COM661 Full Stack Strategies and Development

FE19. Testing the Front-end

Aims

- To introduce the Angular testing framework
- To present test specificactions in Jasmine
- To demonstrate test presentation in Karma
- To design a bespoke test framework for the Biz Directory application
- To test the retrieval of a specified page of businesses
- To test the retrieval of a specified business
- To test the retrieval of the reviews for a specified business
- To test the addition of a review for a business

Table of Contents

19.1 Angular Testing Framework

Software testing is the process of verifying that the expected output is produced for any combination of input. In Angular, it requires that the Components and Services that make up an application respond appropriately when they are activated. In this section, we examine the built-in Component testing provided by the platform, as well as developing a bespoke testing framework for the Biz Directory sample application.

19.1.1 Angular Component Testing

When the Angular framework creates a Component, four files are written to the application. We have so far seen three of these as described below for a component **x**:

x.component.hml	the HTML template
x.component.css	the stylesheet for the component
x.component.ts	the TypeScript definition of the component structure and behaviour.

The fourth "default" file is one that we have not so far created ourselves but can be seen by browsing the **`AppComponent`** files created by Angular when we create a new application. The file ***app.component.spec.ts*** is the default test script for the App Component and checks that the component has been created and is available for use. If we create components using the Angular framework command

```
C:\angularapp> ng generate component newname
```

then ***newname.component.spec.ts*** is generated alongside ***newname.component.html***, ***newname.component.ts*** and ***newname.component.css***. The structure of a default test script can be seen by creating a new Angular application called **calculator** (that we will use later in this practical) and examining the source of ***app.component.spec.ts*** as shown in the code box below.

Do it now!	Create a new Angular application called **calculator** by the command **`ng new calculator`** and verify that the test file ***app.component.spec.ts*** is created and that it has the structure shown in the code box below.

File: calculator/src/app/app.component.spec.ts

```
import { TestBed } from '@angular/core/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [AppComponent],
    }).compileComponents();
  });

  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app).toBeTruthy();
  });

  it(`should have the 'calculator' title`, () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app.title).toEqual('calculator');
  });

  it('should render title', () => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.nativeElement as HTMLElement;
    expect(compiled.querySelector('h1')?.textContent).toContain(
          'Hello, calculator');
  });
});
```

Angular tests are written in a JavaScript framework called **Jasmine** that helps us write test cases in a human-readable way. The Angular testing package includes a utility called **`TestBed`** that is structured around a container called **`describe`**. The **`describe`** container contains a number of blocks, the most important of which are **`beforeEach`**, which runs before each individual test, and **`it`**, which defines a test. The code above defines three tests, verifying (i) that the component has been created, (ii) that the title property of the app is "**calculator**", and (iii) that the HTML template contains a **`<h1>`** element with the value "**Hello, calculator**". The structure of the test specification is shown in Figure 19.1, below, where the **`describe()`** block is a container for the suite of tests, the **`beforeEach()`** block contains configuration or installation code that is run before each test, each **`it()`** block

contains the description of a single test, and the **`expect()`** block describes the condition that must be met for the test to pass.

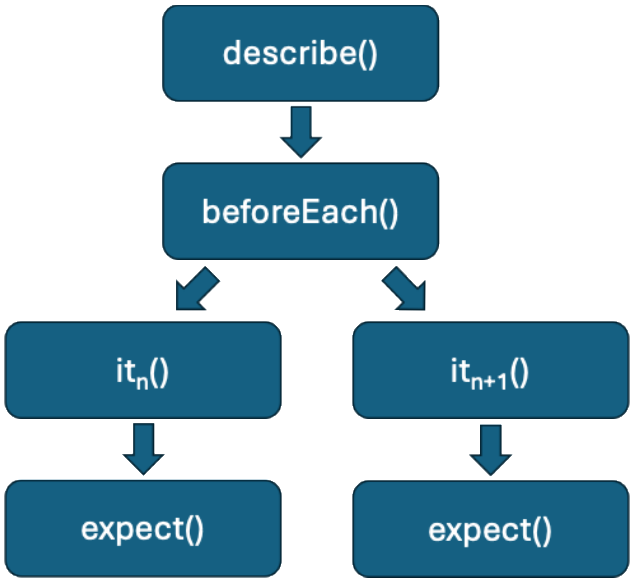

Figure 19.1 Angular Testing Structure

The Angular tests are run by the command **`ng test`**, which opens a Karma server that executes the test code in a browser. Karma is a test runner that executes the tests in a local server. It provides features such as live reloading, code coverage reporting and integration with a range of Continuous Integration tools. The instruction is entered from the command prompt as shown in Figure 19.2, below.

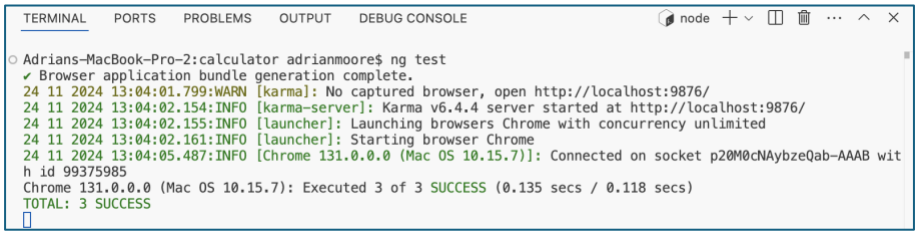

Figure 19.2 Running the Angular tests

When the test is invoked, it spawns a web browser and displays the results of the tests, as illustrated by Figure 19.3, below. Here, we see that the tests described by the three **`it()`** blocks in the code box above have successfully passed. Note the use of the word “should” in the title of each **`it()`** block (provided as the first parameter). This is a convention that should be observed and helps us to write appropriate tests.

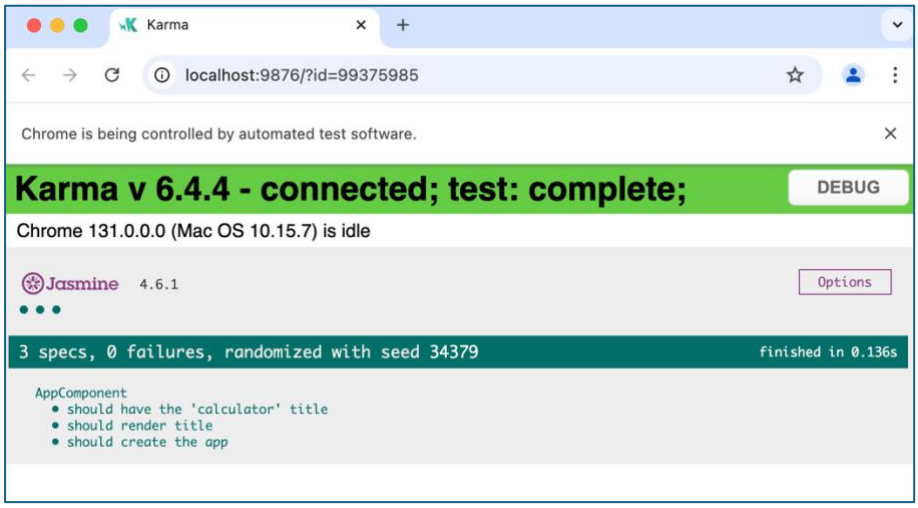

Figure 19.3 Test Results

19.1.2 A Calculator App

As a demonstration of Angular Component testing, we will build a small calculator application on which to specify a test suite. First, we add the files ***calculator.component.ts*** and ***calculator.component.html*** to our calculator application as shown in the code boxes below.

File: calculator/src/app/calculator.component.ts

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

@Component({
  selector: 'calculator',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './calculator.component.html'
})
export class CalculatorComponent { }
```

File: calculator/src/app/calculator.component.html

```
{ file is empty }
```

Next, we add to the component Typescript file, a property called **`calculations`** to hold the collection of operations that we will execute, as well as a method to implement the multiplication operation and add a report to the **`calculations`** collection. We also implement **`ngOnInit()`** to call the **`multiply()`** method for a range of calculations.

File: calculator/src/app/calculator.component.ts

```
...

export class CalculatorComponent {
  calculations: string[] = [];

  public multiply(x: number, y: number)  {
    this.calculations.push(x + " * " + y + " = " + (x * y) );
    return x * y;
  }

  ngOnInit() {
    this.multiply(2, 4);
    this.multiply(-3, -3);
    this.multiply(-3, 2);
    this.multiply(5, 0);
  }

}
```

Then, we add a **`@for`** block to the HTML template to display the collection of reports on the web page.

File: calculator/src/app/calculator.component.html

```
@for( calculation of calculations; track calculation ) {
    <h1>{{ calculation }}</h1>
}
```

Finally, in the setup stage, we replace the default code in ***app.component.html*** with an instance of the **`calculator`** selector so that the list of calculator reports is displayed in the browser and import the Calculator component into the App component specification.

File: calculator/src/app/app.component.html

```
<calculator></calculator>
```

File: calculator/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { CalculatorComponent } from './calculator.component';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, CalculatorComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = calculator;
}
```

Now, when we run the application with **`ng serve`**, we see the report of the four calls to the **`multiply()`** method reported in the browser, as shown in Figure 19.4, below.

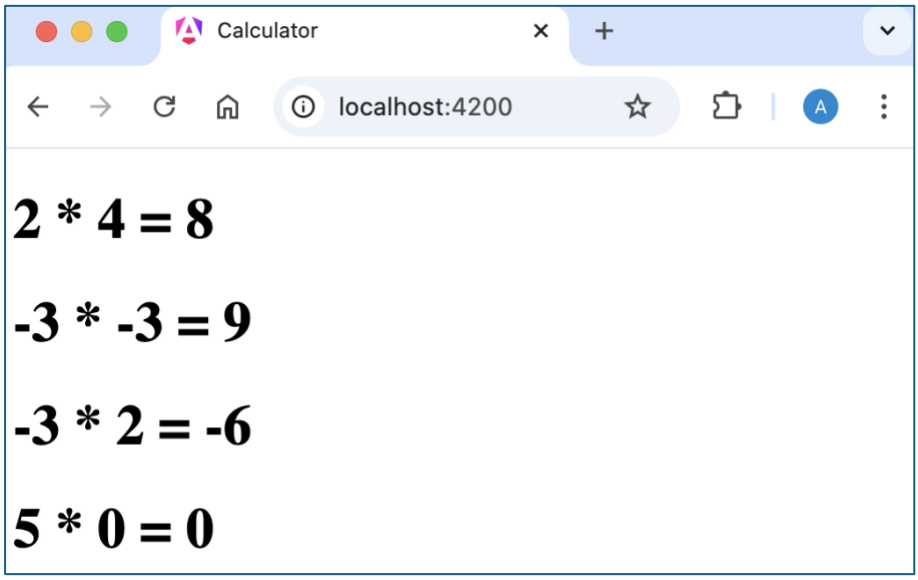

Figure 19.4 A Calculator App

Do it now!	Implement the Calculator component as described in the code boxes above and verify that the calls to the **`multiply()`** method are logged in the browser.

19.1.3 A First Test

Now that the calculator app has a working function, we can specify tests to ensure that the correct result is generated for all combinations of input. Initially, we specify the boilerplate code that imports the test framework components and creates an instance of the **Calculator** component on which to run the tests.

File: calculator/src/app/calculator.component.spec.ts

```
import { ComponentFixture, TestBed }
         from '@angular/core/testing';
import { CalculatorComponent } from './calculator.component';

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

  beforeEach( () => {
    fixture = TestBed.createComponent(CalculatorComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

// tests go here

});
```

Do it now!	Add the file ***calculator.component.spec.ts*** to the application and populate it with the standard test template code as shown in the code box above.

Next, we add a test to ensure that two positive numbers are correctly multiplied. Each test is specified in an `it()` block, where the first parameter is a string that describes the desired outcome of the test and the second parameter is a function that calls the method and compares the result obtained to that which is expected. Here, we expect the result of the multiplication of 2 and 4 to be 8.

File: calculator/src/app/calculator.component.spec.ts

```
...

// tests go here

it('2 times 4 should be 8', () => {
    let result = component.multiply(2, 4);
    expect(result).toEqual(8);
  });

...
```

Now, when we run the tests, the result shown in Figure 19.5 is shown in the browser. Note that we have also commented out the final test in ***app.component.spec.ts*** as the **`title`** property is no longer being rendered to the **`<h1>`** element in the browser.

File: calculator/src/app/app.component.spec.ts

```
...

  /*
  it('should render title', () => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.nativeElement as HTMLElement;
    expect(compiled.querySelector('h1')?.textContent)
          .toContain('Hello, calculator');
  });
  */

});
```

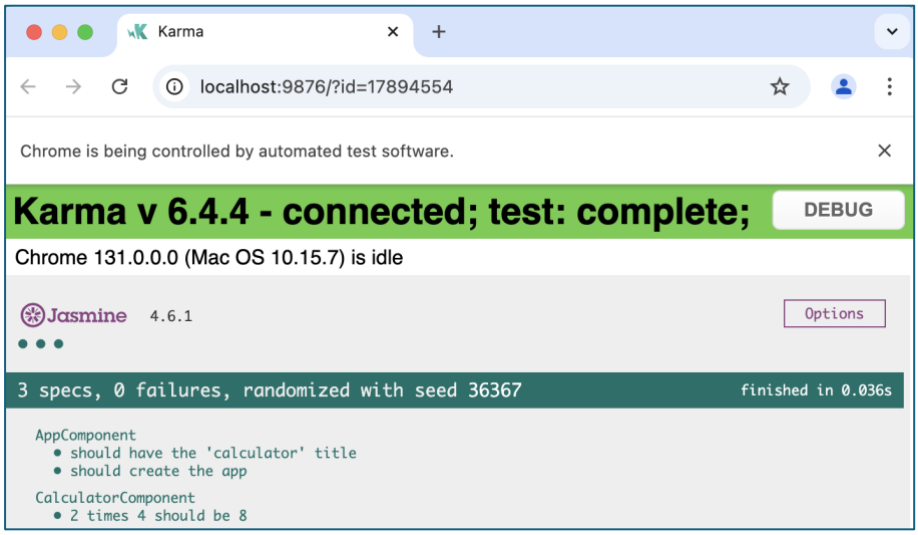

Figure 19.5 Testing the **`multiply()`** *Method*

19.1.4 Adding Tests

With a test framework in place and a first test successfully executing, we can now add additional tests to check the correct operation of the **`multiply()`** method for all combinations of input. Specifically, we want to check that if one of the numbers is negative, the result is negative, if both numbers are negative the result should be positive, and if one of the numbers is zero, the result should be zero. The specification of the new tests is shown in the code box below, while Figures 19.6 shows the sequence of operations in the browser and the result of the tests in Karma.

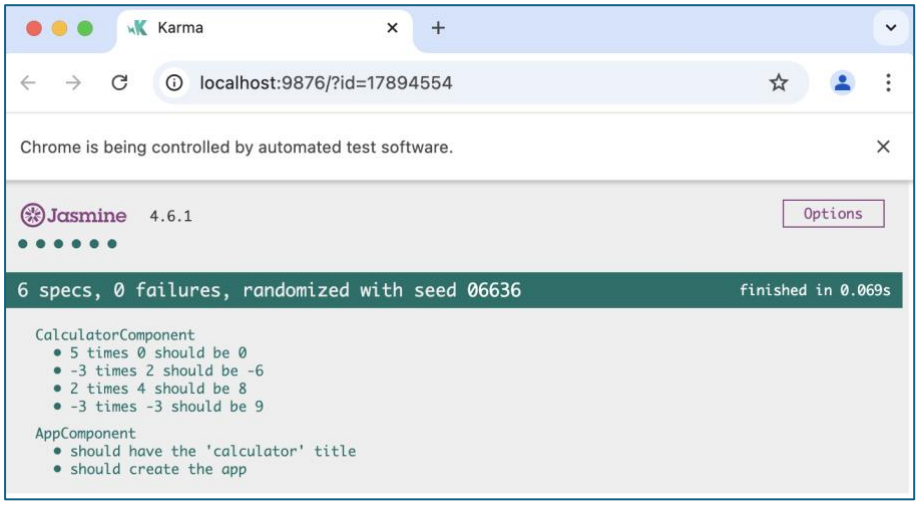

Figure 19.6 Multiplication Test Suite

File: calculator/src/app/calculator.component.spec.ts

```
...

// tests go here

  it('2 times 4 should be 8', () => {
    let result = component.multiply(2, 4);
    expect(result).toEqual(8);
  });

  it('-3 times 2 should be -6', () => {
    let result = component.multiply(-3, 2);
    expect(result).toEqual(-6);
  });

  it('-3 times -3 should be 9', () => {
    let result = component.multiply(-3, -3);
    expect(result).toEqual(9);
  });

  it('5 times 0 should be 0', () => {
    let result = component.multiply(5, 0);
    expect(result).toEqual(0);
  });

...
```

Do it now!	Add the remaining tests to the test specification as shown in the code box above. Verify that all tests run in Karma and that all pass.

Although our tests in this example are quite trivial, it is useful to also see what happens when a test fails. We can manufacture this situation by amending the last test description to expect that 5 multiplied by zero will return the result 5, as shown in the code box below.

File: calculator/src/app/calculator.component.spec.ts

```
...

  it('5 times 0 should be 5', () => {
    let result = componentmultiply(5, 0);
    expect(result).toEqual(5);
  });

...
```

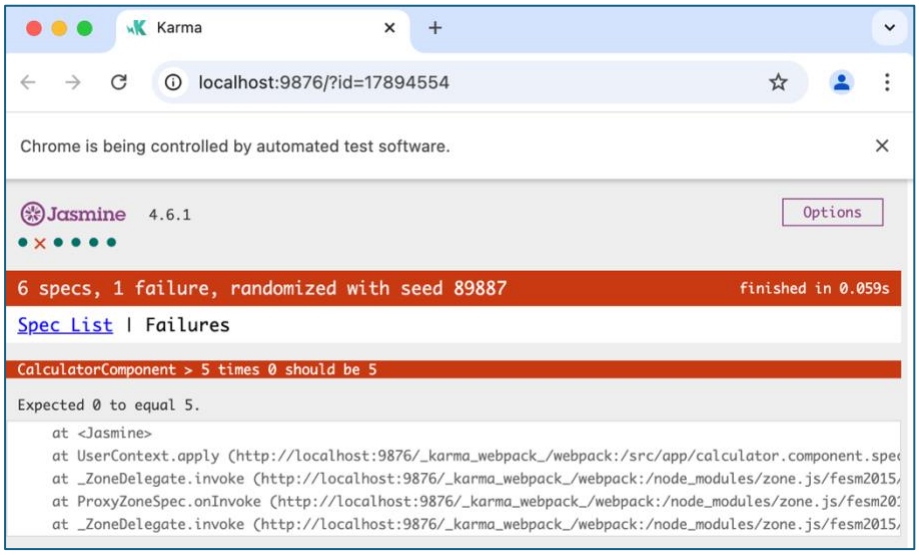

Figure 19.7 A Failing Test

Do it now!	Modify the final test so that is produces a failing result, as shown in Figiure 19.7 above.

19.2 A Custom Testing Framework

The Jasmine framework provides a convenient way of testing the functionality of our Angular applications, but we may also need to supplement it with custom tests that check the operations of specific features. In this section, we will construct a custom testing framework for the Biz Directory application, verifying the correct operation of each element of functionality in our Web Service.

The Web Service is the central element of our application and manages all communication between the front- and back-ends. It provides four functions as follows, that we will test separately

- Fetch a page of businesses
- Fetch a single business specified by ID
- Fetch the reviews of a business
- Add a new review for a business

19.2.1 The Test Component

To test our Web Service, we will create a new component called **`TestWSComponent`** that we specify by TypeScript and HTML files as described below.

File: bizFE/src/app/testWS.component.ts

```
import { Component } from '@angular/core';
import { WebService } from './web.service';

@Component({
  selector: 'testWS',
  standalone: true,
  providers: [WebService],
  templateUrl: './testWS.component.html'
})
export class TestWSComponent {

  test_output: string[] = [];

  constructor(private webService: WebService) {}

}
```

The TypeScript definition imports the WebService as a provider and creates a property **`test_output`** that will hold a series of **`string`** objects that describe the result of the tests. In the HTML template, we iterate over the **`test_output`** collection displaying the strings contained in the browser.

File: bizFE/src/app/testWS.component.html

```
<div style="margin-top: 70px">

    <h1>Test Report</h1>

    <ul>
    @for (report of test_output; track report) {
        <li>{{ report }}</li>
    }
    </ul>

</div>
```

We make the TestWS component available to the user by adding a new route to **app.routes.ts** that causes the component to be displayed when the URL **/test** is presented to the browser.

File: bizFE/src/app/app.routes.ts

```
...

import { TestWSComponent } from './testWS.component';

export const routes: Routes = [
    {
        path: '',
        component: HomeComponent
    },
    {
        path: 'businesses',
        component: BusinessesComponent
    },
    {
        path: 'businesses/:id',
        component: BusinessComponent
    },
    {
        path: 'test',
        component: TestWSComponent
    }
];
```

With the new component in place, we can verify that everything has been properly connected by running the application and visiting the new UR http://localhost:4200/test as shown in Figure 19.8. below.

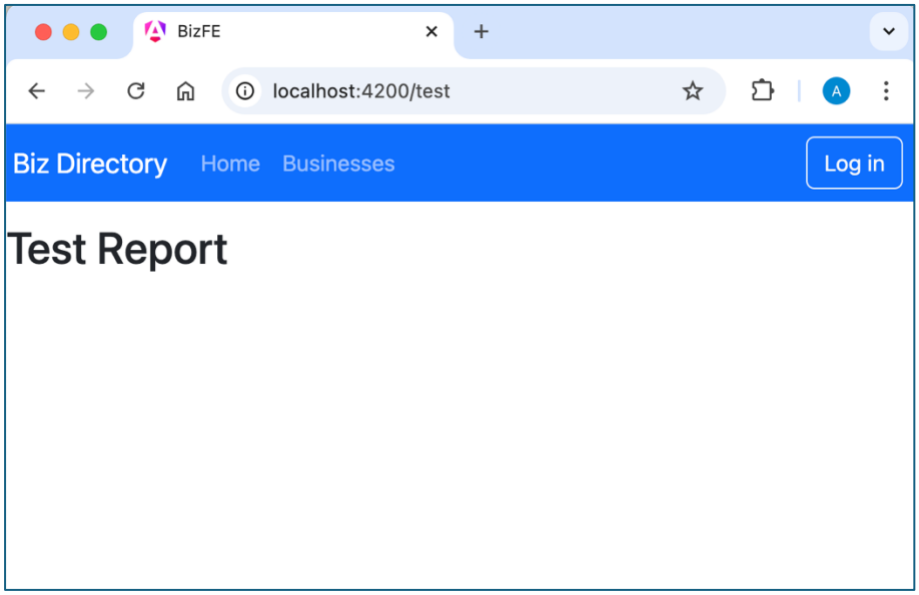

Figure 19.8 Biz Directory Test Framework

Do it now!	Create the new TestWS Component.as shown in the code boxes above and run the application. Confirm that an empty set of test reports is displayed as illustrated in Figure 19.9, above.

19.2.2 Specifying a Test

The first test is to check that the Web Service **`getBusinesses()`** method returns a page of three businesses, where three is the default page size specified in ***web.service.ts***. In this test, we will consider the method to be operating properly if the response object is an array and the length of the array is three, so we add a method to **`TestWSComponent`** to make a call tom the Web Service method and check the **`response`** object. Remember that as the Web Service method returns an Observable, we need to subscribe to it and perform our check in the subscription function.

File: bizFE/src/app/testWS.component.ts

```
...

  constructor(private webService: WebService) {}

  private testBusinessesFetched() {
    this.webService.getBusinesses(1)
      .subscribe( (response) => {
        if (Array.isArray(response) && response.length == 3)
          this.test_output.push(
               "Page of businesses fetched... PASS");
        else
          this.test_output.push(
               "Page of businesses fetched... FAIL");
    });
  }
```

With the **`testBusinessesFetched()`** method in place, we add an **`ngOnInit()`** method to the **`testWSComponent`** to call the test method when the component loads and report the result in the web browser.

File: bizFE/src/app/testWS.component.ts

```
...

    ngOnInit() {
        this.testBusinessesFetched();
    }
}
```

Running the application and visiting the test URL causes the new test to be performed and reported as shown in Figure 19.9, below.

Do it now!	Add the **`testBusinessesFetched()`** method to the TestWS Component as shown in the code boxes above. With the application running, visit the URL http://localhost:4200/test and confirm that the test to check the **`getBusinesses()`** method passes as illustrated in Figure 19.10, below.

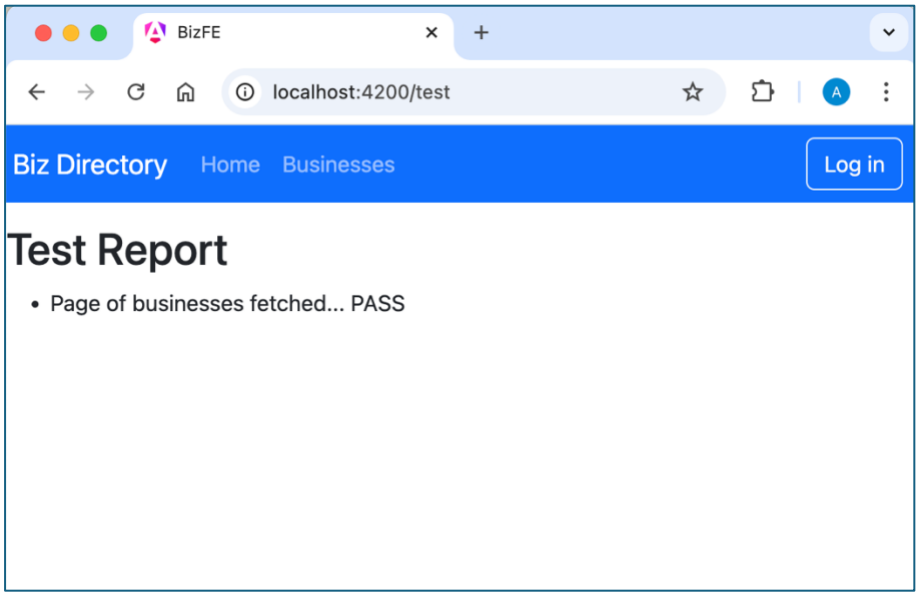

Figure 19.9 Fetching a Page of Businesses

19.2.3 Testing Other Functionality

The next test checks that the pagination in the **`getBusinesses()`** function is working by fetching the first and second pages of businesses and confirming that the first entry in each page is different. To facilitate this we add a pair of new properties to the TestWS Component called **`first_business_list`** and **`second_business_list`** and populate them by calls to **`getBusinesses(1)`** and **`getBusinesses(2)`**, respectively, where the parameter is the page number required.

The code box below presents the code for this test. First, we make a call to **`getBusinesses(1)`**, subscribing to the Observable returned. In the subscription function, we accept the response into **`first_business_list`** and make a call to **`getBusinesses(2)`**. In the subscription function for this, we accept the response into **`second_business_list`** and then check that the first entries in **`first_business_list`** and **`second_business_list`** are different, adding the appropriate response to the **`test_output`** collection. We then add a call to the new test to **`ngOnInit()`**. Figure 19.10 shows the result of adding this test to the collection.

File: bizFE/src/app/testWS.component.ts

```
...

export class TestWSComponent {

  first_business_list: any[] = [];
  second_business_list: any[] = [];

  ...

  private testPagesOfBusinessesAreDifferent() {
      this.webService.getBusinesses(1)
        .subscribe( (response) => {
           this.first_business_list = response;
           this.webService.getBusinesses(2)
             .subscribe( (response) => {
                this.second_business_list = response;
                if (this.first_business_list[0]["_id"] !=
                    this.second_business_list[0]["_id"])
                    this.test_output.push(
                       "Pages 1 and 2 are different... PASS");
                else
                    this.test_output.push(
                      "Pages 1 and 2 are different... FAIL");
           })
      });
  }

  ngOnInit() {
    this.testBusinessesFetched();
    this.testPagesOfBusinessesAreDifferent();
  }
}
```

Do it now!	Add the new test to the TestWS Component and verify that calls to `getBusinesses()` with different parameter values returns different results.

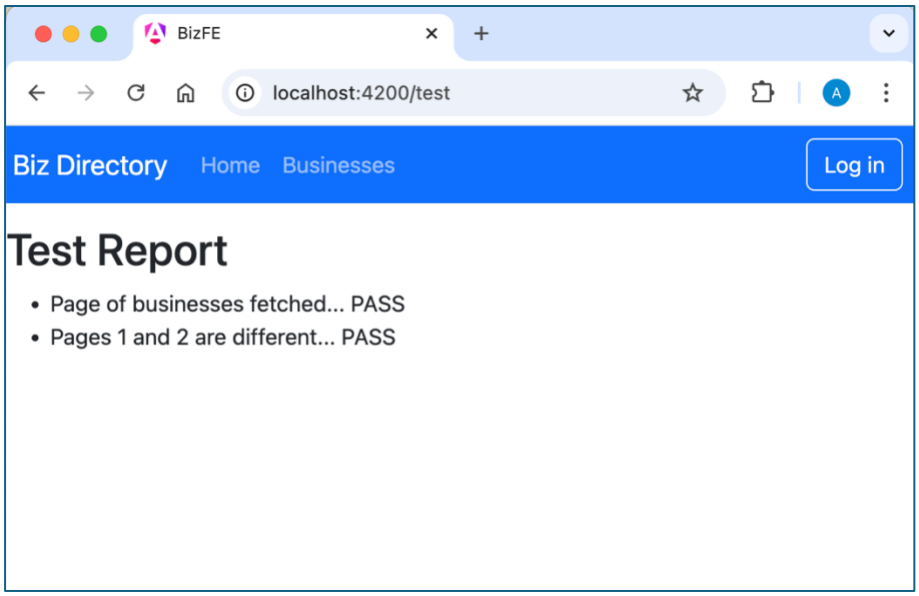

Figure 19.10 Testing that Different Pages Have Different Contents

The next test checks that providing a known `_id` value to a call to the `getBusiness()` method returns the business object expected. Here, we use the MongoDB Shell to retrieve the `_id` of the business with the name "Biz 0" and then make a call to retrieve the business matching the specified `_id`. If the business returned is that with the `name` attribute "Biz 0", then the test is successful.

File: bizFE/src/app/testWS.component.ts

```
        ...

        private testGetBusiness() {
        this.webService.getBusiness('66df1c8fcf0ec82461958517')
          .subscribe( (response) => {
              if (response.name == 'Biz 0')
                this.test_output.push("Fetch Biz 0 by ID... PASS");
              else
                this.test_output.push("Fetch Biz 0 by ID... FAIL");
          })
      }

      ngOnInit() {
        this.testBusinessesFetched();
        this.testPagesOfBusinessesAreDifferent();
        this.testGetBusiness();
      }

    ...
```

Next, we test that the reviews of a business are being fetched by making a call to the **`getReviews()`** function, passing the **`_id`** of a known business. Since a business does not necessarily have any reviews, we consider the test to have passed if an array is returned – even if the array is empty.

File: bizFE/src/app/testWS.component.ts

```
        ...

      private testGetReviews() {
        this.webService.getReviews('66df1c8fcf0ec82461958517')
          .subscribe( (response) => {
              if (Array.isArray(response))
                this.test_output.push(
                     "Fetch Reviews of Biz 0... PASS");
              else
                this.test_output.push(
                     "Fetch Reviews of Biz 0... FAIL");
          })
      }

      ngOnInit() {
        this.testBusinessesFetched();
        this.testPagesOfBusinessesAreDifferent();
        this.testGetBusiness();
        this.testGetReviews();
      }

    ...
```

The final Web Service function is that which adds a review to the collection for a business. To test this, we fetch the list reviews for a known business, storing the length of the reviews structure in a local variable. Then, we pass a dummy review object to the Web Service **`postReview()`** method, along with the **`_id`** of the business. In the subscription function, we again fetch the reviews for the same business and check that the number of reviews is one greater than the previous value.

File: bizFE/src/app/testWS.component.ts

```
      ...

    private testPostReview() {
      let test_review = {
        "username" : "Test User",
        "comment" : "Test Comment",
        "stars" : 5
      };
      this.webService.getReviews('66df1c8fcf0ec82461958517')
      .subscribe( (response) => {
          let numReviews = response.length;
          this.webService.postReview(
                      '66df1c8fcf0ec82461958517', test_review)
            .subscribe( (response) => {
              this.webService.getReviews(
                      '66df1c8fcf0ec82461958517')
                .subscribe( (response) => {
                  if (response.length == numReviews + 1)
                    this.test_output.push("Post review... PASS");
                  else
                    this.test_output.push("Post review... FAIL");
                });
            });
      });
    }

    ngOnInit() {
      this.testBusinessesFetched();
      this.testPagesOfBusinessesAreDifferent();
      this.testGetBusiness();
      this.testGetReviews();
      this.testPostReview();
    }

  ...
```

Adding these tests to the TestWS Component and running the application confirms that all tests are successful as illustrated in Figure 19.11, below.

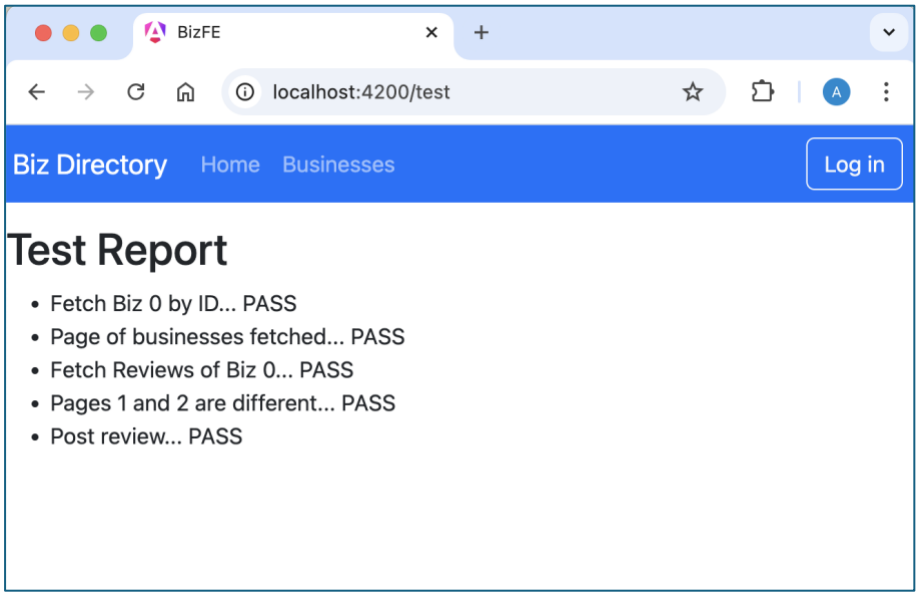

Figure 19.11 The Full Suite of Web Service Tests

Do it now!	Add the remaining tests to the TestWS Component and verify that all tests are successful.

19.3 Further Information

- https://angular.dev/guide/testing
  Angular Testing Documentation

- https://blog.logrocket.com/angular-unit-testing-tutorial/
  Angular Unit Testing Tutorial with Examples

- https://www.angularminds.com/blog/writing-your-first-angular-unit-test-step-by-step-tutorial
  Writing your First Angular Unit Test

- https://www.telerik.com/blogs/testing-angular
  Testing in Angular

- https://angular.dev/tools/libraries/creating-libraries
  How to Write Unit Tests with Jasmine and Karma

- https://www.youtube.com/watch?v=ic_Ez8PO_jc
  Angular Unit Testing Using Jasmine and Karma Tutorial (YouTube)

- https://jasmine.github.io/
  Jasmine Home Page

- https://karma-runner.github.io/latest/index.html
  Karma Test Runner Home Page

- https://www.zymr.com/blog/angular-unit-testing-with-karma-and-jasmine
  Angular Unit Testing with Karma and Jasmine

- https://dev.to/chintanonweb/mastering-angular-unit-testing-a-comprehensive-guide-with-examples-3eg9
  Mastering Angular Unit Testing: A Comprehensive Guide with Examples