Component testing scenarios

This guide explores common component testing use cases.

If you'd like to experiment with the application that this guide describes, run it in your browser or download and run it locally.

Component binding

In the example app, the `BannerComponent` presents static title text in the HTML template.

After a few changes, the `BannerComponent` presents a dynamic title by binding to the component's `title` property like this.

As minimal as this is, you decide to add a test to confirm that component actually displays the right content where you think it should.

Query for the *<h1>* You'll write a sequence of tests that inspect the value of the `<h1>` element that wraps the *title* property interpolation binding.

You update the `beforeEach` to find that element with a standard HTML `querySelector` and assign it to the `h1` variable.

{@a detect-changes}

***createComponent()* does not bind data** For your first test you'd like to see that the screen displays the default `title`. Your instinct is to write a test that immediately inspects the `<h1>` like this:

That test fails with the message:

```
expected '' to contain 'Test Tour of Heroes'.
```

Binding happens when Angular performs **change detection**.

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

The `TestBed.createComponent` does *not* trigger change detection; a fact confirmed in the revised test:

detectChanges() You must tell the `TestBed` to perform data binding by calling `fixture.detectChanges()`. Only then does the `<h1>` have the expected title.

Delayed change detection is intentional and useful. It gives the tester an opportunity to inspect and change the state of the component *before Angular initiates data binding and calls lifecycle hooks.*

Here's another test that changes the component's `title` property *before* calling `fixture.detectChanges()`.

{@a auto-detect-changes}

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:

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

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

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.

{@a dispatch-event} #### Change an input value with *dispatchEvent()*

To simulate user input, find the input element and set its `value` property.

You will call `fixture.detectChanges()` to trigger Angular's change detection. But there is an essential, intermediate step.

Angular doesn't know that you set the input element's `value` property. It won't read that property until you raise the element's `input` event by calling `dispatchEvent()`. *Then* you call `detectChanges()`.

The following example demonstrates the proper sequence.

Component with external files

The preceding `BannerComponent` is defined with an *inline template* and *inline css*, specified in the `@Component.template` and `@Component.styles` properties respectively.

Many components specify *external templates* and *external css* with the `@Component.templateUrl` and `@Component.styleUrls` properties respectively, as the following variant of `BannerComponent` does.

This syntax tells the Angular compiler to read the external files during component compilation.

That's not a problem when you run the CLI `ng test` command because it *compiles the application before running the tests.*

However, if you run the tests in a **non-CLI environment**, tests of this component might fail. For example, if you run the `BannerComponent` tests in a web coding environment such as plunker, you'll see a message like this one:

Error: This test module uses the component BannerComponent which is using a "templateUrl" or "styleUrls", but they were never compiled. Please call "TestBed.compileComponents" before your test.

You get this test failure message when the runtime environment compiles the source code *during the tests themselves.*

To correct the problem, call `compileComponents()` as explained in the following Calling compileComponents section.

{@a component-with-dependency}

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`:

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:

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`.

{@a service-test-doubles}

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 mock of the `UserService` that satisfies the needs of the `WelcomeComponent` and its tests:

{@a get-injected-service}

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`.

{@a testbed-inject}

TestBed.inject() You *might* also be able to get the service from the root injector using `TestBed.inject()`. 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.

In this test suite, the *only* provider of `UserService` is the root testing module, so it is safe to call `TestBed.inject()` as follows:

For a use case in which `TestBed.inject()` does not work, see the *Override component providers* section that explains when and why you must get the service from the component's injector instead.

{@a welcome-spec-setup}

Final setup and tests Here's the complete `beforeEach()`, using `TestBed.inject()`:

And here are some tests:

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

The second parameter to the Jasmine matcher (for example, `'expected name'`) is an optional failure label. If the expectation fails, Jasmine appends this label to 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.

{@a component-with-async-service} ## Component with async service

In this sample, the `AboutComponent` template hosts a `TwainComponent`. The `TwainComponent` displays Mark Twain quotes.

Note that the value of the component's `quote` property passes through an `AsyncPipe`. That means the property returns either a `Promise` or an `Observable`.

In this example, the `TwainComponent.getQuote()` method tells you that the `quote` property returns an `Observable`.

The `TwainComponent` gets quotes from an injected `TwainService`. The component starts the returned `Observable` with a placeholder value (`'...'`), before the service can return its first quote.

The `catchError` intercepts service errors, prepares an error message, and returns the placeholder value on the success channel. It must wait a tick to set the `errorMessage` in order to avoid updating that message twice in the same change detection cycle.

These are all features you'll want to test.

Testing with a spy When testing a component, only the service's public API should matter. In general, tests themselves should not make calls to remote servers. They should emulate such calls. The setup in this `app/twain/twain.component.spec.ts` shows one way to do that:

{@a service-spy}

Focus on the spy.

The spy is designed such that any call to `getQuote` receives an observable with a test quote. Unlike the real `getQuote()` method, this spy bypasses the server and returns a synchronous observable whose value is available immediately.

You can write many useful tests with this spy, even though its `Observable` is synchronous.

{@a sync-tests}

Synchronous tests A key advantage of a synchronous `Observable` is that you can often turn asynchronous processes into synchronous tests.

Because the spy result returns synchronously, the `getQuote()` method updates the message on screen immediately *after* the first change detection cycle during which Angular calls `ngOnInit`.

You're not so lucky when testing the error path. Although the service spy will return an error synchronously, the component method calls `setTimeout()`. The test must wait at least one full turn of the JavaScript engine before the value becomes available. The test must become *asynchronous*.

{@a fake-async}

Async test with *fakeAsync()* To use `fakeAsync()` functionality, you must import `zone.js/testing` in your test setup file. If you created your project with the Angular CLI, `zone-testing` is already imported in `src/test.ts`.

The following test confirms the expected behavior when the service returns an `ErrorObservable`.

Note that the `it()` function receives an argument of the following form.

```
fakeAsync(() => { /* test body */ })
```

The `fakeAsync()` function enables a linear coding style by running the test body in a special `fakeAsync test zone`. The test body appears to be synchronous. There is no nested syntax (like a `Promise.then()`) to disrupt the flow of control.

Limitation: The `fakeAsync()` function won't work if the test body makes an `XMLHttpRequest` (XHR) call. XHR calls within a test are rare, but if you need to call XHR, see the `waitForAsync()` section.

{@a tick}

The *tick()* function You do have to call tick() to advance the (virtual) clock.

Calling tick() simulates the passage of time until all pending asynchronous activities finish. In this case, it waits for the error handler's `setTimeout()`.

The tick() function accepts milliseconds and tickOptions as parameters, the millisecond (defaults to 0 if not provided) parameter represents how much the virtual clock advances. For example, if you have a `setTimeout(fn, 100)` in a `fakeAsync()` test, you need to use tick(100) to trigger the fn callback. The tickOptions is an optional parameter with a property called `processNewMacroTasksSynchronously` (defaults to true) that represents whether to invoke new generated macro tasks when ticking.

The tick() function is one of the Angular testing utilities that you import with `TestBed`. It's a companion to `fakeAsync()` and you can only call it within a `fakeAsync()` body.

tickOptions In this example, you have a new macro task (nested setTimeout), by default, when the `tick` is setTimeout `outside` and `nested` will both be triggered.

And in some case, you don't want to trigger the new macro task when ticking, you can use `tick(milliseconds, {processNewMacroTasksSynchronously: false})` to not invoke new macro task.

Comparing dates inside fakeAsync() `fakeAsync()` simulates passage of time, which lets you calculate the difference between dates inside `fakeAsync()`.

jasmine.clock with fakeAsync() Jasmine also provides a `clock` feature to mock dates. Angular automatically runs tests that are run after `jasmine.clock().install()` is called inside a `fakeAsync()` method until `jasmine.clock().uninstall()` is called. `fakeAsync()` is not needed and throws an error if nested.

By default, this feature is disabled. To enable it, set a global flag before importing `zone-testing`.

If you use the Angular CLI, configure this flag in `src/test.ts`.

```
(window as any)['__zone_symbol__fakeAsyncPatchLock'] = true;
import 'zone.js/testing';
```

Using the RxJS scheduler inside fakeAsync() You can also use RxJS scheduler in `fakeAsync()` just like using `setTimeout()` or `setInterval()`, but you need to import `zone.js/plugins/zone-patch-rxjs-fake-async` to patch RxJS scheduler.

Support more macroTasks By default, `fakeAsync()` supports the following macro tasks.

- `setTimeout`
- `setInterval`
- `requestAnimationFrame`
- `webkitRequestAnimationFrame`
- `mozRequestAnimationFrame`

If you run other macro tasks such as `HTMLCanvasElement.toBlob()`, an *"Unknown macroTask scheduled in fake async test"* error is thrown.

If you want to support such a case, you need to define the macro task you want to support in `beforeEach()`. For example:

Note that in order to make the `<canvas>` element Zone.js-aware in your app, you need to import the `zone-patch-canvas` patch (either in `polyfills.ts` or in the specific file that uses `<canvas>`):

Async observables You might be satisfied with the test coverage of these tests.

However, you might be troubled by the fact that the real service doesn't quite behave this way. The real service sends requests to a remote server. A server takes time to respond and the response certainly won't be available immediately as in the previous two tests.

Your tests will reflect the real world more faithfully if you return an *asynchronous* observable from the `getQuote()` spy like this.

Async observable helpers The async observable was produced by an `asyncData` helper. The `asyncData` helper is a utility function that you'll have to write yourself, or copy this one from the sample code.

This helper's observable emits the `data` value in the next turn of the JavaScript engine.

The RxJS `defer()` operator returns an observable. It takes a factory function that returns either a promise or an observable. When something subscribes to *defer*'s observable, it adds the subscriber to a new observable created with that factory.

The `defer()` operator transforms the `Promise.resolve()` into a new observable that, like `HttpClient`, emits once and completes. Subscribers are unsubscribed after they receive the data value.

There's a similar helper for producing an async error.

More async tests Now that the `getQuote()` spy is returning async observables, most of your tests will have to be async as well.

Here's a `fakeAsync()` test that demonstrates the data flow you'd expect in the real world.

Notice that the quote element displays the placeholder value (`'...'`) after `ngOnInit()`. The first quote hasn't arrived yet.

To flush the first quote from the observable, you call tick(). Then call `detectChanges()` to tell Angular to update the screen.

Then you can assert that the quote element displays the expected text.

{@a waitForAsync}

Async test with *waitForAsync()* To use `waitForAsync()` functionality, you must import `zone.js/testing` in your test setup file. If you created your project with the Angular CLI, `zone-testing` is already imported in `src/test.ts`.

Here's the previous `fakeAsync()` test, re-written with the `waitForAsync()` utility.

The `waitForAsync()` utility hides some asynchronous boilerplate by arranging for the tester's code to run in a special *async test zone*. You don't need to pass Jasmine's `done()` into the test and call `done()` because it is `undefined` in promise or observable callbacks.

But the test's asynchronous nature is revealed by the call to `fixture.whenStable()`, which breaks the linear flow of control.

When using an `intervalTimer()` such as `setInterval()` in `waitForAsync()`, remember to cancel the timer with `clearInterval()` after the test, otherwise the `waitForAsync()` never ends.

{@a when-stable}

whenStable The test must wait for the `getQuote()` observable to emit the next quote. Instead of calling tick(), it calls `fixture.whenStable()`.

The `fixture.whenStable()` returns a promise that resolves when the JavaScript engine's task queue becomes empty. In this example, the task queue becomes empty when the observable emits the first quote.

The test resumes within the promise callback, which calls `detectChanges()` to update the quote element with the expected text.

{@a jasmine-done}

Jasmine *done()* While the `waitForAsync()` and `fakeAsync()` functions greatly simplify Angular asynchronous testing, you can still fall back to the traditional technique and pass `it` a function that takes a `done` callback.

You can't call `done()` in `waitForAsync()` or `fakeAsync()` functions, because the `done parameter` is `undefined`.

Now you are responsible for chaining promises, handling errors, and calling `done()` at the appropriate moments.

Writing test functions with `done()`, is more cumbersome than `waitForAsync()`and `fakeAsync()`, but it is occasionally necessary when code involves the `intervalTimer()` like `setInterval`.

Here are two more versions of the previous test, written with `done()`. The first one subscribes to the `Observable` exposed to the template by the component's `quote` property.

The RxJS `last()` operator emits the observable's last value before completing, which will be the test quote. The `subscribe` callback calls `detectChanges()` to update the quote element with the test quote, in the same manner as the earlier tests.

In some tests, you're more interested in how an injected service method was called and what values it returned, than what appears on screen.

A service spy, such as the `qetQuote()` spy of the fake `TwainService`, can give you that information and make assertions about the state of the view.

{@a marble-testing} ## Component marble tests

The previous `TwainComponent` tests simulated an asynchronous observable response from the `TwainService` with the `asyncData` and `asyncError` utilities.

These are short, simple functions that you can write yourself. Unfortunately, they're too simple for many common scenarios. An observable often emits multiple times, perhaps after a significant delay. A component might coordinate multiple observables with overlapping sequences of values and errors.

RxJS marble testing is a great way to test observable scenarios, both simple and complex. You've likely seen the marble diagrams that illustrate how observables work. Marble testing uses a similar marble language to specify the observable streams and expectations in your tests.

The following examples revisit two of the `TwainComponent` tests with marble testing.

Start by installing the `jasmine-marbles` npm package. Then import the symbols you need.

Here's the complete test for getting a quote:

Notice that the Jasmine test is synchronous. There's no `fakeAsync()`. Marble testing uses a test scheduler to simulate the passage of time in a synchronous test.

The beauty of marble testing is in the visual definition of the observable streams. This test defines a *cold* observable that waits three frames (`---`), emits a value (`x`), and completes (`|`). In the second argument you map the value marker (`x`) to the emitted value (`testQuote`).

The marble library constructs the corresponding observable, which the test sets as the `getQuote` spy's return value.

When you're ready to activate the marble observables, you tell the `TestScheduler` to *flush* its queue of prepared tasks like this.

This step serves a purpose analogous to tick() and `whenStable()` in the earlier `fakeAsync()` and `waitForAsync()` examples. The balance of the test is the same as those examples.

Marble error testing Here's the marble testing version of the `getQuote()` error test.

It's still an async test, calling `fakeAsync()` and tick(), because the component itself calls `setTimeout()` when processing errors.

Look at the marble observable definition.

This is a *cold* observable that waits three frames and then emits an error, The hash (`#`) indicates the timing of the error that is specified in the third argument. The second argument is null because the observable never emits a value.

Learn about marble testing {@a marble-frame} A *marble frame* is a virtual unit of testing time. Each symbol (`-`, `x`, `|`, `#`) marks the passing of one frame.

{@a cold-observable} A *cold* observable doesn't produce values until you subscribe to it. Most of your application observables are cold. All *HttpClient* methods return cold observables.

A *hot* observable is already producing values *before* you subscribe to it. The *Router.events* observable, which reports router activity, is a *hot* observable.

RxJS marble testing is a rich subject, beyond the scope of this guide. Learn about it on the web, starting with the official documentation.

{@a component-with-input-output} ## 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:

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 full definition:

{@a dashboard-hero-component}

While testing a component this simple has little intrinsic value, it's worth knowing how. 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:

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 following discussion 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.

{@a dashboard-standalone}

Test *DashboardHeroComponent* stand-alone Here's the meat of the spec file setup.

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

The following test verifies that the hero name is propagated to the template using a binding.

Because the template passes the hero name through the Angular `UpperCasePipe`, the test must match the element value with the upper-cased name.

This small test demonstrates how Angular tests can verify a component's visual representation—something not possible with component class tests—at low cost and without resorting to much slower and more complicated end-to-end tests.

Clicking Clicking the hero should raise a `selected` event that the host component (`DashboardComponent` presumably) can hear:

The component's `selected` property returns an `EventEmitter`, which looks like an RxJS synchronous `Observable` to consumers. The test subscribes to it *explicitly* just as the host component does *implicitly*.

If the component behaves as expected, clicking the hero's element should tell the component's `selected` property to emit the `hero` object.

The test detects that event through its subscription to `selected`.

{@a trigger-event-handler}

triggerEventHandler The `heroDe` in the previous test is a `DebugElement` that represents the hero `<div>`.

It has Angular properties and methods that abstract interaction with the native element. This test calls the `DebugElement.triggerEventHandler` with the "click" event name. The "click" event binding responds by calling `DashboardHeroComponent.click()`.

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.

The test triggered a "click" event.

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 (if any) was pressed during the click. The `RouterLink` directive throws an error if the event object is missing.

Click the element The following test alternative calls the native element's own `click()` method, which is perfectly fine for *this component*.

{@a click-helper}

***click()* helper** Clicking a button, an anchor, or an arbitrary HTML element is a common test task.

Make that consistent and straightforward by encapsulating the *click-triggering* process in a helper such as the following `click()` function:

The first parameter is the *element-to-click*. If you want, pass a custom event object as the second parameter. The default is a (partial) left-button mouse event object accepted by many handlers including the `RouterLink` directive.

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 the click helper.

{@a component-inside-test-host} ## Component inside a test host

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

You could test with the actual `DashboardComponent`. But doing so could require a lot of setup, especially when its template features an `*ngFor` repeater, other components, layout HTML, additional bindings, a constructor that injects multiple services, and it starts interacting with those services right away.

Imagine the effort to disable these distractions, just to prove a point that can be made satisfactorily with a *test host* like this one:

This test host binds to `DashboardHeroComponent` as the `DashboardComponent` would but without the noise of the `Router`, the `HeroService`, or 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 will be able to check `selectedHero` to verify that the `DashboardHeroComponent.selected` event emitted the expected hero.

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

This testing module configuration shows three important differences:

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

3. The `TestHostComponent` sets the `DashboardHeroComponent.hero` with a binding.

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:

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.

{@a routing-component} ## Routing component

A *routing component* is a component that tells the `Router` to navigate to another component. The `DashboardComponent` is a *routing component* because the user can navigate to the `HeroDetailComponent` by clicking on one of the *hero buttons* on the dashboard.

Routing is pretty complicated. Testing the `DashboardComponent` seemed daunting in part because it involves the `Router`, which it injects together with the `HeroService`.

Mocking the `HeroService` with a spy is a familiar story. But the `Router` has a complicated API and is entwined with other services and application preconditions. Might it be difficult to mock?

Fortunately, not in this case because the `DashboardComponent` isn't doing much with the `Router`

This is often the case with *routing components*. 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.

Providing a router spy for *this component* test suite happens to be as easy as providing a `HeroService` spy.

The following test clicks the displayed hero and confirms that `Router.navigateByUrl` is called with the expected url.

{@a routed-component-w-param}

Routed components

A *routed component* is the destination of a `Router` navigation. It can be trickier to test, especially when the route to the component *includes parameters*. The

`HeroDetailComponent` is a *routed component* that is the destination of such a route.

When a user clicks a *Dashboard* hero, the `DashboardComponent` tells the `Router` to navigate to `heroes/:id`. The `:id` is a route parameter whose value is the `id` of the hero to edit.

The `Router` matches that URL to a route to the `HeroDetailComponent`. It creates an `ActivatedRoute` object with the routing information and injects it into a new instance of the `HeroDetailComponent`.

Here's the `HeroDetailComponent` constructor:

The `HeroDetail` component needs the `id` parameter so it can fetch the corresponding hero using the `HeroDetailService`. The component has to get the `id` from the `ActivatedRoute.paramMap` property which is an `Observable`.

It can't just reference the `id` property of the `ActivatedRoute.paramMap`. The component has to *subscribe* to the `ActivatedRoute.paramMap` observable and be prepared for the `id` to change during its lifetime.

The ActivatedRoute in action section of the Router tutorial: tour of heroes guide covers `ActivatedRoute.paramMap` in more detail.

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

You know how to spy on the `Router` and a data service.

You'll take a different approach with `ActivatedRoute` because

- `paramMap` returns an `Observable` that can emit more than one value during a test.
- You need the router helper function, `convertToParamMap()`, to create a `ParamMap`.
- Other *routed component* tests need a test double for `ActivatedRoute`.

These differences argue for a re-usable stub class.

ActivatedRouteStub The following `ActivatedRouteStub` class serves as a test double for `ActivatedRoute`.

Consider placing such helpers in a `testing` folder sibling to the `app` folder. This sample puts `ActivatedRouteStub` in `testing/activated-route-stub.ts`.

Consider writing a more capable version of this stub class with the *marble testing library*.

{@a tests-w-test-double}

Testing with *ActivatedRouteStub* Here's a test demonstrating the component's behavior when the observed `id` refers to an existing hero:

In the following section, the `createComponent()` method and `page` object are discussed. 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 router spy described above which spies on the router without actually navigating.

This test expects the component to try to navigate to the `HeroListComponent`.

While this application 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:

Nested component tests

Component templates often have nested components, whose templates might contain more components.

The component tree can be very deep and, most of the time, the nested components play no role in testing the component at the top of the tree.

The `AppComponent`, for example, displays a navigation bar with anchors and their `RouterLink` directives.

While the `AppComponent` *class* is empty, you might want to write unit tests to confirm that the links are wired properly to the `RouterLink` directives, perhaps for the reasons as explained in the following section.

To validate the links, you don't need the `Router` to navigate and you don't need the `<router-outlet>` to mark where the `Router` inserts *routed components*.

The `BannerComponent` and `WelcomeComponent` (indicated by `<app-banner>` and `<app-welcome>`) are also irrelevant.

Yet any test that creates the `AppComponent` in the DOM also creates instances of these three components and, if you let that happen, you'll have to configure the `TestBed` to create them.

If you neglect to declare them, the Angular compiler won't recognize the `<app-banner>`, `<app-welcome>`, and `<router-outlet>` tags in the `AppComponent` template and will throw an error.

If you declare the real components, you'll also have to declare *their* nested components and provide for *all* services injected in *any* component in the tree.

That's too much effort just to answer a few simple questions about links.

This section describes two techniques for minimizing the setup. Use them, alone or in combination, to stay focused on testing the primary component.

{@a stub-component}

Stubbing unneeded components In the first technique, you create and declare stub versions of the components and directive that play little or no role in the tests.

The stub selectors match the selectors for the corresponding real components. But their templates and classes are empty.

Then declare them in the `TestBed` configuration next to the components, directives, and pipes that need to be real.

The `AppComponent` is the test subject, so of course you declare the real version.

The `RouterLinkDirectiveStub`, described later, is a test version of the real `RouterLink` that helps with the link tests.

The rest are stubs.

{@a no-errors-schema}

NO_ERRORS_SCHEMA In the second approach, add `NO_ERRORS_SCHEMA` to the `TestBed.schemas` metadata.

The `NO_ERRORS_SCHEMA` tells the Angular compiler to ignore unrecognized elements and attributes.

The compiler recognizes the `<app-root>` element and the `routerLink` attribute because you declared a corresponding `AppComponent` and `RouterLinkDirectiveStub` in the `TestBed` configuration.

But the compiler won't throw an error when it encounters `<app-banner>`, `<app-welcome>`, or `<router-outlet>`. It simply renders them as empty tags and the browser ignores them.

You no longer need the stub components.

Use both techniques together These are techniques for *Shallow Component Testing* , so-named because they reduce the visual surface of the component to just those elements in the component's template that matter for tests.

The `NO_ERRORS_SCHEMA` approach is the easier of the two but don't overuse it.

The `NO_ERRORS_SCHEMA` also prevents the compiler from telling you about the missing components and attributes that you omitted inadvertently or misspelled. You could waste hours chasing phantom bugs that the compiler would have caught in an instant.

The *stub component* approach has another advantage. While the stubs in *this* example were empty, you could give them stripped-down templates and classes if your tests need to interact with them in some way.

In practice you will combine the two techniques in the same setup, as seen in this example.

The Angular compiler creates the `BannerComponentStub` for the `<app-banner>` element and applies the `RouterLinkStubDirective` to the anchors with the `routerLink` attribute, but it ignores the `<app-welcome>` and `<router-outlet>` tags.

{@a routerlink} ## Components with *RouterLink*

The real `RouterLinkDirective` is quite complicated and entangled with other components and directives of the `RouterModule`. It requires challenging setup to mock and use in tests.

The `RouterLinkDirectiveStub` in this sample code replaces the real directive with an alternative version designed to validate the kind of anchor tag wiring seen in the `AppComponent` template.

The URL bound to the `[routerLink]` attribute flows in to the directive's `linkParams` property.

The `HostListener` wires the click event of the host element (the `<a>` anchor elements in `AppComponent`) to the stub directive's `onClick` method.

Clicking the anchor should trigger the `onClick()` method, which sets the stub's telltale `navigatedTo` property. Tests inspect `navigatedTo` to confirm that clicking the anchor sets the expected route definition.

Whether the router is configured properly to navigate with that route definition is a question for a separate set of tests.

{@a by-directive} {@a inject-directive}

***By.directive* and injected directives** A little more setup triggers the initial data binding and gets references to the navigation links:

Three points of special interest:

1. Locate the anchor elements with an attached directive using `By.directive`.
2. The query returns `DebugElement` wrappers around the matching elements.
3. Each `DebugElement` exposes a dependency injector with the specific instance of the directive attached to that element.

The `AppComponent` links to validate are as follows:

{@a app-component-tests}

Here are some tests that confirm those links are wired to the `routerLink` directives as expected:

The "click" test *in this example* is misleading. It tests the `RouterLinkDirectiveStub` rather than the *component*. This is a common failing of directive stubs.

It has a legitimate purpose in this guide. It demonstrates how to find a `RouterLink` element, click it, and inspect a result, without engaging the full router machinery. This is a skill you might need to test a more sophisticated component, one that changes the display, re-calculates parameters, or re-arranges navigation options when the user clicks the link.

{@a why-stubbed-routerlink-tests}

What good are these tests? Stubbed `RouterLink` tests can confirm that a component with links and an outlet is setup properly, that the component has the links it should have, and that they are all pointing in the expected direction. These tests do not concern whether the application will succeed in navigating to the target component when the user clicks a link.

Stubbing the RouterLink and RouterOutlet is the best option for such limited testing goals. Relying on the real router would make them brittle. They could fail for reasons unrelated to the component. For example, a navigation guard could prevent an unauthorized user from visiting the `HeroListComponent`. That's not the fault of the `AppComponent` and no change to that component could cure the failed test.

A *different* battery of tests can explore whether the application navigates as expected in the presence of conditions that influence guards such as whether the user is authenticated and authorized.

A future guide update explains how to write such tests with the `RouterTestingModule`.

{@a page-object} ## Use a *page* object

The `HeroDetailComponent` is a simple view with a title, two hero fields, and two buttons.

But there's plenty of template complexity even in this simple form.

Tests that exercise the component need ...

- to wait until a hero arrives before elements appear in the DOM.
- a reference to the title text.
- a reference to the name input box to inspect and set it.
- references to the two buttons so they can click them.
- spies for some of the component and router methods.

Even a small form such as this one can produce a mess of tortured conditional setup and CSS element selection.

Tame the complexity with a `Page` class that handles access to component properties and encapsulates the logic that sets them.

Here is such a `Page` class for the `hero-detail.component.spec.ts`

Now the important hooks for component manipulation and inspection are neatly organized and accessible from an instance of `Page`.

A `createComponent` method creates a `page` object and fills in the blanks once the `hero` arrives.

The *HeroDetailComponent* tests in an earlier section demonstrate how `createComponent` and `page` keep the tests short and *on message*. There are no distractions: no waiting for promises to resolve and no searching the DOM for element values to compare.

Here are a few more `HeroDetailComponent` tests to reinforce the point.

{@a compile-components} ## Calling *compileComponents()*

Ignore this section if you *only* run tests with the CLI `ng test` command because the CLI compiles the application before running the tests.

If you run tests in a **non-CLI environment**, the tests might fail with a message like this one:

Error: This test module uses the component BannerComponent which is using a "templateUrl" or "styleUrls", but they were never compiled. Please call "TestBed.compileComponents" before your test.

The root of the problem is at least one of the components involved in the test specifies an external template or CSS file as the following version of the `BannerComponent` does.

The test fails when the `TestBed` tries to create the component.

Recall that the application hasn't been compiled. So when you call `createComponent()`, the `TestBed` compiles implicitly.

That's not a problem when the source code is in memory. But the `BannerComponent` requires external files that the compiler must read from the file system, an inherently *asynchronous* operation.

If the `TestBed` were allowed to continue, the tests would run and fail mysteriously before the compiler could finished.

The preemptive error message tells you to compile explicitly with `compileComponents()`.

***compileComponents()* is async** You must call `compileComponents()` within an asynchronous test function.

If you neglect to make the test function async (for example, forget to use `waitForAsync()` as described), you'll see this error message

Error: ViewDestroyedError: Attempt to use a destroyed view

A typical approach is to divide the setup logic into two separate `beforeEach()` functions:

1. An async `beforeEach()` that compiles the components
2. A synchronous `beforeEach()` that performs the remaining setup.

The async *beforeEach* Write the first async `beforeEach` like this.

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

In this example, the `BannerComponent` is the only component to compile. Other examples configure the testing module with multiple components and might import application modules that hold yet more components. Any of them could require external files.

The `TestBed.compileComponents` method asynchronously compiles all components configured in the testing module.

Do not re-configure the `TestBed` after calling `compileComponents()`.

Calling `compileComponents()` 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.

Make `compileComponents()` the last step before calling `TestBed.createComponent()`.

The synchronous *beforeEach* The second, synchronous `beforeEach()` contains the remaining setup steps, which include creating the component and querying for elements to inspect.

Count on the test runner to wait for the first asynchronous `beforeEach` to finish before calling the second.

Consolidated setup You can consolidate the two `beforeEach()` functions into a single, async `beforeEach()`.

The `compileComponents()` method returns a promise so you can perform the synchronous setup tasks *after* compilation by moving the synchronous code after the `await` keyword, where the promise has been resolved.

***compileComponents()* is harmless** There's no harm in calling `compileComponents()` when it's not required.

The component test file generated by the CLI calls `compileComponents()` even though it is never required when running `ng test`.

The tests in this guide only call `compileComponents` when necessary.

{@a import-module} ## Setup with module imports

Earlier component tests configured the testing module with a few `declarations` like this:

The `DashboardComponent` is simple. It needs no help. But more complex components often depend on other components, directives, pipes, and providers and these must be added to the testing module too.

Fortunately, the `TestBed.configureTestingModule` parameter parallels the metadata passed to the `@NgModule` decorator which means you can also specify `providers` and `imports`.

The `HeroDetailComponent` requires a lot of help despite its small size and simple construction. In addition to the support it receives from the default testing module `CommonModule`, it needs:

- `NgModel` and friends in the `FormsModule` to enable two-way data binding.
- The `TitleCasePipe` from the `shared` folder.
- Router services (which these tests are stubbing).
- Hero data access services (also stubbed).

One approach is to configure the testing module from the individual pieces as in this example:

Notice that the `beforeEach()` is asynchronous and calls `TestBed.compileComponents` because the `HeroDetailComponent` has an external template and css file.

As explained in *Calling compileComponents()*, these tests could be run in a non-CLI environment where Angular would have to compile them in the browser.

Import a shared module Because many application components need the `FormsModule` and the `TitleCasePipe`, the developer created a `SharedModule` to combine these and other frequently requested parts.

The test configuration can use the `SharedModule` too as seen in this alternative setup:

It's a bit tighter and smaller, with fewer import statements (not shown).

{@a feature-module-import}

Import a feature module The `HeroDetailComponent` is part of the `HeroModule` Feature Module that aggregates more of the interdependent pieces including the `SharedModule`. Try a test configuration that imports the `HeroModule` like this one:

That's *really* crisp. Only the *test doubles* in the `providers` remain. Even the `HeroDetailComponent` declaration is gone.

In fact, if you try to declare it, Angular will throw an error because `HeroDetailComponent` is declared in both the `HeroModule` and the `DynamicTestModule` created by the `TestBed`.

Importing the component's feature module can be the best way to configure tests when there are many mutual dependencies within the module and the module is small, as feature modules tend to be.

{@a component-override} ## Override component providers

The `HeroDetailComponent` provides its own `HeroDetailService`.

It's not possible to stub the component's `HeroDetailService` in the `providers` of the `TestBed.configureTestingModule`. Those are providers for the *testing module*, not the component. They prepare the dependency injector at the *fixture level*.

Angular creates the component with its *own* injector, which is a *child* of the fixture injector. It registers the component's providers (the `HeroDetailService` in this case) with the child injector.

A test cannot get to child injector services from the fixture injector. And `TestBed.configureTestingModule` can't configure them either.

Angular has created new instances of the real `HeroDetailService` all along!

These tests could fail or timeout if the `HeroDetailService` made its own XHR calls to a remote server. There might not be a remote server to call.

Fortunately, the `HeroDetailService` delegates responsibility for remote data access to an injected `HeroService`.

The previous test configuration replaces the real `HeroService` with a `TestHeroService` that intercepts server requests and fakes their responses.

What if you aren't so lucky. What if faking the `HeroService` is hard? What if `HeroDetailService` makes its own server requests?

The `TestBed.overrideComponent` method can replace the component's `providers` with easy-to-manage *test doubles* as seen in the following setup variation:

Notice that `TestBed.configureTestingModule` no longer provides a (fake) `HeroService` because it's not needed.

{@a override-component-method}

The *overrideComponent* method Focus on the `overrideComponent` method.

It takes two arguments: the component type to override (`HeroDetailComponent`) and an override metadata object. The override metadata object is a generic defined as follows:

type MetadataOverride<T> = { add?: Partial<T>; remove?: Partial<T>; set?: Partial<T>; };

A metadata override object can either add-and-remove elements in metadata properties or completely reset those properties. This example resets the component's `providers` metadata.

The type parameter, `T`, is the kind of metadata you'd pass to the `@Component` decorator:

selector?: string; template?: string; templateUrl?: string; providers?: any[]; . . .

{@a spy-stub}

Provide a *spy stub* (*HeroDetailServiceSpy*) This example completely replaces the component's `providers` array with a new array containing a `HeroDetailServiceSpy`.

The `HeroDetailServiceSpy` is a stubbed version of the real `HeroDetailService` that fakes all necessary features of that service. It neither injects nor delegates to the lower level `HeroService` so there's no need to provide a test double for that.

The related `HeroDetailComponent` tests will assert that methods of the `HeroDetailService` were called by spying on the service methods. Accordingly, the stub implements its methods as spies:

{@a override-tests}

The override tests Now the tests can control the component's hero directly by manipulating the spy-stub's `testHero` and confirm that service methods were called.

{@a more-overrides}

More overrides The `TestBed.overrideComponent` method can be called multiple times for the same or different components. The `TestBed` offers similar `overrideDirective`, `overrideModule`, and `overridePipe` methods for digging into and replacing parts of these other classes.

Explore the options and combinations on your own.