# Testing

Introduction to Angular 6

## Unit test

- A unit test is a test on an individual unit or component of the overall application and verifies its behavior independently from other units or components.
- A typical unit test contains 3 phases:
  - Arrange – Initialize a small piece of an application it wants to test (also known as the system under test, or SUT),
  - Act - call a method on it
  - Assert – evaluate the resulting behavior.
- If the observed behavior is consistent with the expectations, the unit test passes, otherwise, it fails, indicating that there is a problem somewhere in the system under test.
- Unit test usually involve all dependencies completely mocked out.

## Unit test rationale

- It provides immediate feedback that the functionality of a particular unit is correctly achieved.
- It guards existing code base against future regression errors when additional modifications or additions are made to the code base
- Unit tests are great documentation for your code.
  - Comments tend to become outdated quickly
- Unit tests are a great view on how testable and modular your design is.
  - If tests are hard to read or write, it usually is a signal that there might be design flaws or issues in the underlying code.

## State-based vs Interaction-based

- A unit test can verify different behavioral aspects of the system under test in two categories:
- state-based
  - Verifying that the system under test produces correct results, or that its resulting state is correct
- interaction-based
  - verifying that it properly invokes certain methods

# Jasmine

- Behavior-driven development (BDD) framework
  - Test framework that is oriented toward writing specifications rather than traditional unit tests.
- A specification is a series of commands, and expectations on what should have happened as a result of these commands.
- It is a standalone framework that can be used to write tests or specifications for any code, not just Angular.

## Karma

- Test-running framework that complements Jasmine.
- Karma takes any kind of test, runs it across a suite of real browsers and reports the results back.
  - It is highly tuned toward development workflow, as it is heavily oriented toward rapid execution and reporting.
- Spawns a web server that executes source code against test code for each of the browsers connected.
- The results of each test against each browser are examined and displayed via the command line to the developer such that they can see which browsers and tests passed or failed.
- A browser can be captured either manually, by visiting the URL where the Karma server is listening (typically http://localhost:9876/),

## Karma

- Karma also watches all the files, specified within the configuration file
- Whenever any file changes, it triggers the test run by sending a signal to the testing server to inform all of the captured browsers to run the test code again.
- Each browser then loads the source files inside an IFrame, executes the tests and reports the results back to the server.
- The server collects the results from all of the captured browsers and presents them to the developer.

## Protractor

- Protractor is a framework that is built to write and run end-to-end test, allowing to test from the perspective of an end user (User Acceptance Testing)
- This would involve opening the browser, clicking, and interacting with the application.
- Protractor supports this capability of running the real application and simulating actions and verifying behavior, thus completing the circle of testing.

## Lab

-

## Testing setup

- karma.conf.js. - the configuration file for how Karma should find and execute files
- The Karma configuration is responsible for
  - identifying the various plug-ins needed for Karma to run
  - the files it needs to watch or execute
  - configuration of coverage reporting, which port it needs to run on, which browsers to run it on, whether it should rerun every time the file changes, and which level of logs it needs to capture.
- The test.ts file
  - the main entry point for our testing
  - responsible for loading a series of files for the testing framework, and then initializing the Angular testing environment
  - looks for specifications recursively in all the folders
  - It then loads all the relevant modules for them and starts executing Karma.

# Basic unit test

- Describe block is  Jasmine’s way of encapsulating a set of tests as one suite
  - Describe blocks can be nested any number deep
  - This feature is used to create separate describe blocks for Angular-aware tests versus isolated unit tests.
- In an isolated unit test, Angular lifecycle methods are not called automatically
  - Need to manually trigger ngOnInit in the test.
  - Gives us the flexibility to test some other function and avoid the ngOnInit
- Jasmine provides many in-built matchers to use in our specifications.

## Ng Test

- Pick up the configuration from the karma.conf.js Karma configuration file
- Load all the relevant tests and files as per the test.ts file
- Capture the default browser (which is Chrome in our case)
- Execute the tests and report the results in the terminal
- Keep a watch on files to continue executing on change

## Angular aware unit test

- Test that is Angular-aware and goes through the Angular lifecycle.
- TestBed
  - to configure a module for our test with just the component under test
- Async
  - to allow the Jasmine framework to understand Angular’s async behavior
  - ensures that we don’t start executing the test until async tasks are finished.
- In the non-async beforeEach, the fixture is the combination of the template, the component class instance, and Angular’s combination of the two to combine the two.

# Angular aware unit test

- In the beforeEach, we trigger fixture.detectChanges()
  - This is a signal to Angular to trigger its change detection flow, which will look at the values in the component and update the bindings in the corresponding HTML.
  - It is also the trigger to execute the ngOnInit for the component the very first time.
- In the actual tests, we can get access to individual elements from the generated component by using the fixture.debugElement and running CSS queries against it.
  - This allows us to check whether the template has the correct bindings and values

# Unit testing services

- Unit testing services requires additional work on top of testing components.
- Configure the Angular TestBed with a provider for the service we want to test.
  - Inject an instance of the service we want to test either into our test or as a common instance in the beforeEach.
- In the beforeEach, register the provider for the service
  - This ensures that the test is now running in the context of our testing module
- In the it, which is the actual test, we call inject function provided by the Angular testing utilities.
  - The first argument are the Angular services that need to be injected into our test.
  - The second argument is a function that gets the arguments in the same order that we passed to the array.

# Testing services with dependencies

- If our service itself had a dependency on another service, we have a few options:
  - Register the dependency as a service with the Angular TestBed module, and let Angular be responsible for injecting it into the service we are testing.
  - Override/mock the dependent service by registering a fake/stub provider with the TestBed, and rely on that instead of the original service.
- Two slightly different situations:
  - If we had to test a component, and actually use the real service underneath in the test.
  - If we had to test a component, and we wanted to mock out the service it depends on in the test. In this case, we can mock out certain calls instead of creating a brand-new fake service just for our test.

## Testing services with dependencies

- The recommended way to get the handle on an instance of a service, even when using fakes, is through the injector.
  - rely on the inject function from the Angular testing utilities to inject the service instance
  - we can use the injector reference on the element
- Once we have a handle on the service instance, we can use Jasmine spies to spy on different methods on the service.
  - A spy allows us to stub any function or method, and track any calls to it along with its arguments and also define our own return values.
- We can create a fake to replace the actual service
  - The fake is simply an object which has the same API as the service we are mocking, but with a customized hardcoded implementation of the methods underneath.

## Unit testing async

- With any code that involves asynchronous flow
  - important to recognize at which point the hand-off happens from synchronous flow to asynchronous, and deal with it accordingly
- Angular provides helpers to abstract out some of the complexities of this in our test.
- First ensure that the test itself is identified and running as an asynchronous test.
- Next, ensure that we identify when we have to wait for the asynchronous part to finish, and validate the changes after that.

## Unit testing async

- We now pass an async function, which in turn is passed the function containing our test code, to the it block
  - after calling the function under test which actually triggers the asynchronous flow.
- We need to ask Angular’s test fixture to stabilize
  - i.e. wait for the asynchronous parts to finish
- The whenStabilize returns a promise that on completion allows us to run the remaining part of the test.
  - We can tell Angular to detect any changes and update the UI, and then make our assertions.

# fakeAsync

- Used instead of the async function to keep code linear and abstract away the async behavior
- The fakeAsync allows us to write our unit test (for async and sync code) in a completely synchronous manner
- There are two methods to simulate a passage of time in a fakeAsync test,
  - tick() - simulates the passage of time
  - flush() - takes the number of turns as an argument, which is basically how many times the queue of tasks is to be drained.

# Unit testing HTTP

- When writing tests for code that makes XMLHttpRequest (XHR) calls,  use HttpTestingController
- In our unit test, we avoid making the actual underlying network call.
  - Any network call in a test adds unreliable dependencies, and makes our tests nondeterministic.
- We can mock out the XHR calls in our tests
  - Just check that the right XHR calls are made, and if the response was a certain value, then it is handled correctly.
- Use the HttpTestingController to setup what calls to expect and what responses to send when those calls are made.
- HttpClientTestingModule
  - This testing module helps automatically mock out the actual server calls, and replaces it with a HttpTestingController that can intercept and mock all server calls.