Testing services

Contents ›

To check that your services are working as you intend, you can write tests specifically for them.

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

Services are often the easiest files to unit test. Here are some synchronous and asynchronous unit tests of the `ValueService` written without assistance from Angular testing utilities.

app/demo/demo.spec.ts

```
// Straight Jasmine testing without Angular's testing support
describe('ValueService', () => {
  let service: ValueService;
  beforeEach(() => { service = new ValueService(); });

  it('#getValue should return real value', () => {
    expect(service.getValue()).toBe('real value');
  });

  it('#getObservableValue should return value from observable',
    (done: DoneFn) => {
    service.getObservableValue().subscribe(value => {
      expect(value).toBe('observable value');
      done();
    });
  });

  it('#getPromiseValue should return value from a promise',
    (done: DoneFn) => {
    service.getPromiseValue().then(value => {
      expect(value).toBe('promise value');
      done();
    });
  });
});
```

Services with dependencies

Services often depend on other services that Angular injects into the constructor. In many cases, it's easy to create and *inject* these dependencies by hand while calling the service's constructor.

The `MasterService` is a simple example:

app/demo/demo.ts

```
@Injectable()
export class MasterService {
  constructor(private valueService: ValueService) { }
  getValue() { return this.valueService.getValue(); }
}
```

`MasterService` delegates its only method, `getValue`, to the injected `ValueService`.

Here are several ways to test it.

app/demo/demo.spec.ts

```
describe('MasterService without Angular testing support', () => {
  let masterService: MasterService;

  it('#getValue should return real value from the real service', () => {
    masterService = new MasterService(new ValueService());
    expect(masterService.getValue()).toBe('real value');
  });

  it('#getValue should return faked value from a fakeService', () => {
    masterService = new MasterService(new FakeValueService());
    expect(masterService.getValue()).toBe('faked service value');
  });

  it('#getValue should return faked value from a fake object', () => {
    const fake =  { getValue: () => 'fake value' };
    masterService = new MasterService(fake as ValueService);
    expect(masterService.getValue()).toBe('fake value');
  });

  it('#getValue should return stubbed value from a spy', () => {
    // create `getValue` spy on an object representing the ValueService
    const valueServiceSpy =
      jasmine.createSpyObj('ValueService', ['getValue']);

    // set the value to return when the `getValue` spy is called.
    const stubValue = 'stub value';
    valueServiceSpy.getValue.and.returnValue(stubValue);

    masterService = new MasterService(valueServiceSpy);

    expect(masterService.getValue())
      .toBe(stubValue, 'service returned stub value');
    expect(valueServiceSpy.getValue.calls.count())
      .toBe(1, 'spy method was called once');
    expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)
      .toBe(stubValue);
  });
});
```

The first test creates a `ValueService` with `new` and passes it to the `MasterService` constructor.

However, injecting the real service rarely works well as most dependent services are difficult to create and control.

Instead you can mock the dependency, use a dummy value, or create a spy on the pertinent service method.

> Prefer spies as they are usually the easiest way to mock services.

These standard testing techniques are great for unit testing services in isolation.

However, you almost always inject services into application classes using Angular dependency injection and you should have tests that reflect that usage pattern. Angular testing utilities make it easy to investigate how injected services behave.

Testing services with the *TestBed*

Your app relies on Angular dependency injection (DI) to create services. When a service has a dependent service, DI finds or creates that dependent service. And if that dependent service has its own dependencies, DI finds-or-creates them as well.

As service *consumer*, you don't worry about any of this. You don't worry about the order of constructor arguments or how they're created.

As a service *tester*, you must at least think about the first level of service dependencies but you *can* let Angular DI do the service creation and deal with constructor argument order when you use the `TestBed` testing utility to provide and create services.

Angular *TestBed*

The `TestBed` is the most important of the Angular testing utilities. The `TestBed` creates a dynamically-constructed Angular *test* module that emulates an Angular @NgModule.

The `TestBed.configureTestingModule()` method takes a metadata object that can have most of the properties of an @NgModule.

To test a service, you set the `providers` metadata property with an array of the services that you'll test or mock.

app/demo/demo.testbed.spec.ts (provide ValueService in beforeEach)

```
let service: ValueService;

beforeEach(() => {
  TestBed.configureTestingModule({ providers: [ValueService] });
});
```

Then inject it inside a test by calling `TestBed.inject()` with the service class as the argument.

> **Note:** `TestBed.get()` was deprecated as of Angular version 9. To help minimize breaking changes, Angular introduces a new function called `TestBed.inject()`, which you should use instead. For information on the removal of `TestBed.get()`, see its entry in the Deprecations index.

```
it('should use ValueService', () => {
  service = TestBed.inject(ValueService);
  expect(service.getValue()).toBe('real value');
});
```

Or inside the `beforeEach()` if you prefer to inject the service as part of your setup.

```
beforeEach(() => {
  TestBed.configureTestingModule({ providers: [ValueService] });
  service = TestBed.inject(ValueService);
});
```

When testing a service with a dependency, provide the mock in the `providers` array.

In the following example, the mock is a spy object.

```
let masterService: MasterService;
let valueServiceSpy: jasmine.SpyObj<ValueService>;

beforeEach(() => {
  const spy = jasmine.createSpyObj('ValueService', ['getValue']);
```

```
    TestBed.configureTestingModule({
      // Provide both the service-to-test and its (spy) dependency
      providers: [
        MasterService,
        { provide: ValueService, useValue: spy }
      ]
    });
    // Inject both the service-to-test and its (spy) dependency
    masterService = TestBed.inject(MasterService);
    valueServiceSpy = TestBed.inject(ValueService) as jasmine.SpyObj<ValueService>;
  });
```

The test consumes that spy in the same way it did earlier.

```
  it('#getValue should return stubbed value from a spy', () => {
    const stubValue = 'stub value';
    valueServiceSpy.getValue.and.returnValue(stubValue);

    expect(masterService.getValue())
      .toBe(stubValue, 'service returned stub value');
    expect(valueServiceSpy.getValue.calls.count())
      .toBe(1, 'spy method was called once');
    expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)
      .toBe(stubValue);
  });
```

Testing without *beforeEach()*

Most test suites in this guide call `beforeEach()` to set the preconditions for each `it()` test and rely on the `TestBed` to create classes and inject services.

There's another school of testing that never calls `beforeEach()` and prefers to create classes explicitly rather than use the `TestBed`.

Here's how you might rewrite one of the `MasterService` tests in that style.

Begin by putting re-usable, preparatory code in a *setup* function instead of `beforeEach()`.

app/demo/demo.spec.ts (setup)

```
  function setup() {
    const valueServiceSpy =
      jasmine.createSpyObj('ValueService', ['getValue']);
    const stubValue = 'stub value';
    const masterService = new MasterService(valueServiceSpy);
    valueServiceSpy.getValue.and.returnValue(stubValue);
    return { masterService, stubValue, valueServiceSpy };
  }
```

The `setup()` function returns an object literal with the variables, such as `masterService`, that a test might reference. You don't define *semi-global* variables (e.g., `let masterService: MasterService`) in the body of the `describe()`.

Then each test invokes `setup()` in its first line, before continuing with steps that manipulate the test subject and assert expectations.

```
it('#getValue should return stubbed value from a spy', () => {
  const { masterService, stubValue, valueServiceSpy } = setup();
  expect(masterService.getValue())
    .toBe(stubValue, 'service returned stub value');
  expect(valueServiceSpy.getValue.calls.count())
    .toBe(1, 'spy method was called once');
  expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)
    .toBe(stubValue);
});
```

Notice how the test uses *destructuring assignment* to extract the setup variables that it needs.

```
const { masterService, stubValue, valueServiceSpy } = setup();
```

Many developers feel this approach is cleaner and more explicit than the traditional `beforeEach()` style.

Although this testing guide follows the traditional style and the default CLI schematics generate test files with `beforeEach()` and `TestBed`, feel free to adopt *this alternative approach* in your own projects.

Testing HTTP services

Data services that make HTTP calls to remote servers typically inject and delegate to the Angular `HttpClient` service for XHR calls.

You can test a data service with an injected `HttpClient` spy as you would test any service with a dependency.

app/model/hero.service.spec.ts (tests with spies)

```
let httpClientSpy: { get: jasmine.Spy };
let heroService: HeroService;

beforeEach(() => {
  // TODO: spy on other methods too
  httpClientSpy = jasmine.createSpyObj('HttpClient', ['get']);
  heroService = new HeroService(httpClientSpy as any);
});

it('should return expected heroes (HttpClient called once)', () => {
  const expectedHeroes: Hero[] =
    [{ id: 1, name: 'A' }, { id: 2, name: 'B' }];

  httpClientSpy.get.and.returnValue(asyncData(expectedHeroes));

  heroService.getHeroes().subscribe(
    heroes => expect(heroes).toEqual(expectedHeroes, 'expected heroes'),
    fail
  );
  expect(httpClientSpy.get.calls.count()).toBe(1, 'one call');
});

it('should return an error when the server returns a 404', () => {
  const errorResponse = new HttpErrorResponse({
    error: 'test 404 error',
    status: 404, statusText: 'Not Found'
  });

  httpClientSpy.get.and.returnValue(asyncError(errorResponse));

  heroService.getHeroes().subscribe(
    heroes => fail('expected an error, not heroes'),
    error  => expect(error.message).toContain('test 404 error')
  );
});
```

The `HeroService` methods return `Observables`. You must *subscribe* to an observable to (a) cause it to execute and (b) assert that the method succeeds or fails.

The `subscribe()` method takes a success (`next`) and fail (`error`) callback. Make sure you provide *both* callbacks so that you capture errors. Neglecting to do so produces an asynchronous uncaught observable error that the test runner will likely attribute to a completely different test.

HttpClientTestingModule

Extended interactions between a data service and the `HttpClient` can be complex and difficult to mock with spies.

The `HttpClientTestingModule` can make these testing scenarios more manageable.

While the *code sample* accompanying this guide demonstrates `HttpClientTestingModule`, this page defers to the Http guide, which covers testing with the `HttpClientTestingModule` in detail.