Component testing scenarios

Contents ›

This guide explores common component testing use cases.

> For the sample app that the testing guides describe, see the sample app.
>
> For the tests features in the testing guides, see tests.

Component binding

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

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

app/banner/banner.component.ts

```
@Component({
  selector: 'app-banner',
  template: '<h1>{{title}}</h1>',
  styles: ['h1 { color: green; font-size: 350%}']
})
export class BannerComponent {
  title = 'Test Tour of Heroes';
}
```

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

Query for the *<h1>*

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

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

app/banner/banner.component.spec.ts (setup)

```
let component: BannerComponent;
let fixture: ComponentFixture<BannerComponent>;
let h1: HTMLElement;

beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });
  fixture = TestBed.createComponent(BannerComponent);
  component = fixture.componentInstance; // BannerComponent test instance
  h1 = fixture.nativeElement.querySelector('h1');
});
```

createComponent() does not bind data

For your first test you'd like to see that the screen displays the default `title`. Your instinct is to write a test that immediately inspects the `<h1>` like this:

```
it('should display original title', () => {
  expect(h1.textContent).toContain(component.title);
});
```

That test fails with the message:

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

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

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

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

```
it('no title in the DOM after createComponent()', () => {
  expect(h1.textContent).toEqual('');
});
```

detectChanges()

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

```
it('should display original title after detectChanges()', () => {
  fixture.detectChanges();
  expect(h1.textContent).toContain(component.title);
});
```

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

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

```
it('should display a different test title', () => {
  component.title = 'Test Title';
  fixture.detectChanges();
  expect(h1.textContent).toContain('Test Title');
});
```

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:

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

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

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

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

```
TestBed.configureTestingModule({
  declarations: [ BannerComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
});
```

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

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

```
  it('should display original title', () => {
    // Hooray! No `fixture.detectChanges()` needed
    expect(h1.textContent).toContain(comp.title);
  });

  it('should still see original title after comp.title change', () => {
    const oldTitle = comp.title;
    comp.title = 'Test Title';
    // Displayed title is old because Angular didn't hear the change :(
    expect(h1.textContent).toContain(oldTitle);
  });

  it('should display updated title after detectChanges', () => {
    comp.title = 'Test Title';
    fixture.detectChanges(); // detect changes explicitly
    expect(h1.textContent).toContain(comp.title);
  });
```

The first test shows the benefit of automatic change detection.

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

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

Change an input value with *dispatchEvent()*

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

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

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

The following example demonstrates the proper sequence.

app/hero/hero-detail.component.spec.ts (pipe test)

```
  it('should convert hero name to Title Case', () => {
    // get the name's input and display elements from the DOM
    const hostElement = fixture.nativeElement;
    const nameInput: HTMLInputElement = hostElement.querySelector('input');
    const nameDisplay: HTMLElement = hostElement.querySelector('span');

    // simulate user entering a new name into the input box
    nameInput.value = 'quick BROWN  fOx';

    // dispatch a DOM event so that Angular learns of input value change.
    // use newEvent utility function (not provided by Angular) for better browser
  compatibility
    nameInput.dispatchEvent(newEvent('input'));

    // Tell Angular to update the display binding through the title pipe
    fixture.detectChanges();

    expect(nameDisplay.textContent).toBe('Quick Brown  Fox');
  });
```

Component with external files

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

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

app/banner/banner-external.component.ts (metadata)

```
@Component({
  selector: 'app-banner',
  templateUrl: './banner-external.component.html',
  styleUrls:  ['./banner-external.component.css']
})
```

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

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

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

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

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

To correct the problem, call `compileComponents()` as explained below.

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`:

app/welcome/welcome.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UserService } from '../model/user.service';

@Component({
  selector: 'app-welcome',
  template: '<h3 class="welcome"><i>{{welcome}}</i></h3>'
})
export class WelcomeComponent implements OnInit {
  welcome: string;
  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.welcome = this.userService.isLoggedIn ?
      'Welcome, ' + this.userService.user.name : 'Please log in.';
  }
}
```

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

app/welcome/welcome.component.spec.ts

```
TestBed.configureTestingModule({
   declarations: [ WelcomeComponent ],
// providers: [ UserService ],  // NO! Don't provide the real service!
                                // Provide a test-double instead
   providers: [ { provide: UserService, useValue: userServiceStub } ],
});
```

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

Provide service test doubles

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

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

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

app/welcome/welcome.component.spec.ts

```
let userServiceStub: Partial<UserService>;

  userServiceStub = {
    isLoggedIn: true,
    user: { name: 'Test User' },
  };
```

Get injected services

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

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

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

WelcomeComponent's injector

```
// UserService actually injected into the component
userService = fixture.debugElement.injector.get(UserService);
```

TestBed.inject()

You *may* also be able to get the service from the root injector via `TestBed.inject()`. This is easier to remember and less verbose. But it only works when Angular injects the component with the service instance in the test's root injector.

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

TestBed injector

```
// UserService from the root injector
userService = TestBed.inject(UserService);
```

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

Final setup and tests

Here's the complete `beforeEach()`, using `TestBed.inject()`:

app/welcome/welcome.component.spec.ts

```
let userServiceStub: Partial<UserService>;

beforeEach(() => {
  // stub UserService for test purposes
  userServiceStub = {
    isLoggedIn: true,
    user: { name: 'Test User' },
  };

  TestBed.configureTestingModule({
     declarations: [ WelcomeComponent ],
     providers: [ { provide: UserService, useValue: userServiceStub } ],
  });

  fixture = TestBed.createComponent(WelcomeComponent);
  comp    = fixture.componentInstance;

  // UserService from the root injector
  userService = TestBed.inject(UserService);

  //  get the "welcome" element by CSS selector (e.g., by class name)
  el = fixture.nativeElement.querySelector('.welcome');
});
```

And here are some tests:

app/welcome/welcome.component.spec.ts

```
it('should welcome the user', () => {
  fixture.detectChanges();
  const content = el.textContent;
  expect(content).toContain('Welcome', '"Welcome ..."');
  expect(content).toContain('Test User', 'expected name');
});
```

```
  it('should welcome "Bubba"', () => {
    userService.user.name = 'Bubba'; // welcome message hasn't been shown yet
    fixture.detectChanges();
    expect(el.textContent).toContain('Bubba');
  });

  it('should request login if not logged in', () => {
    userService.isLoggedIn = false; // welcome message hasn't been shown yet
    fixture.detectChanges();
    const content = el.textContent;
    expect(content).not.toContain('Welcome', 'not welcomed');
    expect(content).toMatch(/log in/i, '"log in"');
  });
```

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

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

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

Component with async service

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

app/twain/twain.component.ts (template)

```
template: `
  <p class="twain"><i>{{quote | async}}</i></p>
  <button (click)="getQuote()">Next quote</button>
  <p class="error" *ngIf="errorMessage">{{ errorMessage }}</p>`,
```

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

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

app/twain/twain.component.ts (getQuote)

```
getQuote() {
    this.errorMessage = '';
    this.quote = this.twainService.getQuote().pipe(
      startWith('...'),
      catchError( (err: any) => {
        // Wait a turn because errorMessage already set once this turn
        setTimeout(() => this.errorMessage = err.message || err.toString());
        return of('...'); // reset message to placeholder
      })
    );
```

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

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

These are all features you'll want to test.

Testing with a spy

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

app/twain/twain.component.spec.ts (setup)

```
beforeEach(() => {
  testQuote = 'Test Quote';

  // Create a fake TwainService object with a `getQuote()` spy
  const twainService = jasmine.createSpyObj('TwainService', ['getQuote']);
  // Make the spy return a synchronous Observable with the test data
  getQuoteSpy = twainService.getQuote.and.returnValue( of(testQuote) );

  TestBed.configureTestingModule({
    declarations: [ TwainComponent ],
    providers:    [
      { provide: TwainService, useValue: twainService }
    ]
  });

  fixture = TestBed.createComponent(TwainComponent);
  component = fixture.componentInstance;
  quoteEl = fixture.nativeElement.querySelector('.twain');
});
```

Focus on the spy.

```
// Create a fake TwainService object with a `getQuote()` spy
const twainService = jasmine.createSpyObj('TwainService', ['getQuote']);
// Make the spy return a synchronous Observable with the test data
getQuoteSpy = twainService.getQuote.and.returnValue( of(testQuote) );
```

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

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

Synchronous tests

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

```
it('should show quote after component initialized', () => {
  fixture.detectChanges(); // onInit()

  // sync spy result shows testQuote immediately after init
  expect(quoteEl.textContent).toBe(testQuote);
  expect(getQuoteSpy.calls.any()).toBe(true, 'getQuote called');
});
```

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

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

Async test with *fakeAsync()*

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

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

```
it('should display error when TwainService fails', fakeAsync(() => {
  // tell spy to return an error observable
  getQuoteSpy.and.returnValue(
    throwError('TwainService test failure'));

  fixture.detectChanges(); // onInit()
  // sync spy errors immediately after init
```

```
    tick(); // flush the component's setTimeout()

    fixture.detectChanges(); // update errorMessage within setTimeout()

    expect(errorMessage()).toMatch(/test failure/, 'should display error');
    expect(quoteEl.textContent).toBe('...', 'should show placeholder');
  }));
```

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

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

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

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

The *tick()* function

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

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

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

```
it('should run timeout callback with delay after call tick with millis', fakeAsync(()
=> {
     let called = false;
     setTimeout(() => { called = true; }, 100);
     tick(100);
     expect(called).toBe(true);
   }));
```

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

tickOptions

```
it('should run new macro task callback with delay after call tick with millis',
   fakeAsync(() => {
     function nestedTimer(cb: () => any): void { setTimeout(() => setTimeout(() =>
cb())); }
     const callback = jasmine.createSpy('callback');
     nestedTimer(callback);
     expect(callback).not.toHaveBeenCalled();
     tick(0);
     // the nested timeout will also be triggered
     expect(callback).toHaveBeenCalled();
   }));
```

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

```
it('should not run new macro task callback with delay after call tick with millis',
   fakeAsync(() => {
     function nestedTimer(cb: () => any): void { setTimeout(() => setTimeout(() =>
cb())); }
     const callback = jasmine.createSpy('callback');
     nestedTimer(callback);
     expect(callback).not.toHaveBeenCalled();
     tick(0, {processNewMacroTasksSynchronously: false});
     // the nested timeout will not be triggered
     expect(callback).not.toHaveBeenCalled();
     tick(0);
     expect(callback).toHaveBeenCalled();
   }));
```

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

Comparing dates inside fakeAsync()

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

```
it('should get Date diff correctly in fakeAsync', fakeAsync(() => {
     const start = Date.now();
     tick(100);
     const end = Date.now();
     expect(end - start).toBe(100);
   }));
```

jasmine.clock with fakeAsync()

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

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

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

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

```
describe('use jasmine.clock()', () => {
  // need to config __zone_symbol__fakeAsyncPatchLock flag
  // before loading zone.js/dist/zone-testing
  beforeEach(() => { jasmine.clock().install(); });
  afterEach(() => { jasmine.clock().uninstall(); });
  it('should auto enter fakeAsync', () => {
    // is in fakeAsync now, don't need to call fakeAsync(testFn)
    let called = false;
    setTimeout(() => { called = true; }, 100);
    jasmine.clock().tick(100);
    expect(called).toBe(true);
  });
});
```

Using the RxJS scheduler inside fakeAsync()

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

```
it('should get Date diff correctly in fakeAsync with rxjs scheduler', fakeAsync(()
=> {
     // need to add `import 'zone.js/dist/zone-patch-rxjs-fake-async'
     // to patch rxjs scheduler
     let result = null;
     of ('hello').pipe(delay(1000)).subscribe(v => { result = v; });
     expect(result).toBeNull();
     tick(1000);
     expect(result).toBe('hello');

     const start = new Date().getTime();
     let dateDiff = 0;
     interval(1000).pipe(take(2)).subscribe(() => dateDiff = (new Date().getTime() -
start));

     tick(1000);
     expect(dateDiff).toBe(1000);
     tick(1000);
     expect(dateDiff).toBe(2000);
   }));
```

Support more macroTasks

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

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

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

src/app/shared/canvas.component.spec.ts (failing) src/app/shared/canvas.component.ts

```
import { TestBed, async, tick, fakeAsync } from '@angular/core/testing';
import { CanvasComponent } from './canvas.component';

describe('CanvasComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        CanvasComponent
      ],
    }).compileComponents();
  }));

  it('should be able to generate blob data from canvas', fakeAsync(() => {
    const fixture = TestBed.createComponent(CanvasComponent);
    const canvasComp = fixture.componentInstance;

    fixture.detectChanges();
    expect(canvasComp.blobSize).toBe(0);

    tick();
    expect(canvasComp.blobSize).toBeGreaterThan(0);
  }));
});
```

```
});
```

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

src/app/shared/canvas.component.spec.ts (excerpt)

```
beforeEach(() => {
  (window as any).__zone_symbol__FakeAsyncTestMacroTask = [
    {
      source: 'HTMLCanvasElement.toBlob',
      callbackArgs: [{ size: 200 }],
    },
  ];
});
```

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

src/polyfills.ts or src/app/shared/canvas.component.ts

```
// Import patch to make async `HTMLCanvasElement` methods (such as `.toBlob()`)
Zone.js-aware.
// Either import in `polyfills.ts` (if used in more than one places in the app) or in
the component
// file using `HTMLCanvasElement` (if it is only used in a single file).
import 'zone.js/dist/zone-patch-canvas';
```

Async observables

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

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

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

```
// Simulate delayed observable values with the `asyncData()` helper
getQuoteSpy.and.returnValue(asyncData(testQuote));
```

Async observable helpers

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

testing/async-observable-helpers.ts

```
/**
 * Create async observable that emits-once and completes
 * after a JS engine turn
 */
export function asyncData<T>(data: T) {
  return defer(() => Promise.resolve(data));
}
```

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

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

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

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

```
/**
 * Create async observable error that errors
 * after a JS engine turn
 */
export function asyncError<T>(errorObject: any) {
  return defer(() => Promise.reject(errorObject));
}
```

More async tests

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

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

```
it('should show quote after getQuote (fakeAsync)', fakeAsync(() => {
  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  tick(); // flush the observable to get the quote
  fixture.detectChanges(); // update view

  expect(quoteEl.textContent).toBe(testQuote, 'should show quote');
  expect(errorMessage()).toBeNull('should not show error');
}));
```

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

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

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

Async test with *async()*

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

The `fakeAsync()` utility function has a few limitations. In particular, it won't work if the test body makes an `XMLHttpRequest` (XHR) call. XHR calls within a test are rare so you can generally stick with `fakeAsync()`. But if you ever do need to call `XMLHttpRequest`, you'll want to know about `async()`.

> The `TestBed.compileComponents()` method (see below) calls `XHR` to read external template and css files during "just-in-time" compilation. Write tests that call `compileComponents()` with the `async()` utility.

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

```
it('should show quote after getQuote (async)', async(() => {
  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  fixture.whenStable().then(() => { // wait for async getQuote
    fixture.detectChanges();        // update view with quote
    expect(quoteEl.textContent).toBe(testQuote);
    expect(errorMessage()).toBeNull('should not show error');
  });
}));
```

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

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

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

whenStable

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

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

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

Jasmine *done()*

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

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

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

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

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

```
it('should show last quote (quote done)', (done: DoneFn) => {
  fixture.detectChanges();

  component.quote.pipe( last() ).subscribe(() => {
    fixture.detectChanges(); // update view with quote
    expect(quoteEl.textContent).toBe(testQuote);
    expect(errorMessage()).toBeNull('should not show error');
    done();
  });
});
```

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

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

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

```
it('should show quote after getQuote (spy done)', (done: DoneFn) => {
  fixture.detectChanges();

  // the spy's most recent call returns the observable with the test quote
```

```
    getQuoteSpy.calls.mostRecent().returnValue.subscribe(() => {
      fixture.detectChanges(); // update view with quote
      expect(quoteEl.textContent).toBe(testQuote);
      expect(errorMessage()).toBeNull('should not show error');
      done();
    });
  });
```

Component marble tests

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

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

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

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

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

app/twain/twain.component.marbles.spec.ts (import marbles)

```
import { cold, getTestScheduler } from 'jasmine-marbles';
```

Here's the complete test for getting a quote:

```
it('should show quote after getQuote (marbles)', () => {
  // observable test quote value and complete(), after delay
  const q$ = cold('---x|', { x: testQuote });
  getQuoteSpy.and.returnValue( q$ );

  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  getTestScheduler().flush(); // flush the observables

  fixture.detectChanges(); // update view

  expect(quoteEl.textContent).toBe(testQuote, 'should show quote');
  expect(errorMessage()).toBeNull('should not show error');
});
```

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

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

```
const q$ = cold('---x|', { x: testQuote });
```

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

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

```
getTestScheduler().flush(); // flush the observables
```

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

Marble error testing

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

```
it('should display error when TwainService fails', fakeAsync(() => {
  // observable error after delay
  const q$ = cold('---#|', null, new Error('TwainService test failure'));
  getQuoteSpy.and.returnValue( q$ );

  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  getTestScheduler().flush(); // flush the observables
  tick();                     // component shows error after a setTimeout()
  fixture.detectChanges();    // update error message

  expect(errorMessage()).toMatch(/test failure/, 'should display error');
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');
}));
```

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

Look at the marble observable definition.

```
const q$ = cold('---#|', null, new Error('TwainService test failure'));
```

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

Learn about marble testing

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

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

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

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

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:

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

```
<dashboard-hero *ngFor="let hero of heroes"  class="col-1-4"
  [hero]=hero  (selected)="gotoDetail($event)" >
</dashboard-hero>
```

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

Here's the component's full definition:

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

```
@Component({
  selector: 'dashboard-hero',
  template: `
    <div (click)="click()" class="hero">
      {{hero.name | uppercase}}
    </div>`,
  styleUrls: [ './dashboard-hero.component.css' ]
})
```

```
export class DashboardHeroComponent {
  @Input() hero: Hero;
  @Output() selected = new EventEmitter<Hero>();
  click() { this.selected.emit(this.hero); }
}
```

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

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

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

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

```
constructor(
  private router: Router,
  private heroService: HeroService) {
}
```

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

> The discussion below covers testing components that require the router.

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

Test *DashboardHeroComponent* stand-alone

Here's the meat of the spec file setup.

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

```
TestBed.configureTestingModule({
  declarations: [ DashboardHeroComponent ]
})
fixture = TestBed.createComponent(DashboardHeroComponent);
comp    = fixture.componentInstance;

// find the hero's DebugElement and element
heroDe  = fixture.debugElement.query(By.css('.hero'));
heroEl = heroDe.nativeElement;
```

```
// mock the hero supplied by the parent component
expectedHero = { id: 42, name: 'Test Name' };

// simulate the parent setting the input property with that hero
comp.hero = expectedHero;

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

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

```
it('should display hero name in uppercase', () => {
  const expectedPipedName = expectedHero.name.toUpperCase();
  expect(heroEl.textContent).toContain(expectedPipedName);
});
```

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

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

Clicking

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

```
it('should raise selected event when clicked (triggerEventHandler)', () => {
  let selectedHero: Hero;
  comp.selected.subscribe((hero: Hero) => selectedHero = hero);

  heroDe.triggerEventHandler('click', null);
  expect(selectedHero).toBe(expectedHero);
});
```

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

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

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

triggerEventHandler

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

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

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

The test triggered a "click" event with a `null` event object.

```
heroDe.triggerEventHandler('click', null);
```

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

> Other handlers are less forgiving. For example, the `RouterLink` directive expects an object with a `button` property that identifies which mouse button (if any) was pressed during the click. The `RouterLink` directive throws an error if the event object is missing.

Click the element

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

```
it('should raise selected event when clicked (element.click)', () => {
  let selectedHero: Hero;
  comp.selected.subscribe((hero: Hero) => selectedHero = hero);

  heroEl.click();
  expect(selectedHero).toBe(expectedHero);
});
```

click() helper

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

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

testing/index.ts (click helper)

```
/** Button events to pass to `DebugElement.triggerEventHandler` for RouterLink event
handler */
export const ButtonClickEvents = {
   left:  { button: 0 },
   right: { button: 2 }
};

/** Simulate element click. Defaults to mouse left-button click event. */
export function click(el: DebugElement | HTMLElement, eventObj: any =
ButtonClickEvents.left): void {
  if (el instanceof HTMLElement) {
    el.click();
  } else {
    el.triggerEventHandler('click', eventObj);
  }
}
```

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

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

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

app/dashboard/dashboard-hero.component.spec.ts (test with click helper)

```
it('should raise selected event when clicked (click helper)', () => {
  let selectedHero: Hero;
  comp.selected.subscribe((hero: Hero) => selectedHero = hero);

  click(heroDe); // click helper with DebugElement
  click(heroEl); // click helper with native element

  expect(selectedHero).toBe(expectedHero);
});
```

Component inside a test host

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

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

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

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

```
@Component({
  template: `
    <dashboard-hero
      [hero]="hero" (selected)="onSelected($event)">
    </dashboard-hero>`
})
class TestHostComponent {
  hero: Hero = {id: 42, name: 'Test Name' };
  selectedHero: Hero;
  onSelected(hero: Hero) { this.selectedHero = hero; }
}
```

This test host binds to `DashboardHeroComponent` as the `DashboardComponent` would but without the noise of the `Router`, the `HeroService`, or the `*ngFor` repeater.

The test host sets the component's `hero` input property with its test hero. It binds the component's `selected` event with its `onSelected` handler, which records the emitted hero in its `selectedHero` property.

Later, the tests will be able to easily check `selectedHero` to verify that the `DashboardHeroComponent.selected` event emitted the expected hero.

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

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

```
TestBed.configureTestingModule({
  declarations: [ DashboardHeroComponent, TestHostComponent ]
})
// create TestHostComponent instead of DashboardHeroComponent
fixture  = TestBed.createComponent(TestHostComponent);
testHost = fixture.componentInstance;
heroEl   = fixture.nativeElement.querySelector('.hero');
fixture.detectChanges(); // trigger initial data binding
```

This testing module configuration shows three important differences:

1. It *declares* both the `DashboardHeroComponent` and the `TestHostComponent`.
2. It *creates* the `TestHostComponent` instead of the `DashboardHeroComponent`.
3. The `TestHostComponent` sets the `DashboardHeroComponent.hero` with a binding.

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

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

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

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

```
it('should display hero name', () => {
  const expectedPipedName = testHost.hero.name.toUpperCase();
  expect(heroEl.textContent).toContain(expectedPipedName);
});

it('should raise selected event when clicked', () => {
  click(heroEl);
  // selected hero should be the same data bound hero
  expect(testHost.selectedHero).toBe(testHost.hero);
});
```

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

Routing component

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

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

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

```
constructor(
  private router: Router,
  private heroService: HeroService) {
}
```

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

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

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

```
gotoDetail(hero: Hero) {
  const url = `/heroes/${hero.id}`;
  this.router.navigateByUrl(url);
}
```

This is often the case with *routing components*. As a rule you test the component, not the router, and care only if the component navigates with the right address under the given conditions.

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

app/dashboard/dashboard.component.spec.ts (spies)

```
const routerSpy = jasmine.createSpyObj('Router', ['navigateByUrl']);
const heroServiceSpy = jasmine.createSpyObj('HeroService', ['getHeroes']);

TestBed.configureTestingModule({
  providers: [
    { provide: HeroService, useValue: heroServiceSpy },
    { provide: Router,      useValue: routerSpy }
  ]
})
```

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

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

```
it('should tell ROUTER to navigate when hero clicked', () => {

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

  // args passed to router.navigateByUrl() spy
  const spy = router.navigateByUrl as jasmine.Spy;
  const navArgs = spy.calls.first().args[0];

  // expecting to navigate to id of the component's first hero
  const id = comp.heroes[0].id;
  expect(navArgs).toBe('/heroes/' + id,
    'should nav to HeroDetail for first hero');
});
```

Routed components

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

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

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

Here's the `HeroDetailComponent` constructor:

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

```
constructor(
  private heroDetailService: HeroDetailService,
  private route: ActivatedRoute,
  private router: Router) {
}
```

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

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

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

```
ngOnInit(): void {
  // get hero when `id` param changes
  this.route.paramMap.subscribe(pmap => this.getHero(pmap.get('id')));
}
```

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

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

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

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

- `paramMap` returns an `Observable` that can emit more than one value during a test.
- You need the router helper function, `convertToParamMap()`, to create a `ParamMap`.

- Other *routed component* tests need a test double for `ActivatedRoute`.

These differences argue for a re-usable stub class.

ActivatedRouteStub

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

testing/activated-route-stub.ts (ActivatedRouteStub)

```
import { convertToParamMap, ParamMap, Params } from '@angular/router';
import { ReplaySubject } from 'rxjs';

/**
 * An ActivateRoute test double with a `paramMap` observable.
 * Use the `setParamMap()` method to add the next `paramMap` value.
 */
export class ActivatedRouteStub {
  // Use a ReplaySubject to share previous values with subscribers
  // and pump new values into the `paramMap` observable
  private subject = new ReplaySubject<ParamMap>();

  constructor(initialParams?: Params) {
    this.setParamMap(initialParams);
  }

  /** The mock paramMap observable */
  readonly paramMap = this.subject.asObservable();

  /** Set the paramMap observables's next value */
  setParamMap(params?: Params) {
    this.subject.next(convertToParamMap(params));
  }
}
```

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

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

Testing with *ActivatedRouteStub*

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

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

```
describe('when navigate to existing hero', () => {
  let expectedHero: Hero;

  beforeEach(async(() => {
    expectedHero = firstHero;
    activatedRoute.setParamMap({ id: expectedHero.id });
    createComponent();
  }));

  it('should display that hero\'s name', () => {
    expect(page.nameDisplay.textContent).toBe(expectedHero.name);
  });
});
```

> The `createComponent()` method and `page` object are discussed below. Rely on your intuition for now.

When the `id` cannot be found, the component should re-route to the `HeroListComponent`.

The test suite setup provided the same router spy described above which spies on the router without actually navigating.

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

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

```
describe('when navigate to non-existent hero id', () => {
  beforeEach(async(() => {
    activatedRoute.setParamMap({ id: 99999 });
    createComponent();
  }));

  it('should try to navigate back to hero list', () => {
    expect(page.gotoListSpy.calls.any()).toBe(true, 'comp.gotoList called');
    expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
  });
});
```

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

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

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

```
describe('when navigate with no hero id', () => {
  beforeEach(async( createComponent ));

  it('should have hero.id === 0', () => {
    expect(component.hero.id).toBe(0);
  });

  it('should display empty hero name', () => {
    expect(page.nameDisplay.textContent).toBe('');
  });
});
```

Nested component tests

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

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

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

app/app.component.html

```
<app-banner></app-banner>
<app-welcome></app-welcome>
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

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

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

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

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

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

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

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

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

Stubbing unneeded components

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

app/app.component.spec.ts (stub declaration)

```
@Component({selector: 'app-banner', template: ''})
class BannerStubComponent {}

@Component({selector: 'router-outlet', template: ''})
class RouterOutletStubComponent { }

@Component({selector: 'app-welcome', template: ''})
class WelcomeStubComponent {}
```

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

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

app/app.component.spec.ts (TestBed stubs)

```
TestBed.configureTestingModule({
  declarations: [
    AppComponent,
    RouterLinkDirectiveStub,
    BannerStubComponent,
    RouterOutletStubComponent,
    WelcomeStubComponent
  ]
})
```

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

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

The rest are stubs.

NO_ERRORS_SCHEMA

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

app/app.component.spec.ts (NO_ERRORS_SCHEMA)

```
TestBed.configureTestingModule({
  declarations: [
    AppComponent,
    RouterLinkDirectiveStub
  ],
  schemas: [ NO_ERRORS_SCHEMA ]
})
```

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

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

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

You no longer need the stub components.

Use both techniques together

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

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

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

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

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

app/app.component.spec.ts (mixed setup)

```
TestBed.configureTestingModule({
  declarations: [
    AppComponent,
    BannerStubComponent,
    RouterLinkDirectiveStub
  ],
  schemas: [ NO_ERRORS_SCHEMA ]
})
```

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

Components with *RouterLink*

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

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

testing/router-link-directive-stub.ts (RouterLinkDirectiveStub)

```
@Directive({
  selector: '[routerLink]'
})
export class RouterLinkDirectiveStub {
  @Input('routerLink') linkParams: any;
  navigatedTo: any = null;

  @HostListener('click')
  onClick() {
    this.navigatedTo = this.linkParams;
  }
}
```

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

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

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

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

By.directive and injected directives

A little more setup triggers the initial data binding and gets references to the navigation links:

app/app.component.spec.ts (test setup)

```
beforeEach(() => {
  fixture.detectChanges(); // trigger initial data binding
```

```
    // find DebugElements with an attached RouterLinkStubDirective
    linkDes = fixture.debugElement
      .queryAll(By.directive(RouterLinkDirectiveStub));

    // get attached link directive instances
    // using each DebugElement's injector
    routerLinks = linkDes.map(de => de.injector.get(RouterLinkDirectiveStub));
  });
```

Three points of special interest:

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

The `AppComponent` links to validate are as follows:

app/app.component.html (navigation links)

```
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
  <a routerLink="/about">About</a>
</nav>
```

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

app/app.component.spec.ts (selected tests)

```
it('can get RouterLinks from template', () => {
  expect(routerLinks.length).toBe(3, 'should have 3 routerLinks');
  expect(routerLinks[0].linkParams).toBe('/dashboard');
  expect(routerLinks[1].linkParams).toBe('/heroes');
  expect(routerLinks[2].linkParams).toBe('/about');
});

it('can click Heroes link in template', () => {
  const heroesLinkDe = linkDes[1];   // heroes link DebugElement
  const heroesLink = routerLinks[1]; // heroes link directive

  expect(heroesLink.navigatedTo).toBeNull('should not have navigated yet');

  heroesLinkDe.triggerEventHandler('click', null);
```

```
    fixture.detectChanges();

    expect(heroesLink.navigatedTo).toBe('/heroes');
  });
```

> The "click" test *in this example* is misleading. It tests the `RouterLinkDirectiveStub` rather than the *component*. This is a common failing of directive stubs.
>
> It has a legitimate purpose in this guide. It demonstrates how to find a `RouterLink` element, click it, and inspect a result, without engaging the full router machinery. This is a skill you 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.

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.

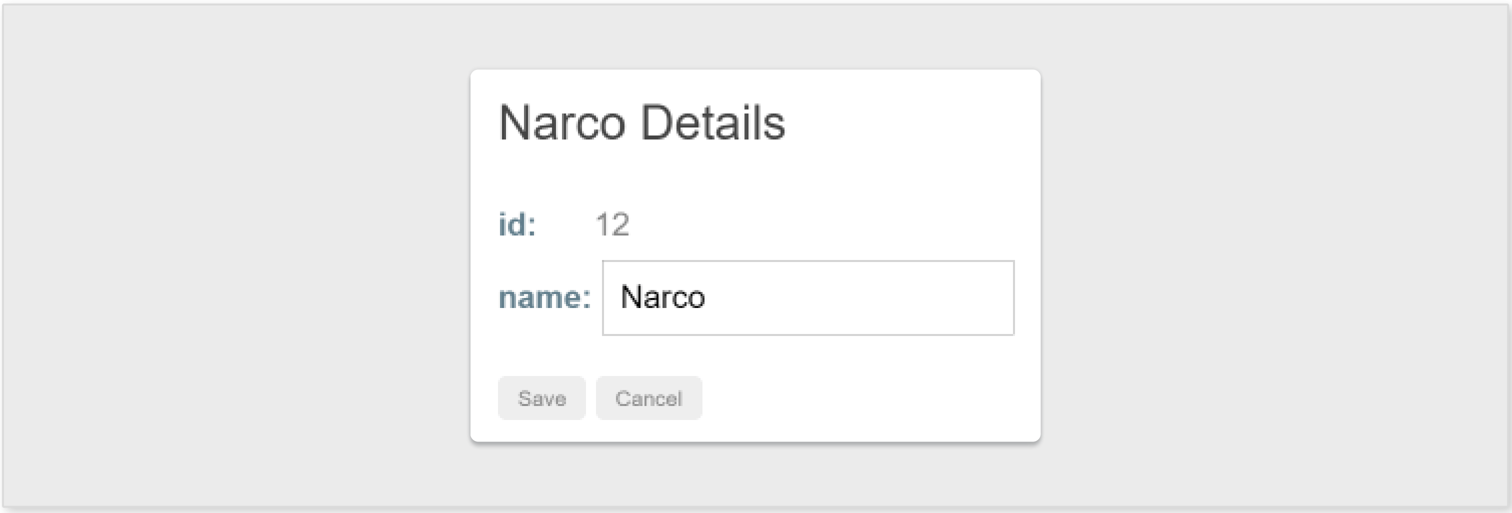
Stubbing the RouterLink and RouterOutlet is the best option for such limited testing goals. Relying on the real router would make them brittle. They could fail for reasons unrelated to the component. For example, a navigation guard could prevent an unauthorized user from visiting the `HeroListComponent`. 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`.

Use a *page* object

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


Narco Details

id: 12

name: Narco

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

app/hero/hero-detail.component.html

```html
<div *ngIf="hero">
  <h2><span>{{hero.name | titlecase}}</span> Details</h2>
  <div>
    <label>id: </label>{{hero.id}}</div>
  <div>
    <label for="name">name: </label>
    <input id="name" [(ngModel)]="hero.name" placeholder="name" />
  </div>
  <button (click)="save()">Save</button>
  <button (click)="cancel()">Cancel</button>
</div>
```

Tests that exercise the component need ...

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

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

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

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

app/hero/hero-detail.component.spec.ts (Page)

```typescript
class Page {
  // getter properties wait to query the DOM until called.
  get buttons()     { return this.queryAll<HTMLButtonElement>('button'); }
  get saveBtn()     { return this.buttons[0]; }
  get cancelBtn()   { return this.buttons[1]; }
  get nameDisplay() { return this.query<HTMLElement>('span'); }
  get nameInput()   { return this.query<HTMLInputElement>('input'); }

  gotoListSpy: jasmine.Spy;
  navigateSpy: jasmine.Spy;

  constructor(someFixture: ComponentFixture<HeroDetailComponent>) {
    // get the navigate spy from the injected router spy object
    const routerSpy = someFixture.debugElement.injector.get(Router) as any;
    this.navigateSpy = routerSpy.navigate;

    // spy on component's `gotoList()` method
    const someComponent = someFixture.componentInstance;
    this.gotoListSpy = spyOn(someComponent, 'gotoList').and.callThrough();
  }

  //// query helpers ////
  private query<T>(selector: string): T {
    return fixture.nativeElement.querySelector(selector);
  }

  private queryAll<T>(selector: string): T[] {
    return fixture.nativeElement.querySelectorAll(selector);
  }
}
```

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.

app/hero/hero-detail.component.spec.ts (createComponent)

```
/** Create the HeroDetailComponent, initialize it, set test variables  */
function createComponent() {
  fixture = TestBed.createComponent(HeroDetailComponent);
  component = fixture.componentInstance;
  page = new Page(fixture);

  // 1st change detection triggers ngOnInit which gets a hero
  fixture.detectChanges();
  return fixture.whenStable().then(() => {
    // 2nd change detection displays the async-fetched hero
    fixture.detectChanges();
  });
}
```

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

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

app/hero/hero-detail.component.spec.ts (selected tests)

```
it('should display that hero\'s name', () => {
    expect(page.nameDisplay.textContent).toBe(expectedHero.name);
  });

  it('should navigate when click cancel', () => {
    click(page.cancelBtn);
    expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
  });

  it('should save when click save but not navigate immediately', () => {
    // Get service injected into component and spy on its`saveHero` method.
    // It delegates to fake `HeroService.updateHero` which delivers a safe test result.
    const hds = fixture.debugElement.injector.get(HeroDetailService);
    const saveSpy = spyOn(hds, 'saveHero').and.callThrough();

    click(page.saveBtn);
    expect(saveSpy.calls.any()).toBe(true, 'HeroDetailService.save called');
    expect(page.navigateSpy.calls.any()).toBe(false, 'router.navigate not called');
  });

  it('should navigate when click save and save resolves', fakeAsync(() => {
    click(page.saveBtn);
    tick(); // wait for async save to complete
    expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
  }));

  it('should convert hero name to Title Case', () => {
    // get the name's input and display elements from the DOM
    const hostElement = fixture.nativeElement;
    const nameInput: HTMLInputElement = hostElement.querySelector('input');
    const nameDisplay: HTMLElement = hostElement.querySelector('span');

    // simulate user entering a new name into the input box
    nameInput.value = 'quick BROWN  fOx';

    // dispatch a DOM event so that Angular learns of input value change.
    // use newEvent utility function (not provided by Angular) for better browser
  compatibility
    nameInput.dispatchEvent(newEvent('input'));

    // Tell Angular to update the display binding through the title pipe
    fixture.detectChanges();

    expect(nameDisplay.textContent).toBe('Quick Brown  Fox');
  });
```

Calling *compileComponents()*

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

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

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

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

app/banner/banner-external.component.ts (external template & css)

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

@Component({
  selector: 'app-banner',
  templateUrl: './banner-external.component.html',
  styleUrls:  ['./banner-external.component.css']
})
export class BannerComponent {
  title = 'Test Tour of Heroes';
}
```

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

app/banner/banner.component.spec.ts (setup that fails)

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });
  fixture = TestBed.createComponent(BannerComponent);
});
```

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

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

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

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

compileComponents() is async

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

> If you neglect to make the test function async (e.g., forget to use `async()` as described below), you'll see this error message
>
> ```
> Error: ViewDestroyedError: Attempt to use a destroyed view
> ```

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

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

To follow this pattern, import the `async()` helper with the other testing symbols.

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

The async *beforeEach*

Write the first async `beforeEach` like this.

app/banner/banner-external.component.spec.ts (async beforeEach)

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  })
  .compileComponents();  // compile template and css
}));
```

The `async()` helper function takes a parameterless function with the body of the setup.

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

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

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

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

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

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

The synchronous *beforeEach*

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

app/banner/banner-external.component.spec.ts (synchronous beforeEach)

```
beforeEach(() => {
  fixture = TestBed.createComponent(BannerComponent);
  component = fixture.componentInstance; // BannerComponent test instance
  h1 = fixture.nativeElement.querySelector('h1');
});
```

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

Consolidated setup

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

The `compileComponents()` method returns a promise so you can perform the synchronous setup tasks *after* compilation by moving the synchronous code into a `then(...)` callback.

app/banner/banner-external.component.spec.ts (one beforeEach)

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  })
  .compileComponents()
  .then(() => {
    fixture = TestBed.createComponent(BannerComponent);
    component = fixture.componentInstance;
    h1 = fixture.nativeElement.querySelector('h1');
  });
}));
```

compileComponents() is harmless

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

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

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

Setup with module imports

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

app/dashboard/dashboard-hero.component.spec.ts (configure TestBed)

```
TestBed.configureTestingModule({
  declarations: [ DashboardHeroComponent ]
})
```

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:

app/hero/hero-detail.component.spec.ts (FormsModule setup)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports:      [ FormsModule ],
    declarations: [ HeroDetailComponent, TitleCasePipe ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService,    useClass: TestHeroService },
      { provide: Router,         useValue: routerSpy},
    ]
  })
  .compileComponents();
}));
```

> Notice that the `beforeEach()` is asynchronous and calls `TestBed.compileComponents` because the `HeroDetailComponent` has an external template and css file.
>
> As explained in *Calling compileComponents()* above, these tests could be run in a non-CLI environment where Angular would have to compile them in the browser.

Import a shared module

Because many 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:

app/hero/hero-detail.component.spec.ts (SharedModule setup)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports:      [ SharedModule ],
    declarations: [ HeroDetailComponent ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService,    useClass: TestHeroService },
      { provide: Router,         useValue: routerSpy},
    ]
  })
  .compileComponents();
}));
```

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

Import a feature module

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

app/hero/hero-detail.component.spec.ts (HeroModule setup)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports:   [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService,    useClass: TestHeroService },
      { provide: Router,         useValue: routerSpy},
    ]
  })
  .compileComponents();
}));
```

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

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

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

Override component providers

The `HeroDetailComponent` provides its own `HeroDetailService`.

app/hero/hero-detail.component.ts (prototype)

```
@Component({
  selector:    'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls:  ['./hero-detail.component.css' ],
  providers:  [ HeroDetailService ]
})
export class HeroDetailComponent implements OnInit {
  constructor(
    private heroDetailService: HeroDetailService,
    private route: ActivatedRoute,
    private router: Router) {
  }
}
```

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`.
>
> app/hero/hero-detail.service.ts (prototype)
>
> ```
> @Injectable()
> export class HeroDetailService {
>   constructor(private heroService: HeroService) {  }
> /* . . . */
> }
> ```
>
> The previous test configuration replaces the real `HeroService` with a `TestHeroService` that intercepts server requests and fakes their responses.

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

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

app/hero/hero-detail.component.spec.ts (Override setup)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports:   [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: Router,         useValue: routerSpy},
    ]
  })
```

```
    // Override component's own provider
    .overrideComponent(HeroDetailComponent, {
      set: {
        providers: [
          { provide: HeroDetailService, useClass: HeroDetailServiceSpy }
        ]
      }
    })

    .compileComponents();
  }));
```

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

The *overrideComponent* method

Focus on the `overrideComponent` method.

app/hero/hero-detail.component.spec.ts (overrideComponent)

```
.overrideComponent(HeroDetailComponent, {
  set: {
    providers: [
      { provide: HeroDetailService, useClass: HeroDetailServiceSpy }
    ]
  }
})
```

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

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

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

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

```
selector?: string;
template?: string;
```

```
  templateUrl?: string;
  providers?: any[];
  ...
```

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:

app/hero/hero-detail.component.spec.ts (HeroDetailServiceSpy)

```
class HeroDetailServiceSpy {
  testHero: Hero = {id: 42, name: 'Test Hero' };

  /* emit cloned test hero */
  getHero = jasmine.createSpy('getHero').and.callFake(
    () => asyncData(Object.assign({}, this.testHero))
  );

  /* emit clone of test hero, with changes merged in */
  saveHero = jasmine.createSpy('saveHero').and.callFake(
    (hero: Hero) => asyncData(Object.assign(this.testHero, hero))
  );
}
```

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.

app/hero/hero-detail.component.spec.ts (override tests)

```
let hdsSpy: HeroDetailServiceSpy;

beforeEach(async(() => {
  createComponent();
  // get the component's injected HeroDetailServiceSpy
  hdsSpy = fixture.debugElement.injector.get(HeroDetailService) as any;
}));
```

```
  it('should have called `getHero`', () => {
    expect(hdsSpy.getHero.calls.count()).toBe(1, 'getHero called once');
  });

  it('should display stub hero\'s name', () => {
    expect(page.nameDisplay.textContent).toBe(hdsSpy.testHero.name);
  });

  it('should save stub hero change', fakeAsync(() => {
    const origName = hdsSpy.testHero.name;
    const newName = 'New Name';

    page.nameInput.value = newName;
    page.nameInput.dispatchEvent(newEvent('input')); // tell Angular

    expect(component.hero.name).toBe(newName, 'component hero has new name');
    expect(hdsSpy.testHero.name).toBe(origName, 'service hero unchanged before save');

    click(page.saveBtn);
    expect(hdsSpy.saveHero.calls.count()).toBe(1, 'saveHero called once');

    tick(); // wait for async save to complete
    expect(hdsSpy.testHero.name).toBe(newName, 'service hero has new name after save');
    expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
  }));
```

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.