ANGULAR TESTING

Naveen Pete

Agenda

- What is Automated Testing?
- Types of Tests
- Fundamentals of Unit Testing
- Set Up and Tear Down
- Spies
- Code Coverage
- Angular Testing Utilities - TestBed
- Working with Components
- Testing Property & Event Bindings
- Handling Component Dependencies
- Testing Async Operations
- Q & A

What is Automated Testing?

- Practice of writing code to test our code
- Run tests in an automated fashion
- Manual testing is time consuming
- Writing automated tests – is this not time consuming?
  - App code
  - Test code
- Automated testing
  - helps you catch defects before releasing your software
  - build software of better quality
  - help you become a better developer
  - enforces you to write better and more reliable code
- Be pragmatic, automated testing may not be good for
  - start ups – limited time, limited budget, not sure about product future
  - frequent change in requirements – will need change in test code

Types of Tests

- Unit tests
  - Test a component in isolation, without external resources like database, file, etc
  - Easy to write
  - Fast
  - Don’t test functionality of app, low confidence
  - Angular – Testing Component code in isolation (without template)
- Integration tests
  - Test a component with external resources
  - Angular – Testing Component code with External Template
- End-to-end tests
  - Test the entire app as a whole
  - Test the app functionality, more confidence
  - Slow and fragile
- Ideal scenario
  - Spend more time to write unit and integration tests
  - Write few end-to-end tests for only the key functionality

Fundamentals of Unit Testing

- Tests are first-class citizens
  - Clean coding practices
  - Apply same principles as the functional code
  - Small functions / methods – 10 lines or less
  - Proper naming
  - Single responsibility – test only one thing

- Angular Testing Tools
- Jasmine
  - A behavior-driven development framework for testing JavaScript code
  - Dependency free and doesn't require a DOM
- Karma
  - A test runner for writing and running unit tests while developing Angular apps
- Protractor
  - Write and run end-to-end (e2e) tests

Fundamentals of Unit Testing

- Test files should have .spec.ts extension
- Running tests using Angular CLI
  - ng test
- describe() – define a suite – group of related tests
  - describe('suite-name', function)
- it() – define a spec or test
  - it('spec-name', function)
- expect()
  - expect(result).toBe('value')
  - expect(result).toContain('value')

Set Up and Tear Down

- Arrange - initialize the system under test
- Act - calling a method / function
- Assert – assertion

- beforeEach(function)
  - Run some shared setup before each of the specs in the describe in which it is called
- afterEach(function)
  - Run some shared teardown after each of the specs in the describe in which it is called
- beforeAll(function)
  - Run some shared setup once before all of the specs in the describe are run
- afterAll(function)
  - Run some shared teardown once after all of the specs in the describe are run

Spies

- spyOn()
  - Install a spy onto an existing object

- spyOn(object, ‘methodName’).and.callFake(function)
  - function should match ‘methodName’ signature

- spyOn(object, ‘methodName’).and.returnValue(observable)

- Observable.from([array])
- Observable.empty()
- Observable.throw(error)

- expect(spy).toHaveBeenCalled()
- expect(spy).toHaveBeenCalledWith(id)

Code Coverage

- How much of our code is covered with tests?
- ng test --code-coverage
- Open ‘index.html’ within ‘coverage’ folder in the project

Angular Testing Utilities - TestBed

- Used to test interaction between a component and its template or with other components
- Include TestBed class and several other helper functions
- Module - '@angular/core/testing'

- TestBed
  - The first and most important Angular testing utility

- TestBed.configureTestingModule(metadataObject)
  - Used to create a dynamic Angular testing module
  - Takes an @NgModule-like metadata object
  - 'metadataObject' can have most of the properties of NgModule.

Working with Components

- TestBed.createComponent(component)
  - Used to create an instance of component-under-test
  - Returns ***component test fixture***

- ComponentFixture
  - Wrapper around a component
  - Gives access to component instance as well as its template (DOM representation)

  - ComponentFixture.componentInstance
    - Returns instance of the component class
  - ComponentFixture.nativeElement
    - Returns the native DOM element at the root of the component
  - ComponentFixture.debugElement
    - Provides a wrapper object around component's root native element
    - Provides useful methods for querying the DOM

Testing Property & Event Bindings

- DebugElement.query(predicate)
  - Used to query the DOM
  - predicate is a function that returns true if a condition is met
  - Returns the first element that matches the predicate
- By.css()
  - Predicates for use with DebugElement's query functions
  - Match elements by the given CSS selector
- ComponentFixture.detectChanges()
  - Trigger a change detection cycle for the component
- DebugElement.triggerEventHandler('eventName', eventObj)
  - Used to trigger an event on an element. For e.g., to invoke 'click' event on a button with id 'save', following code is used
    - const button = fixture.debugElement.query(By.css('#save'));
    - button.triggerEventHandler('click', null);

Handling Component Dependencies

- Providing Dependencies
  - Register the service in the testing module by adding it to the ‘providers’ array
  - Register any other Angular dependency in the testing module by adding it to ‘imports’ array
    - For e.g., if the service internally uses Http, add HttpModule to ‘imports’ array of testing module configuration

- Getting Dependencies
  - TestBed.get(service)
    - Returns a reference to ‘service’ instance injected in a component

Handling Component Dependencies

- Providing Stubs
  - Identify the dependencies and their methods that are used within the component
  - Create a stub class for each of the dependencies
  - Define method stubs
  - Replace the actual dependency with their corresponding stub implementation within the ‘providers’ array

```
TestBed.configureTestingModule({
   declarations: [ UserDetailsComponent ],
   providers: [
      { provide: Router, useClass: RouterStub },
      { provide: ActivatedRoute, useClass: ActivatedRouteStub }
   ]
})
```

Handling Component Dependencies

- Testing the navigation
  - Get Router object from TestBed
  - Spy on ‘navigate()’ method
  - Invoke component’s ‘save()’ method
  - Verify if ‘router.navigate()’ method is called with ‘/users’ route

- Handling route parameters
  - Define a Subject observable object within ActivatedRoute stub class
  - Add public ‘push()’ method. Call ‘next()’ method on subject instance to propagate the parameter to the observer
  - Add public ‘params’ getter that returns subject instance as an observable
  - Within the test, call ‘push()’ and pass an object with ‘id’ property set to 0
  - Verify if ‘router.navigate()’ method is called with ‘not-found’ route

- Testing RouterOutlet components
  - Verify if the component template contains ‘RouterOutlet’ directive
  - Verify if the component template contains ‘RouterLinkWithHref’ directive
  - Add ‘RouterTestingModule’ to ‘imports’ array of testing module

- Misc Tests
  - Shallow Component Tests
    - NO_ERRORS_SCHEMA
  - Testing Custom Directives

Testing Async Operations

- async()
  - Runs the body of a test (it) or setup (beforeEach) function within a special async test zone
- ComponentFixture.whenStable()
  - Returns a promise that resolves when the fixture is stable
  - To resume testing after completion of asynchronous activity or asynchronous change detection, hook that promise
- fakeAsync()
  - Runs the body of a test (it) within a special fakeAsync test zone, enabling a linear control flow coding style
- tick()
  - Simulates the passage of time and the completion of pending asynchronous activities

Q & A

- Thank you!