
12   isUserLoggedIn = false;
13   userDetails;
14   constructor(private userService: UserService) {}
15
16   ngOnInit() {

```

user.component.ts

```

1  <div *ngIf="isUserLoggedIn">
2    <p>
3      Welcome {{user.name}}
4    </p>
5  </div>
6  <div *ngIf="!isUserLoggedIn">
7    <p>

```

user.component.html

```

1  import { Injectable } from '@angular/core';
2
3  @Injectable()
4  export class UserService {
5    user = {
6      name: 'Mannie'
7    };
8

```

user.service.ts

Injecting service in a spec file.

```
const userService =  
  fixture.debugElement.injector.get(UserService);
```

Prefer spies as they are usually the easiest way to mock services.

Spy the service and return the custom values.

```
const mockUser = { name: 'Mannie' };  
  
spyOn(userService, 'getUser').and.returnValue(mockUser);
```

Update the properties, run the change detection. You must tell the **TestBed** to perform data binding by calling ***fixture.detectChanges()***

```
fixture.detectChanges();
```

```
1  import { async, TestBed } from '@angular/core/testing';
2
3  import { UserComponent } from './user.component';
4  import { UserService } from './user.service';
5
6  describe('UserComponent', () => {
7
8    beforeEach(async(() => {
9      TestBed.configureTestingModule({
10        declarations: [UserComponent]
11      });
12
13      // compileComponents(); - compileComponent is not required
14      // created with CLI and uses Web pack. it uses different
15    }));
16
17    describe(':', () => {
18
19      function setup() {
20        const fixture = TestBed.createComponent(UserComponent);
21        const component = fixture.componentInstance;
22        const userService = fixture.debugElement.injector.get(UserService);
23
24        return { fixture, component, userService };
25      }
26
27      it('should create app component', () => {
28        const { component } = setup();
29        expect(component).toBeTruthy();
30      });
31
32      it('should display logged-in user name', () => {
33        const { fixture, component, userService } = setup();
34        const mockUser = { name: 'Mannie' };
```

user.component.spec.ts

Async service(Observables implementation) unit testing

```
1  import { Component, OnInit } from '@angular/core';
2  import { UserAsyncService } from './user-async.service';
3  import { Observable } from 'rxjs';
4
5  @Component({
6    selector: 'app-user-async',
7    templateUrl: './user-async.component.html',
8    styleUrls: ['./user-async.component.scss'],
9    providers: [UserAsyncService]
10  })
11  export class UserAsyncComponent implements OnInit {
12    isLoggedIn = false;
13    user: { name: string };
14    userDetails;
15    systemError = false;
16    systemErrorMessage = '';
17    constructor(private userAsyncService: UserAsyncService) {
18
19    }
20    ngOnInit() {
21      this.userAsyncService = this.userAsyncService.getUserDe
```

user-async.component.ts

```
1  <p *ngIf="systemError">{{systemErrorMessage}}</p>
2  <div *ngIf="isLoggedIn && !systemError">
3    <p>
4      {{userDetails.name}}
5    </p>
6    <p>Note: Above user details returned by async call.</p>
7  </div>
8
9  <div *ngIf="!isLoggedIn && !systemError">
```

user-async.component.html

Creating Async service using Observable, observable returns username after 2 milliseconds

```
1  import { Observable, Observer } from 'rxjs';
2
3  export class UserAsyncService {
4    user = { name: 'Mannie' };
5    getUserDetails() {
6      // Create an observables.
7      const userObservables = Observable.create(
8        (observer: Observer<{ name: string }>) => {
9          setTimeout(() => {
10             observer.next(this.user);
11           }, 2000);
```

user-async.service.ts

tick, ***fakeAsync*** testing utilities are used to test Async service.

fakeAsync—wrap “***it***” with ***fakeAsync*** to create Async environment for test. it allows running Async task and it simulates like Async call running in the browser.

tick—Wait for Async calls finishes and successfully to access the data. Simulates the passage of time and the completion of pending asynchronous activities by flushing both *timer* and *micro-task* queues within the ***fakeAsync*** test zone.

```

1  import { TestBed, tick, fakeAsync } from '@angular/core/testing';
2  import { UserAsyncComponent } from './user-async.component';
3  import { UserAsyncService } from './user-async.service';
4  import { Observable, Observer } from 'rxjs';
5
6  describe('User Async Component:', () => {
7    beforeEach(async () => {
8      TestBed.configureTestingModule({
9        declarations: [UserAsyncComponent]
10      });
11    });
12
13    describe(':', () => {
14      function setup() {
15        const fixture = TestBed.createComponent(UserAsyncComponent);
16        const app = fixture.debugElement.componentInstance;
17        const userAsyncService = fixture.debugElement.injector.get(
18          UserAsyncService
19        );
20
21        return { fixture, app, userAsyncService };
22      }
23
24      it('should create the app component', () => {
25        const { app } = setup();
26        expect(app).toBeTruthy();
27      });
28
29      it('should display user name', fakeAsync(() => {
30        const { fixture, app, userAsyncService } = setup();
31        const mockUser = { name: 'Mannie' };
32        spyOn(userAsyncService, 'getUserDetails').and.returnValue(
33          Observable.create((observer: Observer<{ name: string }>) => {
34            observer.next(mockUser);
35            return observer;
36          })
37        );
38
39        tick();

```

user-async.component.spec.ts

7. Isolation testing

Pipes are easy to test without the Angular testing utilities. These are tests of the pipe *in isolation*. Files which are doesn't require angular dependency can be tested without angular testing utilities.

```
1  import { Pipe, PipeTransform } from '@angular/core';
2
3  @Pipe({
4    name: 'reversePipe'
5  })
6  export class ReversePipe implements PipeTransform {
7    transform(value: any, args?: any): any {
8      return value
9        .split('')
```

reverse-value.pipe.ts

Creating a test for pipe and it is an isolation testing since it doesn't depend on Angular and doesn't use testing utilities.

```
1  import { ReversePipe } from './reverse-value.pipe';
2
3  // Isolated test case.
4  describe('ReversePipe', () => {
5    it('create an instance', () => {
6      const pipe = new ReversePipe();
7      expect(pipe).toBeTruthy();
```

reverse-pipe.pipe.spec.ts

8. Testing HTTP service

Data services that make HTTP calls to remote servers typically inject and delegate to the Angular *HTTPClient* service for XHR calls.

[view raw](#)

17/20

```

39      });
40
41      afterEach(() => {
42          const { httpTestingModule } = setup();

```

9. **fdescribe** and **fit** (skip the test or run only specific tests)

Add **fdescribe** to run the test for a spec file or a test suite. It skips all other files or runs test for the specified file.

```

fdescribe('UserComponent', () => {

  beforeEach(async() => {

    TestBed.configureTestingModule({

      declarations: [UserComponent]

    });

```

Add **fit** to run the test for “**it**” level. It skips all other tests and runs only one test.

```

fit('should create app component', () => {

  const { component } = setup();

  expect(component).toBeTruthy();

});

```


10. Why put spec file next to the file it tests?

It's a good idea to put unit test spec files in the same folder as the application source code files that they test:

- Such tests are easy to find.
- You see at a glance if a part of your application lacks tests.
- Nearby tests can reveal how a part works in context.

- When you move the source (inevitable), you remember to move the test.
- When you rename the source file (inevitable), you remember to rename the test file.

Below is the link for Github projects. It contains all the configuration and code explained above.

manivelarjunan/Angular7-unit-testing Angular 7 application with unit testing and code coverage - manivelarjunan/Angular7-unit-testing github.com	
---	---

Below is the link for Angular official documentation for unit testing.

Angular Docs Edit description angular.io	
---	--

