## Testing Http

**Learning Objectives**

• How to configure a test suite so we can mock the Http service to send back fake responses.

**Test setup**

To demonstrate how to test http requests we will add a test for our iTunes SearchService which we created in the section on Http.

We will use the promise version of the search service that uses JSONP to get around the issue of CORS.

import {Injectable} from '@angular/core';

import {Jsonp} from '@angular/http';

import 'rxjs/add/operator/toPromise';

class SearchItem {

constructor(public name: string,

public artist: string,

public thumbnail: string,

public artistId: string) {

}

}

@Injectable()

export class SearchService {

apiRoot: string = 'https://itunes.apple.com/search';

results: SearchItem[];

constructor(private jsonp: Jsonp) {

this.results = [];

}

search(term: string) {

return new Promise((resolve, reject) => {

this.results = [];

let apiURL =

`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP\_CALLBACK`;

this.jsonp.request(apiURL)

.toPromise()

.then(

res => { // Success

this.results = res.json().results.map(item => {

console.log(item);

return new SearchItem(

item.trackName,

item.artistName,

item.artworkUrl60,

item.artistId

);

});

resolve(this.results);

},

msg => { // Error

reject(msg);

}

);

});

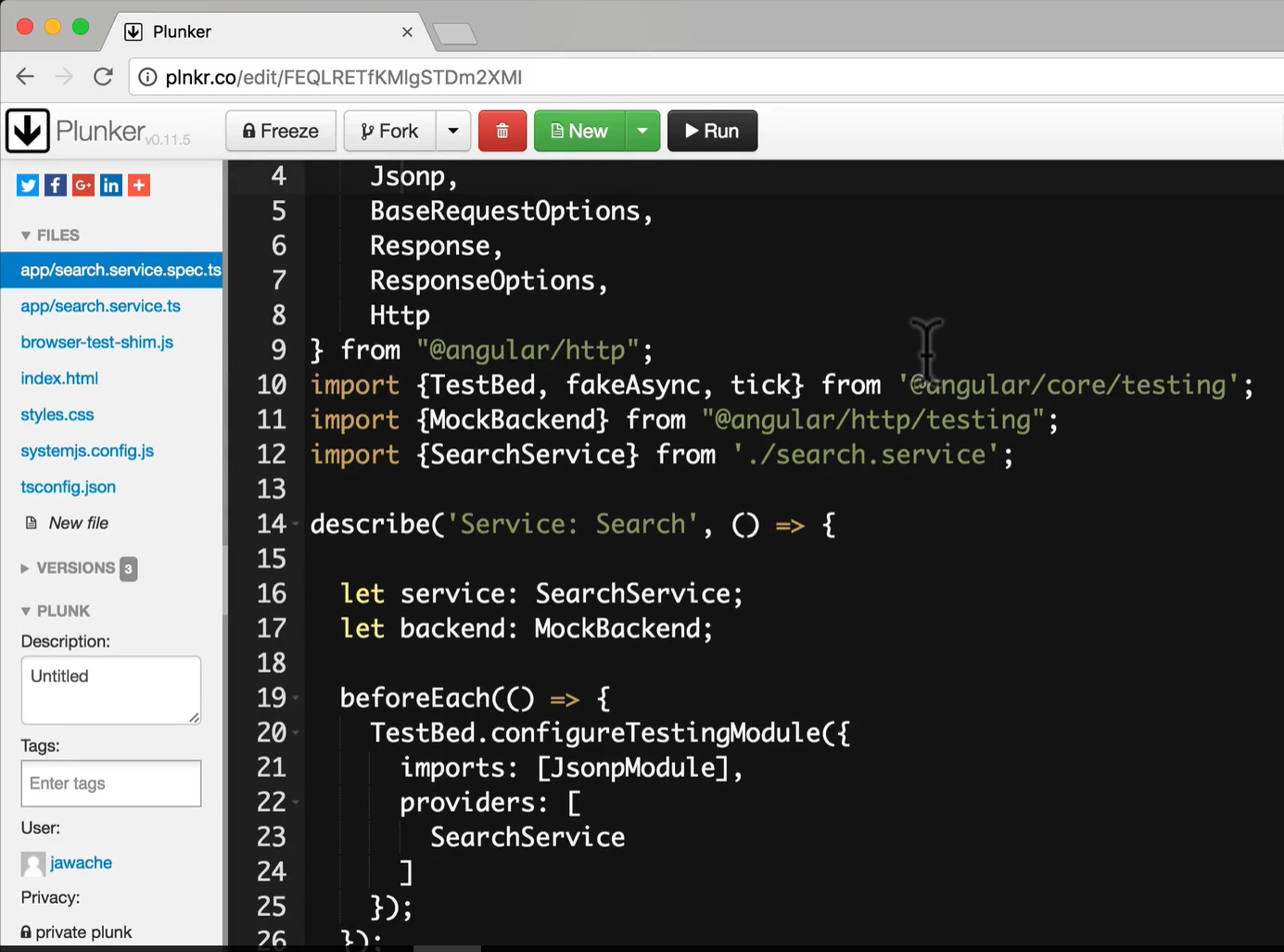
}

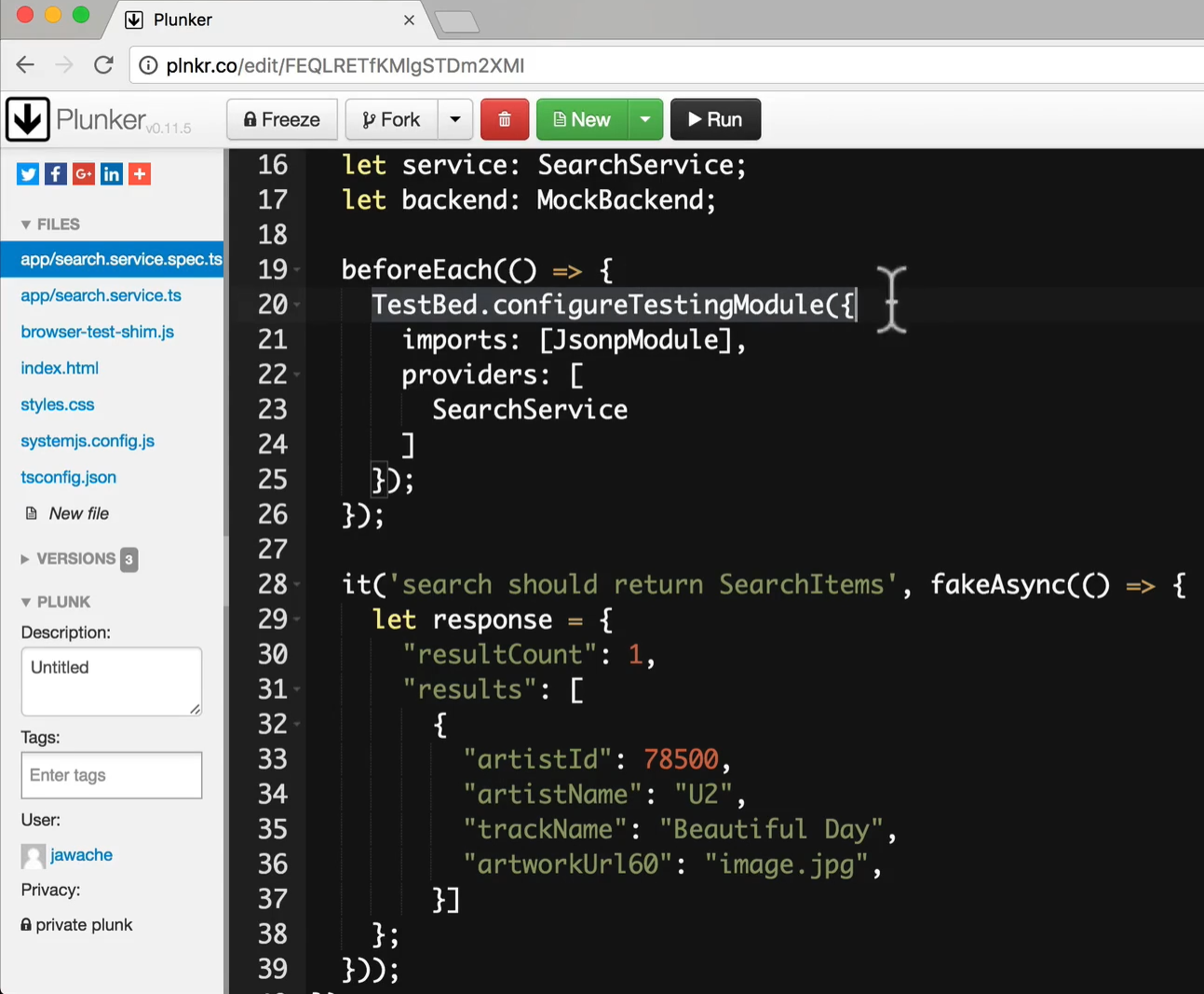
}

Although we are using JSONP here, testing Http and Jsonp is exactly the same. We just replace instances of Jsonp with Http.

### Configuring the test suite

If we look at our test spec here.





You can see, that we have basic test bed and the provider of SearchService and the basic test spec at the bottom and created a sample response and that we want to return from the HTTP request.

Eventhough provided SearchService and imported JsonpModule. If we actually try to run our test the SearchService is using the real JSONP module which is going to send an HTTP request. We don’t want that to happen and in our isolated unit test we never wanted to send an HTTP request for some other service that will make out our test not isolated but will make dependent on some other API and runs slower and have all other issues like dependencies that we don’t control and makes our test brittle.

So, We want the Jsonp and Http services to use the MockBackend instead of the *real* Backend, this is the underling code that actually sends and handles http.

By using the MockBackend we can intercept real requests and simulate responses with test data.

The configuration is slightly more complex since we are using a *factory provider(*which we covered in the section of dependency injection*)*:

Also we need to override the default provider for HTTP(below code) OR JSONP(next code) to use our MockedBackend instead of real backend which we are doing in the below code.

In the below code,

So we are going to add the provider(provide:), the TOKEN is going to be JSONP(OR) HTTP as we are going to override the configuration for the JSONP (or) HTTP, so we are going to provide the same TOKEN.

We are going to use the Factory Function to create the instance that factory function is going to take 2 functions i.e. backend and options and all of this 2nd line (useFactory – line of code) is going to do is create new JSONP (or) HTTP instance and this is going to be passed in by the BACKEND and also the options is passed in.

And the next property for the provider is the deps(dependencies) and this property contains the TOKENS that will resolve the useFactory function parameters. So we pass the real mock instances(MockBackend, BaseRequestOptions) here and this is the reason we have provide these 2 also in the PROVIDERS section. As we need to inject these 2 in for the deps property.

{

provide: Http,

①

useFactory: (backend, options) => new Http(backend, options),

②

deps: [MockBackend, BaseRequestOptions]

③

}

1. We are configuring a dependency for the token Http.
2. The injector calls this function in order to return a new instance of the Http class. The arguments to the useFactory function are themselves *injected* in, see (3).
3. We define the dependencies to our useFactory function via the deps property.

For our API however we are using Jsonp, we can just replace all mention of Http with Jsonp like so:

{

provide: Jsonp,

useFactory: (backend, options) => new Jsonp(backend, options),

deps: [MockBackend, BaseRequestOptions]

}

The above configuration ensures that the Jsonp service is constructed using the MockBackend so we can control it later on in testing.

What we have done above is that when the SearchService requests an instance of the JSONP then our test is going to use the above special configuration and the above configuration is going to return the special instance of the JSONP with MockBackEnd(and this we can control) and mock the HTTP request calls.

Together with the other providers and modules we need our initial test suite file looks like so:

describe('Service: Search', () => {

let service: SearchService;

let backend: MockBackend;

beforeEach(() => {

TestBed.configureTestingModule({

imports: [JsonpModule],

providers: [

SearchService,

MockBackend,

BaseRequestOptions,

{

provide: Jsonp,

useFactory: (backend, options) => new Jsonp(backend, options),

deps: [MockBackend, BaseRequestOptions]

}

]

});

backend = TestBed.get(MockBackend);

①

service = TestBed.get(SearchService);

②

});

})

;

1. We grab a reference to the *mock backend* so we can control the http responses from our test specs.
2. We grab a reference to the SearchService, this has been created using the MockBackend above.

Now JSONP is injected into the SearchService and this JSONP has the MockedBackend injected into it.

**Using the** MockBackend **to simulate a response**

Just by using the MockBackend instead of the real Backend we have stopped the tests from triggering real http requests from being sent out.

Now we need to configure the MockBackend to return dummy test data instead, like so:

it('search should return SearchItems', fakeAsync(() => {

let response = {

①

"resultCount": 1,

"results": [

{

"artistId": 78500,

"artistName": "U2",

"trackName": "Beautiful Day",

"artworkUrl60": "image.jpg",

}]

};

backend.connections.subscribe(connection => {

②

connection.mockRespond(new Response(<ResponseOptions>{

③

body: JSON.stringify(response)

}));

});

}))

;

1. We create some fake data we want the API to response with. i.e. MockedBackend to return this dummy data and to do that we need to grab the reference to one of the connections that JSONP module is trying to make to the server. Backend has a connections observables that we can subscribe to.
2. The mock backend connections property is an observable that emits an connection every time an API request is made. i.e. Everytime we make a request to the API then this connections observable is going to emit a connection and as we are subscribing to that we will get a reference to that connection.
3. For every connection that is requested we tell it to mockRespond with our dummy data.

The above code returns the same dummy data for *every* API request, regardless of the URL. i.e. this is regardless of which API we actually requested. i.e. it can be any API requested.

Continue from 8:55

**Testing the response**

Using HTTP is asynchronous so in order to test we need to use one of the asynchronous testing methods, we’ll use the fakeAsync method.

it('search should return SearchItems', fakeAsync(() => { ①

let response = {

"resultCount": 1,

"results": [

{

"artistId": 78500,

"artistName": "U2",

"trackName": "Beautiful Day",

"artworkUrl60": "image.jpg",

}]

};

// When the request subscribes for results on a connection, return a fake response

backend.connections.subscribe(connection => {

connection.mockRespond(new Response(<ResponseOptions>{

body: JSON.stringify(response)

}));

});

// Perform a request and make sure we get the response we expect

service.search("U2"); ②

tick(); ③

expect(service.results.length).toBe(1); ④

expect(service.results[0].artist).toBe("U2");

expect(service.results[0].name).toBe("Beautiful Day");

expect(service.results[0].thumbnail).toBe("image.jpg");

expect(service.results[0].artistId).toBe(78500);

}));

1. We use the fakeAsync method to execute in the special *fake async zone* and track pending promises.
2. We make the *asynchronous* call to service.search(…)
3. We issue a tick() which blocks execution and waits for all the pending promises to be resolved.
4. We now *know* that the service has received and parsed the response so we can write some expectations.

**Summary**

We can test code that makes Http requests by using a MockBackend.

This requires that we configure our TestBed so that the Jsonp or Http services are created using the MockBackend.

We grab a reference to the instance of MockBackend that was injected and use it to simulate responses.

Since *Http* is asynchronous we use of one of the *async* testing mechanisms so we can write tests specs for our code.

**Listing**

*search.service.ts*

import {Injectable} from '@angular/core';

import {Jsonp} from '@angular/http';

import 'rxjs/add/operator/toPromise';

class SearchItem {

constructor(public name: string,

public artist: string,

public thumbnail: string,

public artistId: string) {

}

}

@Injectable()

export class SearchService {

apiRoot: string = 'https://itunes.apple.com/search';

results: SearchItem[];

constructor(private jsonp: Jsonp) {

this.results = [];

}

search(term: string) {

return new Promise((resolve, reject) => {

this.results = [];

let apiURL =

`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP\_CALLBACK`;

this.jsonp.request(apiURL)

.toPromise()

.then(

res => { // Success

this.results = res.json().results.map(item => {

console.log(item);

return new SearchItem(

item.trackName,

item.artistName,

item.artworkUrl60,

item.artistId

);

});

resolve(this.results);

},

msg => { // Error

reject(msg);

}

);

});

}

}

*script.ts*

/\* tslint:disable:no-unused-variable \*/

import {

JsonpModule,

Jsonp,

BaseRequestOptions,

Response,

ResponseOptions,

Http

}

from "@angular/http";

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

import {MockBackend} from "@angular/http/testing";

import {SearchService} from './search.service';

describe('Service: Search', () => {

let service: SearchService;

let backend: MockBackend;

beforeEach(() => {

TestBed.configureTestingModule({

imports: [JsonpModule],

providers: [

SearchService,

MockBackend,

BaseRequestOptions,

{

provide: Jsonp,

useFactory: (backend, options) => new Jsonp(backend, options),

deps: [MockBackend, BaseRequestOptions]

}

]

});

// Get the MockBackend

backend = TestBed.get(MockBackend);

// Returns a service with the MockBackend so we can test with dummy responses

service = TestBed.get(SearchService);

});

it('search should return SearchItems', fakeAsync(() => {

let response = {

"resultCount": 1,

"results": [

{

"artistId": 78500,

"artistName": "U2",

"trackName": "Beautiful Day",

"artworkUrl60": "image.jpg",

}]

};

// When the request subscribes for results on a connection, return a fake response

backend.connections.subscribe(connection => {

connection.mockRespond(new Response(<ResponseOptions>{

body: JSON.stringify(response)

}));

});

// Perform a request and make sure we get the response we expect

service.search("U2");

tick();

expect(service.results.length).toBe(1);

expect(service.results[0].artist).toBe("U2");

expect(service.results[0].name).toBe("Beautiful Day");

expect(service.results[0].thumbnail).toBe("image.jpg");

expect(service.results[0].artistId).toBe(78500);

}));

});