# Testing

This guide offers tips and techniques for testing Angular applications. Though this page includes some general testing principles and techniques, the focus is on testing applications written with Angular.

{@a top}

## Live examples

This guide presents tests of a sample application that is much like the *Tour of Heroes* tutorial. The sample application and all tests in this guide are available as live examples for inspection, experiment, and download:

- A spec to verify the test environment.
- The first component spec with inline template.
- A component spec with external template.
- The QuickStart seed's AppComponent spec.
- The sample application to be tested.
- All specs that test the sample application.
- A grab bag of additional specs.

---

{@a testing-intro}

## 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.

{@a tools-and-tech}

## 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. |

{@a setup}

## Setup

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

1. Start a new project following the instructions in Setup.
2. Start a new project with the Angular CLI.

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. For a discussion of the unit testing setup files, see below.

{@a isolated-v-testing-utilities}

### Isolated unit tests vs. the Angular testing utilities

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. A comprehensive review of the Angular testing utilities appears later in this guide.

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.

---

{@a 1st-karma-test}

# 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.

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

{@a run-karma}

### Run with karma

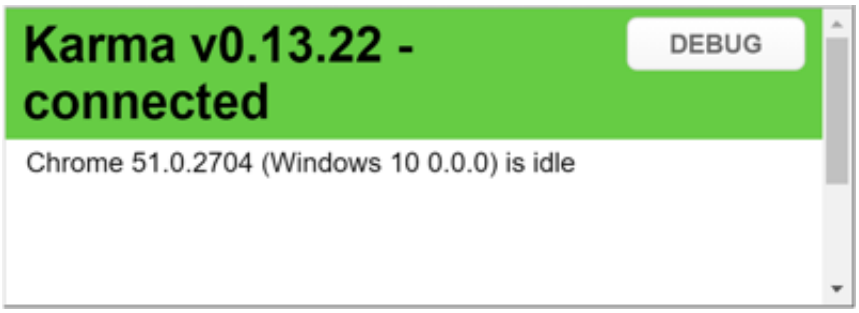
Compile and run it in karma from the command line using the following command:

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:

> 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.

[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.

[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.

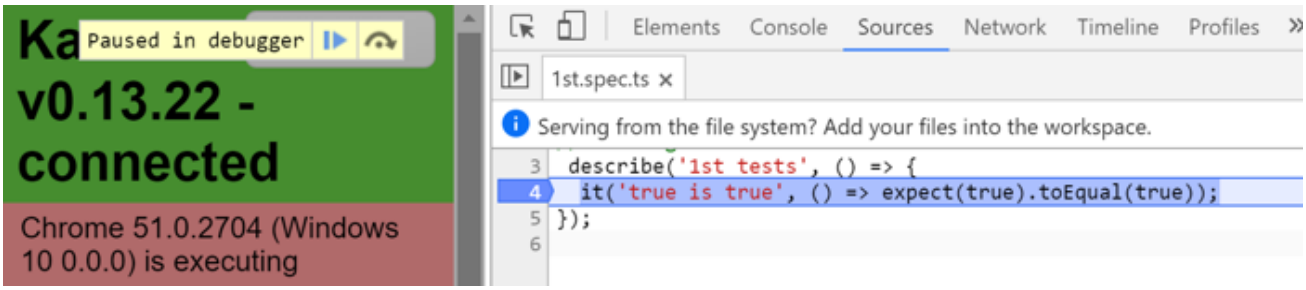
The console log can be quite long. Keep your eye on the last line. When all is well, it reads `SUCCESS`.

{@a test-debugging}

### 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.




{@a live-karma-example}

### Try the live example

You can also try this test as a in plunker. All of the tests in this guide are available as live examples.

---

{@a simple-component-test}

## 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.

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 answer to "Why put specs next to the things they test?".

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

{@a configure-testing-module}

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

{@a testbed}

### *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 NgModule.

*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](guide/testing#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.

{@a create-component}

### *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*.

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.

{@a component-fixture}

### *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 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.

{@a the-tests}

## The tests

Jasmine runs the `beforeEach` function before each of these tests

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

{@a detect-changes}

## *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:

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*.

{@a try-example}

## Try the live example

Take a moment to explore this component spec as a and lock in these fundamentals of component unit testing.

{@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 component-with-external-template}

# 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.

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.

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.

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.

{@a compile-components}

### *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 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.

{@a second-before-each}

## The second synchronous *beforeEach*

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

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.

{@a waiting-compile-components}

### 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.

{@a live-external-template-example}

### Try the live example

Take a moment to explore this component spec as a .

The [Quickstart seed](guide/setup) provides a similar test of its `AppComponent` as you can see in _this_ . 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.

---

{@a component-with-dependency}

## 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`:

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`:

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 `UserService` stub 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-get}

### *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:

The [`inject`](guide/testing#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_](guide/testing#component-override), which explains why you must get the service from the component's injector instead.

{@a service-from-injector}

## 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`.

{@a welcome-spec-setup}

### Final setup and tests

Here's the complete `beforeEach` using `TestBed.get`:

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 (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.

---

{@a component-with-async-service}

## 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`.

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:

{@a service-spy}

## Spying on the real service

This setup is similar to the `welcome.component.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.

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](guide/testing#spy-stub).

Here are the tests with commentary to follow:

{@a sync-tests}

## 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*.

{@a async}

## The *async* function in *it*

Notice the `async` in the third test.

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 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](guide/testing#fake-async), removes this artifact and affords a more linear coding experience.

{@a when-stable}

### *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.

{@a fakeAsync}

{@a fake-async}

## The *fakeAsync* function

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

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

Like 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`.

{@a tick}

## 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.

{@a jasmine-done}

## *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. 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:

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.

---

{@a component-with-input-output}

# 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:

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:

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:

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](guide/testing#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.

{@a dashboard-standalone}

## Test *DashboardHeroComponent* stand-alone

Here's the spec file setup.

The async `beforeEach` was discussed above. Having compiled the components asynchronously with `compileComponents`, the rest of the setup proceeds *synchronously* in a *second* `beforeEach`, using the basic techniques described earlier.

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:

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:

This small test demonstrates how Angular tests can verify a component's visual representation—something not possible with [isolated unit tests](guide/testing#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:

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.

{@a trigger-event-handler}

### *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.

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.

{@a click-helper}

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:

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 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.

---

{@a component-inside-test-host}

# 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:

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:

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:

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 routed-component}

# Test a routed component

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

It also injects the `HeroService`, but faking that is a familiar story. 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`

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:

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.

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

{@a inject}

## The *inject* function

Notice the `inject` function in the second `it` argument.

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:

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`.

{@a routed-component-w-param}

## 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:

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

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 the`id` parameter is missing, the `pluck` operator fails and the `catch` treats failure as a request to edit a new hero. The [Router] (guide/router#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*.

{@a stub-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`:

Notable features of this stub are:

- The stub implements only two of the `ActivatedRoute` capabilities: `params` and `snapshot.params`.
- *BehaviorSubject* 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_](guide/router#snapshot "Router guide: 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.

{@a tests-w-observable-double}

### Testing with the *Observable* test double

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

The `createComponent` method and `page` object are discussed [in the next section](guide/testing#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 which spies on the router without actually navigating. This test supplies a "bad" id and expects the component to try to navigate.

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:

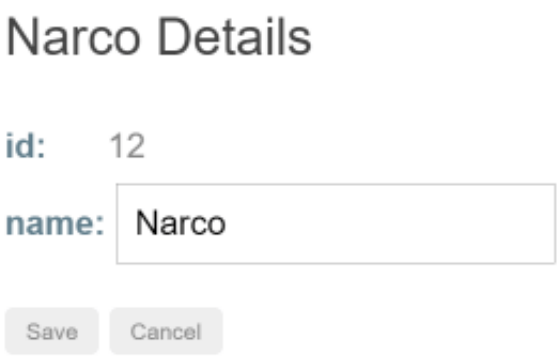
Inspect and download _all_ of the guide's application test code with this live example.

---

{@a page-object}

## Use a *page* object to simplify setup

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




But there's already plenty of template complexity.

To fully exercise the component, the test needs a lot of setup:

- It must wait until a hero arrives before `*ngIf` allows any element in DOM.
- It needs references to the title `<span>` and the name `<input>` so it can inspect their values.
- It needs references to the two buttons so it can click them.

- It needs 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 madness with a `Page` class that simplifies access to component properties and encapsulates the logic that sets them. Here's the `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 observable tests in the previous 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 drive the point home.

---

{@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:

Because many app 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 the 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 throws an error because `HeroDetailComponent` is declared in both the `HeroModule` and the `DynamicTestModule` (the testing module).
Importing the component's feature module is often the easiest way to configure the tests, especially when the feature module is small and mostly self-contained, as feature modules should be.

---

{@a component-override}

## Override a component's 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 been creating 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](guide/testing#feature-module-

import) replaces the real `HeroService` with a `FakeHeroService` 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 overide metadata object is a generic defined as follows:

type MetadataOverride = { add?: T; remove?: T; set?: 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.

---

{@a router-outlet-component}

## Test a *RouterOutlet* component

The `AppComponent` displays routed components in a `<router-outlet>`. It also displays a navigation bar with anchors and their `RouterLink` directives.

{@a app-component-html}

The component class does nothing.

Unit tests can confirm that the anchors are wired properly without engaging the router. See why this is worth doing below.

{@a stub-component}

### Stubbing unneeded components

The test setup should look familiar.

The `AppComponent` is the declared test subject.

The setup extends the default testing module with one real component ( `BannerComponent` ) and several stubs.

- `BannerComponent` is simple and harmless to use as is.
- The real `WelcomeComponent` has an injected service. `WelcomeStubComponent` is a placeholder with no service to worry about.
- The real `RouterOutlet` is complex and errors easily. The `RouterOutletStubComponent` (in `testing/router-stubs.ts`) is safely inert.

The component stubs are essential. Without them, the Angular compiler doesn't recognize the `<app-welcome>` and `<router-outlet>` tags and throws an error.

{@a router-link-stub}

## Stubbing the *RouterLink*

The `RouterLinkStubDirective` contributes substantively to the test:

The `host` metadata property wires the click event of the host element (the `<a>`) to the directive's `onClick` method. The URL bound to the `[routerLink]` attribute flows to the directive's `linkParams` property. Clicking the anchor should trigger the `onClick` method which sets the telltale `navigatedTo` property. Tests can inspect that property to confirm the expected *click-to-navigation* behavior.

{@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:

Two points of special interest:

1. You can locate elements *by directive*, using `By.directive`, not just by css selectors.
2. You can use the component's dependency injector to get an attached directive because Angular always adds attached directives to the component's injector.

{@a app-component-tests}

Here are some tests that leverage this setup:

The "click" test _in this example_ is worthless. It works hard to appear useful when in fact it tests the `RouterLinkStubDirective` 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 may 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 app 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 will explain how to write such tests with the `RouterTestingModule`.

---

{@a shallow-component-test}

## "Shallow component tests" with *NO_ERRORS_SCHEMA*

The previous setup declared the `BannerComponent` and stubbed two other components for *no reason other than to avoid a compiler error*.

Without them, the Angular compiler doesn't recognize the `<app-banner>`, `<app-welcome>` and `<router-outlet>` tags in the *app.component.html* template and throws an error.

Add `NO_ERRORS_SCHEMA` to the testing module's `schemas` metadata to tell the compiler to ignore unrecognized elements and attributes. You no longer have to declare irrelevant components and directives.

These tests are ***shallow*** because they only "go deep" into the components you want to test.

Here is a setup, with `import` statements, that demonstrates the improved simplicity of *shallow* tests, relative to the stubbing setup.

The *only* declarations are the *component-under-test* ( `AppComponent` ) and the `RouterLinkStubDirective` that contributes actively to the tests. The tests in this example are unchanged.

_Shallow component tests_ with `NO_ERRORS_SCHEMA` greatly simplify unit testing of complex templates. However, the compiler no longer alerts you to mistakes such as misspelled or misused components and directives.

---

{@a attribute-directive}

# Test an attribute directive

An *attribute directive* modifies the behavior of an element, component or another directive. Its name reflects the way the directive is applied: as an attribute on a host element.

The sample application's `HighlightDirective` sets the background color of an element based on either a data bound color or a default color (lightgray). It also sets a custom property of the element ( `customProperty` ) to `true` for no reason other than to show that it can.

It's used throughout the application, perhaps most simply in the `AboutComponent` :

Testing the specific use of the `HighlightDirective` within the `AboutComponent` requires only the techniques explored above (in particular the "Shallow test" approach).

However, testing a single use case is unlikely to explore the full range of a directive's capabilities. Finding and testing all components that use the directive is tedious, brittle, and almost as unlikely to afford full coverage.

Isolated unit tests might be helpful, but attribute directives like this one tend to manipulate the DOM. Isolated unit tests don't touch the DOM and, therefore, do not inspire confidence in the directive's efficacy.

A better solution is to create an artificial test component that demonstrates all ways to apply the directive.


The ` ` case binds the `HighlightDirective` to the name of a color value in the input box.

The initial value is the word "cyan" which should be the background color of the input box.

Here are some tests of this component:

A few techniques are noteworthy:

- The `By.directive` predicate is a great way to get the elements that have this directive *when their element types are unknown*.
- The `:not` pseudo-class in `By.css('h2:not([highlight])')` helps find `<h2>` elements that *do not* have the directive. `By.css('*:not([highlight])')` finds *any* element that does not have the directive.
- `DebugElement.styles` affords access to element styles even in the absence of a real browser, thanks to the `DebugElement` abstraction. But feel free to exploit the `nativeElement` when that seems easier or more clear than the abstraction.
- Angular adds a directive to the injector of the element to which it is applied. The test for the default color uses the injector of the second `<h2>` to get its `HighlightDirective` instance and its `defaultColor`.
- `DebugElement.properties` affords access to the artificial custom property that is set by the directive.

---

{@a isolated-unit-tests}

# Isolated Unit Tests

Testing applications with the help of the Angular testing utilities is the main focus of this guide.

However, it's often more productive to explore the inner logic of application classes with *isolated* unit tests that don't depend upon Angular. Such tests are often smaller and easier to read, write, and maintain.

They don't carry extra baggage:

- Import from the Angular test libraries.
- Configure a module.
- Prepare dependency injection `providers`.
- Call `inject` or `async` or `fakeAsync`.

They follow patterns familiar to test developers everywhere:

- Exhibit standard, Angular-agnostic testing techniques.
- Create instances directly with `new`.
- Substitute test doubles (stubs, spys, and mocks) for the real dependencies.

Write both kinds of tests
Good developers write both kinds of tests for the same application part, often in the same spec file. Write simple _isolated_ unit tests to validate the part in isolation. Write _Angular_ tests to validate the part as it interacts with Angular, updates the DOM, and collaborates with the rest of the application.

{@a isolated-service-tests}

## Services

Services are good candidates for isolated unit testing. Here are some synchronous and asynchronous unit tests of the `FancyService` written without assistance from Angular testing utilities.

A rough line count suggests that these isolated unit tests are about 25% smaller than equivalent Angular tests. That's telling but not decisive. The benefit comes from reduced setup and code complexity.

Compare these equivalent tests of `FancyService.getTimeoutValue`.

They have about the same line-count, but the Angular-dependent version has more moving parts including a couple of utility functions (`async` and `inject`). Both approaches work and it's not much of an issue if you're using the Angular testing utilities nearby for other reasons. On the other hand, why burden simple service tests with added complexity?

Pick the approach that suits you.

{@a services-with-dependencies}

## Services with dependencies

Services often depend on other services that Angular injects into the constructor. You can test these services *without* the `TestBed`. In many cases, it's easier to create and *inject* dependencies by hand.

The `DependentService` is a simple example:

It delegates its only method, `getValue`, to the injected `FancyService`.

Here are several ways to test it.

The first test creates a `FancyService` with `new` and passes it to the `DependentService` constructor.

However, it's rarely that simple. The injected service can be difficult to create or control. You can mock the dependency, use a dummy value, or stub the pertinent service method with a substitute method that's easy to control.

These *isolated* unit testing techniques are great for exploring the inner logic of a service or its simple integration with a component class. Use the Angular testing utilities when writing tests that validate how a service interacts with components *within the Angular runtime environment*.

{@a isolated-pipe-tests}

## Pipes

Pipes are easy to test without the Angular testing utilities.

A pipe class has one method, `transform`, that manipulates the input value into a transformed output value. The `transform` implementation rarely interacts with the DOM. Most pipes have no dependence on Angular other than the `@Pipe` metadata and an interface.

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

Anything that uses a regular expression is worth testing thoroughly. Use simple Jasmine to explore the expected cases and the edge cases.

{@a write-tests}

## Write Angular tests too

These are tests of the pipe *in isolation*. They can't tell if the `TitleCasePipe` is working properly as applied in the application components.

Consider adding component tests such as this one:

{@a isolated-component-tests}

## Components

Component tests typically examine how a component class interacts with its own template or with collaborating components. The Angular testing utilities are specifically designed to facilitate such tests.

Consider this `ButtonComp` component.

The following Angular test demonstrates that clicking a button in the template leads to an update of the on-screen message.

The assertions verify that the data values flow from one HTML control (the `<button>`) to the component and from the component back to a *different* HTML control (the `<span>`). A passing test means the component and its template are wired correctly.

Isolated unit tests can more rapidly probe a component at its API boundary, exploring many more conditions with less effort.

Here are a set of unit tests that verify the component's outputs in the face of a variety of component inputs.

Isolated component tests offer a lot of test coverage with less code and almost no setup. This is even more of an advantage with complex components, which may require meticulous preparation with the Angular testing utilities.

On the other hand, isolated unit tests can't confirm that the `ButtonComp` is properly bound to its template or even data bound at all. Use Angular tests for that.

---

{@a atu-apis}

# Angular testing utility APIs

This section takes inventory of the most useful Angular testing features and summarizes what they do.

The Angular testing utilities include the `TestBed`, the `ComponentFixture`, and a handful of functions that control the test environment. The *TestBed* and *ComponentFixture* classes are covered separately.

Here's a summary of the stand-alone functions, in order of likely utility:

| Function | Description |
| --- | --- |
| `async` | Runs the body of a test (`it`) or setup (`beforeEach`) function within a special _async test zone_. See [discussion above](guide/testing#async). |
| `fakeAsync` | Runs the body of a test (`it`) within a special _fakeAsync test zone_, enabling a linear control flow coding style. See [discussion above](guide/testing#fake-async). |
| `tick` | Simulates the passage of time and the completion of pending |

| | |
|---|---|
| | asynchronous activities by flushing both _timer_ and _micro-task_ queues within the _fakeAsync test zone_.<br>The curious, dedicated reader might enjoy this lengthy blog post, ["_Tasks, microtasks, queues and schedules_"](https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/).<br>Accepts an optional argument that moves the virtual clock forward by the specified number of milliseconds, clearing asynchronous activities scheduled within that timeframe. See [discussion above](guide/testing#tick). |
| `inject` | Injects one or more services from the current `TestBed` injector into a test function. See [above](guide/testing#inject). |
| `discardPeriodicTasks` | When a `fakeAsync` test ends with pending timer event _tasks_ (queued `setTimeOut` and `setInterval` callbacks), the test fails with a clear error message. In general, a test should end with no queued tasks. When pending timer tasks are expected, call `discardPeriodicTasks` to flush the _task_ queue and avoid the error. |
| `flushMicrotasks` | When a `fakeAsync` test ends with pending _micro-tasks_ such as unresolved promises, the test fails with a clear error message. In general, a test should wait for micro-tasks to finish. When pending microtasks are expected, call `flushMicrotasks` to flush the _micro-task_ queue and avoid the error. |
| `ComponentFixtureAutoDetect` | A provider token for a service that turns on [automatic change detection](guide/testing#automatic-change-detection). |
| `getTestBed` | Gets the current instance of the `TestBed`. Usually unnecessary because the static class methods of the `TestBed` class are typically sufficient. The `TestBed` instance exposes a few rarely used members that are not available as static methods. |

{@a testbed-class-summary}

## *TestBed* class summary

The `TestBed` class is one of the principal Angular testing utilities. Its API is quite large and can be

overwhelming until you've explored it, a little at a time. Read the early part of this guide first to get the basics before trying to absorb the full API.

The module definition passed to `configureTestingModule` is a subset of the `@NgModule` metadata properties.

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

{@a metadata-override-object}

Each override method takes a `MetadataOverride<T>` where `T` is the kind of metadata appropriate to the method, that is, the parameter of an `@NgModule`, `@Component`, `@Directive`, or `@Pipe`.

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

{@a testbed-methods} {@a testbed-api-summary}

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

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

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

Here are the most important static methods, in order of likely utility.

| Methods | Description |
| --- | --- |
| `configureTestingModule` | The testing shims (`karma-test-shim`, `browser-test-shim`) establish the [initial test environment](guide/testing) and a default testing module. The default testing module is configured with basic declaratives and some Angular service substitutes that every tester needs. Call `configureTestingModule` to refine the testing module configuration for a particular set of tests by adding and removing imports, declarations (of components, directives, and pipes), and providers. |
| `compileComponents` | Compile the testing module asynchronously after you've finished configuring it. You **must** call this method if _any_ of the testing module components have a `templateUrl` or `styleUrls` because fetching component template and style files is necessarily |

| | |
|---|---|
| | asynchronous. See [above](guide/testing#compile-components). After calling `compileComponents`, the `TestBed` configuration is frozen for the duration of the current spec. |
| `createComponent` | Create an instance of a component of type `T` based on the current `TestBed` configuration. After calling `compileComponent`, the `TestBed` configuration is frozen for the duration of the current spec. |
| `overrideModule` | Replace metadata for the given `NgModule`. Recall that modules can import other modules. The `overrideModule` method can reach deeply into the current testing module to modify one of these inner modules. |
| `overrideComponent` | Replace metadata for the given component class, which could be nested deeply within an inner module. |
| `overrideDirective` | Replace metadata for the given directive class, which could be nested deeply within an inner module. |
| `overridePipe` | Replace metadata for the given pipe class, which could be nested deeply within an inner module. |
| {@a testbed-get} `get` | Retrieve a service from the current `TestBed` injector. The `inject` function is often adequate for this purpose. But `inject` throws an error if it can't provide the service. What if the service is optional? The `TestBed.get` method takes an optional second parameter, the object to return if Angular can't find the provider (`null` in this example): After calling `get`, the `TestBed` configuration is frozen for the duration of the current spec. |
| {@a testbed-initTestEnvironment} `initTestEnvironment` | Initialize the testing environment for the entire test run. The testing shims (`karma-test-shim`, `browser-test-shim`) call it for you so there is rarely a reason for you to call it yourself. You may call this method _exactly once_. If you must change this default in the middle of your test run, call `resetTestEnvironment` first. Specify the Angular compiler factory, a `PlatformRef`, and a default Angular testing module. Alternatives for non-browser platforms are available in the general form `@angular/platform-/testing/`. |
| `resetTestEnvironment` | Reset the initial test environment, including the default testing module. |

A few of the `TestBed` instance methods are not covered by static `TestBed` *class* methods. These are rarely needed.

{@a component-fixture-api-summary}

## The *ComponentFixture*

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

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

{@a component-fixture-properties}

## *ComponentFixture* properties

Here are the most important properties for testers, in order of likely utility.

| Properties | Description |
|---|---|
| `componentInstance` | The instance of the component class created by `TestBed.createComponent`. |
| `debugElement` | The `DebugElement` associated with the root element of the component. The `debugElement` provides insight into the component and its DOM element during test and debugging. It's a critical property for testers. The most interesting members are covered [below](guide/testing#debug-element-details). |
| `nativeElement` | The native DOM element at the root of the component. |
| `changeDetectorRef` | The `ChangeDetectorRef` for the component. The `ChangeDetectorRef` is most valuable when testing a component that has the `ChangeDetectionStrategy.OnPush` method or the component's change detection is under your programmatic control. |

{@a component-fixture-methods}

## *ComponentFixture* methods

The *fixture* methods cause Angular to perform certain tasks on the component tree. Call these method to

trigger Angular behavior in response to simulated user action.

Here are the most useful methods for testers.

| Methods | Description |
| --- | --- |
| `detectChanges` | Trigger a change detection cycle for the component. Call it to initialize the component (it calls `ngOnInit`) and after your test code, change the component's data bound property values. Angular can't see that you've changed `personComponent.name` and won't update the `name` binding until you call `detectChanges`. Runs `checkNoChanges`afterwards to confirm that there are no circular updates unless called as `detectChanges(false)`; |
| `autoDetectChanges` | Set this to `true` when you want the fixture to detect changes automatically. When autodetect is `true`, the test fixture calls `detectChanges` immediately after creating the component. Then it listens for pertinent zone events and calls `detectChanges` accordingly. When your test code modifies component property values directly, you probably still have to call `fixture.detectChanges` to trigger data binding updates. The default is `false`. Testers who prefer fine control over test behavior tend to keep it `false`. |
| `checkNoChanges` | Do a change detection run to make sure there are no pending changes. Throws an exceptions if there are. |
| `isStable` | If the fixture is currently _stable_, returns `true`. If there are async tasks that have not completed, returns `false`. |
| `whenStable` | Returns a promise that resolves when the fixture is stable. To resume testing after completion of asynchronous activity or asynchronous change detection, hook that promise. See [above](guide/testing#when-stable). |
| `destroy` | Trigger component destruction. |

{@a debug-element-details}

## *DebugElement*

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

From the test root component's `DebugElement` returned by `fixture.debugElement`, you can walk

(and query) the fixture's entire element and component subtrees.

Here are the most useful `DebugElement` members for testers, in approximate order of utility:

| Member | Description |
|---|---|
| `nativeElement` | The corresponding DOM element in the browser (null for WebWorkers). |
| `query` | Calling `query(predicate: Predicate)` returns the first `DebugElement` that matches the [predicate](guide/testing#query-predicate) at any depth in the subtree. |
| `queryAll` | Calling `queryAll(predicate: Predicate)` returns all `DebugElements` that matches the [predicate](guide/testing#query-predicate) at any depth in subtree. |
| `injector` | The host dependency injector. For example, the root element's component instance injector. |
| `componentInstance` | The element's own component instance, if it has one. |
| `context` | An object that provides parent context for this element. Often an ancestor component instance that governs this element. When an element is repeated within `*ngFor`, the context is an `NgForRow` whose `$implicit` property is the value of the row instance value. For example, the `hero` in `*ngFor="let hero of heroes"`. |
| `children` | The immediate `DebugElement` children. Walk the tree by descending through `children`.<br>`DebugElement` also has `childNodes`, a list of `DebugNode` objects.<br>`DebugElement` derives from `DebugNode` objects and there are often more nodes than elements. Testers can usually ignore plain nodes. |
| `parent` | The `DebugElement` parent. Null if this is the root element. |
| `name` | The element tag name, if it is an element. |
| `triggerEventHandler` | Triggers the event by its name if there is a corresponding listener in the element's `listeners` collection. The second parameter is the _event object_ expected by the handler. See [above](guide/testing#trigger-event-handler). If the event lacks a listener or there's some other problem, consider calling `nativeElement.dispatchEvent(eventObject)`. |
| `listeners` | The callbacks attached to the component's `@Output` properties and/or |

| | |
|---|---|
| | the element's event properties. |
| `providerTokens` | This component's injector lookup tokens. Includes the component itself plus the tokens that the component lists in its `providers` metadata. |
| `source` | Where to find this element in the source component template. |
| `references` | Dictionary of objects associated with template local variables (e.g. `#foo`), keyed by the local variable name. |

{@a query-predicate}

The `DebugElement.query(predicate)` and `DebugElement.queryAll(predicate)` methods take a predicate that filters the source element's subtree for matching `DebugElement` .

The predicate is any method that takes a `DebugElement` and returns a *truthy* value. The following example finds all `DebugElements` with a reference to a template local variable named "content":

The Angular `By` class has three static methods for common predicates:

- `By.all` - return all elements.
- `By.css(selector)` - return elements with matching CSS selectors.
- `By.directive(directive)` - return elements that Angular matched to an instance of the directive class.

{@a setup-files}

# Test environment setup files

Unit testing requires some configuration and bootstrapping that is captured in *setup files*. The setup files for this guide are provided for you when you follow the Setup instructions. The CLI delivers similar files with the same purpose.

Here's a brief description of this guide's setup files:

The deep details of these files and how to reconfigure them for your needs is a topic beyond the scope of this guide .

| File | Description |
| --- | --- |
| `karma.conf.js` | The karma configuration file that specifies which plug-ins to use, which application and test files to load, which browser(s) to use, and how to report test results. It loads three other setup files: * `systemjs.config.js` * `systemjs.config.extras.js` * `karma-test-shim.js` |
| `karma-test-shim.js` | This shim prepares karma specifically for the Angular test environment and launches karma itself. It loads the `systemjs.config.js` file as part of that process. |
| `systemjs.config.js` | [SystemJS] (https://github.com/systemjs/systemjs/blob/master/README.md) loads the application and test files. This script tells SystemJS where to find those files and how to load them. It's the same version of `systemjs.config.js` you installed during [setup] (guide/testing#setup). |
| `systemjs.config.extras.js` | An optional file that supplements the SystemJS configuration in `systemjs.config.js` with configuration for the specific needs of the application itself. A stock `systemjs.config.js` can't anticipate those needs. You fill the gaps here. The sample version for this guide adds the **model barrel** to the SystemJs `packages` configuration. |

## npm packages

The sample tests are written to run in Jasmine and karma. The two "fast path" setups added the appropriate Jasmine and karma npm packages to the `devDependencies` section of the `package.json`. They're installed when you run `npm install`.

## FAQ: Frequently Asked Questions

## Why put specs next to the things they test?

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

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

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

---

## When would I put specs in a test folder?

Application integration specs can test the interactions of multiple parts spread across folders and modules. They don't really belong to any part in particular, so they don't have a natural home next to any one file.

It's often better to create an appropriate folder for them in the `tests` directory.

Of course specs that test the test helpers belong in the `test` folder, next to their corresponding helper files.