Testing Best Practices

Dan Wellman
Senior Front-end Developer

Unit Tests

- Typically written with and residing alongside the app code
- Most commonly written by developers
- Integration/e2e tests usually written by testers

Writing Readable Tests

Write descriptive and meaningful test names.

Descriptive Test Names

- Get a good idea of what the application code does just by reading the test names
- Easily identify failing tests—especially helpful when there are many tests
- Clearly communicate to non-technical stakeholders the app's capabilities

Always use the AAA pattern:

- Arrange
- Act
- Assert

```
it('changes the property value', () => {
  myClass.prop = 'before';


  myClass.changeProp('after');


  expect(myClass.prop).toEqual('after');
});
```

- **Arrange – create the necessary starting conditions for the test**
- **Act – perform the action that will lead to the required code being executed**
- **Assert – check the result of performing the action to make sure the code did what it should**

Test one thing, and test it well!

Keep tests small and focused for readability.

```
describe('outer suite', () => {
  describe('inner suite', () => {
    it('a test', () => {
      // ...
    });

    it('another test', () => {
      // ...
    });
  });
});
```

Test Organization

Test frameworks usually allow both tests and test suites to be created

```
describe('Class name', () => {
  describe('method name', () => {
    it('does this thing', () => {
      // Arrange
      // Act
      // Assert
    });

    it('also does this thing', () => {
      // ...
    });
  });
});
```

- Use `describe` to create suites
- Use `it` to create individual tests
- Group related tests together into suites

Test and Test Data Independence

Unit tests should be completely independent.

Test Independence

- Tests should be able to run in any order
- Modern test frameworks usually run tests in a random order
- Tests should not be coupled to each other to avoid false negatives

Tests should only fail if the code does not work in the expected way.

Reliable tests are a good indicator of the health of the project.

```
import { ExternalClass } from '../services';

describe('MyClass', () => {
  describe('a method of my class', () => {
    it('does something if the method of the external class returns true', () => {
      const instance = new ExternalClass();
      // mock the external class
      jest.spyOn(instance, 'someMethod').mockReturnValue(true);
    });
  });
});
```

External Dependencies

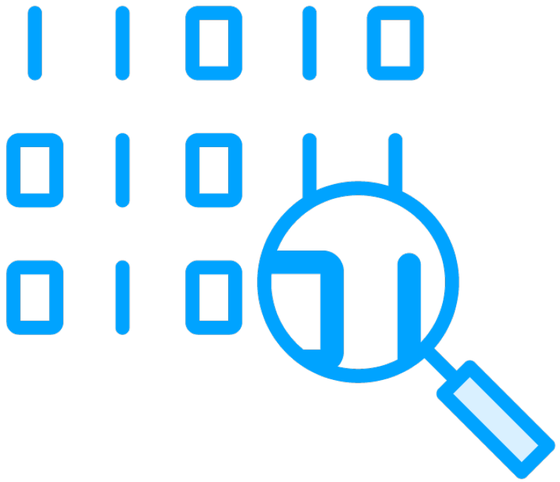
- **All external dependencies should be mocked**
- **Modern frameworks make mocking the methods of external classes very easy**
- **Mock methods are used to control the return value, make sure the code under test is fully exercised, and to keep the tests performant**

```
class MyClass {
  public publicMethod() {
    // do some stuff, then
    this.privateMethod();
  }

  private privateMethod() {
    // code in here will be exercised by the test
    // if we mocked this method it would not
  }
}
```

Avoid Mocking Internal Dependencies

- Try to minimize mocking internal methods
- Improves coverage as private methods can also be exercised

Data Independence



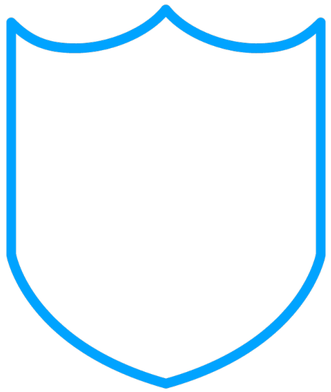
- Ensure that tests don't need specific data to pass
- Avoid global or shared state
- Generate test data instead of using an external data source

```
describe('MyClass', () => {
  describe('a method of my class', () => {

    beforeEach(() => myClass.userData = { name: 'test user' });

    afterEach(() => delete myClass.userData);

    it
  });
});
```

Setup and Tear-down

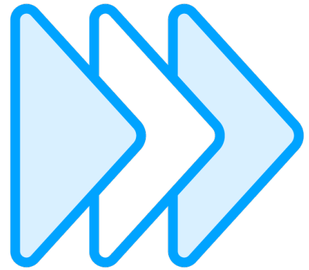
Test frameworks provide setup and tear-down functionality to set up the required data before each test and clean it up after each test

Test and Data Independence Benefits



Resilient to code changes

Easier to maintain and extend

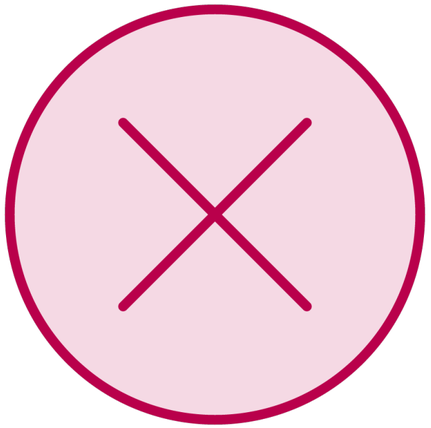


Quicker to run and debug

Test-driven Development

In TDD, you write the tests before you write the actual implementation code.

TDD Cycle – Red, Green, Refactor

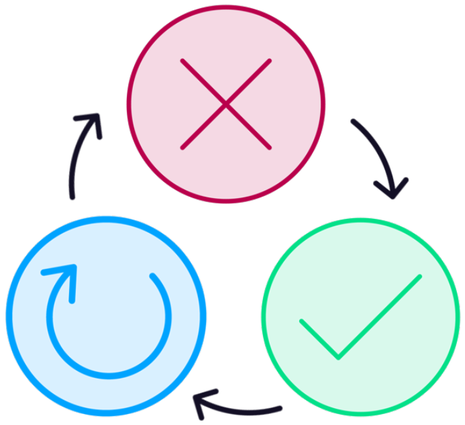


Write a test that fails (because the code is not written)

Write the minimum code needed to make the test pass

Refactor and improve the implementation

TDD Benefits



Cleaner code

- TDD forces developers to think about the design of the code upfront

Better test coverage

- TDD encourages different edge-cases to be tested

Clear requirements

- With TDD, the tests provide the specification

Long-term stability

- TDD helps the code base stay stable and maintainable over time

Increased productivity

- TDD encourages pair programming to improve code quality

Further TDD

Test-driven Development: The Big Picture

Jason Olson

Further TDD

TDD as a Design Tool

Nate Taylor

Use a Modern Test Runner

Jest

- A modern test runner with mocking and assertion APIs, built on top of Jasmine.
- Uses the familiar syntax of describe/it/expect
- Much faster than Karma, which is still the default in Angular applications

Aim for 80%+ code coverage for unit tests.

Avoid chasing the elusive and unreachable 100%