# Testing

After spending hours, days, months on a web app you're finally ready to release it to the world. Plenty of hard work and time has been poured into it and now it's time for it to pay off… and then boom: a blocking bug shows up that prevents anyone from signing up.

## Test driven?

Testing can help reveal bugs before they appear, instill confidence in your web application, and makes it easy to onboard new developers into the application. There is little doubt about the power of testing amongst the world of software development. However, there is debate about how to go about it.

Is it better to write the tests first and then write the implementation to make those tests pass or would it be better to validate that code that we've already written is correct? It's pretty odd to think that this is a source of contention across the development community, but there is a debate that can get pretty heated as to which is the *right* way to handle testing.

In our experience, particularly when coming from a prototype-heavy background, we focus on building test-able code. Although your experience may differ, we have found that while we are prototyping applications, testing individual pieces of code that are likely to change can double or triple the amount of work it takes to keep them up. In contrast, we focus on building our applications in small components, keeping large amounts of functionality broken into several methods which allows us to test the functionality of a part of the larger picture. This is what we mean when we say *testable* code.

An alternative methodology to prototyping (and then testing after) is called "Red-Green-Refactor". The idea is that you **write your tests first** and they fail (red) because you haven't written any code yet. Only after you have failing tests do you go on to write your implementation code until it all passes (green).

Of course, the decision of *what* to test is up to you and your team, however we'll focus on *how* to test your applications in this chapter.

## End-to-end vs. Unit Testing

There are two major ways to test you applications: *end-to-end testing* or *unit testing*.

If you take a top-down approach on testing you write tests that see the application as a "black box" and you interact with the application like a user would and evaluate if the app seems to work from the "outside". This top-down technique of testing is called *End to End testing*.

In the Angular world, the tool that is mostly used is called Protractor[134]. Protractor is a tool that opens a browser and interacts with the application, collecting results, to check whether the testing expectations were met.

The second testing approach commonly used is to isolate each part of the application and test it in isolation. This form of testing is called *Unit Testing*.

In Unit Testing we write tests that provide a given input to a given aspect of that unit and evaluate the output to make sure it matches our expectations.

In this chapter we're going to be covering how to **unit test** your Angular apps.

## Testing Tools

In order to test our apps, we'll use two tools: Jasmine and Karma.

### Jasmine

Jasmine[135] is a behavior-driven development framework for testing JavaScript code.

Using Jasmine, you can set expectations about what your code should do when invoked.

For instance, let's assume we have a `sum` function on a `Calculator` object. We want to make sure that adding 1 and 1 results in 2. We could express that test (also called a _spec), by writing the following code:

```
describe('Calculator', () => {
  it('sums 1 and 1 to 2', () => {
    var calc = new Calculator();
    expect(calc.sum(1, 1)).toEqual(2);
  });
});
```

One of the nice things about Jasmine is how readable the tests are. You can see here that we expect the `calc.sub` operation to equal 2.

We organize our tests with `describe` blocks and `it` blocks.

[134]https://angular.github.io/protractor/#/
[135]http://jasmine.github.io/2.4/introduction.html

Normally we use `describe` for each logical unit we're testing and inside that each we use one `it` for each expectation you want to assert. However, this isn't a hard and fast rule. You'll often see an `it` block contain several expectations.

On the `Calculator` example above we have a very simple object. For that reason, we used one describe block for the whole class and one `it` block for each method.

This is not the case most of the times. For example, methods that produce different outcomes depending on the input will probably have more than one `it` block associated. On those cases, it's perfectly fine to have nested describes: one for the object and one for each method, and then different assertions inside individual `it` blocks.

We'll be looking at a lot of `describe` and `it` blocks throughout this chapter, so don't worry if it isn't clear when to use one vs. the other. We'll be showing lots of examples.

For more information about Jasmine and all its syntax, check out the Jasmine documentation page[136].

### Karma

With Jasmine we can describe our tests and their expectations. Now, in order to actually run the tests we need to have a browser environment.

That's where Karma comes in. Karma allows us to run JavaScript code within a browser like Chrome or Firefox, or on a **headless** browser (or a browser that doesn't expose a user interface) like PhantomJS.

## Writing Unit Tests

Our main focus on this section will be to understand how we write unit tests against different parts of our Angular apps.

We're going to learn to test **Services**, **Components**, **HTTP requests** and more. Along the way we're also going to see a couple of different techniques to make our code more testable.

## Angular Unit testing framework

Angular provides its own set of classes that build upon the Jasmine framework to help writing unit testing for the framework.

The main testing framework can be found on the `@angular/core/testing` package. (Although, for testing components we'll use the `@angular/compiler/testing` package and `@angular/platform-browser/testing` for some other helpers. But more on that later.)

[136] http://jasmine.github.io/2.4/introduction.html

If this is your first time testing Angular I want to prepare you for something: When you write tests for Angular, there is a bit of setup.

For instance, when we have dependencies to inject, we often manually configure them. When we want to test a component, we have to use testing-helpers to initialize them. And when we want to test routing, there are quite a few dependencies we need to structure.

If it feels like there is a lot of setup, don't worry: you'll get the hang of it and find that the setup doesn't change that much from project to project. Besides, we'll walk you through each step in this chapter.

As always, you can find all of the sample code for this chapter in the code download. Looking over the code directly in your favorite editor can provide a good overview of the details we cover in this chapter. We'd encourage you to keep the code open as you go through this chapter.

## Setting Up Testing

Earlier in the Routing Chapter we created an application for searching for music. In this chapter, let's write tests for that application.

Karma requires a configuration in order to run. So the first thing we need to do to setup Karma is to create a `karma.conf.js` file.

Let's `karma.conf.js` file on the root path of our project, like so:

Since we're using Angular CLI, this `karma.conf.js` file is already created for us! However, if your project does not use Angular CLI, you may need to setup Karma on your own.

**code/routes/music/karma.conf.js**

```
// Karma configuration file, see link for more information
// https://karma-runner.github.io/0.13/config/configuration-file.html

module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular/cli'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage-istanbul-reporter'),
      require('@angular/cli/plugins/karma')
    ],
    client:{
      clearContext: false // leave Jasmine Spec Runner output visible in browser
    },
    files: [
      { pattern: './src/test.ts', watched: false }
    ],
    preprocessors: {
      './src/test.ts': ['@angular/cli']
    },
    mime: {
      'text/x-typescript': ['ts','tsx']
    },
    coverageIstanbulReporter: {
      reports: [ 'html', 'lcovonly' ],
      fixWebpackSourcePaths: true
    },
    angularCli: {
      environment: 'dev'
    },
    reporters: config.angularCli && config.angularCli.codeCoverage
              ? ['progress', 'coverage-istanbul']
              : ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false
  });
};
```

Don't worry too much about this file's contents right now, just keep in mind a few things about it:

- sets PhantomJS as the target testing browser;
- uses Jasmine karma framework for testing;
- uses a WebPack bundle called `test.bundle.js` that basically wraps all our testing and app code;

The next step is to create a new `test` folder to hold our test files.

```
mkdir test
```

## Testing Services and HTTP

Services in Angular start out their life as plain classes. In one sense, this makes our services easy to test because we can sometimes test them directly without using Angular.

With Karma configuration done, let's start testing the `SpotifyService` class. If we remember, this service works by interacting with the Spotify API to retrieve album, track and artist information.

Inside the `test` folder, let's create a `service` subfolder where all our service tests will go. Finally, let's create our first test file inside it, called `spotify.service.spec.ts`.

Now we can start putting this test file together. The first thing we need to do is import the test helpers from the `@angular/core/testing` package:

**code/routes/music/src/app/spotify.service.spec.ts**

```
import {
  inject,
  fakeAsync,
  tick,
  TestBed
} from '@angular/core/testing';
```

Next, we'll import a couple more classes:

**code/routes/music/src/app/spotify.service.spec.ts**

```
import {MockBackend} from '@angular/http/testing';
import {
  Http,
  ConnectionBackend,
  BaseRequestOptions,
  Response,
  ResponseOptions
} from '@angular/http';
```

Since our service uses HTTP requests, we'll import the `MockBackend` class from `@angular/http/testing` package. This class will help us set expectations and verify HTTP requests.

The last thing we need to import is the class we're testing:

**code/routes/music/src/app/spotify.service.spec.ts**

```
import { SpotifyService } from './spotify.service';
```

## HTTP Considerations

We could start writing our tests right now, but during each test execution we would be calling out and hitting the Spotify server. This is far from ideal for two reasons:

1. HTTP requests are relatively slow and as our test suite grows, we'd notice it takes longer and longer to run all of the tests.
2. Spotify's API has a quota, and if our whole team is running the tests, we might use up our API call resources needlessly
3. If we are offline or if Spotify is down or inaccessible our tests would start breaking, even though our code might technically be correct

This is a good hint when writing unit tests: isolate everything that you don't control before testing.

In our case, this piece is the Spotify service. The solution is that we will replace the HTTP request with something that would behave like it, but will **not hit the real Spotify server**.

Doing this in the testing world is called *mocking* a dependency. They are sometimes also called *stubbing* a dependency.

You can read more about the difference between Mocks and Stubs in this article Mocks are not Stubs[137]

Let's pretend we're writing code that depends on a given `Car` class.

This class has a bunch of methods: you can `start` a car instance, `stop` it, `park` it and `getSpeed` of that car.

Let's see how we could use stubs and mocks to write tests that depend on this class.

## Stubs

**Stubs** are objects we create on the fly, with a subset of the behaviors our dependency has.

Let's write a test that just interacts with the `start` method of the class.

You could create a *stub* of that `Car` class on-the-fly and inject that into the class you're testing:

[137] http://martinfowler.com/articles/mocksArentStubs.html

```
describe('Speedtrap', function() {
  it('tickets a car at more than 60mph', function() {
    var stubCar = { getSpeed: function() { return 61; } };
    var speedTrap = new SpeedTrap(stubCar);
    speedTrap.ticketCount = 0;
    speedTrap.checkSpeed();
    expect(speedTrap.ticketCount).toEqual(1);
  });
});
```

This would be a typical case for using a stub and we'd probably only use it locally to that test.

## Mocks

**Mocks** in our case will be a more complete representation of objects, that overrides parts or all of the behavior of the dependency. Mocks can, and most of the time will be reused by more than one test across our suite.

They will also be used sometimes to assert that given methods were called the way they were supposed to be called.

One example of a mock version of our `Car` class would be:

```
class MockCar {
  startCallCount: number = 0;

  start() {
    this.startCallCount++;
  }
}
```

And it would be used to write another test like this:

```
describe('CarRemote', function() {
  it('starts the car when the start key is held', function() {
    var car = new MockCar();
    var remote = new CarRemote();
    remote.holdButton('start');
    expect(car.startCallCount).toEqual(1);
  });
});
```

The biggest difference between a mock and a stub is that:

- a stub provides a subset of functionality with "manual" behavior overrides whereas
- a mock generally sets expectations and verifies that certain methods were called

### `Http` `MockBackend`

Now that we have this background in mind, let's go back to writing our service test code.

Interacting with the live Spotify service every time we run our tests is a poor idea but thankfully Angular provides us with a way to create fake HTTP calls with `MockBackend`.

This class can be injected into a `Http` instance and gives us control of how we want the HTTP interaction to act. We can interfere and assert in a variety of different ways: we can manually set a response, simulate an HTTP error, and add expectations, like asserting the URL being requested matches what we want, if the provided request parameters are correct and a lot more.

So the idea here is that we're going to provide our code with a "fake" `Http` library. This "fake" library will appear to our code to be the real `Http` library: all of the methods will match, it will return responses and so on. However, we're not *actually* going to make the requests.

In fact, beyond not making the requests, our `MockBackend` will actually allow us to setup *expectations* and watch for behaviors we expect.

### `TestBed.configureTestingModule` and Providers

When we test our Angular apps we need to make sure we configure the top-level `NgModule` that we will use for this test. When we do this, we can configure providers, declare components, and import other modules: just like you would when using `NgModules` generally.

Sometimes when testing Angular code, we *manually setup injections*. This is good because it gives us more control over what we're actually testing.

So in the case of testing `Http` requests, we don't want to inject the "real" `Http` class, but instead we want to inject something that looks like `Http`, but really intercepts the requests and returns the responses we configure.

To do that, we create a version of the `Http` class that uses `MockBackend` internally.

To do this, we use the `TestBed.configureTestingModule` in the `beforeEach` hook. This hook takes a callback function that will be called before each test is run, giving us a great opportunity to configure alternative class implementations.

**code/routes/music/src/app/spotify.service.spec.ts**

```
describe('SpotifyService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        BaseRequestOptions,
        MockBackend,
        SpotifyService,
        { provide: Http,
          useFactory: (backend: ConnectionBackend,
                       defaultOptions: BaseRequestOptions) => {
                         return new Http(backend, defaultOptions);
                       }, deps: [MockBackend, BaseRequestOptions] },
      ]
    });
  });
```

Notice that `TestBed.configureTestingModule` accepts an **array of providers** in the `providers` key to be used by the test injector.

`BaseRequestOptions` and `SpotifyService` are just the default implementation of those classes. But the last provider is a little more complicated :

**code/routes/music/src/app/spotify.service.spec.ts**

```
      { provide: Http,
        useFactory: (backend: ConnectionBackend,
                     defaultOptions: BaseRequestOptions) => {
                       return new Http(backend, defaultOptions);
                     }, deps: [MockBackend, BaseRequestOptions] },
      ]
```

This code uses `provide` with `useFactory` to create a version of the `Http` class, using a factory (that's what `useFactory` does).

That factory has a signature that expects `ConnectionBackend` and a `BaseRequestOption` instances. The second key on that object is `deps: [MockBackend, BaseRequestOptions]`. That indicates that we'll be using `MockBackend` as the first parameter of the factory and `BaseRequestOptions` (the default implementation) as the second.

Finally, we return our customized `Http` class with the `MockBackend` as a result of that function.

What benefit do we get from this? Well now every time (in our test) that our code requests `Http` as an injection, it will instead receive our customized `Http` instance.

This is a powerful idea that we'll use a lot in testing: use dependency injection to customize dependencies and isolate the functionality you're trying to test.

## Testing `getTrack`

Now, when writing tests for the service, we want to verify that we're calling the correct URL.

If you haven't looked at the Routing chapter music example in a while, you can find the code for this example here

Let's write a test for the `getTrack` method:

**code/routes/music/src/app/spotify.service.ts**

```
  getTrack(id: string): Observable<any[]> {
    return this.query(`/tracks/${id}`);
  }
```

If you remember how that method works, it uses the `query` method, that builds the URL based on the parameters it receives:

**code/routes/music/src/app/spotify.service.ts**

```
  query(URL: string, params?: Array<string>): Observable<any[]> {
    let queryURL = `${SpotifyService.BASE_URL}${URL}`;
    if (params) {
      queryURL = `${queryURL}?${params.join('&')}`;
    }

    return this.http.request(queryURL).map((res: any) => res.json());
  }
```

Since we're passing `/tracks/${id}` we assume that when calling `getTrack('TRACK_ID')` the expected URL will be `https://api.spotify.com/v1/tracks/TRACK_ID`.

Here is how we write the test for this:

```
describe('getTrack', () => {
  it('retrieves using the track ID',
    inject([SpotifyService, MockBackend], fakeAsync((spotifyService, mockBackend\
) => {
      var res;
      mockBackend.connections.subscribe(c => {
        expect(c.request.url).toBe('https://api.spotify.com/v1/tracks/TRACK_ID');
        let response = new ResponseOptions({body: '{"name": "felipe"}'});
        c.mockRespond(new Response(response));
      });
      spotifyService.getTrack('TRACK_ID').subscribe((_res) => {
        res = _res;
      });
      tick();
      expect(res.name).toBe('felipe');
    }))
  );
});
```

This seems like a lot to grasp at first, so let's break it down a bit:

Every time we write tests with dependencies, we need to ask Angular injector to provide us with the instances of those classes. To do that we use:

```
inject([Class1,    /* ..., */ ClassN],
       (instance1, /* ..., */ instanceN) => {
  // ... testing code ...
})
```

When you are testing code that returns either a Promise or an RxJS Observable, you can use `fakeAsync` helper to test that code as if it were synchronous. This way every Promises are fulfilled and Observables are notified immediately after you call `tick()`.

So in this code:

```
inject([SpotifyService, MockBackend],
       fakeAsync((spotifyService, mockBackend) => {
  // ... testing code ...
}));
```

We're getting two variables: `spotifyService` and `mockBackend`. The first one has a concrete instance of the `SpotifyService` and the second is an instance `MockBackend` class. Notice that the arguments to the inner function (`spotifyService, mockBackend`) are injections of the classes specified in the first argument array of the `inject` function (`SpotifyService` and `MockBackend`).

We're also running inside `fakeAsync` which means that async code will be run synchronously when `tick()` is called.

Now that we've setup the injections and context for our test, we can start writing our "actual" test. We start by declaring a `res` variable that will eventually get the HTTP call response. Next we subscribe to `mockBackend.connections`:

```
var res;
mockBackend.connections.subscribe(c => { ... });
```

Here we're saying that whenever a new connection comes in to `mockBackend` we want to be notified (e.g. call this function).

We want to verify that the `SpotifyService` is calling out to the correct URL given the track id `TRACK_ID`. So what we do is specify an *expectation* that the URL is as we would expect. We can get the URL from the connection `c` via `c.request.url`. So we setup an expectation that `c.request.url` should be the string `'https://api.spotify.com/v1/tracks/TRACK_ID'`:

```
expect(c.request.url).toBe('https://api.spotify.com/v1/tracks/TRACK_ID');
```

When our test is run, if the request URL doesn't match, then the test will fail.

Now that we've received our request and verified that it is correct, we need to craft a response. We do this by creating a new `ResponseOptions` instance. Here we specify that it will return the JSON string: `{"name": "felipe"}` as the body of the response.

```
let response = new ResponseOptions({body: '{"name": "felipe"}'});
```

Finally, we tell the connection to replace the response with a `Response` object that wraps the `ResponseOptions` instance we created:

```
c.mockRespond(new Response(response));
```

An interesting thing to note here is that your callback function in `subscribe` can be as sophisticated as you wish it to be. You could have conditional logic based on the URL, query parameters, or anything you can read from the request object etc.

This allows us to write tests for nearly every possible scenario our code might encounter.

We have now everything setup to call the `getTrack` method with `TRACK_ID` as a parameter and tracking the response in our `res` variable:

```
spotifyService.getTrack('TRACK_ID').subscribe((_res) => {
  res = _res;
});
```

If we ended our test here, we would be waiting for the HTTP call to be made and the response to be fulfilled before the callback function would be triggered. It would also happen on a different execution path and we'd have to orchestrate our code to sync things up. Thankfully using `fakeAsync` takes that problem away. All we need to do is call `tick()` and, like magic, our async code will be executed:

```
tick();
```

We now perform one final check just to make sure our response we setup is the one we received:

```
expect(res.name).toBe('felipe');
```

If you think about it, the code for all the methods of this service are *very* similar. So let's extract the snippet we use to setup the URL expectation into a function called expectURL:

**code/routes/music/src/app/spotify.service.spec.ts**

```
  function expectURL(backend: MockBackend, url: string) {
    backend.connections.subscribe(c => {
      expect(c.request.url).toBe(url);
      const response = new ResponseOptions({body: '{"name": "felipe"}'});
      c.mockRespond(new Response(response));
    });
  }
```

Following the same lines, it should be very simple to create similar tests for `getArtist` and `getAlbum` methods:

**code/routes/music/src/app/spotify.service.spec.ts**

```
  describe('getArtist', () => {
    it('retrieves using the artist ID',
      inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
        let res;
        expectURL(backend, 'https://api.spotify.com/v1/artists/ARTIST_ID');
        svc.getArtist('ARTIST_ID').subscribe((_res) => {
          res = _res;
        });
        tick();
        expect(res.name).toBe('felipe');
      }))
    );
  });

  describe('getAlbum', () => {
    it('retrieves using the album ID',
      inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
        let res;
        expectURL(backend, 'https://api.spotify.com/v1/albums/ALBUM_ID');
        svc.getAlbum('ALBUM_ID').subscribe((_res) => {
          res = _res;
        });
        tick();
        expect(res.name).toBe('felipe');
      }))
    );
  });
```

Now `searchTrack` is slightly different: instead of calling `query`, this method uses the `search` method:

**code/routes/music/src/app/spotify.service.ts**

```
  searchTrack(query: string): Observable<any[]> {
    return this.search(query, 'track');
  }
```

And then `search` calls `query` with `/search` as the first argument and an Array containing `q=<query>` and `type=track` as the second argument:

**code/routes/music/src/app/spotify.service.ts**

```
  search(query: string, type: string): Observable<any[]> {
    return this.query(`/search`, [
      `q=${query}`,
      `type=${type}`
    ]);
  }
```

Finally, `query` will transform the parameters into a URL `path` with a `QueryString`. So now, the URL we expect to call ends with `/search?q=<query>&type=track`.

Let's now write the test for `searchTrack` that takes into consideration what we learned above:

**code/routes/music/src/app/spotify.service.spec.ts**

```
  describe('searchTrack', () => {
    it('searches type and term',
      inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
        let res;
        expectURL(backend, 'https://api.spotify.com/v1/search?q=TERM&type=track'\
);
        svc.searchTrack('TERM').subscribe((_res) => {
          res = _res;
        });
        tick();
        expect(res.name).toBe('felipe');
      }))
    );
  });
```

The test ended up also being very similar to the ones we wrote so far. Let's review what this test does:

- it hooks into the HTTP lifecycle, by adding a callback when a new HTTP connection is initiated
- it sets an expectation for the URL we expect the connection to use including the query type and the search term
- it calls the method we're testing, `searchTrack`
- it then tells Angular to complete all the pending async calls
- it finally asserts that we have the expected response

In essence, when testing services our goals should be:

1. Isolate all the dependencies by using stubs or mocks
2. In case of async calls, use `fakeAsync` and `tick` to make sure they are fulfilled
3. Call the service method you're testing
4. Assert that the returning value from the method matches what we expect

Now let's move on to the classes that usually consume the services: components.

## Testing Routing to Components

When testing components, we can either:

1. write tests that will interact with the component from the outside, passing attributes in and checking how the markup is affected or
2. test individual component methods and their output.

Those test strategies are known as **black box** and **white box** testing, respectively. During this section, we'll see a mix of both.

We'll begin by writing tests for the `ArtistComponent` class, which is one of the simpler components we have. This initial set of tests will test the component's internals, so it falls into the **white box** category of testing.

Before we jump into it, let's remember what `ArtistComponent` does:

The first thing we do on the class constructor is retrieve the **id** from the routeParams collection:

**code/routes/music/src/app/artist/artist.component.ts**

```
  constructor(private route: ActivatedRoute, private spotify: SpotifyService,
              private location: Location) {
    route.params.subscribe(params => { this.id = params['id']; });
  }
```

And with that we have our first obstacle. How can we retrieve the ID of a route without an available running router?

## Creating a Router for Testing

Remember that when we write tests in Angular we manually configure many of the classes that are injected. Routing (and testing components) has a daunting number of dependencies that we need to inject. That said, once it's configured, it isn't something we change very much and it's very easy to use.

When we test write tests it's often convenient to use `beforeEach` with `TestBed.configureTestingModule` to set the dependencies that can be injected. In the case of testing our `ArtistComponent` we're going to create a custom function that will create and configure our router for testing:

**code/routes/music/src/app/artist/artist.component.spec.ts**

```
describe('ArtistComponent', () => {
  beforeEach(async(() => {
    configureMusicTests();
  }));
```

We define `configureMusicTests` in the helper file `MusicTestHelpers.ts`. Let's look at that now.

Here's the implementation of `configureMusicTests`. Don't worry, we'll explain each part:

**code/routes/music/src/app/test/test.module.ts**

```
export function configureMusicTests() {
  const mockSpotifyService: MockSpotifyService = new MockSpotifyService();

  TestBed.configureTestingModule({
    imports: [
      { // TODO RouterTestingModule.withRoutes coming soon
        ngModule: RouterTestingModule,
        providers: [provideRoutes(routerConfig)]
      },
      TestModule
    ],
    providers: [
      mockSpotifyService.getProviders(),
      {
        provide: ActivatedRoute,
        useFactory: (r: Router) => r.routerState.root, deps: [ Router ]
      }
    ]
  });
}
```

We start by creating an instance of `MockSpotifyService` that we will use to mock the real implementation of `SpotifyService`.

Next we use a class called `TestBed` and call `configureTestingModule`. `TestBed` is a helper library that ships with Angular to help make testing easier.

In this case, `TestBed.configureTestingModule` is used to configure the `NgModule` used for testing. You can see that we provide an `NgModule` configuration as the argument which has:

- `imports` and
- `providers`

In our `imports` we're importing

- The `RouterTestingModule` and configuring it with our `routerConfig` - this configures the routes for testing
- The `TestModule` - which is the `NgModule` which declares all of the components we will test (see `MusicTestHelpers.ts` for the full details)

In `providers`

- We provide the `MockSpotifyService` (via `mockSpotifyService.getProviders()`)
- and the `ActivatedRoute`

Let's take a closer look at these starting with the `Router`.

#### Router

One thing we haven't talked about yet is what routes we want to use when testing. There are many different ways of doing this. First we'll look at what we're doing here:

**code/routes/music/src/app/test/test.module.ts**

```
@Component({
  selector: 'blank-cmp',
  template: ``
})
export class BlankCmp {
}

@Component({
  selector: 'root-cmp',
  template: `<router-outlet></router-outlet>`
})
export class RootCmp {
}

export const routerConfig: Routes = [
  { path: '', component: BlankCmp },
  { path: 'search', component: SearchComponent },
  { path: 'artists/:id', component: ArtistComponent },
  { path: 'tracks/:id', component: TrackComponent },
  { path: 'albums/:id', component: AlbumComponent }
];
```

Here instead of redirecting (like we do in the real router config) for the empty URL, we're just using `BlankCmp`.

Of course, if you want to use the same `RouterConfig` as in your top-level app then all you need to do is `export` it somewhere and `import` it here.

If you have a more complex scenario where you need to test lots of different route configurations, you could even accept a parameter to the `musicTestProviders` function where you use a new router configuration each time.

There are many possibilities here and you'll need to pick whichever fits best for your team. This configuration works for cases where your routes are relatively static and one configuration works for all of the tests.

Now that we have all of the dependencies, we create the `new Router` and call `r.initialNavigation()` on it.

#### `ActivatedRoute`

The `ActivatedRoute` service keeps track of the "current route". It requires the `Router` itself as a dependency so we put it in `deps` and inject it.

**`MockSpotifyService`**

Earlier we tested our `SpotifyService` by mocking out the HTTP library that backed it. Instead here, we're going to **mock out the whole service itself**. Let's look at how we can mock out this, or any, service.

## Mocking dependencies

If you look inside `music/test` you'll find a `mocks/spotify.ts` file. Let's take a look:

**code/routes/music/src/app/test/spotify.service.mock.ts**

```
import {SpyObject} from './test.helpers';
import {SpotifyService} from '../spotify.service';

export class MockSpotifyService extends SpyObject {
  getAlbumSpy;
  getArtistSpy;
  getTrackSpy;
  searchTrackSpy;
  mockObservable;
  fakeResponse;
```

Here we're declaring the `MockSpotifyService` class, which will be a mocked version of the real `SpotifyService`. These instance variables will act as *spies*.

## Spies

A *spy* is a specific type of mock object that gives us two benefits:

1. we can simulate return values and
2. count how many times the method was called and with which parameters.

In order to use spies with Angular, we're using the internal `SpyObject` class (it's used by Angular to test itself).

You can either declare a class by creating a new `SpyObject` on the fly or you can make your mock class inherit from `SpyObject`, like we're doing in our code.

The great thing inheriting or using this class gives us is the `spy` method. The `spy` method lets us override a method and force a return value (as well as watch and ensure the method was called). We use `spy` on our class constructor:

**code/routes/music/src/app/test/spotify.service.mock.ts**

```
  constructor() {
    super(SpotifyService);

    this.fakeResponse = null;
    this.getAlbumSpy = this.spy('getAlbum').and.returnValue(this);
    this.getArtistSpy = this.spy('getArtist').and.returnValue(this);
    this.getTrackSpy = this.spy('getTrack').and.returnValue(this);
    this.searchTrackSpy = this.spy('searchTrack').and.returnValue(this);
  }
```

The first line of the constructor call's the `SpyObject` constructor, passing the concrete class we're mocking. Calling `super(...)` is optional, but when you do the mock class will inherit all the concrete class methods, so you can override just the pieces you're testing.

> If you're curious about how `SpyObject` is implemented you can check it on the `angular/angular` repository, on the file `/modules/angular2/src/testing/testing_-internal.ts`[138]

After calling `super`, we're intializing the `fakeResponse` field, that we'll use later to `null`.

Next we declare spies that will replace the concrete class methods. Having a reference to them will be helpful to set expectations and simulate responses while writing our tests.

When we use the `SpotifyService` within the `ArtistComponent`, the real `getArtist` method returns an `Observable` and the method we're calling from our components is the subscribe method:

**code/routes/music/src/app/artist/artist.component.ts**

```
  ngOnInit(): void {
    this.spotify
      .getArtist(this.id)
      .subscribe((res: any) => this.renderArtist(res));
  }
```

However, in our mock service, we're going to do something tricky: instead of returning an observable from `getArtist`, we're returning `this`, the `MockSpotifyService` itself. That means the return value of `this.spotify.getArtist(this.id)` above will be the `MockSpotifyService`.

There's one problem with doing this though: our `ArtistComponent` was expecting to call `subscribe` on an Observable. To account for this, we're going to define `subscribe` on our `MockSpotifyService`:

[138] https://github.com/angular/angular/blob/b0cebdba6b65c1e9e7eb5bf801ea42dc7c4a7f25/modules/angular2/src/testing/testing_internal.ts#L205

**code/routes/music/src/app/test/spotify.service.mock.ts**

```
  subscribe(callback) {
    callback(this.fakeResponse);
  }
```

Now when `subscribe` is called on our mock, we're immediately calling the callback, making the async call happen synchronously.

The other thing you'll notice is that we're calling the callback function with `this.fakeResponse`. This leads us to the next method:

**code/routes/music/src/app/test/spotify.service.mock.ts**

```
  setResponse(json: any): void {
    this.fakeResponse = json;
  }
```

This method doesn't replace anything on the concrete service, but is instead a helper method to allow the test code to set a given response (that would come from the service on the concrete class) and with that simulate different responses.

**code/routes/music/src/app/test/spotify.service.mock.ts**

```
  getProviders(): Array<any> {
    return [{ provide: SpotifyService, useValue: this }];
  }
```

This last method is a helper method to be used in `TestBed.configureTestingModule` `providers` like we'll see later when we get back to writing component tests.

Here's what our `MockSpotifyService` looks like altogether:

**code/routes/music/src/app/test/spotify.service.mock.ts**

```
import {SpyObject} from './test.helpers';
import {SpotifyService} from '../spotify.service';

export class MockSpotifyService extends SpyObject {
  getAlbumSpy;
  getArtistSpy;
  getTrackSpy;
  searchTrackSpy;
  mockObservable;
  fakeResponse;

  constructor() {
    super(SpotifyService);

    this.fakeResponse = null;
    this.getAlbumSpy = this.spy('getAlbum').and.returnValue(this);
    this.getArtistSpy = this.spy('getArtist').and.returnValue(this);
    this.getTrackSpy = this.spy('getTrack').and.returnValue(this);
    this.searchTrackSpy = this.spy('searchTrack').and.returnValue(this);
  }

  subscribe(callback) {
    callback(this.fakeResponse);
  }

  setResponse(json: any): void {
    this.fakeResponse = json;
  }

  getProviders(): Array<any> {
    return [{ provide: SpotifyService, useValue: this }];
  }
}
```

## Back to Testing Code

Now that we have all our dependencies under control, it is easier to write our tests. Let's write our test for our `ArtistComponent`.

As usual, we start with imports:

**code/routes/music/src/app/artist/artist.component.spec.ts**

```
import {
  async,
  ComponentFixture,
  TestBed,
  inject,
  fakeAsync,
} from '@angular/core/testing';
import { Router } from '@angular/router';
import { Location } from '@angular/common';
import {
  advance,
  createRoot,
  RootCmp,
  configureMusicTests
} from '../test/test.module';

import { MockSpotifyService } from '../test/spotify.service.mock';
import { SpotifyService } from '../spotify.service';
import { ArtistComponent } from './artist.component';
```

Next, before we can start to describe our tests `configureMusicTests` to ensure we can access our `musicTestProviders` in each test:

**code/routes/music/src/app/artist/artist.component.spec.ts**

```
describe('ArtistComponent', () => {
  beforeEach(async(() => {
    configureMusicTests();
  }));
```

Next, we'll write a test for everything that happens during the initialization of the component. First, let's take a refresh look at what happens on initialization of our `ArtistComponent`:

**code/routes/music/src/app/artist/artist.component.ts**

```
export class ArtistComponent implements OnInit {
  id: string;
  artist: Object;

  constructor(private route: ActivatedRoute, private spotify: SpotifyService,
              private location: Location) {
    route.params.subscribe(params => { this.id = params['id']; });
  }

  ngOnInit(): void {
    this.spotify
      .getArtist(this.id)
      .subscribe((res: any) => this.renderArtist(res));
  }
```

Remember that during the creation of the component, we use `route.params` to retrieve the current route `id` param and store it on the `id` attribute of the class.

When the component is initialized `ngOnInit` is triggered by Angular (because we declared that this component `implements OnInit`. We then use the `SpotifyService` to retrieve the artist for the received `id`, and we subscribe to the returned `observable`. When the artist is finally retrieved, we call renderArtist, passing the artist data.

An important idea here is that we used dependency injection to get the `SpotifyService`, but remember, **we created a `MockSpotifyService`**!

So in order to test this behavior, let's:

1. Use our router to navigate to the `ArtistComponent`, which will initialize the component
2. Check our `MockSpotifyService` and ensure that the `ArtistComponent` did, indeed, try to get the artist with the appropriate id.

Here's the code for our test:

**code/routes/music/src/app/artist/artist.component.spec.ts**

```
  describe('initialization', () => {
    it('retrieves the artist', fakeAsync(
      inject([Router, SpotifyService],
             (router: Router,
              mockSpotifyService: MockSpotifyService) => {
        const fixture = createRoot(router, RootCmp);

        router.navigateByUrl('/artists/2');
        advance(fixture);

        expect(mockSpotifyService.getArtistSpy).toHaveBeenCalledWith('2');
      })));
  });
```

Let's take it step by step.

### `fakeAsync` and `advance`

We start by wrapping the test in `fakeAsync`. Without getting too bogged down in the details, by using `fakeAsync` we're able to have more control over when change detection and asynchronous operations occur. A consequence of this is that we need to explicitly tell our components that they need to detect changes after we make changes in our tests.

Normally you don't need to worry about this when writing your apps, as zones tend to do the right thing, but during tests we manipulate the change detection process more carefully.

If you skip a few lines down you'll notice that we're using a function called `advance` that comes from our `MusicTestHelpers`. Let's take a look at that function:

**code/routes/music/src/app/test/test.module.ts**

```
export function advance(fixture: ComponentFixture<any>): void {
  tick();
  fixture.detectChanges();
}
```

So we see here that `advance` does two things:

1. It tells the component to detect changes and
2. Calls `tick()`

When we use `fakeAsync`, timers are actually synchronous and we use `tick()` to simulate the asynchronous passage of time.

Practically speaking, in our tests we'll call `advance` whenever we want Angular to "work it's magic". So for instance, whenever we navigate to a new route, update a form element, make an HTTP request etc. we'll call `advance` to give Angular a chance to do it's thing.

### `inject`

In our test we need some dependencies. We use `inject` to get them. The `inject` function takes two arguments:

1. An array of *tokens* to inject
2. A function into which to provide the injections

And what classes will `inject` use? The providers we defined in `TestBed.configureTestingModule` `providers`.

Notice that we're injecting:

1. `Router`
2. `SpotifyService`

The `Router` that will be injected is the `Router` we configured in `musicTestProviders` above.

For `SpotifyService`, notice that we're requesting injection of the *token* `SpotifyService`, but we're receiving a `MockSpotifyService`. A little tricky, but hopefully it makes sense given what we've talked about so far.

## Testing `ArtistComponent`'s Initialization

Let's review the contents of our actual test:

**code/routes/music/src/app/artist/artist.component.spec.ts**

```
      const fixture = createRoot(router, RootCmp);

      router.navigateByUrl('/artists/2');
      advance(fixture);

      expect(mockSpotifyService.getArtistSpy).toHaveBeenCalledWith('2');
```

We start by creating an instance of our `RootCmp` by using `createRoot`. Let's look at the `createRoot` helper function:

**code/routes/music/src/app/test/test.module.ts**

```
export function createRoot(router: Router,
                           componentType: any): ComponentFixture<any> {
  const f = TestBed.createComponent(componentType);
  advance(f);
  (<any>router).initialNavigation();
  advance(f);
  return f;
}
```

Notice here that when we call `createRoot` we

1. Create an instance of the root component
2. `advance` it
3. Tell the router to setup it's `initialNavigation`
4. `advance` again
5. return the new root component.

This is something we'll do a lot when we want to test a component that depends on routing, so it's handy to have this helper function around.

Notice that we're using the `TestBed` library again to call `TestBed.createComponent`. This function creates a component of the appropriate type.

`RootCmp` is an empty component that we created in `MusicTestHelpers`. You definitely don't need to create an empty component for your root component, but I like to do it this way because it lets us test our child component (`ArtistComponent`) more-or-less in isolation. That is, we don't have to worry about the effects of the parent app component.

That said, maybe you *want* to make sure that the child component operates correctly in context. In that case instead of using `RootCmp` you'd probably want to use your app's normal parent component.

Next we use `router` to navigate to the url `/artists/2` and `advance`. When we navigate to that URL, `ArtistComponent` should be initialized, so we assert that the `getArtist` method of the `SpotifyService` was called with the proper value.

## Testing `ArtistComponent` Methods

Recall that the `ArtistComponent` has an `href` which calls the `back()` function.

**code/routes/music/src/app/artist/artist.component.ts**

```
  back(): void {
    this.location.back();
  }
```

Let's test that when the `back` method is called, the router will redirect the user back to the previous location.

The current location state is controlled by the `Location` service. When we need to send the user back to the previous location, we use the `Location`'s `back` method.

Here is how we test the `back` method:

**code/routes/music/src/app/artist/artist.component.spec.ts**

```
  describe('back', () => {
    it('returns to the previous location', fakeAsync(
      inject([Router, Location],
             (router: Router, location: Location) => {
        const fixture = createRoot(router, RootCmp);
        expect(location.path()).toEqual('/');

        router.navigateByUrl('/artists/2');
        advance(fixture);
        expect(location.path()).toEqual('/artists/2');

        const artist = fixture.debugElement.children[1].componentInstance;
        artist.back();
        advance(fixture);

        expect(location.path()).toEqual('/');
      })));
  });
```

The initial structure is similar: we inject our dependencies and create a new component.

We have a new `expectation` - we assert that the `location.path()` is equal to what we expect it to be.

We also have another new idea: we're accessing the methods on the `ArtistComponent` itself. We get a reference to our `ArtistComponent` instance through the line

`fixture.debugElement.children[1].componentInstance`.

Now that we have the instance of the component, we're able to call methods on it directly, like `back()`.

After we call `back()` we `advance` and then verify that the `location.path()` is what we `expected` it to be.

## Testing `ArtistComponent` DOM Template Values

The last thing we need to test on `ArtistComponent` is the template that renders the artist.

**code/routes/music/src/app/artist/artist.component.html**

```
<div *ngIf="artist">
  <h1>{{ artist.name }}</h1>

  <p>
    <img src="{{ artist.images[0].url }}">
  </p>

  <p><a href (click)="back()">Back</a></p>
</div>
```

Remember that the instance variable `artist` is set by the result of the `SpotifyService getArtist` call. Since we're mocking the `SpotifyService` with `MockSpotifyService`, the data we should have in our template should be whatever the `mockSpotifyService` returns. Let's look at how we do this:

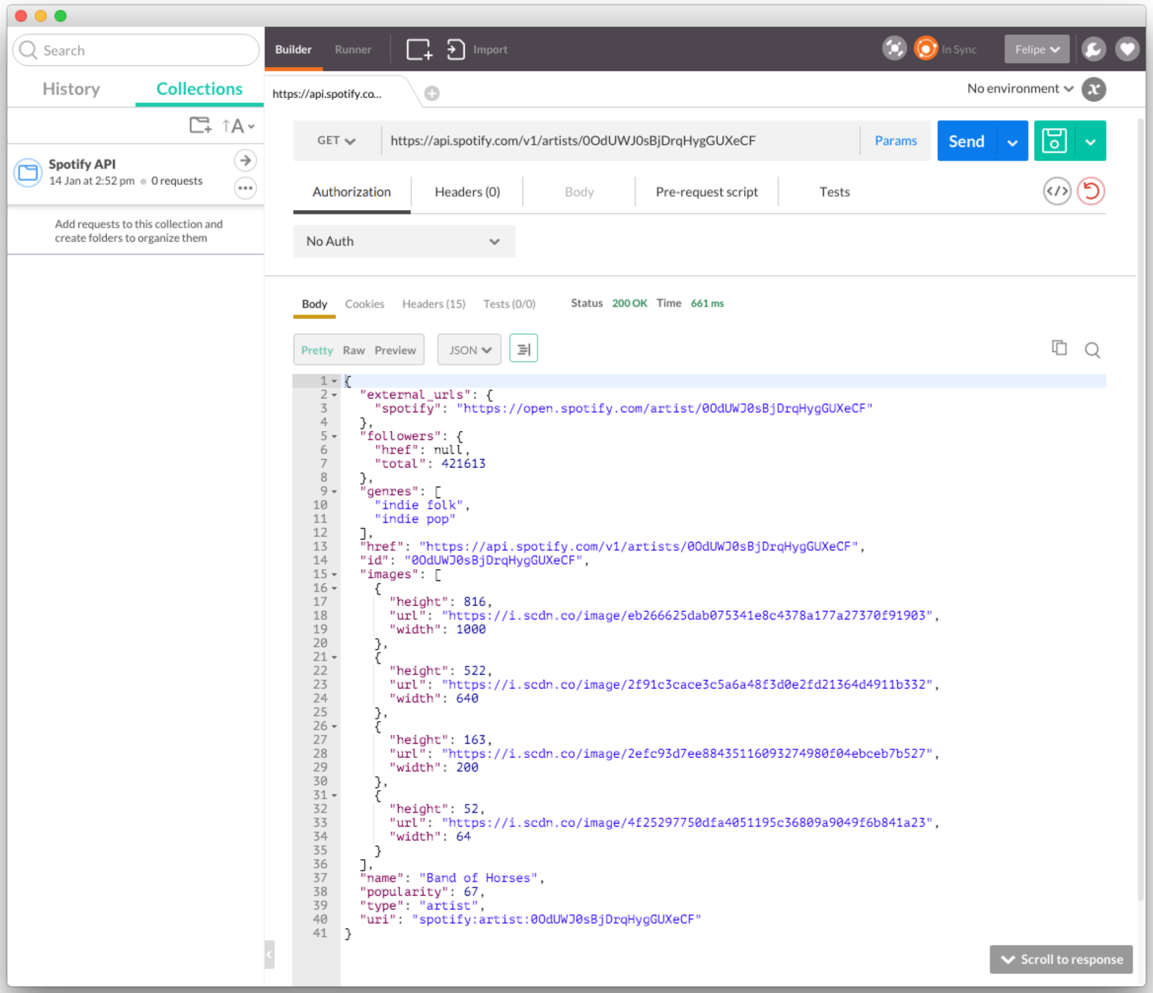
**code/routes/music/src/app/artist/artist.component.spec.ts**

```
  describe('renderArtist', () => {
    it('renders album info', fakeAsync(
      inject([Router, SpotifyService],
             (router: Router,
              mockSpotifyService: MockSpotifyService) => {
        const fixture = createRoot(router, RootCmp);

        const artist = {name: 'ARTIST NAME', images: [{url: 'IMAGE_1'}]};
        mockSpotifyService.setResponse(artist);

        router.navigateByUrl('/artists/2');
        advance(fixture);

        const compiled = fixture.debugElement.nativeElement;

        expect(compiled.querySelector('h1').innerHTML).toContain('ARTIST NAME');
        expect(compiled.querySelector('img').src).toContain('IMAGE_1');
      })));
  });
```

The first thing that's new here is that we're *manually setting the response* of the `mockSpotifyService` with `setResponse`.

The `artist` variable is a *fixture* that represents what we get from the Spotify API when we call the `artists` endpoint at `GET https://api.spotify.com/v1/artists/{id}`.

Here's what the real JSON looks like:



```
https://api.spotify.com/v1/artists/0OdUWJ0sBjDrqHygGUXeCF
```

```
{
  "external_urls": {
    "spotify": "https://open.spotify.com/artist/0OdUWJ0sBjDrqHygGUXeCF"
  },
  "followers": {
    "href": null,
    "total": 421613
  },
  "genres": [
    "indie folk",
    "indie pop"
  ],
  "href": "https://api.spotify.com/v1/artists/0OdUWJ0sBjDrqHygGUXeCF",
  "id": "0OdUWJ0sBjDrqHygGUXeCF",
  "images": [
    {
      "height": 816,
      "url": "https://i.scdn.co/image/eb266625dab075341e8c4378a177a27370f91903",
      "width": 1000
    },
    {
      "height": 522,
      "url": "https://i.scdn.co/image/2f91c3cace3c5a6a48f3d0e2fd21364d4911b332",
      "width": 640
    },
    {
      "height": 163,
      "url": "https://i.scdn.co/image/2efc93d7ee88435116093274980f04ebceb7b527",
      "width": 200
    },
    {
      "height": 52,
      "url": "https://i.scdn.co/image/4f25297750dfa4051195c36809a9049f6b841a23",
      "width": 64
    }
  ],
  "name": "Band of Horses",
  "popularity": 67,
  "type": "artist",
  "uri": "spotify:artist:0OdUWJ0sBjDrqHygGUXeCF"
}
```

**Postman - Spotify Get Artist Endpoint**

However, for this test we need only the `name` and `images` properties.

When we call the `setResponse` method, that response will be used for the next call we make to any of the service methods. In this case, we want the method `getArtist` to return this response.

Next we navigate with the router and `advance`. Now that the view is rendered, we can use the DOM representation of the component's view to check if the artist was properly rendered.

We do that by getting the `nativeElement` property of the `DebugElement` with the line `fixture.debugElement.nativeElement`.

In our assertions, we expect to see H1 tag containing the artist's name, in our case the string `ARTIST NAME` (because of our `artist` fixture above).

To check those conditions, we use the `NativeElement`'s `querySelector` method. This method will return the first element that matches the provided CSS selector.

For the H1 we check that the text is indeed `ARTIST NAME` and for the image, we check its `src` property is `IMAGE 1`.

With this, we are done testing the `ArtistComponent` class.

## Testing Forms

To write form tests, let's use the `DemoFormWithEventsComponent` component we created back in the `Forms` chapter. This example is a good candidate because it uses a few features of Angular's forms:

- it uses a `FormBuilder`
- has validations
- handles events

As a reminder, here's the full code for that class:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.ts**

```
import { Component, OnInit } from '@angular/core';
import {
  FormBuilder,
  FormGroup,
  Validators,
  AbstractControl
} from '@angular/forms';

@Component({
  selector: 'app-demo-form-with-events',
  templateUrl: './demo-form-with-events.component.html',
  styles: []
})
export class DemoFormWithEventsComponent implements OnInit {
  myForm: FormGroup;
  sku: AbstractControl;

  ngOnInit() {
  }

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];

    this.sku.valueChanges.subscribe(
      (value: string) => {
        console.log('sku changed to:', value);
      }
    );

    this.myForm.valueChanges.subscribe(
      (form: any) => {
        console.log('form changed to:', form);
      }
    );

  }

  onSubmit(form: any): void {
    console.log('you submitted value:', form.sku);
  }

}
```

And the template:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.html**

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: with events</h2>
  <form [formGroup]="myForm"
        (ngSubmit)="onSubmit(myForm.value)"
        class="ui form">

    <div class="field"
        [class.error]="!sku.valid && sku.touched">
      <label for="skuInput">SKU</label>
      <input type="text"
             class="form-control"
             id="skuInput"
             placeholder="SKU"
             [formControl]="sku">
       <div *ngIf="!sku.valid"
```

```
          class="ui error message">SKU is invalid</div>
        <div *ngIf="sku.hasError('required')"
          class="ui error message">SKU is required</div>
    </div>

    <div *ngIf="!myForm.valid"
      class="ui error message">Form is invalid</div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

Just to recap, this code will have the following behavior:

- when no value is present for the SKU field, two validation error will be displayed: *SKU is invalid* and *SKU is required*
- when the value of the SKU field changes, we are logging a message to the console
- when the form changes, we are also logging to the console
- when the form is submitted, we log yet another final message to the console

It seems that one obvious external dependency we have is the console. As we learned before, we need to somehow mock all external dependencies.

## Creating a `ConsoleSpy`

This time, instead of using a `SpyObject` to create a mock, let's do something simpler, since all we're using from the `console` is the `log` method.

We will replace the original `console` instance, that is held on the `window.console` object and replace by an object we control: a `ConsoleSpy`.

**code/forms/src/app/utils.ts**

```
export class ConsoleSpy {
  public logs: string[] = [];
  log(...args) {
    this.logs.push(args.join(' '));
  }
  warn(...args) {
    this.log(...args);
  }
}
```

The `ConsoleSpy` is an object that will take whatever is `logged`, naively convert it to a string, and store it in an internal list of things that were logged.

To accept a variable number of arguments on our version of the `console.log` method, we are using ES6 and TypeScript's *Rest parameters*[139].

This operator, represented by an ellipsis, like `...theArgs` as our function argument. In a nutshell using it indicates that we're going to capture all the remaining arguments from that point on. If we had something like `(a, b, ...theArgs)` and called `func(1, 2, 3, 4, 5)`, `a` would be `1`, `b` would be `2` and `theArgs` would have `[3, 4, 5]`.

You can play with it yourself if you have a recent version of Node.js[140] installed:

```
$ node --harmony
> var test = (a, b, ...theArgs) => console.log('a=',a,'b=',b,'theArgs=',theArgs);
undefined
> test(1,2,3,4,5);
a= 1 b= 2 theArgs= [ 3, 4, 5 ]
```

So instead of writing it to the console itself, we'll be storing them on an array. If the code under test calls `console.log` three times:

```
console.log('First message', 'is', 123);
console.log('Second message');
console.log('Third message');
```

We expect the `_logs` field to have an array of `['First message is 123', 'Second message', 'Third message']`.

## Installing the `ConsoleSpy`

To use our spy in our test we start by declaring two variables: `originalConsole` will keep a reference to the original console instance, and `fakeConsole` that will hold the *mocked* version of the console. We also declare a few variables that will be helpful in testing our `input` and `form` elements.

[139]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/rest_parameters
[140]https://nodejs.org/en/

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

```
describe('DemoFormWithEventsComponent', () => {
  let component: DemoFormWithEventsComponent;
  let fixture: ComponentFixture<DemoFormWithEventsComponent>;

  let originalConsole, fakeConsole;
  let el, input, form;
```

And then we can install the fake console and specify our providers:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

```
  beforeEach(async(() => {
    // replace the real window.console with our spy
    fakeConsole = new ConsoleSpy();
    originalConsole = window.console;
    (<any>window).console = fakeConsole;

    TestBed.configureTestingModule({
      imports: [ FormsModule, ReactiveFormsModule ],
      declarations: [ DemoFormWithEventsComponent ]
    })
    .compileComponents();
  }));
```

Back to the testing code, the next thing we need to do is replace the real console instance with ours, saving the original instance.

Finally, on the `afterAll` method, we restore the original console instance to make sure it doesn't *leak* into other tests.

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

```
  // restores the real console
  afterAll(() => (<any>window).console = originalConsole);
```

## Configuring the Testing Module

Notice that in the `beforeEach` we call `TestBed.configureTestingModule` - remember that `configureTestingModule` sets up the root `NgModule` for our tests.

In this case we're importing the two forms modules and declaring the `DemoFormWithEvents` component.

Now that we have control of the console, let's begin testing our form.

## Testing The Form

Now we need to test the validation errors and the events of the form.

The first thing we need to do is to get the references to the SKU input field and to the form elements:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts**

```
  it('validates and triggers events', fakeAsync( () => {
    fixture = TestBed.createComponent(DemoFormWithEventsComponent);
    component = fixture.componentInstance;
    el = fixture.debugElement.nativeElement;
    input = fixture.debugElement.query(By.css('input')).nativeElement;
    form = fixture.debugElement.query(By.css('form')).nativeElement;
    fixture.detectChanges();
```

The last line tells Angular to commit all the pending changes, similar to what we did in the routing section above. Next, we will set the SKU input value to the empty string:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts**

```
    input.value = '';
    dispatchEvent(input, 'input');
    fixture.detectChanges();
    tick();
```

Here we use `dispatchEvent` to notify Angular that the input element changed, and then we trigger the change detection a second time. Finally we use `tick()` to make sure all asynchronous code triggered up to this point gets executed.

The reason we are using `fakeAsync` and `tick` on this test, is to assure the form events are triggered. If we used `async` and `inject` instead, we would finish the code before the events were triggered.

Now that we have changed the input value, let's make sure the validation is working. We ask the component element (using the `el` variable) for all child elements that are error messages and then making sure we have both error messages displayed:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts**

```
    let msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs[0].innerHTML).toContain('SKU is invalid');
    expect(msgs[1].innerHTML).toContain('SKU is required');
```

Next, we will do something similar, but this time we set a value to the SKU field:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts**

```
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    fixture.detectChanges();
    tick();
```

And make sure all the error messages are gone:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts**

```
    msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs.length).toEqual(0);
```

Finally, we will trigger the submit event of the form:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts**

```
    fixture.detectChanges();
    dispatchEvent(form, 'submit');
    tick();
```

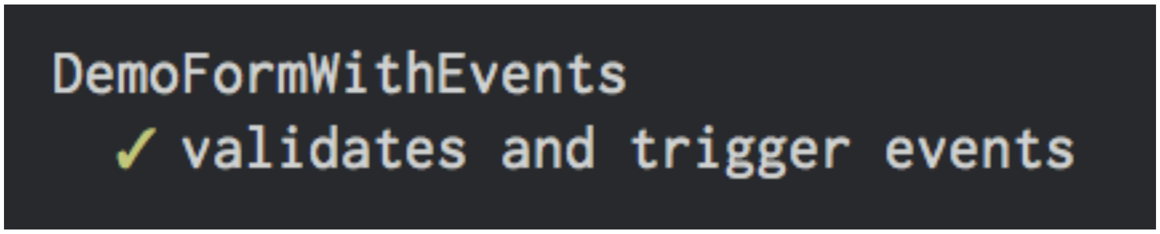
And finally we make sure the event was kicked by checking that the message we log to the console when the form is submitted is there:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts**

```
    // checks for the form submitted message
    expect(fakeConsole.logs).toContain('you submitted value: XYZ');
```

We could continue and add new verifications for the other two events our form triggers: the SKU change and the form change events. However, our test is growing quite long.

When we run our tests, we see it passes:



```
DemoFormWithEvents
  ✓ validates and trigger events
```

**DemoFormWithEvents test output**

This test works, but stylistically we have some code smells:

- a really long `it` condition (more than 5-10 lines)
- more than one or two `expects` per `it` condition
- the word **and** on the test description

## Refactoring Our Form Test

Let’s fix that by first extracting the code that creates the component and gets the component element and also the elements for the input and for the form:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

```
    fixture = TestBed.createComponent(DemoFormWithEventsComponent);
```

The `createComponent` code is pretty straightforward: Creates the component with `TestBed.createComponent`, retrieves all the elements we need and calls `detectChanges`.

Now the first thing we want to test is that given an empty SKU field, we should see two error messages:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

```
  it('displays errors with no sku', fakeAsync( () => {
    input.value = '';
    dispatchEvent(input, 'input');
    fixture.detectChanges();

    // no value on sku field, all error messages are displayed
    const msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs[0].innerHTML).toContain('SKU is invalid');
    expect(msgs[1].innerHTML).toContain('SKU is required');
  }));
```

See how much cleaner this is? Our test is focused and tests only one thing. Great job!

This new structure makes adding the second test easy. This time we want to test that, once we add a value to the SKU field, the error messages are gone:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

```
  it('displays no errors when sku has a value', fakeAsync( () => {
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    fixture.detectChanges();

    const msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs.length).toEqual(0);
  }));
```

One thing you may have noticed is that so far, our tests are not using `fakeAsync`, but `async` plus `inject` instead.

That's another bonus of this refactoring: we will only use `fakeAsync` and `tick()` when we want to check if something was added to the console, because that's all our form's event handlers do.

The next test will do exactly that - when the SKU value changes, we should have a message logged to the console:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

```
  it('handles sku value changes', fakeAsync( () => {
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    tick();

    expect(fakeConsole.logs).toContain('sku changed to: XYZ');
  }));
```

We can write similar code for both the form change…

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

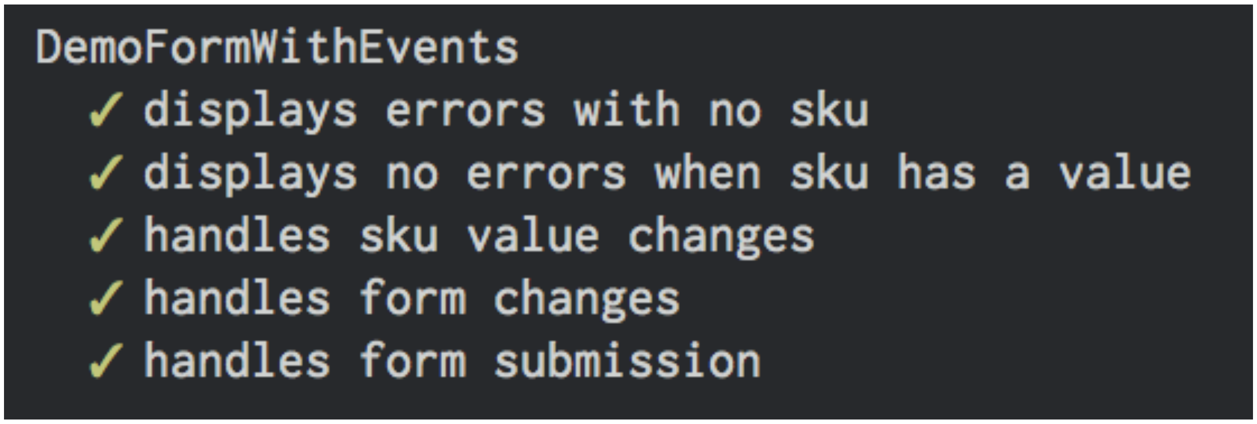
```
  it('handles form changes', fakeAsync(() => {
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    tick();

    expect(fakeConsole.logs).toContain('form changed to: [object Object]');
  }));
```

… and the form submission events:

**code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts**

```
  it('handles form submission', fakeAsync((tcb) => {
    input.value = 'ABC';
    dispatchEvent(input, 'input');
    tick();

    fixture.detectChanges();
    dispatchEvent(form, 'submit');
    tick();

    expect(fakeConsole.logs).toContain('you submitted value: ABC');
  }));
```

When we run the tests now, we get a much nicer output:



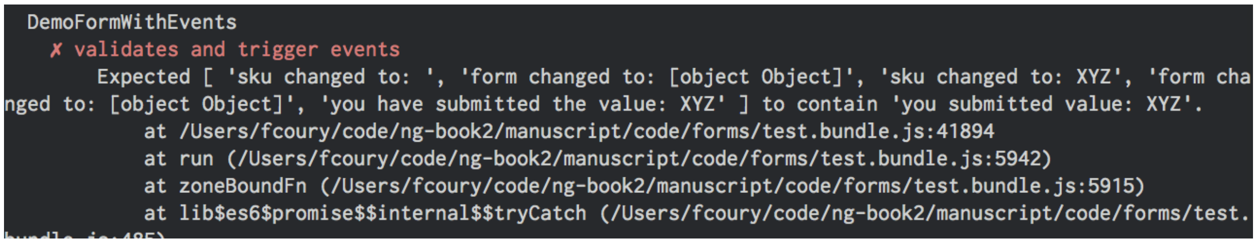
```
DemoFormWithEvents
  ✓ displays errors with no sku
  ✓ displays no errors when sku has a value
  ✓ handles sku value changes
  ✓ handles form changes
  ✓ handles form submission
```

**DemoFormWithEvents test output after refactoring**

Another great benefit from this refactor can be seen when something goes wrong. Let's go back to the component code and change the message when the form gets submitted, in order to force one of our tests to fail:

```
onSubmit(form: any): void {
  console.log('you have submitted the value:', form.sku);
}
```

If we ran the previous version of the test, here's what would happen:

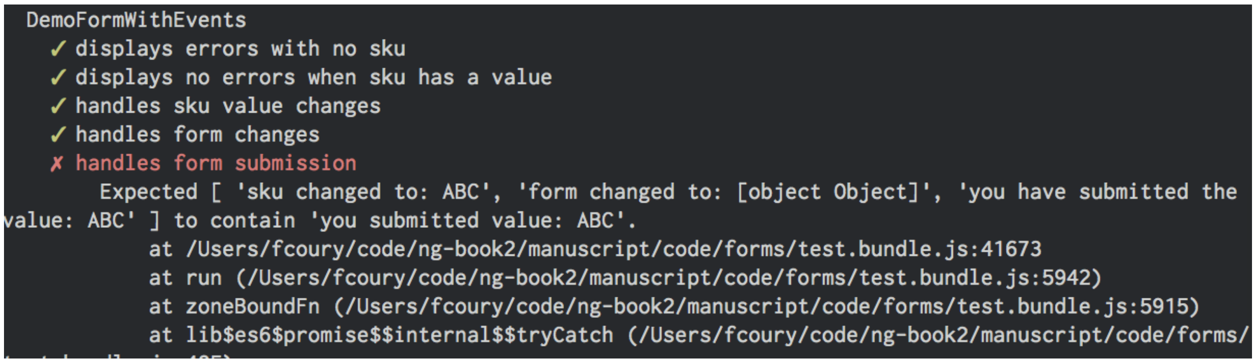


```
 DemoFormWithEvents
   ✗ validates and trigger events
       Expected [ 'sku changed to: ', 'form changed to: [object Object]', 'sku changed to: XYZ', 'form cha
nged to: [object Object]', 'you have submitted the value: XYZ' ] to contain 'you submitted value: XYZ'.
           at /Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:41894
           at run (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:5942)
           at zoneBoundFn (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:5915)
           at lib$es6$promise$$internal$$tryCatch (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.
```

**DemoFormWithEvents error output before refactoring**

It's not immediately obvious what failed. We have to read the error code to realize it was the submission message that failed. We also can't be sure if that was the only thing that broke on the component code, since we may have other test conditions after the one that failed that never had a chance to be executed.

Now, compare that to the error we get from our refactored code:



```
 DemoFormWithEvents
   ✓ displays errors with no sku
   ✓ displays no errors when sku has a value
   ✓ handles sku value changes
   ✓ handles form changes
   ✗ handles form submission
        Expected [ 'sku changed to: ABC', 'form changed to: [object Object]', 'you have submitted the
value: ABC' ] to contain 'you submitted value: ABC'.
            at /Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:41673
            at run (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:5942)
            at zoneBoundFn (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:5915)
            at lib$es6$promise$$internal$$tryCatch (/Users/fcoury/code/ng-book2/manuscript/code/forms/
```

**DemoFormWithEvents error output after refactoring**

This version makes it pretty obvious that the only thing that failed was the form submission event.

## Testing HTTP requests

We could test the HTTP interaction in our apps using the same strategy as we used so far: write a mock version of the `Http` class, since it is an external dependency.

But since the vast majority of single page apps written using frameworks like Angular use HTTP interaction to talk to APIs, the Angular testing library already provides a built in alternative: `MockBackend`.

We have used this class before in this chapter when we were testing the `SpotifyService` class.

Let's dive a little deeper now and see some more testing scenarios and also some good practices. In order to do this, let's write tests for the examples from the *HTTP chapter*.

First, let's see how we test different HTTP methods, like POST or DELETE and how to test the correct HTTP headers are being sent.

Back on the HTTP chapter, we created this example that covered how to do those things using `Http`.

### Testing a `POST`

The first test we'll write is to make sure we're doing a proper POST request on the `makePost` method:

**code/http/src/app/more-http-requests/more-http-requests.component.ts**

```
makePost(): void {
  this.loading = true;
  this.http.post(
    'http://jsonplaceholder.typicode.com/posts',
    JSON.stringify({
      body: 'bar',
      title: 'foo',
      userId: 1
    }))
    .subscribe((res: Response) => {
      this.data = res.json();
      this.loading = false;
    });
}
```

When writing our test for this method, our goal is to test two things:

1. the request method (`POST`) is correct and that
2. the URL we're hitting is also correct.

Here's how we turn that into a test:

**code/src/app/more-http-requests/more-http-requests.component.spec.ts**

```
it('performs a POST', async(() => {
  backend.connections.subscribe(c => {
    expect(c.request.url)
    .toBe('http://jsonplaceholder.typicode.com/posts');
    expect(c.request.method).toBe(RequestMethod.Post);
    c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));
  });
  component.makePost();
  expect(component.data).toEqual({'response': 'OK'});
}));
```

Notice how we have a `subscribe` call to `backend.connections`. This will trigger our code whenever a new HTTP connection is established, giving us an opportunity to peek into the request and also provide the response we want.

This place is where you can:

- add request assertions, like checking the correct URL or HTTP method was requested
- set a mocked response, to force your code to deal with different responses, given different test scenarios

Angular uses an `enum` called `RequestMethod` to identify HTTP methods. Here are the supported methods:

```
export enum RequestMethod {
  Get,
  Post,
  Put,
  Delete,
  Options,
  Head,
  Patch
}
```

Finally, after the call `makePost()` we're doing another check to make sure that the mock response we set was the one that was assigned to our component.

Now that we understand how this work, adding a second test for a DELETE method is easy.

## Testing `DELETE`

Here's how the `makeDelete` method is implemented:

**code/http/src/app/more-http-requests/more-http-requests.component.ts**

```
  makeDelete(): void {
    this.loading = true;
    this.http.delete('http://jsonplaceholder.typicode.com/posts/1')
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
  }
```

And this is the code we use to test it:

**src/app/more-http-requests/more-http-requests.component.spec.ts**

```
  it('performs a DELETE', async(() => {
    backend.connections.subscribe(c => {
      expect(c.request.url)
      .toBe('http://jsonplaceholder.typicode.com/posts/1');
      expect(c.request.method).toBe(RequestMethod.Delete);
      c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));
    });

    component.makeDelete();
    expect(component.data).toEqual({'response': 'OK'});
  }));
```

Everything here is the same, except for the URL that changes a bit and the HTTP method, which is now `RequestMethod.Delete`.

## Testing HTTP Headers

The last method we have to test on this class is `makeHeaders`:

**code/http/src/app/more-http-requests/more-http-requests.component.ts**

```
  makeHeaders(): void {
    const headers: Headers = new Headers();
    headers.append('X-API-TOKEN', 'ng-book');

    const opts: RequestOptions = new RequestOptions();
    opts.headers = headers;

    this.http.get('http://jsonplaceholder.typicode.com/posts/1', opts)
      .subscribe((res: Response) => {
        this.data = res.json();
      });
  }
```

In this case, what our test should focus on is making sure the header `X-API-TOKEN` is being properly set to `ng-book`:

**src/app/more-http-requests/more-http-requests.component.spec.ts**

```
  it('sends correct headers', async(() => {
    backend.connections.subscribe(c => {
      expect(c.request.url)
      .toBe('http://jsonplaceholder.typicode.com/posts/1');
      expect(c.request.headers.has('X-API-TOKEN')).toBeTruthy();
      expect(c.request.headers.get('X-API-TOKEN')).toEqual('ng-book');
      c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));
    });

    component.makeHeaders();
    expect(component.data).toEqual({'response': 'OK'});
  }));
```

The connection's `request.headers` attribute returns a `Headers` class instance and we're using two methods to perform two different assertions:

- the `has` method to check whether a given header was set, ignoring it's value
- the `get` method, that returns the value that was set

If having the header set is sufficient, use `has`. Otherwise, if you need to inspect the set value, use `get`.

And with that we finish the tests of different methods and headers on Angular. Time to move to a more complex example, that will be closer to what you will encounter when coding real world applications.

## Testing `YouTubeSearchService`

The other example we built back on the HTTP chapter was a YouTube video search. The HTTP interaction for that example takes place on a service called `YouTubeSearchService`:

**code/http/src/app/you-tube-search/you-tube-search.service.ts**

```
/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeSearchService {
  constructor(private http: Http,
    @Inject(YOUTUBE_API_KEY) private apiKey: string,
    @Inject(YOUTUBE_API_URL) private apiUrl: string) {
    }

    search(query: string): Observable<SearchResult[]> {
      const params: string = [
        `q=${query}`,
        `key=${this.apiKey}`,
        `part=snippet`,
        `type=video`,
        `maxResults=10`
      ].join('&');
      const queryUrl = `${this.apiUrl}?${params}`;
      return this.http.get(queryUrl)
      .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
          // console.log("raw item", item); // uncomment if you want to debug
          return new SearchResult({
            id: item.id.videoId,
            title: item.snippet.title,
            description: item.snippet.description,
            thumbnailUrl: item.snippet.thumbnails.high.url
          });
        });
      });
    }
  }
```

It uses the YouTube API to search for videos and parse the results into a `SearchResult` instance:

**code/http/src/app/you-tube-search/search-result.model.ts**

```
export class SearchResult {
  id: string;
  title: string;
  description: string;
  thumbnailUrl: string;
  videoUrl: string;

  constructor(obj?: any) {
    this.id              = obj && obj.id             || null;
    this.title           = obj && obj.title          || null;
    this.description     = obj && obj.description    || null;
    this.thumbnailUrl    = obj && obj.thumbnailUrl   || null;
    this.videoUrl        = obj && obj.videoUrl       ||
                             `https://www.youtube.com/watch?v=${this.id}`;
  }
}
```

The important aspects of this service we need to test are that:

- given a JSON response, the service is able to parse the video id, title, description and thumbnail
- the URL we are requesting uses the provided search term
- the URL starts with what is set on the `YOUTUBE_API_URL` constant
- the API key used matches the `YOUTUBE_API_KEY` constant

With that in mind, let's start writing our test:

**code/http/src/app/you-tube-search/you-tube-search.component.before.spec.ts**

```
describe('YouTubeSearchComponent (before)', () => {
  let component: YouTubeSearchComponent;
  let fixture: ComponentFixture<YouTubeSearchComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        YouTubeSearchComponent,
        SearchResultComponent,
        SearchBoxComponent
      ],
      providers: [
        YouTubeSearchService,
        BaseRequestOptions,
        MockBackend,
        { provide: YOUTUBE_API_KEY, useValue: 'YOUTUBE_API_KEY' },
        { provide: YOUTUBE_API_URL, useValue: 'YOUTUBE_API_URL' },
        { provide: Http,
          useFactory: (backend: ConnectionBackend,
                       defaultOptions: BaseRequestOptions) => {
                         return new Http(backend, defaultOptions);
                       }, deps: [MockBackend, BaseRequestOptions] }
      ]
    })
    .compileComponents();
  }));
```

As we did for every test we wrote on this chapter, we start by declaring how we want to setup our dependencies: we're using the real `YouTubeSearchService` instance, but setting fake values for `YOUTUBE_API_KEY` and `YOUTUBE_API_URL` constants. We also setting up the `Http` class to use a `MockBackend`.

Now, let's begin to write our first test case:

**code/http/src/app/you-tube-search/you-tube-search.component.before.spec.ts**

```
  describe('search', () => {
    it('parses YouTube response',
      inject([YouTubeSearchService, MockBackend],
        fakeAsync((service, backend) => {
        let res;

        backend.connections.subscribe(c => {
          c.mockRespond(new Response(<any>{
            body: `
            {
              "items": [
                {
                  "id": { "videoId": "VIDEO_ID" },
                  "snippet": {
                    "title": "TITLE",
                    "description": "DESCRIPTION",
                    "thumbnails": {
                      "high": { "url": "THUMBNAIL_URL" }
                    }
                  }
                }
              ]
            }`
          }));
        });

        service.search('hey').subscribe(_res => {
          res = _res;
        });
        tick();

        const video = res[0];
        expect(video.id).toEqual('VIDEO_ID');
        expect(video.title).toEqual('TITLE');
        expect(video.description).toEqual('DESCRIPTION');
        expect(video.thumbnailUrl).toEqual('THUMBNAIL_URL');
      }))
    );
  });
```

Here we are telling `Http` to return a fake response that will match the relevant fields what we expect the YouTube API to respond when we call the real URL. We do that by using the `mockRespond` method of the connection.

**code/http/src/app/you-tube-search/you-tube-search.component.before.spec.ts**

```
      service.search('hey').subscribe(_res => {
        res = _res;
      });
      tick();
```

Next, we're calling the method we're testing: `search`. We're calling it with the term *hey* and capturing the response on the `res` variable.

If you noticed before, we're using `fakeAsync` that requires us to manually sync asynchronous code by calling `tick()`. When we do that here, we expect that the search finished executing and our `res` variable to have a value.

Now is the time to evaluate that value:

**code/http/src/app/you-tube-search/you-tube-search.component.before.spec.ts**

```
        const video = res[0];
        expect(video.id).toEqual('VIDEO_ID');
        expect(video.title).toEqual('TITLE');
        expect(video.description).toEqual('DESCRIPTION');
        expect(video.thumbnailUrl).toEqual('THUMBNAIL_URL');
      }))
```

We are getting the first element from the list of responses. We know it's a `SearchResult`, so we're now checking that each attribute was set correctly, based on our provided response: the id, title, description and thumbnail URL should all match.

With this, we completed our first goal when writing this test. However, didn't we just say that having a huge `it` method and having too many `expects` are testing code smells?

We did, so before we continue let's refactor this code to make isolated assertions easier.

Add the following helper fuction inside our `describe('search', ...)`:

**code/http/src/app/you-tube-search/you-tube-search.component.spec.ts**

```
    function search(term: string, response: any, callback) {
      return inject([YouTubeSearchService, MockBackend],
        fakeAsync((service, backend) => {
          let req;
          let res;

          backend.connections.subscribe(c => {
            req = c.request;
            c.mockRespond(new Response(<any>{body: response}));
          });

          service.search(term).subscribe(_res => {
            res = _res;
          });
          tick();

          callback(req, res);
        })
      );
    }
```

Let's see what this function does: it uses `inject` and `fakeAsync` to perform the same thing we were doing before, but in a configurable way. We take a *search term*, a *response* and a *callback function*.

We use those parameters to call the `search` method with the search term, set the fake response and call the callback function after the request is finished, providing the request and the response objects.

This way, all our test need to do is call the function and check one of the objects.

Let's break the test we had before into four tests, each testing one specific aspect of the response:

**code/http/src/app/you-tube-search/you-tube-search.component.spec.ts**

```
    it('parses YouTube video id',
      search('hey', defaultResponse, (req, res) => {
        const video = res[0];
        expect(video.id).toEqual('VIDEO_ID');
    }));

    it('parses YouTube video title',
      search('hey', defaultResponse, (req, res) => {
        const video = res[0];
        expect(video.title).toEqual('TITLE');
    }));

    it('parses YouTube video description',
      search('hey', defaultResponse, (req, res) => {
        const video = res[0];
        expect(video.description).toEqual('DESCRIPTION');
    }));

    it('parses YouTube video thumbnail',
      search('hey', defaultResponse, (req, res) => {
        const video = res[0];
        expect(video.description).toEqual('DESCRIPTION');
    }));
```

Doesn't it look good? Small, focused tests that test only one thing. Great!

Now it should be really easy to add tests for the remaining goals we had:

**code/http/src/app/you-tube-search/you-tube-search.component.spec.ts**

```
    it('sends the query',
      search('term', defaultResponse, (req, res) => {
        expect(req.url).toContain('q=term');
    }));

    it('sends the API key',
      search('term', defaultResponse, (req, res) => {
        expect(req.url).toContain('key=YOUTUBE_API_KEY');
    }));

    it('uses the provided YouTube URL',
      search('term', defaultResponse, (req, res) => {
        expect(req.url).toMatch(/^YOUTUBE_API_URL\?/);
    }));
```

Feel free to add more tests as you see fit. For example, you could add a test for when you have more than one `item` on the response, with different attributes. See if you can find other aspects of the code you'd like to test.

## Conclusion

The Angular team has done a great job building testing right into Angular. It's easy to test all of the aspects of our application: from controllers, to services, forms and HTTP. Even testing asynchronous code that was a difficult to test is now a breeze.