

Testing Utility APIs



Contents

- TestBed class summary
- The ComponentFixture
- ComponentFixture properties
- ComponentFixture methods

This page describes the most useful Angular testing features.

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 |
|------------------------|--|
| <code>async</code> | Runs the body of a test (<code>it</code>) or setup (<code>beforeEach</code>) function within a special <i>async test zone</i> . See discussion above . |
| <code>fakeAsync</code> | Runs the body of a test (<code>it</code>) within a special <i>fakeAsync test zone</i> , enabling a linear control flow coding style. See discussion above . |
| <code>tick</code> | <div>Simulates the passage of time and the completion of pending asynchronous activities by flushing both <i>timer</i> and <i>micro-task</i> queues within the <i>fakeAsync test zone</i>.</div> <div><div></div><div>The curious, dedicated reader might enjoy this lengthy blog post, "Tasks, microtasks, queues and schedules" ↗.</div></div> <div>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.</div> |
| <code>inject</code> | Injects one or more services from the current <code>TestBed</code> injector into a test function. It cannot inject a service provided by the component itself. See |

discussion of the [debugElement.injector](#).

| | |
|---|---|
| <code>discardPeriodicTasks</code> | <p>When a <code>fakeAsync()</code> test ends with pending timer event <i>tasks</i> (queued <code>setTimeout</code> and <code>setInterval</code> callbacks), the test fails with a clear error message.</p> <p>In general, a test should end with no queued tasks. When pending timer tasks are expected, call <code>discardPeriodicTasks</code> to flush the <i>task</i> queue and avoid the error.</p> |
| <code>flushMicrotasks</code> | <p>When a <code>fakeAsync()</code> test ends with pending <i>micro-tasks</i> such as unresolved promises, the test fails with a clear error message.</p> <p>In general, a test should wait for micro-tasks to finish. When pending microtasks are expected, call <code>flushMicrotasks</code> to flush the <i>micro-task</i> queue and avoid the error.</p> |
| <code>ComponentFixtureAutoDetect</code> | A provider token for a service that turns on automatic change detection . |
| <code>getTestBed</code> | <p>Gets the current instance of the <code>TestBed</code>. Usually unnecessary because the static class methods of the <code>TestBed</code> class are typically sufficient.</p> <p>The <code>TestBed</code> instance exposes a few rarely used members that are not available as static methods.</p> |

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[]>;  
};
```



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<T> = {  
  add?: Partial<T>;
```



```
remove?: Partial<T>;  
set?: Partial<T>;  
};
```

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 |
|-------------------------------------|--|
| <code>configureTestingModule</code> | <p>The testing shims (<code>karma-test-shim</code>, <code>browser-test-shim</code>) establish the initial test environment and a default testing module. The default testing module is configured with basic declaratives and some Angular service substitutes that every tester needs.</p> <p>Call <code>configureTestingModule</code> 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.</p> |
| <code>compileComponents</code> | <p>Compile the testing module asynchronously after you've finished configuring it. You must call this method if <i>any</i> of the testing module components have a <code>templateUrl</code> or <code>styleUrls</code> because fetching component template and style files is necessarily asynchronous. See above.</p> <p>After calling <code>compileComponents</code>, the <code>TestBed</code> configuration is frozen for the duration of the current spec.</p> |
| <code>createComponent</code> | <p>Create an instance of a component of type <code>T</code> based on the current <code>TestBed</code> configuration. After calling <code>compileComponent</code>, the <code>TestBed</code> configuration is frozen for the duration of the current spec.</p> |
| <code>overrideModule</code> | <p>Replace metadata for the given <code>NgModule</code>. Recall that modules can import other modules. The <code>overrideModule</code> method can reach deeply into the current testing module to modify one of these inner modules.</p> |
| <code>overrideComponent</code> | <p>Replace metadata for the given component class, which could be nested deeply within an inner module.</p> |
| <code>overrideDirective</code> | <p>Replace metadata for the given directive class, which could be nested deeply within an inner module.</p> |

| | |
|----------------------|--|
| overridePipe | Replace metadata for the given pipe class, which could be nested deeply within an inner module. |
| inject | <p>Retrieve a service from the current <code>TestBed</code> injector.</p> <p>The <code>inject</code> function is often adequate for this purpose. But <code>inject</code> throws an error if it can't provide the service.</p> <p>What if the service is optional?</p> <p>The <code>TestBed.inject()</code> method takes an optional second parameter, the object to return if Angular can't find the provider (<code>null</code> in this example):</p> <pre>app/demo/demo.testbed.spec.ts service = TestBed.inject(NotProvided, null); // service is null</pre> <p>After calling <code>TestBed.inject</code>, the <code>TestBed</code> configuration is frozen for the duration of the current spec.</p> |
| initTestEnvironment | <p>Initialize the testing environment for the entire test run.</p> <p>The testing shims (<code>karma-test-shim</code>, <code>browser-test-shim</code>) call it for you so there is rarely a reason for you to call it yourself.</p> <p>You may call this method <i>exactly once</i>. If you must change this default in the middle of your test run, call <code>resetTestEnvironment</code> first.</p> <p>Specify the Angular compiler factory, a <code>PlatformRef</code>, and a default Angular testing module. Alternatives for non-browser platforms are available in the general form <code>@angular/platform-<platform_name>/testing/<platform_name></code>.</p> |
| 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.

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.

ComponentFixture properties

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

| Properties | Description |
|--------------------------------|---|
| <code>componentInstance</code> | The instance of the component class created by <code>TestBed.createComponent</code> . |
| <code>debugElement</code> | <p>The <code>DebugElement</code> associated with the root element of the component.</p> <p>The <code>debugElement</code> 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.</p> |
| <code>nativeElement</code> | The native DOM element at the root of the component. |
| <code>changeDetectorRef</code> | <p>The <code>ChangeDetectorRef</code> for the component.</p> <p>The <code>ChangeDetectorRef</code> is most valuable when testing a component that has the <code>ChangeDetectionStrategy.OnPush</code> method or the component's change detection is under your programmatic control.</p> |

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 |
|--------------------------------|---|
| <code>detectChanges</code> | <p>Trigger a change detection cycle for the component.</p> <p>Call it to initialize the component (it calls <code>ngOnInit</code>) and after your test code, change the component's data bound property values. Angular can't see that you've changed <code>personComponent.name</code> and won't update the name binding until you call <code>detectChanges</code>.</p> <p>Runs <code>checkNoChanges</code> afterwards to confirm that there are no circular updates unless called as <code>detectChanges(false)</code>;</p> |
| <code>autoDetectChanges</code> | <p>Set this to <code>true</code> when you want the fixture to detect changes automatically.</p> <p>When <code>autodetect</code> is <code>true</code>, the test fixture calls <code>detectChanges</code> immediately after creating the component. Then it listens for pertinent zone events and calls <code>detectChanges</code> accordingly. When your test code modifies component property</p> |

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`.

| | |
|-----------------------------|---|
| <code>checkNoChanges</code> | Do a change detection run to make sure there are no pending changes. Throws an exceptions if there are. |
| <code>isStable</code> | If the fixture is currently <i>stable</i> , returns <code>true</code> . If there are async tasks that have not completed, returns <code>false</code> . |
| <code>whenStable</code> | 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 . |
| <code>destroy</code> | Trigger component destruction. |

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 |
|--------------------------------|--|
| <code>nativeElement</code> | The corresponding DOM element in the browser (null for WebWorkers). |
| <code>query</code> | Calling <code>query(predicate: Predicate<DebugElement>)</code> returns the first <code>DebugElement</code> that matches the predicate at any depth in the subtree. |
| <code>queryAll</code> | Calling <code>queryAll(predicate: Predicate<DebugElement>)</code> returns all <code>DebugElements</code> that matches the predicate at any depth in subtree. |
| <code>injector</code> | The host dependency injector. For example, the root element's component instance injector. |
| <code>componentInstance</code> | The element's own component instance, if it has one. |
| | |

| | |
|---------------------|--|
| context | <p>An object that provides parent context for this element. Often an ancestor component instance that governs this element.</p> <p>When an element is repeated within <code>*ngFor</code>, the context is an <code>NgForRow</code> whose <code>\$implicit</code> property is the value of the row instance value. For example, the hero in <code>*ngFor="let hero of heroes"</code>.</p> |
| children | <p>The immediate <code>DebugElement</code> children. Walk the tree by descending through children.</p> <div><p><code>DebugElement</code> also has <code>childNodes</code>, a list of <code>DebugNode</code> objects. <code>DebugElement</code> derives from <code>DebugNode</code> objects and there are often more nodes than elements. Testers can usually ignore plain nodes.</p></div> |
| parent | The <code>DebugElement</code> parent. Null if this is the root element. |
| name | The element tag name, if it is an element. |
| triggerEventHandler | <p>Triggers the event by its name if there is a corresponding listener in the element's <code>listeners</code> collection. The second parameter is the <i>event object</i> expected by the handler. See above.</p> <p>If the event lacks a listener or there's some other problem, consider calling <code>nativeElement.dispatchEvent(eventObject)</code>.</p> |
| listeners | The callbacks attached to the component's <code>@Output</code> 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 <code>providers</code> metadata. |
| source | Where to find this element in the source component template. |
| references | Dictionary of objects associated with template local variables (e.g. <code>#foo</code>), keyed by the local variable name. |

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":

```
// Filter for DebugElements with a #content reference
const contentRefs = el.queryAll( de => de.references.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.

```
// Can find DebugElement either by css selector or by directive
const h2 = fixture.debugElement.query(By.css('h2'));
const directive = fixture.debugElement.query(By.directive(HighlightDirective));
```



RESOURCES

[About](#)
[Resource Listing](#)
[Press Kit](#)
[Blog](#)
[Usage Analytics](#)

HELP

[Stack Overflow](#)
[Gitter](#)
[Report Issues](#)
[Code of Conduct](#)

COMMUNITY

[Events](#)
[Meetups](#)
[Twitter](#)
[GitHub](#)
[Contribute](#)

LANGUAGES

[简体中文版](#)
[正體中文版](#)
[日本語版](#)
[한국어](#)

Super-powered by Google ©2010-2020. Code licensed under an MIT-style License. Documentation licensed under CC BY 4.0.

Version 10.0.10-local+sha.84d1ba792b.