Testing Angular Components and Services

Objectives
Key objectives of this chapter ■ Angular testing technologies and setup ■ Jasmine unit test basics ■ Angular TestBed and test configuration ■ Testing external templates ■ Testing components with dependencies

1.1 Testing Angular Components

- Just like any other development project, it is important to test Angular applications
- Angular applications are modular so with the right tools and techniques it is possible to perform robust unit/integration testing
- Angular has several testing utilities built in to use
  - Several other JavaScript testing technologies are also used for testing Angular applications
- Some situations like external HTML templates and asynchronous interaction can present challenges for testing but are possible to test

1.2 Testing Tools

- **Jasmine** - The Jasmine test framework is for testing JavaScript code
  - Jasmine is probably the most important tool used, besides Angular itself, as the unit tests themselves are written according to Jasmine
- **Angular testing utilities** - Creates a test environment for application code being tested

- Can control parts of the application as they interact with Angular

- **Karma** - Helps simplify running Jasmine tests although it can also work with other frameworks
  - Karma is a Node.js tool and requires Node.js and npm installed
  - You can run your Jasmine specifications (test scripts) in multiple browsers like Firefox and Chrome
  - Karma will automatically watch all JavaScript files and re-run the tests if any one of them is modified
- **Protractor** - Simulate user activities on a browser for "end to end" testing
  - This chapter will not focus on Protractor as there is not as much low-level integration required as with Jasmine

Testing Tools

With Jasmine tests you can generate an HTML "test runner" to run in a browser without Karma. Karma can automatically generate this for running tests and run the tests in a browser automatically so Karma simplifies the testing process.

The main focus of this chapter is the Angular testing utilities and integration with Jasmine tests. Protractor testing is less Angular-specific and does not require the Angular testing utilities.

Jasmine – https://jasmine.github.io/

Karma – https://github.com/karma-runner/karma

Protractor – http://www.protractortest.org

1.3 Testing Setup

- One of the easiest ways to get setup for testing is to use the process in the Angular documentation for the setup of a project or use Angular CLI
- Karma requires Node.js and npm although you most likely already have

these as part of regular Angular setup

- Although Jasmine is most important for writing the tests, the setup is mostly about Karma and how it hosts Angular files and runs Jasmine tests
- Important configuration files:
  - **package.json** – Development dependencies like Karma, Jasmine, Protractor and lite-server along with various script definitions
  - **karma.conf.js** – Main config file for Karma with some basic configuration including the Karma/Jasmine integration
  - **karma-test-shim.js** – Prepares Karma for the Angular testing environment and runs Karma, loads systemjs.config.js
  - **src/systemjs.config.js** – Same file used to load SystemJS in testing as when running the Angular application in a browser

Testing Setup

This chapter will focus on using the Angular project setup (or "quickstart") to setup for testing although Angular CLI setup should be similar.

The details of some of the configuration files above are not provided but can be found in the documentation of the relevant framework. This slide is mainly to be familiar with some of the files important in setting up testing.

There is also a 'protractor.config.js' file for Protractor configuration although that is not covered here.

Karma can integrate with other frameworks besides Jasmine but the Angular testing configuration uses this combination.

1.4 Important Test Configuration Settings

- **package.json** has several scripts that are related to testing
  - **'test' & 'test:once'** – Most important, builds the app then runs Karma from the Karma config file

    - '**e2e**' – Starts up a server for "end 2 end" tests and runs Protractor
- The Karma configuration in **karma.conf.js** has a few settings that are important or might need customization
    - **appBase** variable – location of transpiled application JavaScript and map files, often 'src/' unless the build process puts them somewhere different
    - **testingBase** variable – location of application components only defined for the testing environment, often 'testing/'
    - '**frameworks**' and '**plugins**' properties – Karma-Jasmine integration
    - '**files**' array – All the framework and application code loaded in the test environment, the file patterns that use 'appBase' or 'testingBase' as a root will pick up application code
    - '**port**' property – The local port the Karma process will host files from
    - '**browsers**' array – Array of browsers used for testing
    - '**autoWatch**' and '**usePolling**' properties – Behavior of Karma when left running to watch application files to pick up changes

Important Test Configuration Settings

The main 'test' script also does a build and watches the source files for any changes. It leaves Karma running to automatically pick up those changes to simplify the process.

By default the '**autoWatch**' property is present in the configuration and set to true. The '**usePolling**' property is typically not present and therefore defaults to false. There have been issues where the file watching does not properly pick up changes, even with the 'autoWatch' property set to true. **If you see problems where the running Karma process does not pick up changes to application or test files**, try adding the 'usePolling' property and set it to true. One example of this would be something like adding and compiling a new unit test definition but the number of tests run and reported by Karma doesn't change. Unfortunately the 'usePolling' property is not very well documented in Karma.

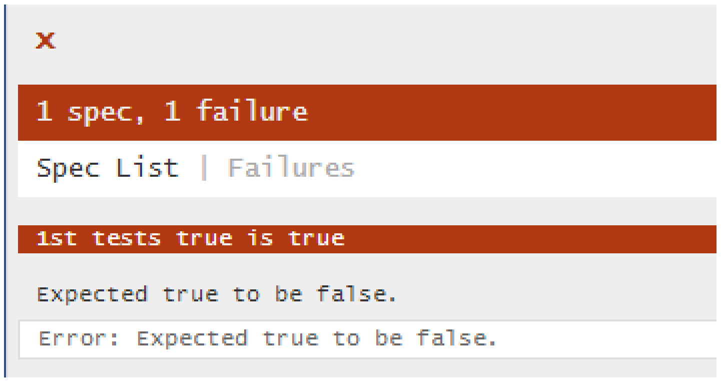
By default, the Karma configuration is just setup for Chrome browser testing. You could configure other supported browsers to be used during testing also although this would also require some Karma browser "launcher" package(s) to be configured in package.json. Consult Karma documentation for more.

1.5 Typical Testing Process

- Typical steps involved with testing
  - Define application code to be tested
  - Create code to define the tests
    - By default, a filename ending in '.spec.ts' identifies it as testing code
    - Typically in the same folder as the code being tested with the same filename except adding '.spec' to the end of the filename

`banner.component.ts is tested by banner.component.`**`spec`**`.ts`

  - Run the '**npm test**' command to build code and run the Karma tests
  - Examine the test output for any test failures



```
x
1 spec, 1 failure
Spec List | Failures
1st tests true is true
Expected true to be false.
Error: Expected true to be false.
```

  - Repeat the process to fix or add application or test code

1.6 Jasmine Test Suites

- A tests suite in Jasmine is defined with the **describe** function, which takes two parameters:
  - A string, which acts as a title (it is displayed on the spec runner page).
  - A function that implements the test suite:

```
describe("Test Suite #1", function() {
 // . . . unit tests (specs) or other describe functions
});
```

- A test suite acts as a container for:
  - Actual unit tests
  - Other test suites
    - So, you may have a hierarchy of nested test suites.
- The test suites are linked to the spec runner page via the <script> tag.
- You may also have multiple test suites in the same JavaScript file.
- Test suites help with grouping related tests in large projects.

1.7 Jasmine Specs (Unit Tests)

- In Jasmine, a unit test is referred to as a **spec**.
- Specs are defined by the **it** function which takes two parameters:
  - A string, which acts as a title (it is displayed on the spec runner page facilitating the behavior-driven development).
  - A function that implements the actual unit test.
- Variables declared in the parent *describe* function (the test suite container) are visible in the unit tests (*it* function(s))

- A spec is a container for one or more expectations (assertions) about the code outcome (pass/fail).

1.8 Expectations (Assertions)

- Expectations are defined by the **expect** function that performs an assertion on the expected value of a variable or a function passed to the *expect* function as a parameter.
  - The *expect* function's parameter is called the **actual**.
- An expectation is evaluated to either *true* or *false.*
  - An expectation evaluated to *true* is treated as test success (test passed).
  - A *false* result is treated as a failed unit test.
- The *expect* function is chained with a *matcher* function, which takes the *expected* value.
- Example:

```
expect(<actual>).toBe(<expected value>);
```

where *toBe* is a *matcher* function chained to the *expect* function via the dot-notation.

  - **Note**: You can also chain other matchers.

1.9 Matchers

- A **matcher** is a function that performs a boolean comparison between the actual value (passed to the chained *expect* function) and the expected value (passed as a parameter to the *matcher*).
- Jasmine comes with a rich catalog of built-in matchers:

```
toBe                       toBeUndefined
toBeCloseTo                toContain
toBeDefined                toEqual
toBeFalsy                  toHaveBeenCalled
toBeGreaterThan            toHaveBeenCalledWith
toBeLessThan               toMatch
toBeNaN                    toThrow
toBeNull                   toThrowError
toBeTruthy
```

- Developers can also create their own (custom) matchers.

Matchers

toBe compares using the triple equal sign: ===

toEqual works for simple literals and variables

toMatch is used for regular expressions

toBeDefined and **toBeUndefined** compare against *undefined*

toBeNull compares against null

toBeTruthy checks for true

toBeFalsy checks for false

toContain is used for finding an element in an Array

toBeLessThan is used for mathematical '<'

toBeGreaterThan is used for mathematical '>'

toBeCloseTo is used for precision math comparison

toThrow is used for testing if a function throws an exception

1.10 Examples of Using Matchers

- Here is a full example of a spec used to test a simple function

```
it ("Test #234", function(){

    var testFunction = function (){
        return 1000 + 1;
    };

    expect(testFunction()).toEqual(1001);
});
```

  - **Note**: The *testFunction* function would normally be placed in a separate JavaScript file of functions to be tested.

- Similarly, you can test a variable.
- **Note**: The *expect* function performs an expression evaluation, so you can assert math expressions as well, e.g.

```
expect(1000 + 1).toEqual(1001);
```

1.11 Using the not Property

- To reverse the expected value (e.g. from *false* to *true*), matchers are chained to the *expect* function via the **not** property.
- For example, the following assertions are functionally equivalent:

```
expect(<variable or a function>).toBe(true);
expect(<variable or a function>).not.toBe(false);
```

1.12 Setup and Teardown in Unit Test Suites

- Jasmine framework provides two global functions that are called before and after each spec (test) in the *describe* block: *beforeEach* and *afterEach*.
- The *beforeEach* function runs the common setup code (if needed).
- The *afterEach* function contains the tear-down code (if needed).
- Both functions take a function as a parameter.

1.13 Example of beforeEach and afterEach Functions

```
describe("A Test suite with setup and teardown", function(){
  var obj = {};

  beforeEach(function() {
    obj.prop1 = true;
    // obj.fooBar - undefined !
  });

  afterEach(function() {
    obj = {};
  });

  it("has one property, for sure", function() {
    expect(obj.prop1).toBeTruthy(); // is true, indeed
  });

  it("has undefined property", function() {
    expect(obj.fooBar).toBeUndefined();
  });
});
```

Example of beforeEach and afterEach Functions

Non of the code above is specific to Angular. This just shows the Jasmine-related code and the structure of a Jasmine test.

1.14 Angular TestBed

- The Angular **TestBed** class is the most important Angular testing utility
- The **TestBed.configureTestingModule** method produces a module environment for the class to be tested
  - This dynamically constructed Angular module is just for defining the test environment
  - This method takes a metadata object similar to the one used by Angular @NgModule classes
- The **TestBed.createComponent** method creates an instance of a component and returns a **ComponentFixture** object
  - All test configuration must be done before creating Angular components
- A **ComponentFixture** object is a handle on the test environment surrounding the created component
  - This allows access to the component itself and to a **DebugElement** that is the component's DOM element
  - Allows the test to check anything that is needed to determine if the test has "passed"
- You tell Angular when to perform change detection with the **ComponentFixture.detectChanges** method
  - Allows modification of a component for a test before triggering Angular

Angular TestBed

The components DOM element represented by the **DebugElement** in the **ComponentFixture** replicates what the component would cause a browser to do in the structure or properties of the document when run as part of the application in a browser. Access to this and the component instance itself will allow tests to confirm either or both to decide if the component behaves as expected and therefore if the test should "pass".

1.15 Typical Test Structure

- Imports of Angular testing classes and application components
- Jasmine 'describe' function containing all of the test code
  - Variables to be initialized and then manipulated in tests
  - 'beforeEach' function(s) with test environment configuration
  - 'it' function(s) to define individual tests
    - Modifying component properties and/or taking actions with components
    - Defining expectations by querying Angular objects to check for a passing test
- There will be lots of nested function declarations so balanced parenthesis and curly bracket syntax is a common source of errors

1.16 Example of Basic Angular Test

```
import { ComponentFixture, TestBed } from
'@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';
import { BannerComponent } from './banner-
```

```
inline.component';
describe('BannerComponent (inline template)', () => {
  let comp:    BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;
  let de:      DebugElement;
  let el:      HTMLElement;

  beforeEach(() => { ... });  // shown next slide
  it('should display original title', () => {
    fixture.detectChanges();
    expect(el.textContent).toContain(comp.title);
  });
  it('should display a different test title', () => {
    comp.title = 'Test Title';
    fixture.detectChanges();
    expect(el.textContent).toContain('Test Title');
  });
});
```

Example of Basic Angular Test

The 'By' class is used in a "predicate" that determines what will match for the component in the DOM tree. This lets the tests determine if the DOM objects have changed as they should to indicate the test passes.

Notice in the code above that there are several places JavaScript functions are passed as parameters. Above this is typically done with the "arrow" syntax of '() => { … }'. This is the second parameter of the 'describe' and 'it' functions and the first parameter of the 'beforeEach' function. This can make syntax difficult since you will have several nested function declarations with sets of parenthesis and brackets.

1.17 Basic beforeEach Configuration

```
beforeEach(() => {
  // pass object with module @NgModule properties
  // declare the test component
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });

  // create component and store ComponentFixture
  fixture = TestBed.createComponent(BannerComponent);

  // BannerComponent test instance
  comp = fixture.componentInstance;

  // query for the title <h1> by CSS element selector
  // store DebugElement and HTMLElement for test usage
  de = fixture.debugElement.query(By.css('h1'));
  el = de.nativeElement;
});
```

1.18 Automatically Detecting Component Changes

- By default, Angular will not detect changes in components and trigger the Angular lifecycle unless the **ComponentFixture.detectChanges** method is called
  - You can change this for a specific test using the '**ComponentFixtureAutoDetect**' class
- Import the class into the test

```
import { ComponentFixtureAutoDetect } from
'@angular/core/testing';
```

- Change the test module configuration to use it as a 'provider'

```
TestBed.configureTestingModule({
  declarations: [ BannerComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
})
```

- Being able to precisely control when Angular performs change detection is often beneficial though so this ability is not often used in tests

Automatically Detecting Component Changes

Tests are often written so that several properties of a component are changed and then an expectation is checked to see the behavior of the component. Requiring a call to 'detectChanges' allows the test to indicate when it is ready for Angular to pick up the component changes introduced for the test.

1.19 Testing External Templates

- Most Angular components have HTML templates in external files
  - This can make testing difficult as configuring components with this information is an asynchronous process
- You must have an initial '**beforeEach**' that calls the '**compileComponents**' method after test module configuration and waits for this asynchronous process to finish
  - Then a synchronous '**beforeEach**' can create the tested component
  - Compilation must happen after test module configuration
- Import the '**async**' function from Angular testing module

```
import { async } from '@angular/core/testing';
```

- Add a new '**beforeEach**' at the top of the test code that calls the '**async**' function

- Move the test module configuration into the asynchronous '**beforeEach**' and call '**compileComponents**' on the configured TestBed

```
// async beforeEach, make sure this is first
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ]  })
  .compileComponents();  // compile template and css
}));
```

Testing External Templates

Note that since '**async**' is a function, there is an extra set of parenthesis around the entire function which does the test module configuration.

Although the synchronous 'beforeEach' (that was used before) is not shown above, make sure the asynchronous one is first in the test code. The 'beforeEach' methods will execute in order and this will guarantee the component is completely configured and compiled with external template files before the component is created.

The synchronous 'beforeEach' still creates the component and initializes some of the variables in the test to reference the tested elements. The only thing that changes is where the testing module configuration is done and the new call to the 'compileComponents' method.

```
// synchronous beforeEach
beforeEach(() => {
  fixture = TestBed.createComponent(BannerComponent);

  comp = fixture.componentInstance; // BannerComponent test instance

  // query for the title <h1> by CSS element selector
  de = fixture.debugElement.query(By.css('h1'));
  el = de.nativeElement;
});
```

1.20 Testing Components With Dependencies

- Many Angular components depend on Angular services to function
  - This makes testing these components difficult
- There are different approaches for testing these components
  - Provide service test "doubles" or "stubs"
    - This is a completely fake version of the service
  - Use the real service but "spy" on the method(s) of interest
    - You can define the behavior of the required methods without needing a completely fake service

1.21 Getting Injected Services

- A test will often need to refer to a service that has been injected
  - You might need to change properties of the service and then test that the component behaves properly in those conditions
- Even if you have an object created in the test that is a stand-in for the service, it should not be used
  - The instance actually injected by Angular into the component being tested will be different with Angular functionality wrapped around it
- You should always use a '**get**' method to obtain the injected service from Angular
  - This method takes a parameter for the class being requested
- The method that will always work is to get the injected service from the injector of the component being tested

```
userService =
fixture.debugElement.injector.get(UserService);
```

1.22 Getting Injected Services

- It is also possible to call '**get**' directly on TestBed but this may return a different service instance in some cases
  - If the root testing module is not the only provider of the service, this method would return a different instance than the one injected into the component

```
userService = TestBed.get(UserService);
```

- Another mechanism to obtain injected services is the '**inject**' function which is an Angular testing utility function
  - This also uses the TestBed injector so the same warnings apply
  - The function takes two parameters, an array of Angular dependency injection tokens and a test function with parameters that match the array contents

```
inject([Router], (router: Router) => { ... } )
```

Getting Injected Services

The problem with getting a different instance of the service than is injected in the component is that you may get false test failures. If you change the state of a service instance that is not injected into the component, the component will not see this and will not exhibit any change in behavior. This may cause the test to fail but the reason for the failure is perhaps not that the tested component behaves incorrectly but that the test is not modifying the service instance injected into the component and therefore not triggering any change in the component.

1.23 Testing With Dependencies – Test Double

- If a service dependency is fairly simple, it may be possible to define a completely "fake" version and use that instead of the real service
  - This "Test Double" or "stub" would be defined as part of the test
- Define the "Test Double" as an object local to the test, often done in test setup

```
userServiceStub = {
  isLoggedIn: true,
  user: { name: 'Test User'}
};
```

- Modify the testing module configuration to use the test double as the service provider instead of the actual service

```
  TestBed.configureTestingModule({
     declarations: [ WelcomeComponent ],
     // Provide a test-double instead of the real service
     providers:    [
       {provide: UserService, useValue: userServiceStub } ]
  });
```

- Store the injected service in a variable to be used as part of the tests

```
userService =
fixture.debugElement.injector.get(UserService);
```

Testing With Dependencies – Test Double

If a service is used quite frequently in several tests, it might be useful to centralize the test double service code in the 'testingBase' source code location of Karma so it is picked up for testing but is not part of the actual application code.

Complete example of setup and tests (imports and 'describe' function not shown):

```
beforeEach(() => {
  // stub UserService for test purposes
  userServiceStub = {
    isLoggedIn: true,
    user: { name: 'Test User'}
  };

  TestBed.configureTestingModule({
     declarations: [ WelcomeComponent ],
     providers:    [ {provide: UserService, useValue: userServiceStub } ]
  });

  fixture = TestBed.createComponent(WelcomeComponent);
  comp    = fixture.componentInstance;

  // UserService from the root injector
  userService = TestBed.get(UserService);

  //  get the "welcome" element by CSS selector (e.g., by class name)
  de = fixture.debugElement.query(By.css('.welcome'));
  el = de.nativeElement;
});

it('should welcome the user', () => {
  fixture.detectChanges();
  const content = el.textContent;
  expect(content).toContain('Welcome', '"Welcome ..."');
  expect(content).toContain('Test User', 'expected name');
});

it('should welcome "Bubba"', () => {
  userService.user.name = 'Bubba'; // welcome message hasn't been shown yet
  fixture.detectChanges();
  expect(el.textContent).toContain('Bubba');
});

it('should request login if not logged in', () => {
  userService.isLoggedIn = false; // welcome message hasn't been shown yet
  fixture.detectChanges();
  const content = el.textContent;
  expect(content).not.toContain('Welcome', 'not welcomed');
  expect(content).toMatch(/log in/i, '"log in"');
});
```

1.24 Testing With Dependencies – Spy

- Rather than trying to create a complete, alternate definition of a service, the real service can be injected with a Jasmine "Spy" modifying the behavior of any methods used
- Declare a variable in the test to store the Jasmine Spy

```
let spy: jasmine.Spy;
```

- Inject the real service in test module configuration

```
TestBed.configureTestingModule({
   declarations: [ WelcomeComponent ],
   providers:    [ UserService ]  // provide real service
});
```

- Get the injected service

```
userService =
fixture.debugElement.injector.get(UserService);
```

- Configure the Spy behavior for the relevant method of the service

```
// Setup spy on the `isLoggedIn` method
spy = spyOn(userService, 'isLoggedIn')
      .and.returnValue(false);
```

Testing With Dependencies – Spy

The creation of spy objects has nothing specifically to do with Angular. This is defined completely by Jasmine. Although details are not provided here, there is nothing different from "normal" Jasmine you would do to configure Spies. The most important thing related to Angular is to make sure to spy on the injected service instance.

1.25 Testing With Asynchronous Dependencies

- Testing a component with asynchronous dependencies is difficult because the delay in interaction during a test can cause a false failure
  - The test must wait for the asynchronous process to finish
- First, you might need to return a Promise from a Spy to replicate the behavior

```
spy = spyOn(twainService, 'getQuote')
      .and.returnValue(Promise.resolve(testQuote));
```

- There are different methods to handle asynchronous interaction in tests
  - Using the Angular '**async**' function and the '**whenStable**' function

```
it('getQuote promise (async)', async( () => {
  fixture.detectChanges();
  // wait for async getQuote
  fixture.whenStable().then( () => {
    // update view with quote after the asynchronous return
    fixture.detectChanges();
    expect(el.textContent).toBe(testQuote);
  });
}));
```

Testing With Asynchronous Dependencies

Besides dealing with the asynchronous interaction, it is also common to use a test double or spy on the dependency to include the interaction in the test. This is un-related to dealing with the asynchronous interaction.

The '**whenStable**' function returns a Promise that resolves when all pending asynchronous activities complete. That allows this to be used even when the test doesn't have direct access to the Promise used by the component being tested.

1.26 Testing With Asynchronous Dependencies

- Using the Angular '**fakeAsync**' function and the '**tick**' function

```
it('getQuote promise (fakeAsync)', fakeAsync( () => {
   fixture.detectChanges();
   // wait for async getQuote
   tick();
   // update view with quote
   fixture.detectChanges();
   expect(el.textContent).toBe(testQuote);
}));
```

- Using the Jasmine '**done**' callback

```
it('getQuote promise (done)', done => {
  fixture.detectChanges();
  // get the spy promise and wait for it to resolve
  spy.calls.mostRecent().returnValue.then( () => {
    // update view with quote
    fixture.detectChanges();
    expect(el.textContent).toBe(testQuote);
    done();
  });
});
```

Testing With Asynchronous Dependencies

The '**fakeAsync**' and '**tick**' functions are Angular testing utility functions.

The advantage of the '**fakeAsync**' method is the test appears synchronous. The asynchronous behavior is controlled with the '**tick**' method and you don't need to work with the Promise API or the 'then' method.

1.27 Testing Components With @Input and @Output

- A component with @Input and @Output is used within a "host" component
  - The test must validate the input and output binding
- A test can set input directly, manually trigger events with the '**triggerEventHandler**' function, and then check output

```
// manually wired to something that supplied a hero
expectedHero = new Hero(42, 'Test Name');
comp.hero = expectedHero;
fixture.detectChanges(); // trigger data binding

// listen then trigger event to check output
comp.selected.subscribe((hero: Hero)=>selectedHero = hero);
heroEl.triggerEventHandler('click', null);
expect(selectedHero).toBe(expectedHero);
```

- You can also test the component by creating a host component specifically for the test
  - This would use a template that uses the component being tested just like a real host component from the application
  - You would define this test host @Component directly in the test code

Testing Components With @Input and @Output

The host component will often set input for the component and then listen to events raised by the output binding.

1.28 Testing Routed Components

- Although testing components that use the Angular Router can seem daunting, remember the test is only testing the component and no the Router itself
  - In general you are only testing if the component navigates with the correct address under given conditions
- You could simply define a test double, or "stub", which echoes the URL of navigation

```
class RouterStub {
  navigateByUrl(url: string) { return url; }
}
```

- Provide the test double as an alternative to the Router in the test module configuration

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

- The test simply spies on the method and checks the URL

```
const spy = spyOn(router, 'navigateByUrl');
// simulate a click
const navArgs = spy.calls.first().args[0];
expect(navArgs).toBe('/heroes/' + id);
```

1.29 Summary

- Jasmine and Karma are the two main technologies used in Angular component unit testing
  - Test code is written using Jasmine
  - Karma provides an automation for configuring and running the tests
- Jasmine tests are defined within 'describe' functions that have 'beforeEach' functions for test setup and 'it' functions for the actual tests
- The Angular TestBed class is the most important class for test configuration and execution
- Common things for Angular components, like external templates and service dependencies, can make testing more complex but not impossible