INTRO TO TESTING

minimizing mistaqes

Through some workshops and the take home checkpoints, you are familiar with tests. You can read them and use them to figure out what it is that you need to write so that the tests pass. And we've done that on purpose.

But out in the wild, tests serve a different purpose than to guide you in what code to write. And writing a test is a different skill set than reading one.

QUESTIONS YOU MIGHT HAVE

- **Why** test?
- **How** to test?
- **What** to test?

Why do you think testing is important?

WHY DO I NEED TO WRITE TESTS?

WHY

- Reliability
- Refactorability
- Documentation
- Accuracy
- Value in Industry

Reliability: Ensure code is working - most common
Refactorability: Ensure code WILL continue to work after someone changes it - refactor with more confidence
Documentation: Documents what the code actually does.
Accuracy: Precision/Accuracy/certainty of behavior
Value in Industry: Professional software developers are expected to be able to write tests, read tests, and understand why we write tests

HOW TO WRITE MY TESTS?

ANATOMY OF A SPEC

Describe blocks: contains [sub-groups of] specs

1) descriptive labels of entity to be tested
2) function to nest further `describe`s (sub-entities) or `it`s

```
describe('Kittens', function() {
  describe('eat', function() {
    it('returns yum', function() {
      var k = new Kitten();
      expect(k.eat()).to.equal('yum');
    });
  });
});
```

ANATOMY OF A SPEC

Describe blocks: contains [sub-groups of] specs

1) descriptive labels of entity to be tested
2) function to nest further `describes` (sub-entities) or `its`

```
describe('Kittens', function() {
  describe('eat', function() {
    it('returns yum', function() {
      var k = new Kitten();
      expect(k.eat()).to.equal('yum');
    });
  });
});
```

It block: constitutes a single spec

1) descriptive label of one thing that should happen
2) function for testing that actually happens

It block - constitues an single spec
It’s similar to a try catch: if an error is thrown, the test fails, but the process will not crash, instead it will continue running all the tests

ANATOMY OF A SPEC

Describe blocks: contains [sub-groups of] specs

1) descriptive labels of entity to be tested
2) function to nest further `describes` (sub-entities) or `its`

```
describe('Kittens', function() {
  describe('eat', function() {
    it('returns yum', function() {
      var k = new Kitten();
      expect(k.eat()).to.equal('yum');
    });
  });
});
```

It block: constitutes a single spec

1) descriptive label of one thing that should happen
2) function for testing that actually happens

Assertion: making your expectations

1) There can be many assertions within one it block
2) If something is `thrown` *at any point* in an it block, the spec stops and **fails**

Based on this code, what can you tell me about k (the kitten) and .eat()?

describe, it, and expect functions are NOT something that node understands. So we need to install tools that can define these functions for us.

The only way to make a test spec fail is to throw an error. This is what the expect will do. What we return doesn't determine if the spec passes.

TOOLS

- Jasmine and Mocha/Chai are two popular testing frameworks in the JavaScript ecosystem
- We chose mocha/chai for the popularity and flexibility
- However, there are many other options out there!

Jasmine is a one size fits all. Less flexible, but more included. We used Jasmine with Testem during Foundations.
Mocha/chai is more flexible and very popular.
Mocha is the testing framework (has describe and it) chai is the assertion library, so it gives us access to ‘expect’

Many others!

Example 1: Installation and basic tests

Workshop link
https://learn.fullstackacademy.com/workshop/5a68bf8a9f9cb600048448ef/landing

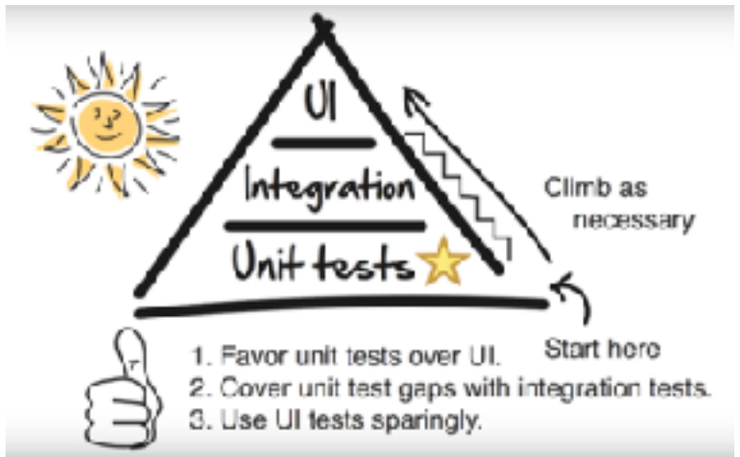
WHAT DO I TEST FOR?

WHAT

- **Test for behavior, *not* implementation**
  - ✗ “I expect this multiply function to use the add function”
  - ✓ “I expect this multiply function to return 6 given the inputs 2 and 3”
- **Implementation details change all the time, but intended behaviors generally do not**

This is DIFFERENT from how you’ve seen tests up to this point. Tests in checkpoints and workshops are often meant to guide you in the implementation.

TEST PYRAMID



source http://www.agilenutshell.com/

Unit tests: testing individual, independent functions
Integration tests: testing that different parts of my application work together appropriately
UI tests: does this show correctly to the user?

We are going to be focusing on unit tests, but in a production app, you will also need integration and UI tests

TEST-DRIVEN DEVELOPMENT

- **Write tests first, *then* write code to pass the tests**

- Focus on what code does
- Have a goal
- Ensure you don't blow off automated testing
- Improve design and modularity

Just one philosophy for test-writing, separate from automated testing.

It allows you to focus on what the code does and keep your goal in mind.

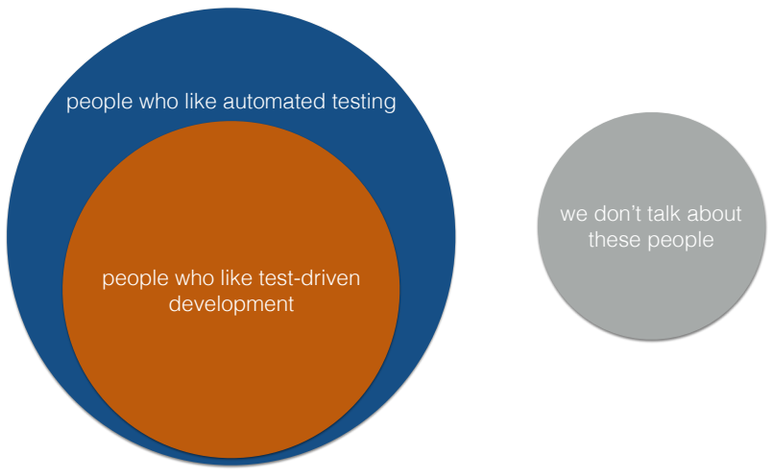
Also makes sure you have

TEST-DRIVEN DEVELOPMENT

Helpful to make sure that your test fails before you actually write any code!

Then you write the bare bones necessary to make it pass. Then you write another test.

AUTOMATED TESTING ≠ TEST-DRIVEN DEVELOPMENT



Don't have to love strict TDD, but you should at least embrace tests

TDD use automated testing, but a more strict paradigm

Not all developers are required to do TDD