



# Testing in Angular

---

.NET

*Testing an application offers the benefits of improved application design, preemptive refactoring, and helps prevent breaking legacy code when adding new features*

[HTTPS://MEDIUM.COM/SWLH/ANGULAR-UNIT-TESTING-JASMINE-KARMA-STEP-BY-STEP-E3376D110AB4](https://medium.com/swlh/angular-unit-testing-jasmine-karma-step-by-step-e3376d110ab4)

# Testing in Angular – Overview

<https://angular.io/guide/testing>

[https://jasmine.github.io/pages/docs\\_home.html](https://jasmine.github.io/pages/docs_home.html)

---

When you run **ng new** in the command line, the *Angular CLI* downloads, installs, and configures everything needed to test an *Angular* application with the *Jasmine* test framework and *Karma* test runner.

Any default project is immediately ready to test and have test suites built.

The **ng test** command builds the app in “watch mode” and launches the *Karma* test runner.

*Jasmine* and *Karma* work together to run your tests and output test results to the command line and to a browser window that displays the results of the tests.

**ng test** also watches for any code changes and re-runs the tests every time there are new saved changes to the code base.



# Jasmine Testing Framework

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

<https://jasmine.github.io/setup/nodejs.html>

---

**Jasmine** is a development framework for testing **JavaScript** code. It does not depend on any other **JavaScript** frameworks. It does not require a DOM and it has a simple syntax for easily written tests.

**Jasmine** is installed along with the default **Angular** Application created with **ng new**.

You can also add it locally to your project as a **Node.js** package.

There is also a global install. Click [here](#) to learn how.

1. If needed, install Jasmine locally.
  - `npm install --save-dev jasmine`
2. Initialize a Jasmine project with default configuration.
  - `npx jasmine init`
3. Generate Jasmine folders and spec (test) files.
  - `jasmine examples`
4. Run tests.
  - `npx jasmine`

# Karma Test Runner

<https://karma-runner.github.io/latest/index.html>

---

**Jasmine** and **Karma** work together to run tests and display results.

**Karma** launches a browser window with a context page that spawns a web server that executes source code against test code. The results of each test are displayed on the command line to the developer.

**Jasmine** runs the 'spec' tests and reports results by messaging through the client page. **Karma** watches all files specified in its configuration file.

When changes are saved to these files, **Karma** reports the tests again.



# Karma - Installation

<https://karma-runner.github.io/latest/index.html>

<https://karma-runner.github.io/5.0/intro/installation.html>

---

*Jasmine* and *Angular* install *Karma* automatically. *Angular* does this along with the default *Angular* Application created with **ng new**.

If needed, *karma*, *karma-jasmine*, *karma-chrome-launcher* and *jasmine-core* can be installed as npm packages into *node\_modules* in your current working directory and saved as devDependencies in **package.json**. When installing *Karma* manually, it is recommended to install locally in the target project's root directory.

1. Install Karma in the app root directory
  - **npm install karma --save-dev**
2. Install plugins that your project needs
  - **npm install karma-jasmine karma-chrome-launcher jasmine-core --save-dev**
3. Run Karma:
  - **./node\_modules/karma/bin/karma start**



# Angular Testing – Set-up

<https://angular.io/guide/testing#test-file-name-and-location>

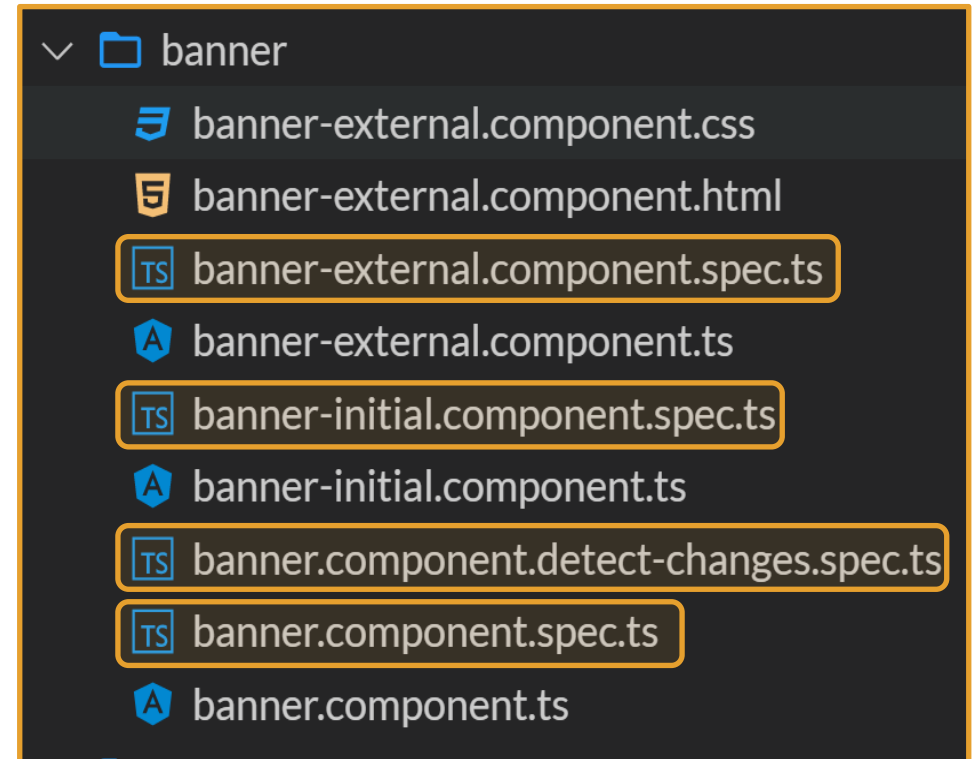
---

Testing best practices:

- Put **Unit Test** “spec” files in the same folder as the component or source code files they test.
- Put **Integration Test** “spec” files in a general “Tests” folder.
- Implement Continuous Integration with **Circle CI** or **Travis CI**.

All test files have extension **.spec.ts**.

This example shows the **banner** component. It contains all the associated files for the component.



# Testing Components

<https://angular.io/guide/testing-components-basics#cli-generated-tests>

[https://jasmine.github.io/tutorials/mocking\\_ajax](https://jasmine.github.io/tutorials/mocking_ajax)

---

ng generate component  
<componentName> creates a  
new component in the  
app/<componentName> folder  
with a .css, .html, .spec.ts, and  
.ts files preconfigured.

The testing file is the .spec.ts file.  
It has testing preconfigured.

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { BannerComponent } from './banner.component';

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

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ] // declares which component to test
    })
    .compileComponents();
  }));

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

  it('should create', () => {
    expect(component).toBeDefined(); // verify the component is created
  });
});
```



# Angular Testing Template

<https://angular.io/guide/testing-components-basics#cli-generated-tests>  
<https://medium.com/swlh/angular-unit-testing-jasmine-karma-step-by-step-e3376d110ab4>

1. The whole Angular test suite is contained within **describe()**. **describe()** has two parameters:

1. A string. It declares the function or component to test.
2. A callback function containing the actions to take.

2. **describe()**'s callback function has zero parameters but lists all the actions to take on the component.

- **beforeEach()** sets up the test environment.

- **it()** declares an individual test. **it()** has two parameters:

1. The first is a string that states the expected result.
2. The second contains a callback with an **expect()** assertion.

**expect()** is similar to **assert()** in C#.

```
describe('BannerComponent (inline template)', () => {
  let component: BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;
  let h1: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ],
    });
    fixture = TestBed.createComponent(BannerComponent);
    // BannerComponent test instance
    component = fixture.componentInstance;
    h1 = fixture.nativeElement.querySelector('h1');
  });

  it('no title in DOM after createComponent()', () => {
    expect(h1.textContent).toEqual('');
  });

  it('should display original title', () => {
    fixture.detectChanges();
    expect(h1.textContent).toContain(component.title);
  });
});
```

# beforeEach(), beforeAll()

<https://angular.io/guide/testing-components-basics#beforeeach>  
<https://jasmine.github.io/api/edge/global.html#beforeAll>  
<https://jasmine.github.io/api/edge/global.html#beforeEach>

Rather than duplicate the **TestBed** configuration for each test, *Jasmine* provides **beforeEach()** and **beforeAll()** to set up the testing environment. **beforeEach()** runs before each **it()**. **beforeAll()** runs just once before all **it()**'s.

Here, in **beforeEach()**,

- **Testbed** declares which component will be tested.
- **Testbed** creates **fixture**, a complete component environment.
- Extracts the **component** class from **fixture**.
- Extracts the 'h1' tag from the component .html.

```
describe('BannerComponent (inline template)', () => {
  let component: BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;
  let h1: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ],
    });
    fixture = TestBed.createComponent(BannerComponent);
    // BannerComponent test instance
    component = fixture.componentInstance;
    h1 = fixture.nativeElement.querySelector('h1');
  });

  it('no title in DOM after createComponent()', () => {
    expect(h1.textContent).toEqual('');
  });

  it('should display original title', () => {
    fixture.detectChanges();
    expect(h1.textContent).toContain(component.title);
  });
});
```

# it() and expect()

[https://jasmine.github.io/tutorials/your\\_first\\_suite](https://jasmine.github.io/tutorials/your_first_suite)  
<https://jasmine.github.io/api/edge/global.html#it>

---

**Specs** (tests) are defined by calling the *Jasmine* function **it()**. **it()** takes a string and a callback function.

- The string serves as the title of the spec and describes what the spec does and expects.
- The callback function is the actual **spec**. It contains one or more **expect()** assertions.

An **expect()** is an assertion that evaluates to true or false.

- **expect()** takes an “actual” value to be evaluated.
- It is chained with a “Matcher” function (**toBe()**, **toEqual()**, **toHaveBeenCalled()**, etc), which takes the “expected” value.
- A **spec** with all true expectations passes.
- If there is even one false **expect()**, the whole **spec** fails.

```
describe("A test suite is a function",  
() => {  
  var a;  
  it("and so is a spec", () => {  
    a = true;  
    expect(a).toBe(true);  
    expect(a).not.toBe(false);  
  });  
});
```

# Testing with TestBed

<https://angular.io/guide/testing-services#testing-services-with-the-testbed>

<https://angular.io/guide/testing-utility-apis#testbed-class-summary>

<https://duncanhunter.gitbook.io/testing-angular/testbed-and-fixtures>

**TestBed API** is an **Angular** testing utility. **TestBed** creates a dynamically-constructed **Angular module** (for testing) that emulates (mocks) a component and its dependencies.

Use **TestBed** when:

- tests require creating the **component's** host element in the browser DOM and verifying the components interaction with it.
- You need to create a component and its dependencies (services) all at once.

**.configureTestingModule()** takes optional arrays of providers (services), declarations (components), imports, and schemas. It provides static **component** class methods that either update or reference a global instance of the **TestBed**.

```
type TestModuleMetadata = {  
  providers?: any[];  
  declarations?: any[];  
  imports?: any[];  
  schemas?: Array<SchemaMetadata | any[]>;  
};
```

```
13 beforeEach(() => {  
14   TestBed.configureTestingModule({  
15     declarations: [AppComponent],  
16     providers: [  
17       {  
18         provide: AppService,  
19         useValue: { getNames: () => (of([])) }  
20       }  
21     ]  
22   });  
23   fixture = TestBed.createComponent(AppComponent);  
24   component = fixture.componentInstance;  
25   appService = TestBed.get(AppService);  
26 });  
27  
28 it('add 1+1 - PASS', () => {  
29   expect(1 + 1).toEqual(2);  
30 });
```

# Mock vs Spy vs Stub

<https://www.amadousall.com/unit-testing-angular-stubs-vs-spies-vs-mocks/>  
<https://martinfowler.com/bliki/TestDouble.html>

---

Mock	Spy	Stub
<p>Jasmine does not natively have mocking behavior.</p> <p>To mock means to provide an implementation of a function, class, or service as a dependency of the item under test. The SUT will use the mocked implementation instead of its normal dependency.</p> <p>The emphasis with mocks is that the SUT is calling the correct dependencies.</p> <p>The mock has its own verification mechanisms.</p>	<p>A Spy is a special stub that intercepts the call to a dependency function and returns a predefined value. Spies give additional information like</p> <ul style="list-style-type: none"><li>• how many times the spy was called,</li><li>• what args were sent, etc.</li></ul> <p>The emphasis with spies is to verify that the SUT is calling the correct dependencies.</p>	<p>The mocked instance that replaces the dependency. The stub is used only for the unit test.</p> <p>The emphasis with stubs is verifying that the SUT does the correct things with the data received from its dependencies.</p>

# Spies

<https://jasmine.github.io/api/edge/Spy>

<https://scriptverse.academy/tutorials/jasmine-spyon.html>

[https://www.tutorialspoint.com/jasminejs/jasminejs\\_spies.htm](https://www.tutorialspoint.com/jasminejs/jasminejs_spies.htm)

---

If you want to test a function without actually calling its dependency function, Jasmine provides `spyOn()`. A **Spy** can stub any function and track calls to it and all its arguments. `spyOn()` is used to **stub** a function. `spyOn()` takes two parameters:

1. The name of the object to be acted upon.
2. The name of the method to be spied upon.

`spyOn()` replaces the spied function with a **stub** (a stand-in function). It does not execute the real method called. `spyOn()` can only be called on existing methods.

In this example, we see the original function above and the **spec** below. The function `nextNumber` is spied so what is being called when `s.getNextNumber` is invoked is not the actual function but a temporary stub that returns 10 instead of 3.

```
function Number() {  
  this.number = 2;  
  this.nextNumber = () => {  
    this.number = this.number + 1;  
    return this.number;  
  },  
  this.getNextNumber = () => {  
    return this.nextNumber();  
  }  
};
```

```
describe('spyOn() Number', () => {  
  it('should be 10', () => {  
    let s = new Number();  
    spyOn(s, 'nextNumber').and.returnValue(10);  
    s.getNextNumber();  
    expect(s.number).toEqual(10);  
  });  
});
```



# Spying on Properties

[https://jasmine.github.io/tutorials/spying\\_on\\_properties](https://jasmine.github.io/tutorials/spying_on_properties)

<https://jasmine.github.io/api/2.7/global.html#spyOnProperty>

---

In Jasmine, anything a **property spy** can do, can also be done with a **function spy**, but potentially with different syntax. Use `spyOnProperty()` to create a “getter” or a “setter”. `spyOnProperty()` returns a spy.

`spyOnProperty(object, propertyName, accessType?)` has three parameters:

1. an object (the **component class**) as it's first parameter.
2. the name (as a string) of the property.
3. the type of **spy** (“**get**” or “**set**”) as it's third.

```
it("allows you to create spies for either type", function() {  
  spyOnProperty(someObject, "myValue", "get").and.returnValue(30);  
  spyOnProperty(someObject, "myValue", "set").and.callThrough();  
});
```

# Spying on Properties

[https://jasmine.github.io/tutorials/spying\\_on\\_properties](https://jasmine.github.io/tutorials/spying_on_properties)

<https://jasmine.github.io/api/2.7/global.html#spyOnProperty>

---

You cannot refer to a property without calling its “getter” method. To gain access to the value of an existing **spy**, save a **reference** to the **spy** for later changes.

```
beforeEach(function() {  
  this.propertySpy = spyOnProperty(someObject, "myValue", "get").and.returnValue(1);  
});  
  
it("lets you change the spy strategy later", function() {  
  this.propertySpy.and.returnValue(3);  
  expect(someObject.myValue).toEqual(3);  
});
```

Create a spy object with several properties by passing an array of properties as a third argument to **createSpyObj()**.

To change the spy property value, use **Object.getOwnPropertyDescriptor()**.

```
it("creates a spy object with properties", function() {  
  let obj = createSpyObj("myObject", {}, { x: 3, y: 4 });  
  expect(obj.x).toEqual(3);  
  
  Object.getOwnPropertyDescriptor(obj, "x").get.and.returnValue(7);  
  expect(obj.x).toEqual(7);  
});
```

# Testing Reactive Forms

<https://angular.io/guide/forms-overview#testing>

<https://angular.io/guide/forms-overview#testing-reactive-forms>

<https://codecraft.tv/courses/angular/unit-testing/model-driven-forms/>

---

**Reactive** forms can be tested without rendering the UI because they provide synchronous access to the *form models* and *data models*.

In the below spec, status and data are queried and manipulated through the **control** without interacting with the *change detection cycle*.

This test verifies the data flow from template to model in a **reactive** Form.

```
it('should update the value of the input field', () => {
  const input = fixture.nativeElement.querySelector('input'); //Grab "input" element from the form.
  const event = createNewEvent('input'); //create an 'input' type event.

  input.value = 'Red'; //Set value of input element to 'Red'.
  input.dispatchEvent(event); //Fire off the event.

  //assert that the value of 'favoriteColor' in the model now matches the input.
  expect(fixture.componentInstance.favoriteColor.value).toEqual('Red');
});
```

# Testing Template-driven Forms

<https://angular.io/guide/forms-overview#testing>

<https://angular.io/guide/forms-overview#testing-template-driven-forms>

<https://codecraft.tv/courses/angular/unit-testing/model-driven-forms/>

---

Writing tests with **template-driven** forms requires a detailed knowledge of the change detection process and an understanding of how **directives** run on each cycle to ensure that elements are queried, tested, or changed at the correct time.

This test verifies the data flows from view to model and model to view for a **template-driven** form.

```
it('should update the favorite color in the component', fakeAsync(() => {  
  const input = fixture.nativeElement.querySelector('input'); //Grab the "input" element from the form.  
  const event = createNewEvent('input'); //create an 'input' type event.  
  input.value = 'Red'; //Set the value of the input to 'Red'.  
  input.dispatchEvent(event); //Fire off the event.  
  
  fixture.detectChanges(); //tell fixture to detect changes  
  expect(component.favoriteColor).toEqual('Red'); //assert the value in the model now matches the input.  
}));
```

# Testing HTTP Services with Spies

<https://stackblitz.com/angular/mkjgxjnxek?file=src%2Fapp%2Fmodel%2Fhero.service.spec.ts>

<https://angular.io/guide/testing-services#testing-http-services>

<https://levelup.gitconnected.com/test-angular-components-and-services-with-http-mocks-e143d90fa27d>

---

**Services** that make HTTP calls to remote servers typically inject and delegate to an *Angular HttpClient* service. You can test a data service with an injected *HttpClient spy* as you would test any service with a dependency.

```
describe('HeroesService (with spies)', () => {
  let httpClientSpy: { get: jasmine.Spy };
  let heroService: HeroService;

  beforeEach(() => {
    // TODO: spy on other methods too
    httpClientSpy = jasmine.createSpyObj('HttpClient', ['get']);
    heroService = new HeroService(httpClientSpy as any);
  });

  it('should return expected heroes (HttpClient called once)', () => {
    const expectedHeroes: Hero[] =
      [{ id: 1, name: 'A' }, { id: 2, name: 'B' }];
    httpClientSpy.get.and.returnValue(of(expectedHeroes));
    heroService.getHeroes().subscribe(
      heroes => expect(heroes).toEqual(expectedHeroes, 'expected heroes'), fail
    );
    expect(httpClientSpy.get.calls.count()).toBe(1, 'one call');
  });
});
```

# Mocking AJAX calls

[https://jasmine.github.io/tutorials/mocking\\_ajax](https://jasmine.github.io/tutorials/mocking_ajax)

```
describe("mocking ajax", function() {  
  it("allows use in a single spec", function() {  
    let doneFn = jasmine.createSpy('success');           //this is the mock response  
    // .withMock() takes a function that's called after AJAX has been mocked and is  
    // automatically uninstalled afterwards.  
    jasmine.Ajax.withMock(function() {                  //this is normal AJAX setup  
      let xhr = new XMLHttpRequest();                    //but using .withMock()  
      xhr.onreadystatechange = function(args) {  
        if (this.readyState == this.DONE) {  
          doneFn(this.responseText);  
        }  
      };  
      xhr.open("GET", "/reature.com/associates");        //set up the request  
      xhr.send();                                         //send the request  
      expect(doneFn).not.toHaveBeenCalled();  
      jasmine.Ajax.requests.mostRecent().respondWith({ //tell 'doneFn' how to respond  
        "status": 200,  
        "responseText": 'successful mock!'  
      });  
      expect(doneFn).toHaveBeenCalled('successful mock!');//assert  
    });  
  });  
});
```



# Mock a Service

When a component that consumes a service is under test, any time a function under test that uses the service is called, that service needs to be mocked to properly isolate the function.

In order to mock a service, you must declare a substitute service class that contains the methods expected to be called in the test suites along with a mock implementation.

Then, in `TestBed.configureTestingModule()`, in the `providers[]`, add an object that states the service name and substitute class. The substitute class will be called instead of the service during the test suite.

```
class memesaverServiceStub {
  login(): Observable<StringPerson> {
    return of({
      personId: 'guid', fname: 'Mwq', lname: 'A',
      userName: 'R', passwordHash: 'K', memberSince: new Date(),
      memes: [], memesILiked: []
    });
  }
  EditPerson(): Observable<boolean> {
    return of(false);
  }
};

beforeEach(async () => {
  await TestBed.configureTestingModule([
    {
      provide: MemesaverService, useClass: memesaverServiceStub
    }
  ]).compileComponents();
});

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

it('should return false to myBool from ChangedPerson()', () => {
  const editPerson: EditPerson = {
    personId: 'guid', fname: 'M', lname: 'A',
    username: 'R', newPassword: 'O', passwordHash: 'K', newUsername: 'O'
  };
  component.ChangedPerson(editPerson);
  expect(component.myBool).toBeFalsey;
  expect(component.ChangedPerson(editPerson)).toHaveBeenCalled(); // make su
});

it('should give \'stringPerson\' the fname of "M"', () => {
  component.login();
  expect(component.stringPerson.fname).toBe("Mwq");
});
```

# Testing pipe()

<https://angular.io/guide/testing-pipes>

All *pipe classes* have one method, *transform*. Transform converts the input value into a transformed output value.

The *transform* implementation rarely interacts with the DOM and most pipes have no dependence on Angular other than the *@Pipe* metadata and an interface

Consider a **TitleCasePipe** that capitalizes the first letter of each word. Here's an implementation with a regular expression.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'titlecase', pure: true})
/** Transform to Title Case: uppercase the first letter of
the words in a string. */
export class TitleCasePipe implements PipeTransform {
  transform(input: string): string {
    return input.length === 0 ? '' :
      input.replace(/\w\S*/g, (txt => txt[0].toUpperCase()
+ txt.substr(1).toLowerCase() ));
  }
}
```

```
describe('TitleCasePipe', () => {
  // This pipe is a pure, stateless function so no need for
  beforeEach

  const pipe = new TitleCasePipe();

  it('transforms "abc" to "Abc"', () => {
    expect(pipe.transform('abc')).toBe('Abc');
  });

  it('transforms "abc def" to "Abc Def"', () => {
    expect(pipe.transform('abc def')).toBe('Abc Def');
  });

  // ... more tests ...
});
```