**Introduction to Angular Testing**

This page guides you through writing tests to explore and confirm the behavior of the application. Testing does the following:

1. Guards against changes that break existing code (“regressions”).
2. Clarifies what the code does both when used as intended and when faced with deviant conditions.
3. Reveals mistakes in design and implementation. Tests shine a harsh light on the code from many angles. When a part of the application seems hard to test, the root cause is often a design flaw, something to cure now rather than later when it becomes expensive to fix.

**Tools and technologies**

You can write and run Angular tests with a variety of tools and technologies. This guide describes specific choices that are known to work well.

|  |  |
| --- | --- |
| **Technology** | **Purpose** |
| Jasmine | The [Jasmine test framework](http://jasmine.github.io/2.4/introduction.html) provides everything needed to write basic tests. It ships with an HTML test runner that executes tests in the browser. |
| Angular testing utilities | Angular testing utilities create a test environment for the Angular application code under test. Use them to condition and control parts of the application as they interact *within* the Angular environment. |
| Karma | The [karma test runner](https://karma-runner.github.io/1.0/index.html) is ideal for writing and running unit tests while developing the application. It can be an integral part of the project's development and continuous integration processes. This guide describes how to set up and run tests with karma. |
| Protractor | Use protractor to write and run *end-to-end* (e2e) tests. End-to-end tests explore the application *as users experience it*. In e2e testing, one process runs the real application and a second process runs protractor tests that simulate user behavior and assert that the application respond in the browser as expected. |

**Setup**

There are two fast paths to getting started with unit testing.

1. Start a new project following the instructions in [Setup](https://v2.angular.io/docs/ts/latest/guide/setup.html).
2. Start a new project with the [Angular CLI](https://github.com/angular/angular-cli/blob/master/README.md).

Both approaches install npm packages, files, and scripts pre-configured for applications built in their respective modalities. Their artifacts and procedures differ slightly but their essentials are the same and there are no differences in the test code.

In this guide, the application and its tests are based on the [setup instructions](https://v2.angular.io/docs/ts/latest/guide/setup.html). For a discussion of the unit testing setup files, [see below](https://v2.angular.io/docs/ts/latest/guide/testing.html#setup-files).

**Isolated unit tests vs. the Angular testing utilites**

[Isolated unit tests](https://v2.angular.io/docs/ts/latest/guide/testing.html#isolated-unit-tests) examine an instance of a class all by itself without any dependence on Angular or any injected values. The tester creates a test instance of the class with new, supplying test doubles for the constructor parameters as needed, and then probes the test instance API surface.

*You should write isolated unit tests for pipes and services.*

You can test components in isolation as well. However, isolated unit tests don't reveal how components interact with Angular. In particular, they can't reveal how a component class interacts with its own template or with other components.

Such tests require the **Angular testing utilities**. The Angular testing utilities include the TestBed class and several helper functions from @angular/core/testing. They are the main focus of this guide and you'll learn about them when you write your [first component test](https://v2.angular.io/docs/ts/latest/guide/testing.html#simple-component-test). A comprehensive review of the Angular testing utilities appears [later in this guide](https://v2.angular.io/docs/ts/latest/guide/testing.html#atu-apis).

But first you should write a dummy test to verify that your test environment is set up properly and to lock in a few basic testing skills.

[Back to top](https://v2.angular.io/docs/ts/latest/guide/testing.html#top)

**The first karma test**

Start with a simple test to make sure that the setup works properly.

Create a new file called 1st.spec.ts in the application root folder, src/app/

Tests written in Jasmine are called *specs* . **The filename extension must be .spec.ts**, the convention adhered to by karma.conf.js and other tooling.

**Put spec files somewhere within the src/app/ folder.** The karma.conf.js tells karma to look for spec files there, for reasons explained [below](https://v2.angular.io/docs/ts/latest/guide/testing.html#q-spec-file-location).

Add the following code to src/app/1st.spec.ts.

**src/app/1st.spec.ts**

Copy Code

describe('1st tests', () => {

it('true is true', () => expect(true).toBe(true));

});

**Run with karma**

Compile and run it in karma from the command line using the following command:

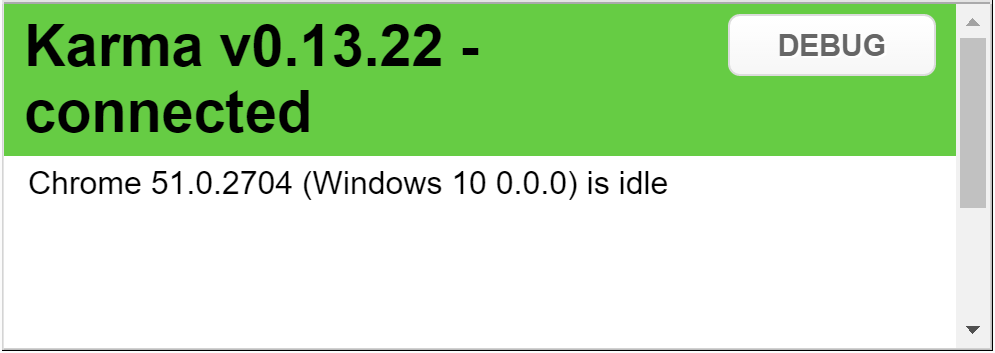
Copy Code

npm test

The command compiles the application and test code and starts karma. Both processes watch pertinent files, write messages to the console, and re-run when they detect changes.

The documentation setup defines the test command in the scripts section of npm's package.json. The Angular CLI has different commands to do the same thing. Adjust accordingly.

After a few moments, karma opens a browser and starts writing to the console.



Hide (don't close!) the browser and focus on the console output, which should look something like this:

Copy Code

> npm test

...

[0] 1:37:03 PM - Compilation complete. Watching for file changes.

...

[1] Chrome 51.0.2704: Executed 0 of 0 SUCCESS

Chrome 51.0.2704: Executed 1 of 1 SUCCESS

SUCCESS (0.005 secs / 0.005 secs)

Both the compiler and karma continue to run. The compiler output is preceded by [0]; the karma output by [1].

Change the expectation from true to false.

The *compiler* watcher detects the change and recompiles.

Copy Code

[0] 1:49:21 PM - File change detected. Starting incremental compilation...

[0] 1:49:25 PM - Compilation complete. Watching for file changes.

The *karma* watcher detects the change to the compilation output and re-runs the test.

Copy Code

[1] Chrome 51.0.2704 1st tests true is true FAILED

[1] Expected false to equal true.

[1] Chrome 51.0.2704: Executed 1 of 1 (1 FAILED) (0.005 secs / 0.005 secs)

It fails of course.

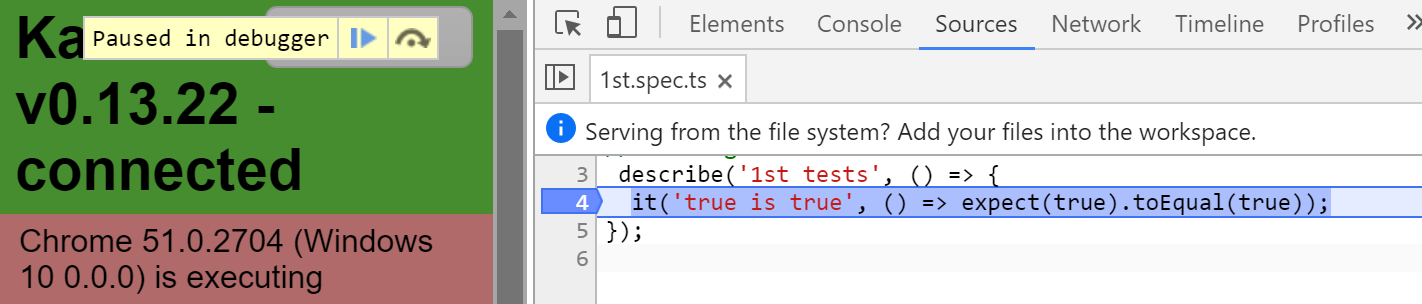
Restore the expectation from false back to true. Both processes detect the change, re-run, and karma reports complete success.

The console log can be quite long. Keep your eye on the last line. When all is well, it reads SUCCESS.

**Test debugging**

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 1st.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.



**Try the live example**

You can also try this test as a [live example](https://v2.angular.io/resources/live-examples/testing/ts/1st-specs.eplnkr.html) / [downloadable example](https://v2.angular.io/resources/zips/testing/1st-specs.testing.zip) in plunker. All of the tests in this guide are available as [live examples](https://v2.angular.io/docs/ts/latest/guide/testing.html#live-examples).

[Back to top](https://v2.angular.io/docs/ts/latest/guide/testing.html#top)

**Test a component**

An Angular component is the first thing most developers want to test. The BannerComponent in src/app/banner-inline.component.ts is the simplest component in this application and a good place to start. It presents the application title at the top of the screen within an <h1> tag.

**src/app/banner-inline.component.ts**

Copy Code

import { Component } from '@angular/core';

@Component({

selector: 'app-banner',

template: '<h1>{{title}}</h1>'

})

export class BannerComponent {

title = 'Test Tour of Heroes';

}

This version of the BannerComponent has an inline template and an interpolation binding. The component is probably too simple to be worth testing in real life but it's perfect for a first encounter with the Angular testing utilities.

The corresponding src/app/banner-inline.component.spec.ts sits in the same folder as the component, for reasons explained in the [FAQ](https://v2.angular.io/docs/ts/latest/guide/testing.html#faq) answer to ["Why put specs next to the things they test?"](https://v2.angular.io/docs/ts/latest/guide/testing.html#q-spec-file-location).

Start with ES6 import statements to get access to symbols referenced in the spec.

**src/app/banner-inline.component.spec.ts (imports)**

Copy Code

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { By } from '@angular/platform-browser';

import { DebugElement } from '@angular/core';

import { BannerComponent } from './banner-inline.component';

Here's the describe and the beforeEach that precedes the tests:

**src/app/banner-inline.component.spec.ts (beforeEach)**

Copy Code

describe('BannerComponent (inline template)', () => {

let comp: BannerComponent;

let fixture: ComponentFixture<BannerComponent>;

let de: DebugElement;

let el: HTMLElement;

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [ BannerComponent ], // declare the test component

});

fixture = TestBed.createComponent(BannerComponent);

comp = fixture.componentInstance; // BannerComponent test instance

// query for the title <h1> by CSS element selector

de = fixture.debugElement.query(By.css('h1'));

el = de.nativeElement;

});

});

***TestBed***

TestBed is the first and most important of the Angular testing utilities. It creates an Angular testing module—an @NgModule class—that you configure with the configureTestingModule method to produce the module environment for the class you want to test. In effect, you detach the tested component from its own application module and re-attach it to a dynamically-constructed Angular test module tailored specifically for this battery of tests.

The configureTestingModule method takes an @NgModule-like metadata object. The metadata object can have most of the properties of a normal [Angular module](https://v2.angular.io/docs/ts/latest/guide/ngmodule.html).

*This metadata object* simply declares the component to test, BannerComponent. The metadata lack imports because (a) the default testing module configuration already has what BannerComponent needs and (b) BannerComponent doesn't interact with any other components.

Call configureTestingModule within a beforeEach so that TestBed can reset itself to a base state before each test runs.

The base state includes a default testing module configuration consisting of the declarables (components, directives, and pipes) and providers (some of them mocked) that almost everyone needs.

The testing shims mentioned [later](https://v2.angular.io/docs/ts/latest/guide/testing.html#testbed-methods) initialize the testing module configuration to something like the BrowserModule from @angular/platform-browser.

This default configuration is merely a *foundation* for testing an app. Later you'll call TestBed.configureTestingModule with more metadata that define additional imports, declarations, providers, and schemas to fit your application tests. Optional override methods can fine-tune aspects of the configuration.

***createComponent***

After configuring TestBed, you tell it to create an instance of the *component-under-test*. In this example, TestBed.createComponent creates an instance of BannerComponent and returns a [*component test fixture*](https://v2.angular.io/docs/ts/latest/guide/testing.html#component-fixture).

Do not re-configure TestBed after calling createComponent.

The createComponent method closes the current TestBed instance to further configuration. You cannot call any more TestBed configuration methods, not configureTestingModule nor any of the override... methods. If you try, TestBed throws an error.

***ComponentFixture*, *DebugElement*, and *query(By.css)***

The createComponent method returns a **ComponentFixture**, a handle on the test environment surrounding the created component. The fixture provides access to the component instance itself and to the **DebugElement**, which is a handle on the component's DOM element.

The title property value is interpolated into the DOM within <h1> tags. Use the fixture's DebugElement to query for the <h1> element by CSS selector.

The **query** method takes a predicate function and searches the fixture's entire DOM tree for the *first* element that satisfies the predicate. The result is a *different* DebugElement, one associated with the matching DOM element.

The queryAll method returns an array of *all* DebugElements that satisfy the predicate.

A *predicate* is a function that returns a boolean. A query predicate receives a DebugElement and returns true if the element meets the selection criteria.

The **By** class is an Angular testing utility that produces useful predicates. Its By.css static method produces a [standard CSS selector](https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_started/Selectors) predicate that filters the same way as a jQuery selector.

Finally, the setup assigns the DOM element from the DebugElement **nativeElement** property to el. The tests assert that el contains the expected title text.

**The tests**

Jasmine runs the beforeEach function before each of these tests

**src/app/banner-inline.component.spec.ts (tests)**

Copy Code

it('should display original title', () => {

fixture.detectChanges();

expect(el.textContent).toContain(comp.title);

});

it('should display a different test title', () => {

comp.title = 'Test Title';

fixture.detectChanges();

expect(el.textContent).toContain('Test Title');

});

These tests ask the DebugElement for the native HTML element to satisfy their expectations.

***detectChanges*: Angular change detection within a test**

Each test tells Angular when to perform change detection by calling fixture.detectChanges(). The first test does so immediately, triggering data binding and propagation of the title property to the DOM element.

The second test changes the component's title property *and only then* calls fixture.detectChanges(); the new value appears in the DOM element.

In production, change detection kicks in automatically when Angular creates a component or the user enters a keystroke or an asynchronous activity (e.g., AJAX) completes.

The TestBed.createComponent does *not* trigger change detection. The fixture does not automatically push the component's title property value into the data bound element, a fact demonstrated in the following test:

**src/app/banner-inline.component.spec.ts (no detectChanges)**

Copy Code

it('no title in the DOM until manually call `detectChanges`', () => {

expect(el.textContent).toEqual('');

});

This behavior (or lack of it) is intentional. It gives the tester an opportunity to inspect or change the state of the component *before Angular initiates data binding or calls lifecycle hooks*.

**Try the live example**

Take a moment to explore this component spec as a [live example](https://v2.angular.io/resources/live-examples/testing/ts/banner-inline-specs.eplnkr.html) / [downloadable example](https://v2.angular.io/resources/zips/testing/banner-inline-specs.testing.zip) and lock in these fundamentals of component unit testing.

**Automatic change detection**

The BannerComponent tests frequently call detectChanges. Some testers prefer that the Angular test environment run change detection automatically.

That's possible by configuring the TestBed with the ComponentFixtureAutoDetect provider. First import it from the testing utility library:

**src/app/banner.component.detect-changes.spec.ts (import)**

Copy Code

import { ComponentFixtureAutoDetect } from '@angular/core/testing';

Then add it to the providers array of the testing module configuration:

**src/app/banner.component.detect-changes.spec.ts (AutoDetect)**

Copy Code

TestBed.configureTestingModule({

declarations: [ BannerComponent ],

providers: [

{ provide: ComponentFixtureAutoDetect, useValue: true }

]

})

Here are three tests that illustrate how automatic change detection works.

**src/app/banner.component.detect-changes.spec.ts (AutoDetect Tests)**

Copy Code

it('should display original title', () => {

// Hooray! No `fixture.detectChanges()` needed

expect(el.textContent).toContain(comp.title);

});

it('should still see original title after comp.title change', () => {

const oldTitle = comp.title;

comp.title = 'Test Title';

// Displayed title is old because Angular didn't hear the change :(

expect(el.textContent).toContain(oldTitle);

});

it('should display updated title after detectChanges', () => {

comp.title = 'Test Title';

fixture.detectChanges(); // detect changes explicitly

expect(el.textContent).toContain(comp.title);

});

The first test shows the benefit of automatic change detection.

The second and third test reveal an important limitation. The Angular testing environment does *not* know that the test changed the component's title. The ComponentFixtureAutoDetect service responds to *asynchronous activities* such as promise resolution, timers, and DOM events. But a direct, synchronous update of the component property is invisible. The test must call fixture.detectChanges() manually to trigger another cycle of change detection.

Rather than wonder when the test fixture will or won't perform change detection, the samples in this guide *always call* detectChanges() *explicitly*. There is no harm in calling detectChanges() more often than is strictly necessary.

[Back to top](https://v2.angular.io/docs/ts/latest/guide/testing.html#top)

**Test a component with an external template**

The application's actual BannerComponent behaves the same as the version above but is implemented differently. It has *external* template and css files, specified in templateUrl and styleUrls properties.

**src/app/banner.component.ts**

Copy Code

import { Component } from '@angular/core';

@Component({

selector: 'app-banner',

templateUrl: './banner.component.html',

styleUrls: ['./banner.component.css']

})

export class BannerComponent {

title = 'Test Tour of Heroes';

}

That's a problem for the tests. The TestBed.createComponent method is synchronous. But the Angular template compiler must read the external files from the file system before it can create a component instance. That's an asynchronous activity. The previous setup for testing the inline component won't work for a component with an external template.

**The first asynchronous *beforeEach***

The test setup for BannerComponent must give the Angular template compiler time to read the files. The logic in the beforeEach of the previous spec is split into two beforeEach calls. The first beforeEach handles asynchronous compilation.

**src/app/banner.component.spec.ts (first beforeEach)**

Copy Code

// async beforeEach

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [ BannerComponent ], // declare the test component

})

.compileComponents(); // compile template and css

}));

Notice the async function called as the argument to beforeEach. The async function is one of the Angular testing utilities and has to be imported.

Copy Code

import { async } from '@angular/core/testing';

It takes a parameterless function and *returns a function* which becomes the true argument to the beforeEach.

The body of the async argument looks much like the body of a synchronous beforeEach. There is nothing obviously asynchronous about it. For example, it doesn't return a promise and there is no done function to call as there would be in standard Jasmine asynchronous tests. Internally, async arranges for the body of the beforeEach to run in a special *async test zone* that hides the mechanics of asynchronous execution.

All this is necessary in order to call the asynchronous TestBed.compileComponents method.

***compileComponents***

The TestBed.configureTestingModule method returns the TestBed class so you can chain calls to other TestBed static methods such as compileComponents.

The TestBed.compileComponents method asynchronously compiles all the components configured in the testing module. In this example, the BannerComponent is the only component to compile. When compileComponents completes, the external templates and css files have been "inlined" and TestBed.createComponent can create new instances of BannerComponent synchronously.

WebPack developers need not call compileComponents because it inlines templates and css as part of the automated build process that precedes running the test.

In this example, TestBed.compileComponents only compiles the BannerComponent. Tests later in the guide declare multiple components and a few specs import entire application modules that hold yet more components. Any of these components might have external templates and css files. TestBed.compileComponents compiles all of the declared components asynchronously at one time.

Do not configure the TestBed after calling compileComponents. Make compileComponents the last step before calling TestBed.createComponent to instantiate the *component-under-test*.

Calling compileComponents closes the current TestBed instance is further configuration. You cannot call any more TestBed configuration methods, not configureTestingModule nor any of the override... methods. The TestBed throws an error if you try.

**The second synchronous *beforeEach***

A *synchronous* beforeEach containing the remaining setup steps follows the asynchronous beforeEach.

**src/app/banner.component.spec.ts (second beforeEach)**

Copy Code

// synchronous beforeEach

beforeEach(() => {

fixture = TestBed.createComponent(BannerComponent);

comp = fixture.componentInstance; // BannerComponent test instance

// query for the title <h1> by CSS element selector

de = fixture.debugElement.query(By.css('h1'));

el = de.nativeElement;

});

These are the same steps as in the original beforeEach. They include creating an instance of the BannerComponent and querying for the elements to inspect.

You can count on the test runner to wait for the first asynchronous beforeEach to finish before calling the second.

**Waiting for *compileComponents***

The compileComponents method returns a promise so you can perform additional tasks *immediately after* it finishes. For example, you could move the synchronous code in the second beforeEach into a compileComponents().then(...) callback and write only one beforeEach.

Most developers find that hard to read. The two beforeEach calls are widely preferred.

**Try the live example**

Take a moment to explore this component spec as a [live example](https://v2.angular.io/resources/live-examples/testing/ts/banner-specs.eplnkr.html) / [downloadable example](https://v2.angular.io/resources/zips/testing/banner-specs.testing.zip).

The [Quickstart seed](https://v2.angular.io/docs/ts/latest/guide/setup.html) provides a similar test of its AppComponent as you can see in *this* [live example](https://v2.angular.io/resources/live-examples/setup/ts/quickstart-specs.eplnkr.html) / [downloadable example](https://v2.angular.io/resources/zips/setup/quickstart-specs.setup.zip). It too calls compileComponents although it doesn't have to because the AppComponent's template is inline.

There's no harm in it and you might call compileComponents anyway in case you decide later to re-factor the template into a separate file. The tests in this guide only call compileComponents when necessary.

[Back to top](https://v2.angular.io/docs/ts/latest/guide/testing.html#top)

**Test a component with a dependency**

Components often have service dependencies.

The WelcomeComponent displays a welcome message to the logged in user. It knows who the user is based on a property of the injected UserService:

**src/app/welcome.component.ts**

Copy Code

import { Component, OnInit } from '@angular/core';

import { UserService } from './model';

@Component({

selector: 'app-welcome',

template: '<h3 class="welcome" ><i>{{welcome}}</i></h3>'

})

export class WelcomeComponent implements OnInit {

welcome = '-- not initialized yet --';

constructor(private userService: UserService) { }

ngOnInit(): void {

this.welcome = this.userService.isLoggedIn ?

'Welcome, ' + this.userService.user.name :

'Please log in.';

}

}

The WelcomeComponent has decision logic that interacts with the service, logic that makes this component worth testing. Here's the testing module configuration for the spec file, src/app/welcome.component.spec.ts:

**src/app/welcome.component.spec.ts**

Copy Code

TestBed.configureTestingModule({

declarations: [ WelcomeComponent ],

// providers: [ UserService ] // NO! Don't provide the real service!

// Provide a test-double instead

providers: [ {provide: UserService, useValue: userServiceStub } ]

});

This time, in addition to declaring the *component-under-test*, the configuration adds a UserService provider to the providers list. But not the real UserService.

**Provide service test doubles**

A *component-under-test* doesn't have to be injected with real services. In fact, it is usually better if they are test doubles (stubs, fakes, spies, or mocks). The purpose of the spec is to test the component, not the service, and real services can be trouble.

Injecting the real UserService could be a nightmare. The real service might ask the user for login credentials and attempt to reach an authentication server. These behaviors can be hard to intercept. It is far easier and safer to create and register a test double in place of the real UserService.

This particular test suite supplies a minimal UserService stub that satisfies the needs of the WelcomeComponent and its tests:

Copy Code

userServiceStub = {

isLoggedIn: true,

user: { name: 'Test User'}

};

**Get injected services**

The tests need access to the (stub) UserService injected into the WelcomeComponent.

Angular has a hierarchical injection system. There can be injectors at multiple levels, from the root injector created by the TestBed down through the component tree.

The safest way to get the injected service, the way that ***always works***, is to **get it from the injector of the *component-under-test***. The component injector is a property of the fixture's DebugElement.

**WelcomeComponent's injector**

Copy Code

// UserService actually injected into the component

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

***TestBed.get***

You *may* also be able to get the service from the root injector via TestBed.get. This is easier to remember and less verbose. But it only works when Angular injects the component with the service instance in the test's root injector. Fortunately, in this test suite, the *only* provider of UserService is the root testing module, so it is safe to call TestBed.get as follows:

**TestBed injector**

Copy Code

// UserService from the root injector

userService = TestBed.get(UserService);

The [inject](https://v2.angular.io/docs/ts/latest/guide/testing.html#inject) utility function is another way to get one or more services from the test root injector.

For a use case in which inject and TestBed.get do not work, see the section [*Override a component's providers*](https://v2.angular.io/docs/ts/latest/guide/testing.html#component-override), which explains why you must get the service from the component's injector instead.

**Always get the service from an injector**

Do *not* reference the userServiceStub object that's provided to the testing module in the body of your test. **It does not work!** The userService instance injected into the component is a completely *different* object, a clone of the provided userServiceStub.

Copy Code

it('stub object and injected UserService should not be the same', () => {

expect(userServiceStub === userService).toBe(false);

// Changing the stub object has no effect on the injected service

userServiceStub.isLoggedIn = false;

expect(userService.isLoggedIn).toBe(true);

});

**Final setup and tests**

Here's the complete beforeEach using TestBed.get:

**src/app/welcome.component.spec.ts**

Copy Code

beforeEach(() => {

// stub UserService for test purposes

userServiceStub = {

isLoggedIn: true,

user: { name: 'Test User'}

};

TestBed.configureTestingModule({

declarations: [ WelcomeComponent ],

providers: [ {provide: UserService, useValue: userServiceStub } ]

});

fixture = TestBed.createComponent(WelcomeComponent);

comp = fixture.componentInstance;

// UserService from the root injector

userService = TestBed.get(UserService);

// get the "welcome" element by CSS selector (e.g., by class name)

de = fixture.debugElement.query(By.css('.welcome'));

el = de.nativeElement;

});

And here are some tests:

**src/app/welcome.component.spec.ts**

Copy Code

it('should welcome the user', () => {

fixture.detectChanges();

const content = el.textContent;

expect(content).toContain('Welcome', '"Welcome ..."');

expect(content).toContain('Test User', 'expected name');

});

it('should welcome "Bubba"', () => {

userService.user.name = 'Bubba'; // welcome message hasn't been shown yet

fixture.detectChanges();

expect(el.textContent).toContain('Bubba');

});

it('should request login if not logged in', () => {

userService.isLoggedIn = false; // welcome message hasn't been shown yet

fixture.detectChanges();

const content = el.textContent;

expect(content).not.toContain('Welcome', 'not welcomed');

expect(content).toMatch(/log in/i, '"log in"');

});

The first is a sanity test; it confirms that the stubbed UserService is called and working.

The second parameter to the Jasmine it (e.g., 'expected name') is an optional addendum. If the expectation fails, Jasmine displays this addendum after the expectation failure message. In a spec with multiple expectations, it can help clarify what went wrong and which expectation failed.

The remaining tests confirm the logic of the component when the service returns different values. The second test validates the effect of changing the user name. The third test checks that the component displays the proper message when there is no logged-in user.

[Back to top](https://v2.angular.io/docs/ts/latest/guide/testing.html#top)

**Test a component with an async service**

Many services return values asynchronously. Most data services make an HTTP request to a remote server and the response is necessarily asynchronous.

The "About" view in this sample displays Mark Twain quotes. The TwainComponent handles the display, delegating the server request to the TwainService.

Both are in the src/app/shared folder because the author intends to display Twain quotes on other pages someday. Here is the TwainComponent.

**src/app/shared/twain.component.ts**

Copy Code

@Component({

selector: 'twain-quote',

template: '<p class="twain"><i>{{quote}}</i></p>'

})

export class TwainComponent implements OnInit {

intervalId: number;

quote = '...';

constructor(private twainService: TwainService) { }

ngOnInit(): void {

this.twainService.getQuote().then(quote => this.quote = quote);

}

}

The TwainService implementation is irrelevant for this particular test. It is sufficient to see within ngOnInit that twainService.getQuote returns a promise, which means it is asynchronous.

In general, tests should not make calls to remote servers. They should emulate such calls. The setup in this src/app/shared/twain.component.spec.ts shows one way to do that:

**src/app/shared/twain.component.spec.ts (setup)**

Copy Code

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [ TwainComponent ],

providers: [ TwainService ],

});

fixture = TestBed.createComponent(TwainComponent);

comp = fixture.componentInstance;

// TwainService actually injected into the component

twainService = fixture.debugElement.injector.get(TwainService);

// Setup spy on the `getQuote` method

spy = spyOn(twainService, 'getQuote')

.and.returnValue(Promise.resolve(testQuote));

// Get the Twain quote element by CSS selector (e.g., by class name)

de = fixture.debugElement.query(By.css('.twain'));

el = de.nativeElement;

});

**Spying on the real service**

This setup is similar to the [welcome.component.spec setup](https://v2.angular.io/docs/ts/latest/guide/testing.html#welcome-spec-setup). But instead of creating a stubbed service object, it injects the *real* service (see the testing module providers) and replaces the critical getQuote method with a Jasmine spy.

Copy Code

spy = spyOn(twainService, 'getQuote')

.and.returnValue(Promise.resolve(testQuote));

The spy is designed such that any call to getQuote receives an immediately resolved promise with a test quote. The spy bypasses the actual getQuote method and therefore does not contact the server.

Faking a service instance and spying on the real service are *both* great options. Pick the one that seems easiest for the current test suite. Don't be afraid to change your mind.

Spying on the real service isn't always easy, especially when the real service has injected dependencies. You can *stub and spy* at the same time, as shown in [an example below](https://v2.angular.io/docs/ts/latest/guide/testing.html#spy-stub).

Here are the tests with commentary to follow:

**src/app/shared/twain.component.spec.ts (tests)**

Copy Code

1. it('should not show quote before OnInit', () => {
2. expect(el.textContent).toBe('', 'nothing displayed');
3. expect(spy.calls.any()).toBe(false, 'getQuote not yet called');
4. });
5. it('should still not show quote after component initialized', () => {
6. fixture.detectChanges();
7. // getQuote service is async => still has not returned with quote
8. expect(el.textContent).toBe('...', 'no quote yet');
9. expect(spy.calls.any()).toBe(true, 'getQuote called');
10. });
11. it('should show quote after getQuote promise (async)', async(() => {
12. fixture.detectChanges();
13. fixture.whenStable().then(() => { // wait for async getQuote
14. fixture.detectChanges(); // update view with quote
15. expect(el.textContent).toBe(testQuote);
16. });
17. }));
18. it('should show quote after getQuote promise (fakeAsync)', fakeAsync(() => {
19. fixture.detectChanges();
20. tick(); // wait for async getQuote
21. fixture.detectChanges(); // update view with quote
22. expect(el.textContent).toBe(testQuote);
23. }));

**Synchronous tests**

The first two tests are synchronous. Thanks to the spy, they verify that getQuote is called *after* the first change detection cycle during which Angular calls ngOnInit.

Neither test can prove that a value from the service is displayed. The quote itself has not arrived, despite the fact that the spy returns a resolved promise.

This test must wait at least one full turn of the JavaScript engine before the value becomes available. The test must become *asynchronous*.

**The *async* function in *it***

Notice the async in the third test.

**src/app/shared/twain.component.spec.ts (async test)**

Copy Code

it('should show quote after getQuote promise (async)', async(() => {

fixture.detectChanges();

fixture.whenStable().then(() => { // wait for async getQuote

fixture.detectChanges(); // update view with quote

expect(el.textContent).toBe(testQuote);

});

}));

The async function is one of the Angular testing utilities. It simplifies coding of asynchronous tests by arranging for the tester's code to run in a special *async test zone* as [discussed earlier](https://v2.angular.io/docs/ts/latest/guide/testing.html#async-in-before-each) when it was called in a beforeEach.

Although async does a great job of hiding asynchronous boilerplate, some functions called within a test (such as fixture.whenStable) continue to reveal their asynchronous behavior.

The fakeAsync alternative, [covered below](https://v2.angular.io/docs/ts/latest/guide/testing.html#fake-async), removes this artifact and affords a more linear coding experience.

***whenStable***

The test must wait for the getQuote promise to resolve in the next turn of the JavaScript engine.

This test has no direct access to the promise returned by the call to twainService.getQuote because it is buried inside TwainComponent.ngOnInit and therefore inaccessible to a test that probes only the component API surface.

Fortunately, the getQuote promise is accessible to the *async test zone*, which intercepts all promises issued within the *async* method call *no matter where they occur*.

The ComponentFixture.whenStable method returns its own promise, which resolves when the getQuote promise finishes. In fact, the *whenStable* promise resolves when *all pending asynchronous activities within this test* complete—the definition of "stable."

Then the test resumes and kicks off another round of change detection (fixture.detectChanges), which tells Angular to update the DOM with the quote. The getQuote helper method extracts the display element text and the expectation confirms that the text matches the test quote.

**The *fakeAsync* function**

The fourth test verifies the same component behavior in a different way.

**src/app/shared/twain.component.spec.ts (fakeAsync test)**

Copy Code

it('should show quote after getQuote promise (fakeAsync)', fakeAsync(() => {

fixture.detectChanges();

tick(); // wait for async getQuote

fixture.detectChanges(); // update view with quote

expect(el.textContent).toBe(testQuote);

}));

Notice that fakeAsync replaces async as the it argument. The fakeAsync function is another of the Angular testing utilities.

Like [async](https://v2.angular.io/docs/ts/latest/guide/testing.html#async), it *takes* a parameterless function and *returns* a function that becomes the argument to the Jasmine it call.

The fakeAsync function enables a linear coding style by running the test body in a special *fakeAsync test zone*.

The principle advantage of fakeAsync over async is that the test appears to be synchronous. There is no then(...) to disrupt the visible flow of control. The promise-returning fixture.whenStable is gone, replaced by tick().

There *are* limitations. For example, you cannot make an XHR call from within a fakeAsync.

**The *tick* function**

The tick function is one of the Angular testing utilities and a companion to fakeAsync. You can only call it within a fakeAsync body.

Calling tick() simulates the passage of time until all pending asynchronous activities finish, including the resolution of the getQuote promise in this test case.

It returns nothing. There is no promise to wait for. Proceed with the same test code that appeared in the whenStable.then() callback.

Even this simple example is easier to read than the third test. To more fully appreciate the improvement, imagine a succession of asynchronous operations, chained in a long sequence of promise callbacks.

***jasmine.done***

While the async and fakeAsync functions greatly simplify Angular asynchronous testing, you can still fall back to the traditional Jasmine asynchronous testing technique.

You can still pass it a function that takes a [done callback](http://jasmine.github.io/2.0/introduction.html#section-Asynchronous_Support). Now you are responsible for chaining promises, handling errors, and calling done at the appropriate moment.

Here is a done version of the previous two tests:

**src/app/shared/twain.component.spec.ts (done test)**

Copy Code

it('should show quote after getQuote promise (done)', done => {

fixture.detectChanges();

// get the spy promise and wait for it to resolve

spy.calls.mostRecent().returnValue.then(() => {

fixture.detectChanges(); // update view with quote

expect(el.textContent).toBe(testQuote);

done();

});

});

Although there is no direct access to the getQuote promise inside TwainComponent, the spy has direct access, which makes it possible to wait for getQuote to finish.

Writing test functions with done, while more cumbersome than async and fakeAsync, is a viable and occasionally necessary technique. For example, you can't call async or fakeAsync when testing code that involves the intervalTimer, as is common when testing async Observable methods.

[Back to top](https://v2.angular.io/docs/ts/latest/guide/testing.html#top)

**Test a component with inputs and outputs**

A component with inputs and outputs typically appears inside the view template of a host component. The host uses a property binding to set the input property and an event binding to listen to events raised by the output property.

The testing goal is to verify that such bindings work as expected. The tests should set input values and listen for output events.

The DashboardHeroComponent is a tiny example of a component in this role. It displays an individual hero provided by the DashboardComponent. Clicking that hero tells the DashboardComponent that the user has selected the hero.

The DashboardHeroComponent is embedded in the DashboardComponent template like this:

**src/app/dashboard/dashboard.component.html (excerpt)**

Copy Code

<dashboard-hero \*ngFor="let hero of heroes" class="col-1-4"

[hero]=hero (selected)="gotoDetail($event)" >

</dashboard-hero>

The DashboardHeroComponent appears in an \*ngFor repeater, which sets each component's hero input property to the looping value and listens for the component's selected event.

Here's the component's definition:

**src/app/dashboard/dashboard-hero.component.ts (component)**

Copy Code

@Component({

selector: 'dashboard-hero',

templateUrl: './dashboard-hero.component.html',

styleUrls: [ './dashboard-hero.component.css' ]

})

export class DashboardHeroComponent {

@Input() hero: Hero;

@Output() selected = new EventEmitter<Hero>();

click() { this.selected.emit(this.hero); }

}

While testing a component this simple has little intrinsic value, it's worth knowing how. You can use one of these approaches:

* Test it as used by DashboardComponent.
* Test it as a stand-alone component.
* Test it as used by a substitute for DashboardComponent.

A quick look at the DashboardComponent constructor discourages the first approach:

**src/app/dashboard/dashboard.component.ts (constructor)**

Copy Code

constructor(

private router: Router,

private heroService: HeroService) {

}

The DashboardComponent depends on the Angular router and the HeroService. You'd probably have to replace them both with test doubles, which is a lot of work. The router seems particularly challenging.

The [discussion below](https://v2.angular.io/docs/ts/latest/guide/testing.html#routed-component) covers testing components that require the router.

The immediate goal is to test the DashboardHeroComponent, not the DashboardComponent, so, try the second and third options.

**Test *DashboardHeroComponent* stand-alone**

Here's the spec file setup.

**src/app/dashboard/dashboard-hero.component.spec.ts (setup)**

Copy Code

// async beforeEach

beforeEach( async(() => {

TestBed.configureTestingModule({

declarations: [ DashboardHeroComponent ],

})

.compileComponents(); // compile template and css

}));

// synchronous beforeEach

beforeEach(() => {

fixture = TestBed.createComponent(DashboardHeroComponent);

comp = fixture.componentInstance;

heroEl = fixture.debugElement.query(By.css('.hero')); // find hero element

// pretend that it was wired to something that supplied a hero

expectedHero = new Hero(42, 'Test Name');

comp.hero = expectedHero;

fixture.detectChanges(); // trigger initial data binding

});

The async beforeEach was discussed [above](https://v2.angular.io/docs/ts/latest/guide/testing.html#component-with-external-template). Having compiled the components asynchronously with compileComponents, the rest of the setup proceeds *synchronously* in a *second* beforeEach, using the basic techniques described [earlier](https://v2.angular.io/docs/ts/latest/guide/testing.html#simple-component-test).

Note how the setup code assigns a test hero (expectedHero) to the component's hero property, emulating the way the DashboardComponent would set it via the property binding in its repeater.

The first test follows:

**src/app/dashboard/dashboard-hero.component.spec.ts (name test)**

Copy Code

it('should display hero name', () => {

const expectedPipedName = expectedHero.name.toUpperCase();

expect(heroEl.nativeElement.textContent).toContain(expectedPipedName);

});

It verifies that the hero name is propagated to template with a binding. Because the template passes the hero name through the Angular UpperCasePipe, the test must match the element value with the uppercased name:

Copy Code

<div (click)="click()" class="hero">

{{hero.name | uppercase}}

</div>

This small test demonstrates how Angular tests can verify a component's visual representation—something not possible with [isolated unit tests](https://v2.angular.io/docs/ts/latest/guide/testing.html#isolated-component-tests)—at low cost and without resorting to much slower and more complicated end-to-end tests.

The second test verifies click behavior. Clicking the hero should raise a selected event that the host component (DashboardComponent presumably) can hear:

**src/app/dashboard/dashboard-hero.component.spec.ts (click test)**

Copy Code

it('should raise selected event when clicked', () => {

let selectedHero: Hero;

comp.selected.subscribe((hero: Hero) => selectedHero = hero);

heroEl.triggerEventHandler('click', null);

expect(selectedHero).toBe(expectedHero);

});

The component exposes an EventEmitter property. The test subscribes to it just as the host component would do.

The heroEl is a DebugElement that represents the hero <div>. The test calls triggerEventHandler with the "click" event name. The "click" event binding responds by calling DashboardHeroComponent.click().

If the component behaves as expected, click() tells the component's selected property to emit the hero object, the test detects that value through its subscription to selected, and the test should pass.

***triggerEventHandler***

The Angular DebugElement.triggerEventHandler can raise *any data-bound event* by its *event name*. The second parameter is the event object passed to the handler.

In this example, the test triggers a "click" event with a null event object.

Copy Code

heroEl.triggerEventHandler('click', null);

The test assumes (correctly in this case) that the runtime event handler—the component's click() method—doesn't care about the event object.

Other handlers are less forgiving. For example, the RouterLink directive expects an object with a button property that identifies which mouse button was pressed. This directive throws an error if the event object doesn't do this correctly.

Clicking a button, an anchor, or an arbitrary HTML element is a common test task.

Make that easy by encapsulating the *click-triggering* process in a helper such as the click function below:

**testing/index.ts (click helper)**

Copy Code

/\*\* Button events to pass to `DebugElement.triggerEventHandler` for RouterLink event handler \*/

export const ButtonClickEvents = {

left: { button: 0 },

right: { button: 2 }

};

/\*\* Simulate element click. Defaults to mouse left-button click event. \*/

export function click(el: DebugElement | HTMLElement, eventObj: any = ButtonClickEvents.left): void {

if (el instanceof HTMLElement) {

el.click();

} else {

el.triggerEventHandler('click', eventObj);

}

}

The first parameter is the *element-to-click*. If you wish, you can pass a custom event object as the second parameter. The default is a (partial) [left-button mouse event object](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/button) accepted by many handlers including the RouterLink directive.

click() is not an Angular testing utility

The click() helper function is **not** one of the Angular testing utilities. It's a function defined in *this guide's sample code*. All of the sample tests use it. If you like it, add it to your own collection of helpers.

Here's the previous test, rewritten using this click helper.

**src/app/dashboard/dashboard-hero.component.spec.ts (click test revised)**

Copy Code

it('should raise selected event when clicked', () => {

let selectedHero: Hero;

comp.selected.subscribe((hero: Hero) => selectedHero = hero);

click(heroEl); // triggerEventHandler helper

expect(selectedHero).toBe(expectedHero);

});

**Test a component inside a test host component**

In the previous approach, the tests themselves played the role of the host DashboardComponent. But does the DashboardHeroComponent work correctly when properly data-bound to a host component?

Testing with the actual DashboardComponent host is doable but seems more trouble than its worth. It's easier to emulate the DashboardComponent host with a *test host* like this one:

**src/app/dashboard/dashboard-hero.component.spec.ts (test host)**

Copy Code

@Component({

template: `

<dashboard-hero [hero]="hero" (selected)="onSelected($event)"></dashboard-hero>`

})

class TestHostComponent {

hero = new Hero(42, 'Test Name');

selectedHero: Hero;

onSelected(hero: Hero) { this.selectedHero = hero; }

}

The test host binds to DashboardHeroComponent as the DashboardComponent would but without the distraction of the Router, the HeroService, or even the \*ngFor repeater.

The test host sets the component's hero input property with its test hero. It binds the component's selected event with its onSelected handler, which records the emitted hero in its selectedHero property. Later, the tests check that property to verify that the DashboardHeroComponent.selected event emitted the right hero.

The setup for the test-host tests is similar to the setup for the stand-alone tests:

**src/app/dashboard/dashboard-hero.component.spec.ts (test host setup)**

Copy Code

beforeEach( async(() => {

TestBed.configureTestingModule({

declarations: [ DashboardHeroComponent, TestHostComponent ], // declare both

}).compileComponents();

}));

beforeEach(() => {

// create TestHostComponent instead of DashboardHeroComponent

fixture = TestBed.createComponent(TestHostComponent);

testHost = fixture.componentInstance;

heroEl = fixture.debugElement.query(By.css('.hero')); // find hero

fixture.detectChanges(); // trigger initial data binding

});

This testing module configuration shows two important differences:

1. It *declares* both the DashboardHeroComponent and the TestHostComponent.
2. It *creates* the TestHostComponent instead of the DashboardHeroComponent.

The createComponent returns a fixture that holds an instance of TestHostComponent instead of an instance of DashboardHeroComponent.

Creating the TestHostComponent has the side-effect of creating a DashboardHeroComponent because the latter appears within the template of the former. The query for the hero element (heroEl) still finds it in the test DOM, albeit at greater depth in the element tree than before.

The tests themselves are almost identical to the stand-alone version:

**src/app/dashboard/dashboard-hero.component.spec.ts (test-host)**

Copy Code

it('should display hero name', () => {

const expectedPipedName = testHost.hero.name.toUpperCase();

expect(heroEl.nativeElement.textContent).toContain(expectedPipedName);

});

it('should raise selected event when clicked', () => {

click(heroEl);

// selected hero should be the same data bound hero

expect(testHost.selectedHero).toBe(testHost.hero);

});

Only the selected event test differs. It confirms that the selected DashboardHeroComponent hero really does find its way up through the event binding to the host component.

[Back to top](https://v2.angular.io/docs/ts/latest/guide/testing.html#top)

**Test a routed component**

Testing the actual DashboardComponent seemed daunting because it injects the Router.

**src/app/dashboard/dashboard.component.ts (constructor)**

Copy Code

constructor(

private router: Router,

private heroService: HeroService) {

}

It also injects the HeroService, but faking that is a [familiar story](https://v2.angular.io/docs/ts/latest/guide/testing.html#component-with-async-service). The Router has a complicated API and is entwined with other services and application preconditions.

Fortunately, the DashboardComponent isn't doing much with the Router

**src/app/dashboard/dashboard.component.ts (goToDetail)**

Copy Code

gotoDetail(hero: Hero) {

let url = `/heroes/${hero.id}`;

this.router.navigateByUrl(url);

}

This is often the case. As a rule you test the component, not the router, and care only if the component navigates with the right address under the given conditions. Stubbing the router with a test implementation is an easy option. This should do the trick:

**src/app/dashboard/dashboard.component.spec.ts (Router Stub)**

Copy Code

class RouterStub {

navigateByUrl(url: string) { return url; }

}

Now set up the testing module with the test stubs for the Router and HeroService, and create a test instance of the DashboardComponent for subsequent testing.

**src/app/dashboard/dashboard.component.spec.ts (compile and create)**

Copy Code

beforeEach( async(() => {

TestBed.configureTestingModule({

providers: [

{ provide: HeroService, useClass: FakeHeroService },

{ provide: Router, useClass: RouterStub }

]

})

.compileComponents().then(() => {

fixture = TestBed.createComponent(DashboardComponent);

comp = fixture.componentInstance;

});

The following test clicks the displayed hero and confirms (with the help of a spy) that Router.navigateByUrl is called with the expected url.

**src/app/dashboard/dashboard.component.spec.ts (navigate test)**

Copy Code

it('should tell ROUTER to navigate when hero clicked',

inject([Router], (router: Router) => { // ...

const spy = spyOn(router, 'navigateByUrl');

heroClick(); // trigger click on first inner <div class="hero">

// args passed to router.navigateByUrl()

const navArgs = spy.calls.first().args[0];

// expecting to navigate to id of the component's first hero

const id = comp.heroes[0].id;

expect(navArgs).toBe('/heroes/' + id,

'should nav to HeroDetail for first hero');

}));

**The *inject* function**

Notice the inject function in the second it argument.

Copy Code

it('should tell ROUTER to navigate when hero clicked',

inject([Router], (router: Router) => { // ...

}));

The inject function is one of the Angular testing utilities. It injects services into the test function where you can alter, spy on, and manipulate them.

The inject function has two parameters:

1. An array of Angular dependency injection tokens.
2. A test function whose parameters correspond exactly to each item in the injection token array.

inject uses the TestBed Injector

The inject function uses the current TestBed injector and can only return services provided at that level. It does not return services from component providers.

This example injects the Router from the current TestBed injector. That's fine for this test because the Router is, and must be, provided by the application root injector.

If you need a service provided by the component's *own* injector, call fixture.debugElement.injector.get instead:

**Component's injector**

Copy Code

// UserService actually injected into the component

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

Use the component's own injector to get the service actually injected into the component.

The inject function closes the current TestBed instance to further configuration. You cannot call any more TestBed configuration methods, not configureTestingModule nor any of the override... methods. The TestBed throws an error if you try.

Do not configure the TestBed after calling inject.

**Test a routed component with parameters**

Clicking a *Dashboard* hero triggers navigation to heroes/:id, where :id is a route parameter whose value is the id of the hero to edit. That URL matches a route to the HeroDetailComponent.

The router pushes the :id token value into the ActivatedRoute.params *Observable* property, Angular injects the ActivatedRoute into the HeroDetailComponent, and the component extracts the id so it can fetch the corresponding hero via the HeroDetailService. Here's the HeroDetailComponent constructor:

**src/app/hero/hero-detail.component.ts (constructor)**

Copy Code

constructor(

private heroDetailService: HeroDetailService,

private route: ActivatedRoute,

private router: Router) {

}

HeroDetailComponent subscribes to ActivatedRoute.params changes in its ngOnInit method.

**src/app/hero/hero-detail.component.ts (ngOnInit)**

Copy Code

ngOnInit(): void {

// get hero when `id` param changes

this.route.params.subscribe(p => this.getHero(p && p['id']));

}

The expression after route.params chains an *Observable* operator that *plucks* the id from the params and then chains a forEach operator to subscribe to id-changing events. The id changes every time the user navigates to a different hero.

The forEach passes the new id value to the component's getHero method (not shown) which fetches a hero and sets the component's hero property. If theid parameter is missing, the pluck operator fails and the catch treats failure as a request to edit a new hero.

The [Router](https://v2.angular.io/docs/ts/latest/guide/router.html#route-parameters) guide covers ActivatedRoute.params in more detail.

A test can explore how the HeroDetailComponent responds to different id parameter values by manipulating the ActivatedRoute injected into the component's constructor.

By now you know how to stub the Router and a data service. Stubbing the ActivatedRoute follows the same pattern except for a complication: the ActivatedRoute.params is an *Observable*.

**Create an *Observable* test double**

The hero-detail.component.spec.ts relies on an ActivatedRouteStub to set ActivatedRoute.params values for each test. This is a cross-application, re-usable *test helper class*. Consider placing such helpers in a testing folder sibling to the app folder. This sample keeps ActivatedRouteStub in testing/router-stubs.ts:

**testing/router-stubs.ts (ActivatedRouteStub)**

Copy Code

import { BehaviorSubject } from 'rxjs/BehaviorSubject';

@Injectable()

export class ActivatedRouteStub {

// ActivatedRoute.params is Observable

private subject = new BehaviorSubject(this.testParams);

params = this.subject.asObservable();

// Test parameters

private \_testParams: {};

get testParams() { return this.\_testParams; }

set testParams(params: {}) {

this.\_testParams = params;

this.subject.next(params);

}

// ActivatedRoute.snapshot.params

get snapshot() {

return { params: this.testParams };

}

}

Notable features of this stub are:

* The stub implements only two of the ActivatedRoute capabilities: params and snapshot.params.
* [*BehaviorSubject*](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/behaviorsubject.md) drives the stub's params *Observable* and returns the same value to every params subscriber until it's given a new value.
* The HeroDetailComponent chains its expressions to this stub params *Observable* which is now under the tester's control.
* Setting the testParams property causes the subject to push the assigned value into params. That triggers the HeroDetailComponent *params* subscription, described above, in the same way that navigation does.
* Setting the testParams property also updates the stub's internal value for the snapshot property to return.

The [*snapshot*](https://v2.angular.io/docs/ts/latest/guide/router.html#snapshot) is another popular way for components to consume route parameters.

The router stubs in this guide are meant to inspire you. Create your own stubs to fit your testing needs.

**Testing with the *Observable* test double**

Here's a test demonstrating the component's behavior when the observed id refers to an existing hero:

**src/app/hero/hero-detail.component.spec.ts (existing id)**

Copy Code

describe('when navigate to existing hero', () => {

let expectedHero: Hero;

beforeEach( async(() => {

expectedHero = firstHero;

activatedRoute.testParams = { id: expectedHero.id };

createComponent();

}));

it('should display that hero\'s name', () => {

expect(page.nameDisplay.textContent).toBe(expectedHero.name);

});

});

The createComponent method and page object are discussed [in the next section](https://v2.angular.io/docs/ts/latest/guide/testing.html#page-object). Rely on your intuition for now.

When the id cannot be found, the component should re-route to the HeroListComponent. The test suite setup provided the same RouterStub [described above](https://v2.angular.io/docs/ts/latest/guide/testing.html#routed-component) which spies on the router without actually navigating. This test supplies a "bad" id and expects the component to try to navigate.

**src/app/hero/hero-detail.component.spec.ts (bad id)**

Copy Code

describe('when navigate to non-existant hero id', () => {

beforeEach( async(() => {

activatedRoute.testParams = { id: 99999 };

createComponent();

}));

it('should try to navigate back to hero list', () => {

expect(page.gotoSpy.calls.any()).toBe(true, 'comp.gotoList called');

expect(page.navSpy.calls.any()).toBe(true, 'router.navigate called');

});

});

While this app doesn't have a route to the HeroDetailComponent that omits the id parameter, it might add such a route someday. The component should do something reasonable when there is no id.

In this implementation, the component should create and display a new hero. New heroes have id=0 and a blank name. This test confirms that the component behaves as expected:

**src/app/hero/hero-detail.component.spec.ts (no id)**

Copy Code

describe('when navigate with no hero id', () => {

beforeEach( async( createComponent ));

it('should have hero.id === 0', () => {

expect(comp.hero.id).toBe(0);

});

it('should display empty hero name', () => {

expect(page.nameDisplay.textContent).toBe('');

});

});