Advanced Angular Testing

Agenda

Recap: Unit Testing

Integration Tests

Http & Async Testing

Marble Testing

RxJS Testing

Using Jest for Unit Testing

End-2-End Testing using Cypress

Recap: Unit Testing in Angular

Test Runners

Execute Tests written by using a Testing Framework

Popular Testrunners are:

- Unit Testing:
  - Karma (Default Angular Test Runner)
  - Jest
  - Wallaby.js (not free)
- E2E / UI Testing
  - Protractor
  - Cypress

Jasmine

Popular Testing Framework used by Angular CLI by default

Jasmine has the following features:

- Easy-to-read (expressional) syntax
- Testing async code
- Spies (mocking objects and methods)
- DOM testing
- Jasmine Testing Syntax is JEST compatible

Jasmine Overview

Suite – a suite of related tests, created using "describe"

Setup and teardown

- beforeEach() and
- afterEach() methods

Specs – expectations to test for, created using "it“

- use Matchers

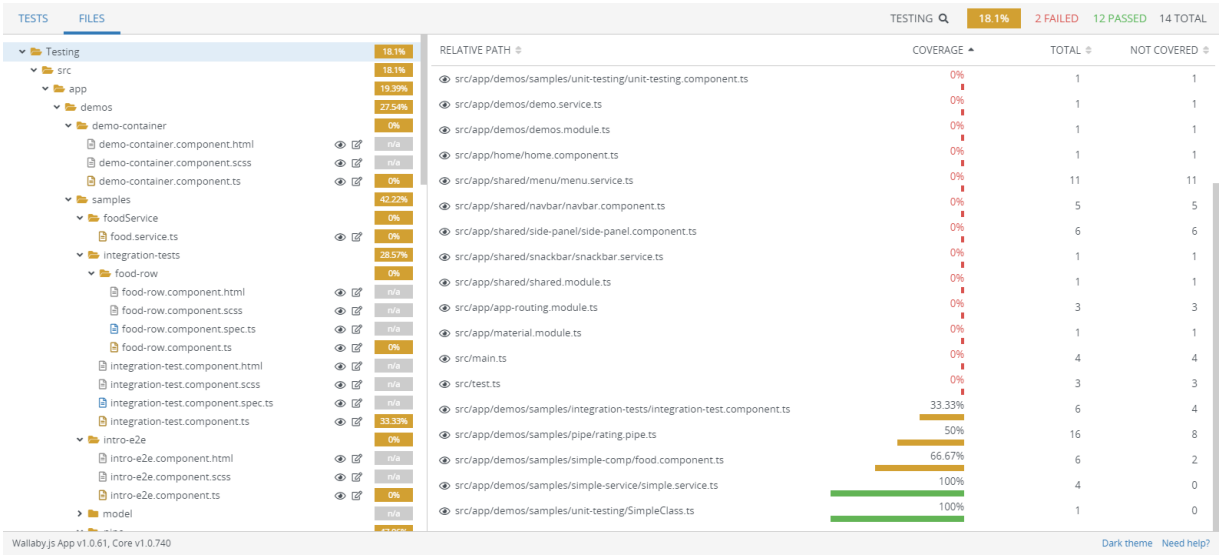
```
import { SimpleMessageService } from './simple.service';

describe('Testing a simple Service: SimpleMessageService', () => {
  let service: SimpleMessageService;

  beforeEach(() => {});

  it('should have no messages to start', () => {
    service = new SimpleMessageService();

    expect(service.messages.length).toBe(0);
  });

  it('should add a message when add is called', () => {
    service = new SimpleMessageService();
    service.add('message1');

    expect(service.messages.length).toBe(1);
  });
```

Matchers

- toBe / toEqual
- toMatch: RegExp match()
- toBeDefined / toBeUndefined
- toBeNull
- toContain
- toBeLessThan/toBeGreaterThan
- toThrow: for catching expected exceptions

Coverage

- Is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs
- Expresses the amount of Code that is tested
- Several Coverage Reports available
- Can be use as Quality Gate for DevOps



Pipes & Services

Testing Pipes

Angular pipes are the simplest ones to test

Pipes take the input, transform it and gives the output

```
transform(value: any, args?: any): any {
  if (undefined !== value && value.length === 10) {
    return (
      '(' +
      value.substring(0, 3) +
      ') ' +
      value.substring(3, 6) +
      ' ' +
      value.substring(6)
    );
  }
  return '';
}
```

```
it('should display in phone format', () => {
  const phoneNumber = '3333333333';
  const pipe = new PhonenumberPipe();
  const result = pipe.transform(phoneNumber);

  expect(result).toBe('(333) 333 3333');
});
```

Services

Two kind of Services:

- With no depencies
  - Simple to test – just like a class
- With dependencies / injections -> ie HttpClient
  - Use (manual) Mocks
  - Use Spies
  - Use HttpClientTestingModule

Mocking

Mocking

Mocking is primarily used in unit testing

Object tested may have dependencies on others

Replace the other objects by mocks that simulate the behavior of the real objects.

Mocking can be done with:

- Fake objects & overriding functions
- Using a real instance with Spy

Spy

Many times you have to test Objects (Components, Services) that exprect Injections

Jasmine provides Spies to allow mocking of

- Objects
  - Methods
    - Results

Many ways to create Spy Objects:

- spyOn | spyOnProperty
- jasmine.createSpy | jasmine.createSpy

Using jasmine.createSpyObj

Spies can mock all or certain methods ob Objects

```
mockHeroService = jasmine.createSpyObj([
  "getHeroes",
  "addHero",
  "deleteHero"
]);

component = new HeroComponent(mockHeroService);
```

```
@Injectable()
export class HeroService {

  constructor(
    private http: HttpClient
  ) {}

  getHeroes(): Observable<Hero[]> {···
  }

  getHero(id: number): Observable<Hero> {···
  }

  searchHeroes(term: string): Observable<Hero[]> {···
  }


  addHero(hero: Hero): Observable<Hero> {···
  }

  deleteHero(hero: Hero | number): Observable<Hero> {···
  }

  updateHero(hero: Hero): Observable<any> {···
  }
```

Method can be called & returnValue can be set!

```
describe("delete", () => {
  it("should remove the indicated hero from the heroes list", () => {  7ms
    mockHeroService.deleteHero.and.returnValue(of(true));
    component.heroes = HEROES;

    component.delete(HEROES[2]);

    expect(component.heroes.length).toBe(2);
  });
```

Integrations Tests

Testbed

Allows Testing the Component and Template together

Creates a seperate Module only used for testing using the configureTestingModule method

Mock instances of Services are provides using the: provide -> useValue notation

Uses Fixture as a Variable to hold the Component for testing

```
describe('HeroesComponent (shallow tests)', () => {
  let fixture: ComponentFixture<HeroesComponent>;
  let mockHeroService;
  let HEROES;
```

```
TestBed.configureTestingModule({
  declarations: [
    HeroesComponent,
    FakeHeroComponent
  ],
  providers: [
    { provide: HeroService, useValue: mockHeroService }
  ],
  // schemas: [NO_ERRORS_SCHEMA]
})
fixture = TestBed.createComponent(HeroesComponent);
```

Mocking Child-Components

Child Components can tested using:

- Real Commponents (covered later)
- Mock-Child Components

Mocked Components must be added to the Test-Module

Integrations Tests

Tests that Components & Template work together as expected

- Make sure that the Component renders what is exprected with a give input
- Class and DOM testing

Renders Components using Karma

Uses Testbed and Fixtures

Shallow

- Ignores Child Components & Directives

Deep

- Includes Child Components & Directives

Integration Test Setup

Test the Integration of Components and their Templates

Uses { TestBed, async, ComponentFixture} from '@angular/core/testing'

Uses { DebugElement, Component, NO_ERRORS_SCHEMA } from '@angular/core'

Uses { By } from '@angular/platform-browser'

NO_ERRORS_SCHEMA

Suppresses Errors in Angular Modules to avoid Error Messages related to unknown:

- Attributes & Properties
- Directions
- Child Components
- Injections

Use with care because it also suppresses Errors that you might want to see

Testing the DOM

Two approaches

- fixture.nativeElement
  - Refers to the DOM Element
  - nativeElement.querySelector(‘a‘).textContent
- fixture.debugElement
  - Uses By-Library (from "@angular/platform-browser“) to query
    - By.css
    - By.directive

Can access Directives

Use fixture.detectChanges() to apply bindings

Http & Async Testing

Http Testing

Used to test the http operations of a Service

Use classes from @angular/common/http/testing:

- HttpClientTestingModule
  - Allows you to easily mock HTTP requests by providing you with the HttpTestingController service
- HttpTestingController
  - Mock requests instead of making real API requests to our API back-end when testing

Using HttpTestingController

Testing Controller allows us to mock the result of an async operation using:

- controller.expectOne
  - can set http-verb
- flush
  - set response
- contoller.verify()
  - make sure all async operations are completed

```
18  beforeEach(() => {
19    TestBed.configureTestingModule({
20      imports: [HttpClientTestingModule],
21      providers: [FoodService]
22    });
23
24    service = TestBed.get(FoodService);
25    controller = TestBed.get(HttpTestingController);
26  });
27
28  it('should be created & made the initialized the data', done => {  15ms
29    expect(service).toBeTruthy();
30
31    //setup the service mock
32    const req = controller.expectOne(`${environment.apiUrl}food`);
33    expect(req.request.method).toEqual('GET');
34
35    //flushing down mock data
36    req.flush(data);
37    //make sure all requests are completed
38    controller.verify();
39
40    //testing a private method - use any type to fool the compiler
41    let fs: any = service;
42    fs.setState(data);
43
44    service.getItems().subscribe(data => {
45      expect(data.length).toEqual(2);
46      done();
47    });
48  });
```

Async Testing

When testing Components with async Implementations (very likely when working with Observables) we often have to deal with problems arising out of latency

Three way of handling this problems

- Jasmine done() funtion together with spy callbacks
  - Problem: We have to know & handle all Promises & Observables & other async
- async ... whenStable
- fakeAsync & tick
  - Let's us write linear test code

Marble Testing

Marble Testing

RxJS marble testing is a great way to test Observables

Mocks observable and subscription

Uses scheduler to provide timing (10 frames)

Serveral Libs based on TestScheduler:

- jasmine-marbles
- jest-marbles

TestScheduler

Introduced in RxJS 6 to provide build-in Marble Testing

Two main creation methods:

- createColdObservable
- createHotObservable

For Testing

- flush
- expectObservable | expectSubscription

Marble Testing Syntax

- simulate the passage of timea-z (a to z): represent an emission,
  -a--b---c stands for “emit a at 20ms, b at 50ms, c at 90ms”

| emit a completed (end of the stream),
  ---a-| stands for emit ‘a’ at 40ms then complete (60ms)

indicate an error (end of the stream),
  --a--# emit a at 40ms then an error at 70ms

( ) group multiple values together in the same unit of time,
  --(ab|) stands for emit a b at 40ms then complete (40ms)

^ indicate a subscription point,

- ^-- subscription starting at ^

! indicate the end of a subscription point,

- ^--! subscription starting at ^ and ending at !

Hot vs Cold Observalbes

Two types of Observables exist

- Cold Observables
  - Need Observer to emit values
- Hot Observalbes
  - Don't need Subscriber to emit values

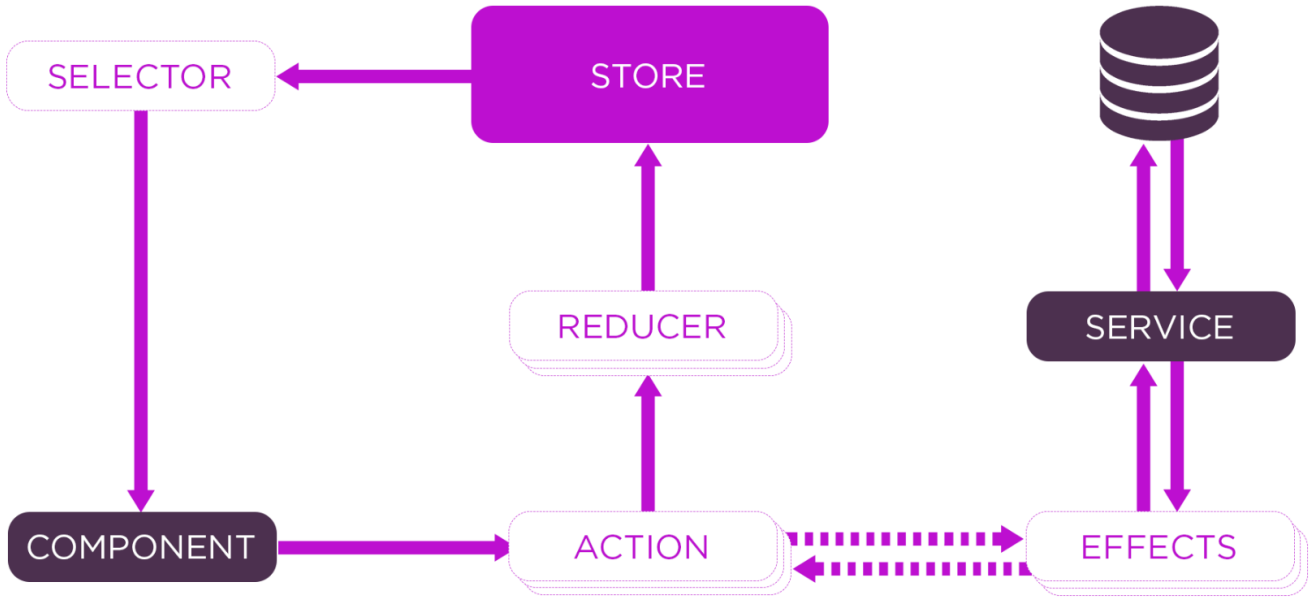
```
it('should multiply by "2" each value emitted', () => {
  const values = { a: 1, b: 2, c: 3 };
  const expectedVals = { a: 2, b: 4, c: 6 };
  const source = cold('-a-b-c-|', values);
  const expected = cold('-a-b-c-|', expectedVals);

  const result = source.pipe(
    map(x => x * 2),
    tap(console.log)
  );

  expect(result).toBeObservable(expected);
});
```

NgRx Testing

What to Test in NgRx

In RxJS usually we usually test the following items:

- Reducers
- Selectors
- Effects



Testing Reducers

Always test for unknow actions to retur default state

Are your Reducers that complex

When using ngrx Entity it makes more sense to test

Component interaction

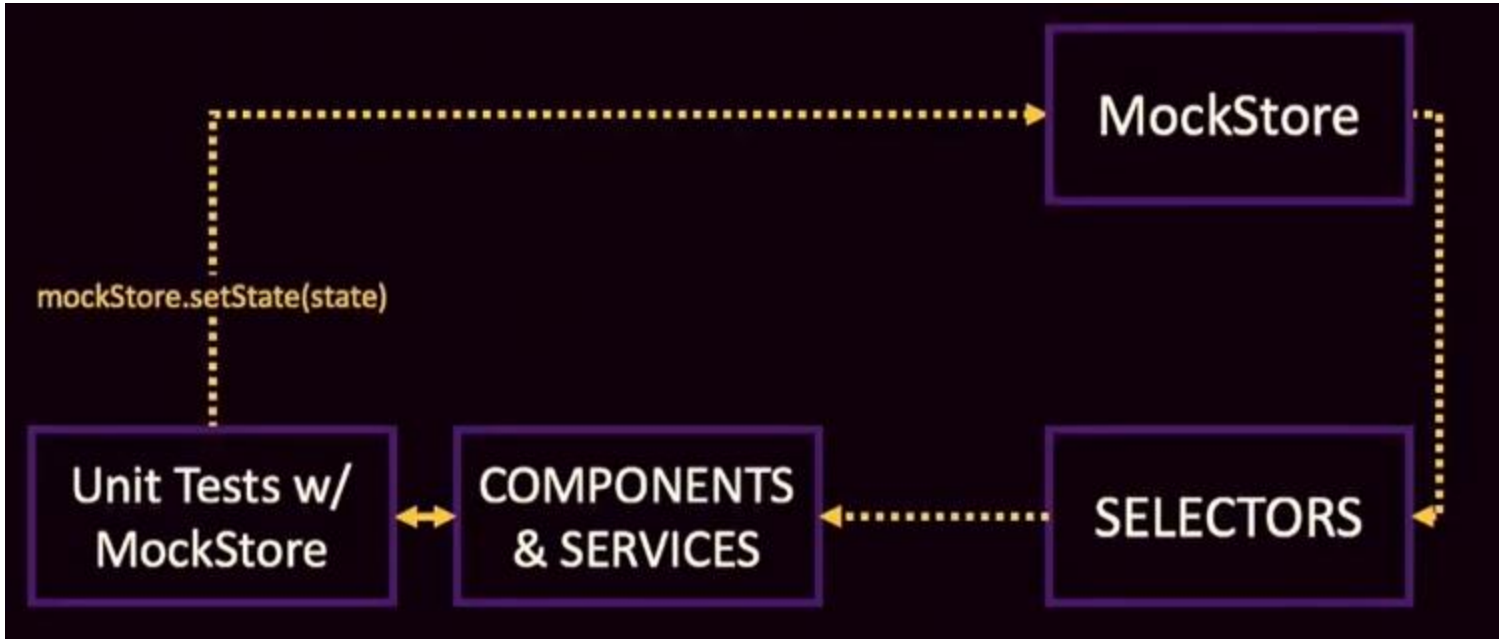
```
 8  describe('Demos Reducer', () => {
 9    describe('an unknown action', () => {
10      it('should return the previous state', () => {  6ms
11        const action = {} as any;
12        const result = DemosReducer(initialState, action);
13        expect(result).toBe(initialState);
14      });
15    });
16
17    describe('[Demos] Add Demo', () => {
18      it('should return a new state with the added item', () => {
19        let item = {···
27        };
28
29        let designated = {···
39        };
40
41        const action = new AddDemo(item);
42        const result = DemosReducer(initialState, action);
43        expect(result.entities).toEqual(designated);
44      });
```

MockStore

@ngrx/store/testing v7 provides a MockStore that makes RxJS Testing easier

Provides useful methods:

- setState()
- overrideSelector()



Test AuthGuard using MockStore

```spec_navigation.spec.ts
describe('Auth Guard', () => {
  let guard: FBAuthGuard;
  let store: MockStore<AuthState>;

  const initialState = {
    user: null,
    token: null,
    isLoggedIn: false
  };

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [FBAuthGuard, provideMockStore({ initialState })]
    });

    store = TestBed.get(Store);
    guard = TestBed.get(FBAuthGuard);
  });

  it('should return false if the user state is not logged in', () => {  20ms
    const expected = cold('(a|)', { a: false });

    expect(guard.canLoad()).toBeObservable(expected);
  });

  it('should return true if the user state is logged in', () => {  10ms
    store.setState({
      user: {},
      token: null,
      isLoggedIn: false
    });
    const expected = cold('(a|)', { a: true });
    expect(guard.canLoad()).toBeObservable(expected);
  });
});
```

```auth.guard.ts
import { Injectable } from '@angular/core';
import { CanLoad } from '@angular/router';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';
import { map, take } from 'rxjs/operators';
import { AuthState } from './store/reducers/auth.reducer';
import { LoginRedirect } from './store/actions/auth.actions';
@Injectable({
  providedIn: 'root'
})
export class FBAuthGuard implements CanLoad {
  constructor(private store: Store<AuthState>) {}

  canLoad(): boolean | Observable<boolean> | Promise<boolean> {
    return this.store
      .select(appState => appState.user)
      .pipe(
        map(fbUser => {
          if (!fbUser) {
            this.store.dispatch(new LoginRedirect());
            return false;
          }
          return true;
        }),
        take(1)
      );
  }
}
```

JEST

JEST

Jest is a delightful JavaScript Testing Framework with a focus on simplicity

Works with any current JS based Framework

Features:

- Zero Config
- Snapshots
- Isolated

Why Jest

Sensible faster; parallelized test runs

Snapshot testing; to make sure your UI does not change unexpectedly

Rich CLI options; only run failed tests, filter on filename and/or test name, only run related tests since the latest commit

Readable and useful tests reports

Sandboxed tests; which means automatic global state resets

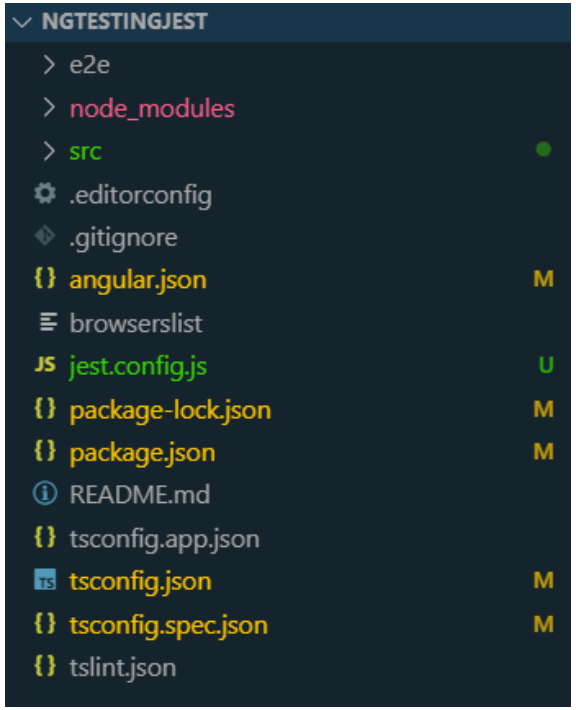
Built in code coverage

Personally I also like that I don’t need a browser in order to run my tests

JEST Installation

Two ways to install JEST

- Manual
  - Remove Karma and Related
  - Add JEST
  - Detailed Instructions in readme.md
- Using Schematics
  - npm install -g @briebug/jest-schematic
  - ng g @briebug/jest-schematic:add
  - OR
  - ng add @briebug/jest-schematic



Jest Mocking

Similar to Jasmine Mocking ...

- Jest provides jest.spyOn to replace jasmin.spyOn

If you want to use advanced Mocking use

- Spectator
- https://github.com/ngneat/spectator


```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [IntegrationTestComponent, FoodRowComponent, RatingPipe],
    imports: [HttpClientTestingModule],
    schemas: [NO_ERRORS_SCHEMA],
    providers: [FoodService]
  });

  fixture = TestBed.createComponent(IntegrationTestComponent);
  service = TestBed.get(FoodService);
});

it('component has been created', () => {
  expect(service).toBeTruthy();
});

it('should render each FoodItem as FoodItemRow', () => {
  jest.spyOn(service, 'getItems').mockImplementation(() => of(foodData));
  fixture.detectChanges();

  const rows = fixture.debugElement.queryAll(By.directive(FoodRowComponent));
  expect(rows.length).toEqual(4);
  expect(rows[0].componentInstance.food.name).toEqual('Pad Thai');
});
```

E2E using Cypress

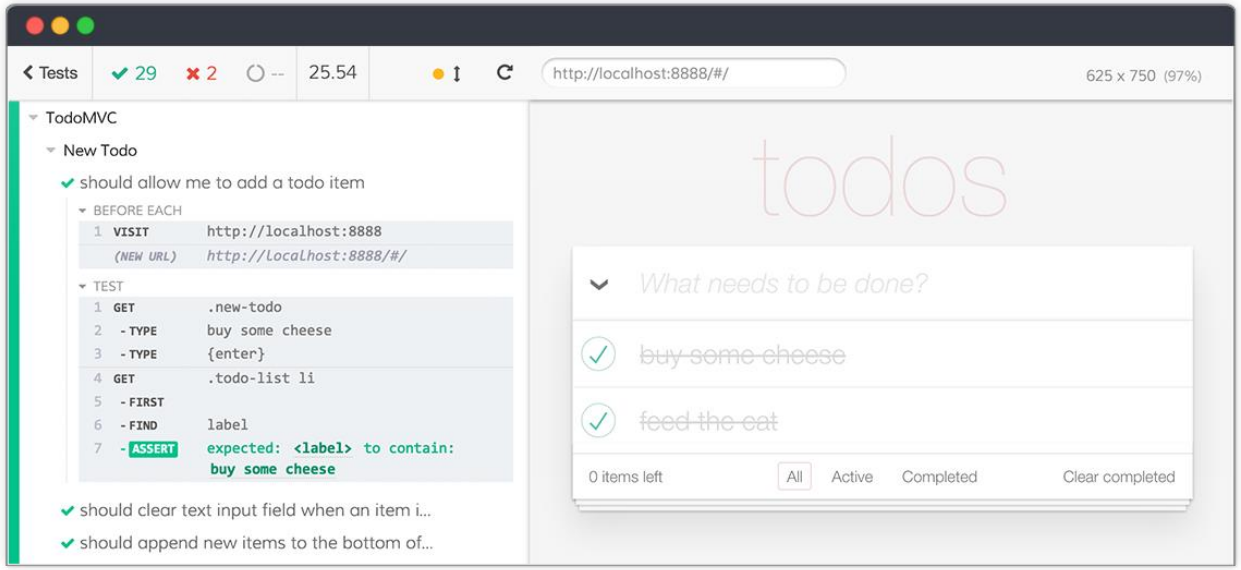
What is Cypress

Electron based End-to-End Test Runner

Alternative to Protactor

Features

- App Previw
- Before / After State
- Time Travel
- Debugging
- Custom Methods



Installation & Setup

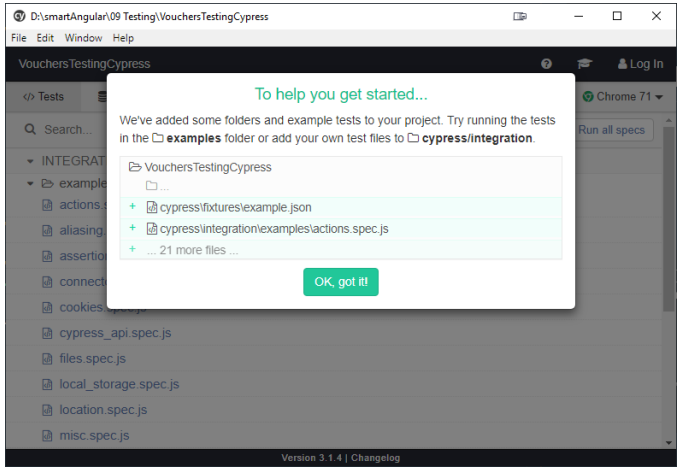
Install Cypress & Chance

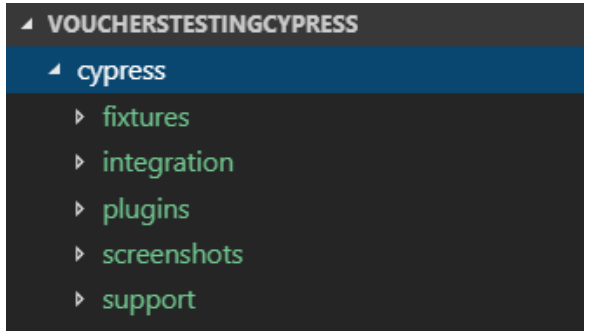
- npm i --save-dev cypress chance

Config written to cypress.json

- ie ViewPort Size

Tests written to cypress folder






```
describe('My First Test', function() {
  it('finds the content "type"', function() {
    cy.visit('https://example.cypress.io')

    cy.contains('type')
  })
})
```