Testing Policy and Guidelines

Developers
Tegan Carton-Barber
Emma Coetzer
Aeron Land
Luveshan Marimuthu
Kendra Riddle

July 2019

Contents

1 Introduction

1.1 Purpose

The purpose of this document serves to provide an overview of the testing policies and guidelines that are to be implemented and followed. This document contains two sections which define the foundations of the testing culture followed with regards to the FABI Mobile system.

2 Testing Policy

2.1 Purpose of Policy

The purpose of a well defined testing policy is to provide an overview of how testing is to be conducted. This section describes the benefits of a test policy, the objectives, evaluation, and general testing approach.

2.2 Benefits of Policy

The primary benefits of a testing policy are the following:

- Define a structured policy and guideline which provides structure
- Promotes transparency
- Encourages testing to be done continually and consistently
- Promotes effective team work

2.3 Test Policy Objectives

The objectives of the testing policy aim to accomplish the following:

- Plan tests in advance in order to ensure that testing can commence soon into the development
- Provide an indication as to the priority of certain use cases
- Promote early detection of defects
- Encourage the documentation of test cases

3 Testing Guidelines

3.1 Purpose of Guidelines

These guideline set out to inform interested stakeholders about the fundamental structure, implementation, and evaluation regarding the testing process.

3.2 Definition of Testing

Testing is a process which tests the actual results against those results which were expected. Testing is used, not only to test for errors and gaps in logic, but also internal and external aspects which relate to the system. Testing can also help identify missing functional requirements or requirements which have been incorrectly transposed into code. Testing also ensures that system integration is done correctly.

3.3 Testing Approach

The approach used is **Test Driven Development(TDD)**. The use of TDD will ensure that the development and testing team achieve good quality code and good test coverage.

The rules of TDD are:

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

TDD is beneficial in a number of ways, including:

- TDD requires extensive understanding of the functionality on the programmers part. This will help in understanding and improving the system requirements as well as writing valuable test cases.
- TDD constantly validates the implementation with respect to the tests. It will help the team to detect and remove defects, and as a result, produce high-quality code.
- TDD focuses on testing the desired functionality first, but also address the other quality aspects of the system.

This approach uses a **Red Green Refactor Cycle**:

- **Red Phase:** This phase the team must write a test on the behaviour that is about to be implemented. The test must be written before any production code is done.
- **Green Phase:** In this phase the team must write straightforward production code that makes the appropriate test case pass. The solution must be as simple as possible. Any duplicate or dirty code will be handled in the refactor phase.
- **Refactor Phase:** In this phase the team is allowed to change code. The team must be hypercritical in this phase to ensure performance and clean code is achieved.

3.4 Test Levels

The purpose of test levels is to promote decomposition in terms of tests. This allows testing to be broken into levels which start at a the lowest level, unit testing, to the highest level, acceptance testing.

Level	Owner	Objective	Key areas of Testing
Unit	Development	Detect defects code in units	Functionality and resource utilization
System	Development	Detect defects in end-to-end communications	Functionality, resource utilization, performance, reliability, portability, and interoperability
System Integration	Development	Detects defects in unit interfaces and reduces data and work flow failures	Functionality, data quality, and compatibility
Acceptance	Business	Demonstrate that the product works as expected in the target environment and detects defects in user work flow	Functionality in context of target environment

Table 1: Test Level Description

3.5 Guiding Principles

This section sets out to ensure that each phase in the testing process is performed effectively buy providing an overview of each phase that is followed.

3.5.1 Test Planning

The planning phase ensures that all required resources and test cases are ready to be used in the design and execution phases.

3.5.2 Test Design

The design phase ensure that appropriate test cases are deigned in order to maximize test coverage of the application. These tests need to be designed, reviewed, and approved before test execution can take place.

3.5.3 Test Execution

The execution phase is when the testing team will run all tests that are set out in the above phases. An informal meeting should be held in order to clarify any concerns or problems prior to the execution of this phase.

3.5.4 Test Closure

The closure phase encapsulates the review of the execution phase in order to find any defects that occurred. These defects are then prioritized.

3.6 Approach to Automated Testing

Automated testing is done in two ways:

- Automatic Unit Testing: This is done using the Jasmine Framework and the Karma task runner provided by the Angular framework.
- Automatic Integration Testing: This is done using TravisCI. Each time code is pushed to the repository TravisCI runs a series of tests ensuring that the newly added code is compatible and integrates with the rest of the system.

3.7 Test Priorities

Every test that is created and executed has a priority. These are used in order to manage task prioritization, specifically with regard to regression testing.

The priorities are classified as follows:

- **High:** This priority is used for tests that affect the core functionality of the system. These must be resolved first.
- **Medium:** This priority is used for tests that, if failed do not severely impact the system. These must be handled only after the high priority.
- **Low:** This priority is used for tests that represent ”nice-to-haves” and are not critically important.

3.8 Unit Tests

Due to the Angular framework being used for the development of the system, unit tests will need to be written for every component created, all services created and used, as well as any other elements/API calls that may need to be tested. All API calls need to be tested in the .spec.ts file of the component that uses its service.

The benefits of unit testing include:

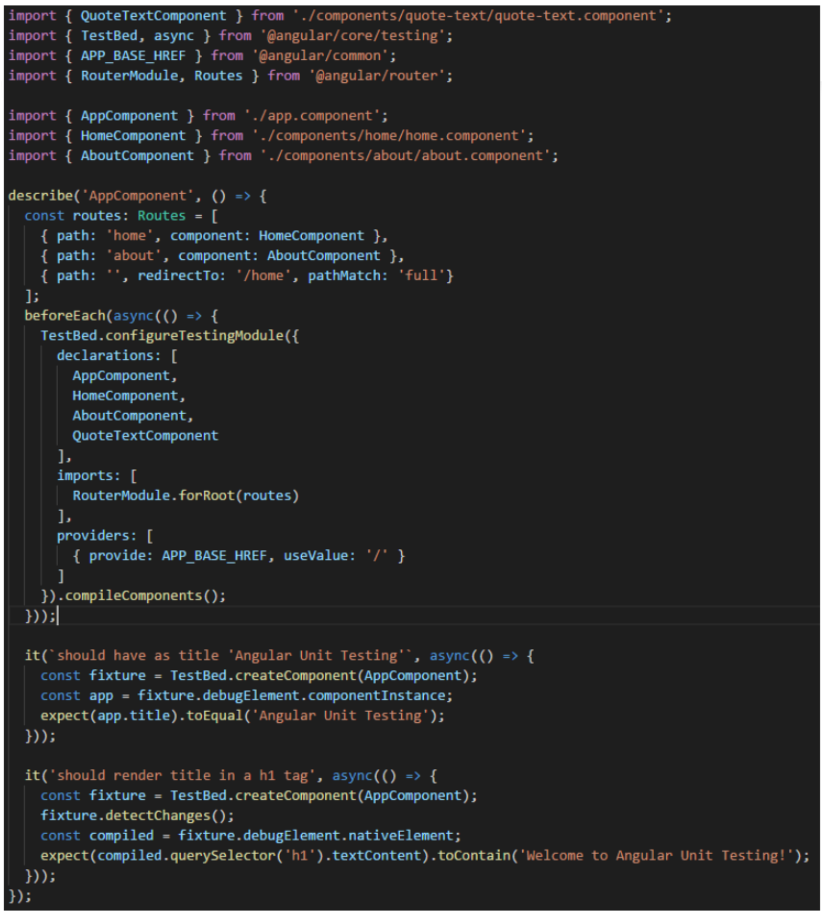
- Improvements in the design of implementations: this will ensure that the team does not start coding a feature without giving it much thought. Unit tests will enforce design thinking.
- Allows refactoring: since there will be written tests that check whether the code is executing as expected, it will be easier to ass changes to the code with the certainty that no defects are being created.
- Add new features without breaking the system: when a new feature is added, the test can be run to ensure that no other part of the system will break due to the new feature.

The Angular framework provides two tools to aid in testing, ie. Jasmine (the framework to create the tests) and Karma (the task runner for the tests). To run the created tests, using the ’ng test’ command in the terminal will execute the tests as well as open a browser to show the results of the executed tests.

The unit tests for the individual components will be written in their respective .spec.ts files. Within the .spec.ts file, the following elements will be available:

- **’describe’** is used to start the test block with the title matching the tested component name.
- The asynchronous **’beforeEach’** is used to let all the possible asynchronous code finish executing before continuing.

The below image shows an example of how the unit tests for each component should be set out:



```
import { QuoteTextComponent } from './components/quote-text/quote-text.component';
import { TestBed, async } from '@angular/core/testing';
import { APP_BASE_HREF } from '@angular/common';
import { RouterModule, Routes } from '@angular/router';

import { AppComponent } from './app.component';
import { HomeComponent } from './components/home/home.component';
import { AboutComponent } from './components/about/about.component';

describe('AppComponent', () => {
  const routes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'about', component: AboutComponent },
    { path: '', redirectTo: '/home', pathMatch: 'full'}
  ];
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent,
        HomeComponent,
        AboutComponent,
        QuoteTextComponent
      ],
      imports: [
        RouterModule.forRoot(routes)
      ],
      providers: [
        { provide: APP_BASE_HREF, useValue: '/' }
      ]
    }).compileComponents();
  }));

  it(`should have as title 'Angular Unit Testing'`, async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app.title).toEqual('Angular Unit Testing');
  }));

  it('should render title in a h1 tag', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.querySelector('h1').textContent).toContain('Welcome to Angular Unit Testing!');
  }));
});
```

Figure 1: Angular Component Unit Test