Getting Started with Unit Testing

K. DEVIN MCINTYRE

CONFOO 2022

Overview

How do I find enough time to write unit tests?

What should I be testing?

How do I write unit tests?

How do I find enough time to write unit tests?

Typical process without unit tests

1. Prototype / First draft
2. Trial-and-Error
   - Trial: Run the app (manual test).
   - Error: See a defect / see functionality missing.
   - Code: Fix the defect / add missing functionality.
3. Confirmation
   - Many manual tests; unit tests added if time allows
   - Return to step 2 if defects are found

Test "During" Development

1. Prototype / First draft
   - Minimize this step
2. Trial-and-Error
   - Trial: ~~Run the app (manual test).~~ Write some unit tests and run the full suite.
   - Error: See a ~~defect~~ test failure / see functionality missing.
   - Code: Fix the ~~defect~~ test failure / add missing functionality.
   - Refactor: The tests + code should follow best practices.
3. Confirmation
   - ~~Many~~ fewer manual tests; unit tests ~~added if time allows~~ run automatically at build
   - Return to step 2 if defects are found

Not only did we find enough time …

Faster trial-and-error

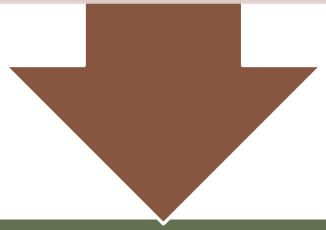
Less manual testing

Fewer bugs to fix later

What should I be testing?

Each function has a contract:

What it is supposed to do	What parameters it takes	What it returns or changes



Unit tests prove each externally-available function contract is correct

Externally-available contracts

An external source determines the state and/or parameters

- Public functions
- Initialization functions
- Error handlers
- Asynchronous callbacks

Note: there are more externalized things in a UI than in a server

Choosing test cases

- Exercise important logic branches
- Exercise logic branches difficult to reach by manual testing
- Confirm boundary conditions
- Test previously-fixed bugs to confirm they don't reoccur

Things **not** to test

- Anything third-party
- Theoretical cases that will not be practically reached
  - Edge cases: user accidentally enters nothing; malicious user enters 5000-character string
  - Theoretical cases: someone calls the method incorrectly; third-party APIs change
- Typically, direct tests of private method contracts – but this has gray areas
  - OK: splitting a complex chain of steps apart
  - OK: testing asynchronous things separately

How do I write unit tests?

Setup

1. Choose a test runner appropriate to your project.
   - Node.js recommendation: Mocha + Chai + Sinon
   - Client-side JS recommendation: Karma + Mocha + Chai + Sinon
2. Provide configurations and globals appropriate to your project.
3. Choose appropriate conventions for where to put your test files.
   - Same directory as source?
   - Separate "tests" directory?
4. Create a test file for each source code file using an appropriate naming convention.
   - JavaScript: sourcecode.js is tested by file sourcecode.spec.js
   - Python: sourcecode.py is tested by file test_sourcecode.py

General test file content

1. Declare the file being tested.
2. Define the code state the tests will run in, a.k.a. the "fixture."
   - Provide a set up function to create the code state.
   - Provide a tear down function to return to the original code state. Restore state changes and destroy references to objects to avoid memory leaks.
   - State changes should not bleed over between tests. Each test should start in the same starting environment.
3. Define individual tests.

Style comparison

	Spec-style	Test-style – NOT RECOMMENDED
Declare the file being tested	outer "describe" statement	class name
Set up the code state the tests will run in, a.k.a. "fixture"	"beforeEach" statement	setup method
Restore state changes between tests	"afterEach" statement	tear down method
Provide variations of fixtures	use nested "describe" statements	use multiple classes
Define each individual test	"it" statement	the other methods in the class
Test runner examples	• Java Javaspec • Ruby RSpec • Python minitest • most JavaScript test runners	• Java JUnit • Python unittest

Spec-style structure

One **spec** file for each source code file

An **outer describe** statement that states the file being tested

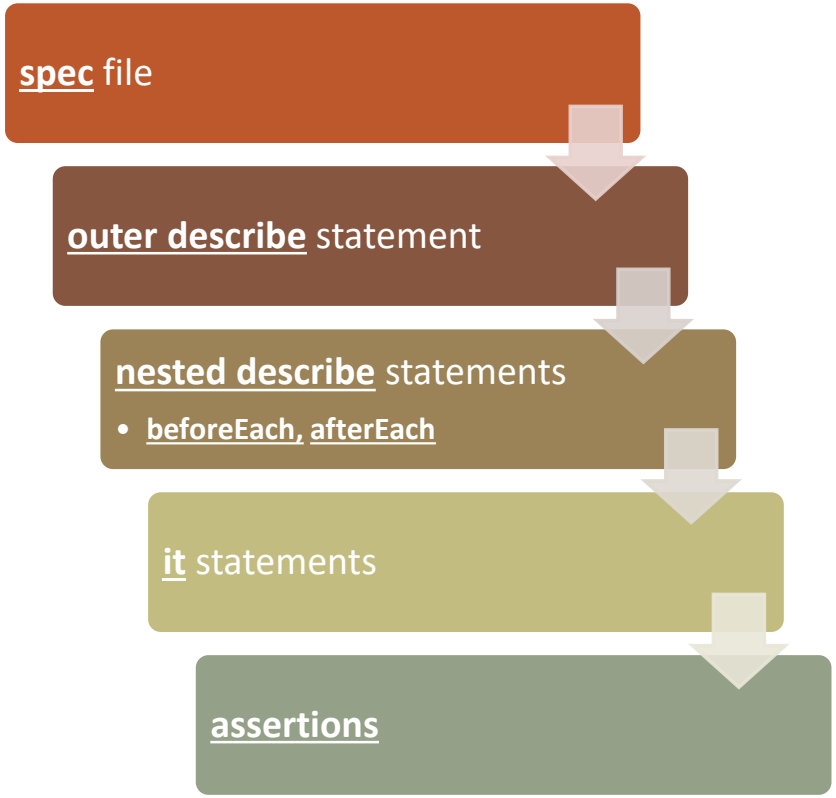
Any number of **nested describe** statements using one **beforeEach** and one **afterEach** statement to control fixture state

Any number of **it** statements, one per call of a function being tested

One or more **assertions** to find out the outcomes of each function call

Spec-style structure

Example uses Mocha + Chai.assert




```
/** @file widget.spec.js */
import Widget from '../factory/widget.js';
import app from '../app.js';

describe( 'factory/widget', () => {
    describe( 'creates app widget', () => {
        let widget;
        beforeEach( () => {
            widget = app.setWidget( new Widget( 'red' ) );
        } );

        afterEach( () => {
            app.reset();
        } );

        it( 'app has red widget', () => {
            assert.equals( widget.color, 'red' );
            assert.equals( app.mainWidget, widget );
        } );
    } );
} );
```

Assertion styles

assert	expect	should – NOT RECOMMENDED
assert .equal(foo, 'foo');	expect(foo) .to.equal('foo');	foo .should.equal('foo');
assert .deepEqual(bar, ['a']);	expect(bar) .to.deep.equal(['a']);	bar .should.deep.equal(['a']);
assert .notExists(baz);	expect(baz) .not.to.exist();	should.not.exist(baz);
assert .throws(()=>fn(), /Message/);	expect(()=>fn()) .to.throw(/Message/);	(()=>fn()) .should.throw(/Message/);

Test contents

1. Ready initial state/fixture
   - Includes mocks/stubs/spies: fake versions of parts of code
   - Includes creating instances, assigning values, etc.
2. Assert the initial value of any changeable states
3. Call the function
4. Assert the function return value
   - Example: assert returned value equals expected value
   - Example: assert no error was thrown
5. Assert any states that could have changed
   - Example: assert changed value equals expected value
   - Example: assert no change has occurred (equals initial value)
6. Assert correct mocking

Test contents

Example uses Mocha + Chai.expect

```
it( 'app main widget color change updates app color', () => {
    // 1. Ready initial state/fixture
    const widget = new Widget( 'red' );
    app.setWidget( widget );

    // 2. Assert the initial value of any changeable states
    expect( widget.color ).to.equal( 'red' );
    expect( app.mainColor ).to.equal( 'red' );

    // 3. Call the function
    const colorChange = widget.setColor( 'blue' );

    // 4. Assert the function return value
    expect( colorChange ).to.deep.equal( { previous: 'red', current: 'blue' } );

    // 5. Assert any states that could have changed
    expect( widget.color ).to.equal( 'blue' );
    expect( app.mainColor ).to.equal( 'blue' );

    // (not applicable) 6. Assert correct mocking
} );
```

Mocking: goals

- Prevent all network calls
  - Network call = integration test, not unit test
- Simulate complicated code states using simplified models
- Contain asynchronous actions inside each test case – avoid memory leaks and zombies
- Check whether methods are called, how many times, in what order, etc.
- Intercept console logging so it does not clutter the tests' console output
- UI: Simulate page reloads and navigation
  - Your tests will halt if these happen
- UI: Simulate DOM state to avoid unnecessary repaints (see: containing async actions)

Mocking: schools of thought

LONDON / MOCKIST

- Should mock outside resources **and usually** dependent modules
- *Tests involving dependencies are …*
  - technically integration tests because multiple modules were involved
  - used sparingly to augment normal unit tests
- *Isolating source code using mocks …*
  - facilitates refactoring by narrowing scope
  - prevents redundant code coverage

DETROIT / CLASSICAL

- Should mock outside resources **but not** dependent modules
- *Tests involving dependencies are …*
  - unit tests because only one function call occurred
  - ideal because they are closest to reality
- *Isolating source code using mocks …*
  - increases overhead of test creation
  - risks mocks becoming out-of-date

Recommendation: London school

- Isolation of files lets you work on one thing at a time
- Easier to control state
- No need to understand 100% of logic to write each test
- No need to force every file in the system to conform to unit test structure
- Clearer boundaries of what code has been tested vs covered
  - Code coverage tools only tell you whether lines of code are reached, not tested

Mocking: types

- *Note: not everyone agrees on what these are*
- Mock: generated object simulating a real object
  - You must swap the real object for the mock
  - Highly-coupled; requires good maintenance
- Stub: generated method ("fake") on an object simulating the real method
  - The real method does not get called
  - Always remove all stubs after each test!
- Spy: interceptor that records calls to a real method
  - The real method gets called
- Order of preference for test maintainability: spy > stub/fake > mock

Note: Jasmine.js "spy" function creates a stub, not a spy

Mocking: when to use mocks

- Simulating API response data
- Testing abstract class files
  - Create mock classes that descend from the abstract class
- Simulating a complex third-party library
  - Prefer existing third-party mocks
  - Examples: aws-sdk-mock, nodemailer-mock, mongo-mock

Mocking: when to use stubs

- To make external sources deterministic (e.g. random, date/time, API responses)
- To return mocks instead of real objects
- To test triggered events separately from their triggers (especially asynchronous ones)
- To confirm whether a method was called with specific parameters
- To find out how many times a method was called
- To simulate simple third-party-library actions

Mocking: when to use spies

- Confirm whether a method was called with specific parameters
- Find out how many times a recursive method calls itself

Sinon: JavaScript mocking tool

- Pronounced *sigh non*
- Can create mocks, stubs, and spies easily
- Works with most JavaScript test runners
- Comes with method `useFakeTimers` to mock Date, setTimeout, setInterval, etc.

Example: mocking

Example uses Mocha + Chai.expect + Sinon

```
it( 'gets configurations from configuration file', () => {
    // 1. Ready initial state/fixture
    const contents = { type: 'xyz' };
    sinon.stub( fs, 'readFileSync' ).returns( JSON.stringify( contents ) );

    // 2. Assert the initial value of any changeable states
    expect( service.type ).not.to.exist();

    // 3. Call the function
    const config = service.getConfigurationsFromConfigurationFile();

    // 4. Assert the function return value
    expect( config ).to.deep.equal( contents );

    // 5. Assert any states that could have changed
    expect( service.type ).to.equal( contents.type );

    // 6. Assert correct mocking
    const configPath = path.join( process.cwd(), service.CONFIG_FILE_PATH );
    expect( fs.readFileSync ).to.have.been.calledWith( configPath, 'utf8' );
} );
```

Review: Setup

1. Choose a test runner appropriate to your project.
2. Provide configurations and globals appropriate to your project.
3. Choose appropriate conventions for where to put your test files.
4. Create a test file for each source code file using an appropriate naming convention.

Review: Spec-style structure

One **spec** file for each source code file

An **outer describe** statement that states the file being tested

Any number of **nested describe** statements using one **beforeEach** and one **afterEach** statement to control fixture state

Any number of **it** statements, one per call of a function being tested

One or more **assertions** to find out the outcomes of each function call

Review: Test contents

1. Ready initial state/fixture
2. Assert the initial value of any changeable states
3. Call the function
4. Assert the function return value
5. Assert any states that could have changed
6. Assert correct mocking

Working example

https://github.com/miyasudokoro/web-component-demo

What about Test-Driven Development?

Test "During" Development

1. Prototype / First draft
   - Minimize this step
2. Trial-and-Error
   - Trial: ~~Run the app (manual test).~~ Write some unit tests and run the full suite.
   - Error: See a ~~defect~~ test failure / see functionality missing.
   - Code: Fix the ~~defect~~ test failure / add missing functionality.
   - Refactor: The tests + code should follow best practices.
3. Confirmation
   - ~~Many~~ fewer manual tests; unit tests ~~added if time allows~~ run automatically at build
   - Return to step 2 if defects are found

Test-Driven Development (TDD)

1. Prototype ~~/ First draft~~
   - Minimize this step
   - Only use your prototype as a reference; discard its code
2. Trial-and-Error
   - Trial: ~~Run the app (manual test).~~ Write ~~some~~ one unit test~~s~~ that should fail and run the full suite.
   - Error: See a ~~defect~~ test failure of the new test ~~/ see functionality missing.~~
   - Code: Fix the ~~defect~~ test failure ~~/ add missing functionality.~~
   - Refactor: The tests + code should follow best practices.
   - See functionality missing and use that to decide the next test to write.
3. Confirmation
   - ~~Many~~ fewer manual tests; unit tests ~~added if time allows~~ run automatically at build
   - Return to step 2 if defects are found

Starting Test-Driven Development

1. Is it right for you and your project?
   - Read http://neopragma.com/index.php/2019/09/29/against-tdd/
2. Gain experience with unit testing best practices.
   - Poorly-written unit tests lock in the bad rather than uncovering it
3. Understand how unit tests and code reflect each other.
4. Mentally prepare for continuous “failure.”
5. Try it with defects first.

Working example

https://github.com/miyasudokoro/web-component-demo