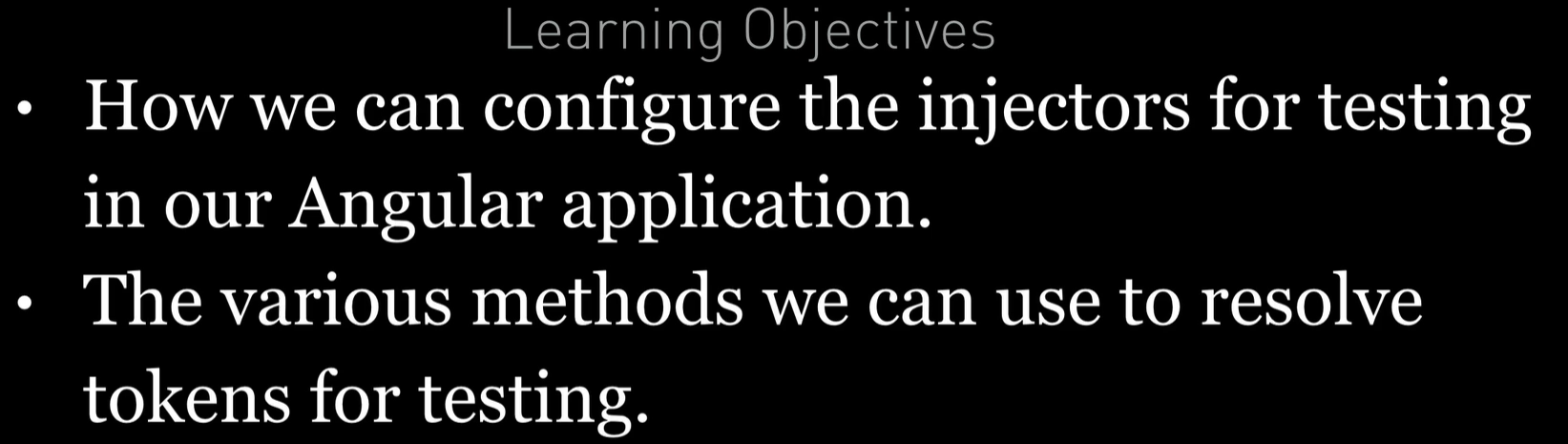
## Testing Dependency Injection

**Learning Objectives**

* Know how we can configure the injectors for testing in our Angular application.
* Know the various methods we can use to resolve tokens for testing.



**Resolving via** TestBed

This is how we’ve configured and injected dependencies so far in this section.

The TestBed acts as a dummy Angular Module and using it we can configure it with a set of providers like so:

Just like adding the providers to the Provider property in the same way as we do with the normal angular module.

TestBed.configureTestingModule({

providers: [AuthService]

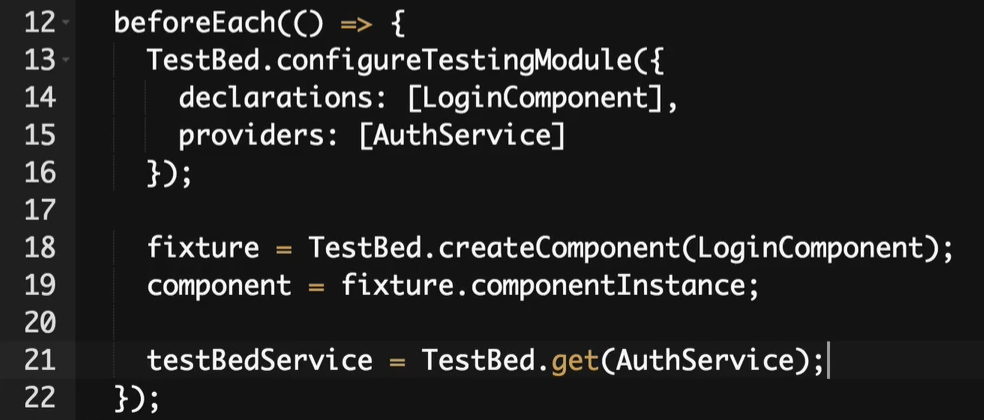
})

;

We can then ask the TestBed to resolve a token into a dependency using it’s internal injector, like so:

testBedService = TestBed.get(AuthService);

If most of our test specs need the same dependency mocked the same way we can resolve it once in the beforeEach function and store in something like testBedService variable here and then if we need to mock it with spy then we can mock it.



This is one way of resolving token in Angular

**Resolving via the** inject **function**

it('Service injected via inject(...) and TestBed.get(...) should be the same

instance',

inject([AuthService], (injectService: AuthService) => {

expect(injectService).toBe(testBedService);

})

);

We can inject explicitly in each test spec using inject function.

The inject function wraps the test spec function but lets us also *inject* dependencies using the parent injector in the TestBed.

We use it like so:

inject(

[token1, token2, token2],

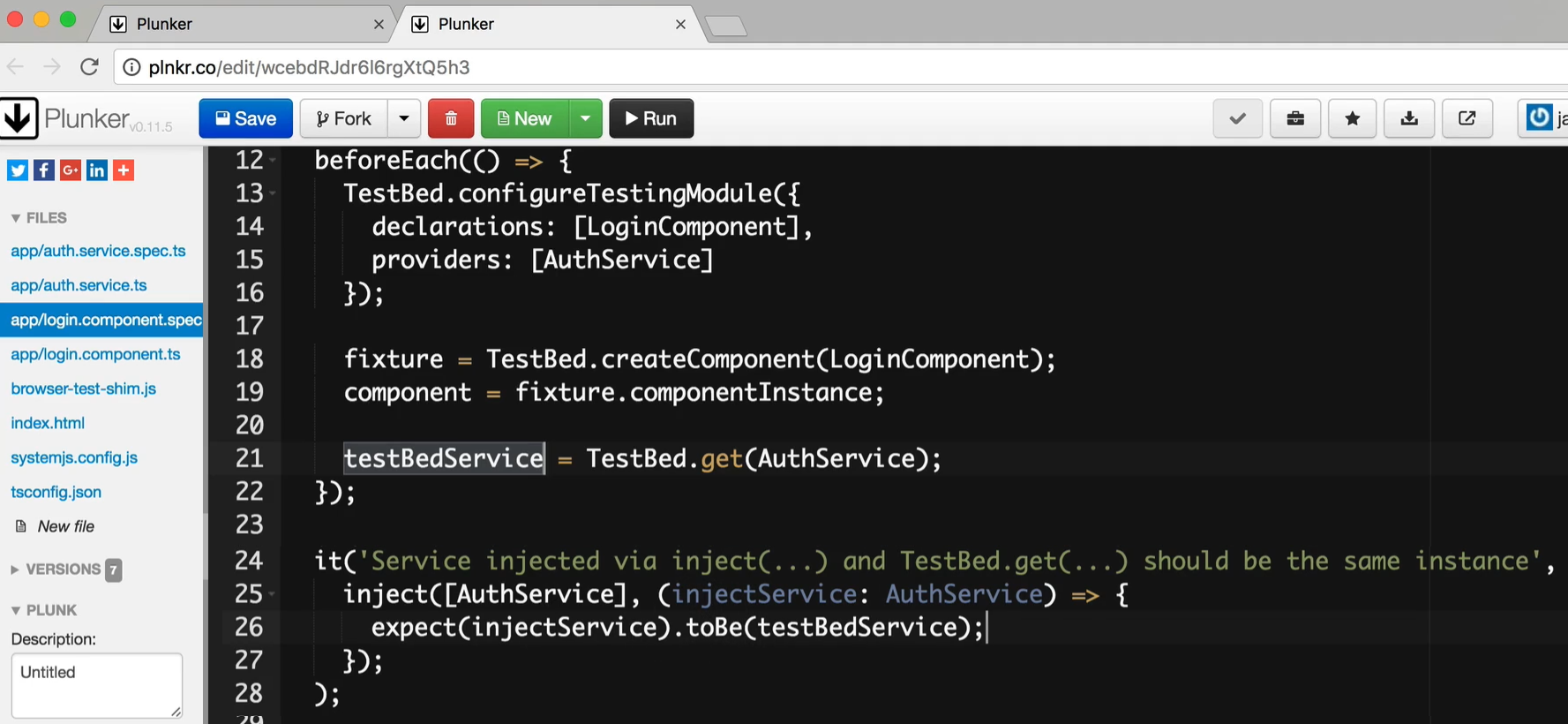
(dep1, dep2, dep3) => { }

)

The first param is an array of tokens we want to resolve dependencies for, the second parameter is a function whose arguments are the resolved dependencies.

The inject function below is going to use the same injector that is configured on the TESTBED – THE GLOBLA INJECTOR.

So the dependencies being returned from both the injectors are going to be the same. So we will test the same below



Using the inject function:

* Makes it clear what dependencies each spec function uses.
* If each test spec requires different mocks and spys this is a better solution that resolving it once per test suite. i.e. if this test spec require some custom mocks and spies for our instance of inject service which is useful only for this test spec well then we may inject our own instance into our test spec rather than reuse the an instance of created in beforeEach function.



This will eventually move to becoming an function decorator like so:

@Inject (dep1: Token1, dep2: Token2) => { ... }

**Overriding the components providers**

*Before* we create a component via the TestBed we can *override* it’s providers. Lets imagine we have a mock AuthService like so:

class MockAuthService extends AuthService {

isAuthenticated() {

return 'Mocked';

}

}

We can override the components providers to use this mocked AuthService like so.

TestBed.overrideComponent(

LoginComponent,

{set: {providers: [{provide: AuthService, useClass: MockAuthService}]}}

)

;

The first parameter for overrideComponent is Component we want to override and the second parameter is something called MetaDataOverride.

The syntax is pretty specific, it’s called a MetaDataOverride and it can have the properties set, add and remove. We use set to **completely replace(swapping)** the providers array with the values we’ve set on the LoginComponent.

If we use add then we are adding the list of providers to LoginComponent.

**Resolving via the component injector (using the above approach – overriding the components providers)**

Now our component has been configured with it’s own providers it will therefore have a child injector.

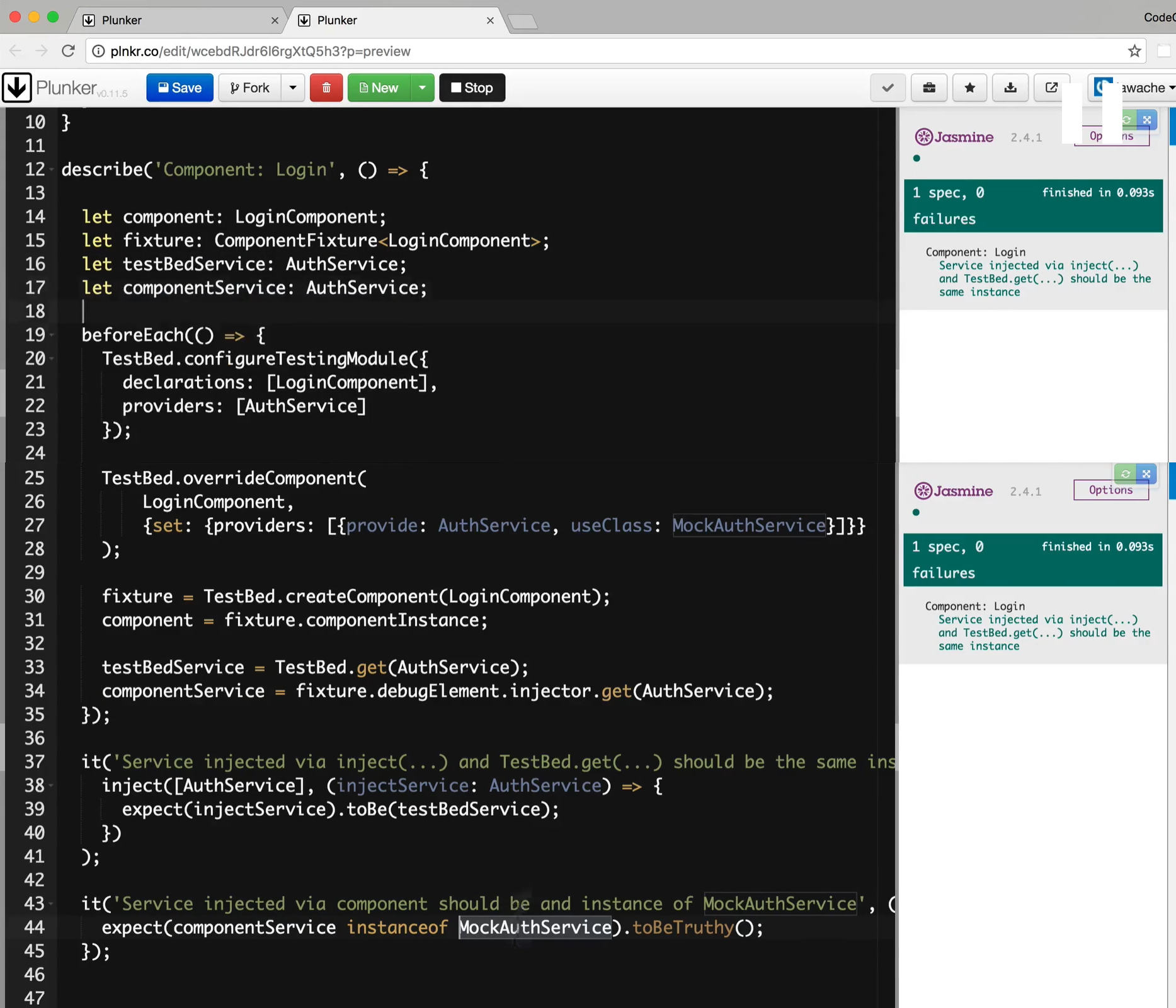
When the component is created since it has it’s own injector it will resolve the AuthService itself and not forward the request to it’s parent TestBed injector.

If we wanted to get the *same* instance of dependency that was passed to the component constructor we need to resolve using the component injector, we can do that through the component fixture like so:

componentService = fixture.debugElement.injector.get(AuthService);

The above code resolves the token using the components child injector.

* LINE 33 uses the injector associated with the LoginComponent of the child injector to resolve the token AuthService token. To get it we get it through Fixture.debug…..



The last test is testing that the componentService is instance of MockAuthService. As we have overridden the providers on the LoginComponent of the AuthService using MockAuthService.

**Summary**

We can resolve dependencies in our tests using a number of methods.

We can resolve using the the test bed itself, usually in the beforeEach function and store the resolved dependencies for variables used in our test specs.

We can resolve using the inject function at the start of each test spec.

We can also override the default providers for our components using the TestBed.

We can then also use the components *child injector* to resolve tokens.

