# Basics of testing components

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

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

The Angular `TestBed` facilitates this kind of testing as you'll see in the following sections. But in many cases, *testing the component class alone*, without DOM involvement, can validate much of the component's behavior in a straightforward, more obvious way.

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

{@a component-class-testing}

## Component class testing

Test a component class on its own as you would test a service class.

Component class testing should be kept very clean and simple. It should test only a single unit. At first glance, you should be able to understand what the test is testing.

Consider this `LightswitchComponent` which toggles a light on and off (represented by an on-screen message) when the user clicks the button.

You might decide only to test that the `clicked()` method toggles the light's *on/off* state and sets the message appropriately.

This component class has no dependencies. To test these types of classes, follow the same steps as you would for a service that has no dependencies:

1. Create a component using the new keyword.
2. Poke at its API.
3. Assert expectations on its public state.

Here is the `DashboardHeroComponent` from the *Tour of Heroes* tutorial.

It appears within the template of a parent component, which binds a *hero* to the `@Input` property and listens for an event raised through the *selected* `@Output` property.

You can test that the class code works without creating the `DashboardHeroComponent` or its parent component.

When a component has dependencies, you might want to use the `TestBed` to both create the component and its dependencies.

The following `WelcomeComponent` depends on the `UserService` to know the name of the user to greet.

You might start by creating a mock of the `UserService` that meets the minimum needs of this component.

Then provide and inject *both the* **component** *and the service* in the `TestBed` configuration.

Then exercise the component class, remembering to call the lifecycle hook methods as Angular does when running the application.

## Component DOM testing

Testing the component *class* is as straightforward as testing a service.

But a component is more than just its class. A component interacts with the DOM and with other components. The *class-only* tests can tell you about class behavior. They cannot tell you if the component is going to render properly, respond to user input and gestures, or integrate with its parent and child components.

None of the preceding *class-only* tests can answer key questions about how the components actually behave on screen.

- Is `Lightswitch.clicked()` bound to anything such that the user can invoke it?
- Is the `Lightswitch.message` displayed?
- Can the user actually select the hero displayed by `DashboardHeroComponent` ?
- Is the hero name displayed as expected (i.e, in uppercase)?
- Is the welcome message displayed by the template of `WelcomeComponent` ?

These might not be troubling questions for the preceding simple components illustrated. But many components have complex interactions with the DOM elements described in their templates, causing HTML to appear and disappear as the component state changes.

To answer these kinds of questions, you have to create the DOM elements associated with the components, you must examine the DOM to confirm that component state displays properly at the appropriate times, and you must simulate user interaction with the screen to determine whether those interactions cause the component to behave as expected.

To write these kinds of test, you'll use additional features of the `TestBed` as well as other testing helpers.

### CLI-generated tests

The CLI creates an initial test file for you by default when you ask it to generate a new component.

For example, the following CLI command generates a `BannerComponent` in the `app/banner` folder (with inline template and styles):

ng generate component banner --inline-template --inline-style --module app

It also generates an initial test file for the component, `banner-external.component.spec.ts` , that looks like this:

Because `compileComponents` is asynchronous, it uses the `waitForAsync` utility function imported from `@angular/core/testing` .

Refer to the waitForAsync section for more details.

### Reduce the setup

Only the last three lines of this file actually test the component and all they do is assert that Angular can create the component.

The rest of the file is boilerplate setup code anticipating more advanced tests that *might* become necessary if the component evolves into something substantial.

You'll learn about these advanced test features in the following sections. For now, you can radically reduce this test file to a more manageable size:

In this example, the metadata object passed to `TestBed.configureTestingModule` simply declares `BannerComponent`, the component to test.

There's no need to declare or import anything else. The default test module is pre-configured with something like the `BrowserModule` from `@angular/platform-browser`.

Later you'll call `TestBed.configureTestingModule()` with imports, providers, and more declarations to suit your testing needs. Optional `override` methods can further fine-tune aspects of the configuration.

{@a create-component}

### *createComponent()*

After configuring `TestBed`, you call its `createComponent()` method.

`TestBed.createComponent()` creates an instance of the `BannerComponent`, adds a corresponding element to the test-runner DOM, and returns a `ComponentFixture`.

Do not re-configure `TestBed` after calling `createComponent`.

The `createComponent` method freezes the current `TestBed` definition, closing it to further configuration.

You cannot call any more `TestBed` configuration methods, not `configureTestingModule()`, nor `get()`, nor any of the `override...` methods. If you try, `TestBed` throws an error.

{@a component-fixture}

### *ComponentFixture*

The ComponentFixture is a test harness for interacting with the created component and its corresponding element.

Access the component instance through the fixture and confirm it exists with a Jasmine expectation:

### *beforeEach()*

You will add more tests as this component evolves. Rather than duplicate the `TestBed` configuration for each test, you refactor to pull the setup into a Jasmine `beforeEach()` and some supporting variables:

<code-example path="testing/src/app/banner/banner-initial.component.spec.ts" region="v3"

Now add a test that gets the component's element from `fixture.nativeElement` and looks for the expected text.

{@a native-element}

### *nativeElement*

The value of `ComponentFixture.nativeElement` has the `any` type. Later you'll encounter the `DebugElement.nativeElement` and it too has the `any` type.

Angular can't know at compile time what kind of HTML element the `nativeElement` is or if it even is an HTML element. The application might be running on a *non-browser platform*, such as the server or a Web Worker, where the element might have a diminished API or not exist at all.

The tests in this guide are designed to run in a browser so a `nativeElement` value will always be an `HTMLElement` or one of its derived classes.

Knowing that it is an `HTMLElement` of some sort, use the standard HTML `querySelector` to dive deeper into the element tree.

Here's another test that calls `HTMLElement.querySelector` to get the paragraph element and look for the banner text:

{@a debug-element}

#### *DebugElement*

The Angular *fixture* provides the component's element directly through the `fixture.nativeElement`.

This is actually a convenience method, implemented as `fixture.debugElement.nativeElement`.

There's a good reason for this circuitous path to the element.

The properties of the `nativeElement` depend upon the runtime environment. You could be running these tests on a *non-browser* platform that doesn't have a DOM or whose DOM-emulation doesn't support the full `HTMLElement` API.

Angular relies on the `DebugElement` abstraction to work safely across *all supported platforms*. Instead of creating an HTML element tree, Angular creates a `DebugElement` tree that wraps the *native elements* for the runtime platform. The `nativeElement` property unwraps the `DebugElement` and returns the platform-specific element object.

Because the sample tests for this guide are designed to run only in a browser, a `nativeElement` in these tests is always an `HTMLElement` whose familiar methods and properties you can explore within a test.

Here's the previous test, re-implemented with `fixture.debugElement.nativeElement`:

The `DebugElement` has other methods and properties that are useful in tests, as you'll see elsewhere in this guide.

You import the `DebugElement` symbol from the Angular core library.

{@a by-css}

#### *By.css()*

Although the tests in this guide all run in the browser, some applications might run on a different platform at least some of the time.

For example, the component might render first on the server as part of a strategy to make the application launch faster on poorly connected devices. The server-side renderer might not support the full HTML element API. If it doesn't support `querySelector`, the previous test could fail.

The `DebugElement` offers query methods that work for all supported platforms. These query methods take a *predicate* function that returns `true` when a node in the `DebugElement` tree matches the selection criteria.

You create a *predicate* with the help of a `By` class imported from a library for the runtime platform. Here's the `By` import for the browser platform:

The following example re-implements the previous test with `DebugElement.query()` and the browser's `By.css` method.

Some noteworthy observations:

- The `By.css()` static method selects `DebugElement` nodes with a standard CSS selector.
- The query returns a `DebugElement` for the paragraph.
- You must unwrap that result to get the paragraph element.

When you're filtering by CSS selector and only testing properties of a browser's *native element*, the `By.css` approach might be overkill.

It's often straightforward and more clear to filter with a standard `HTMLElement` method such as `querySelector()` or `querySelectorAll()`.