ANGULAR TDD

GOALS FOR THIS SECTION

1. TDD Review
2. Testing Frameworks
3. Testing Angular
    - Services
    - Controllers
    - Spies and Mocks

TDD REVIEW

QUESTION

What do you already know about TDD?

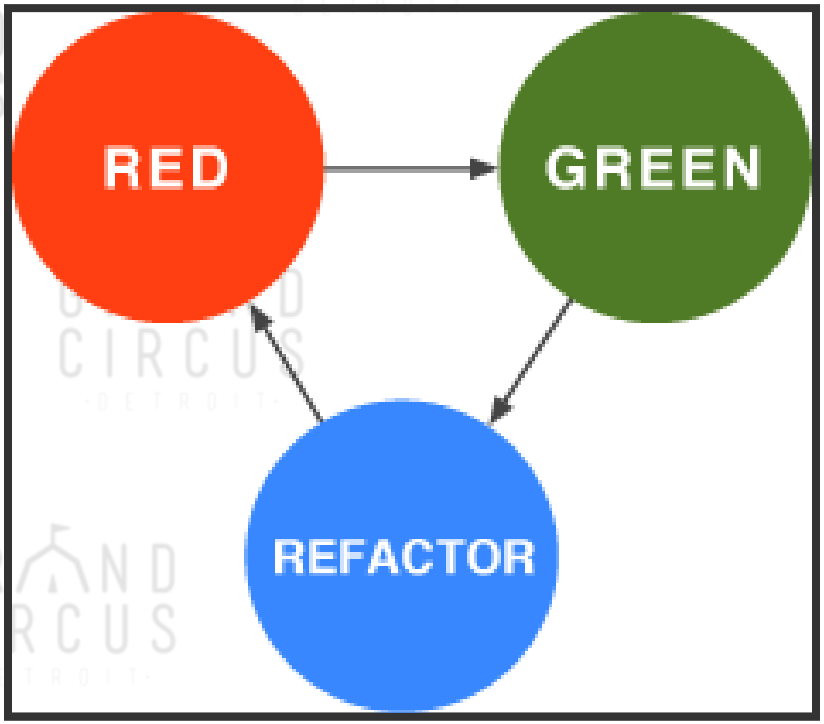
TEST FIRST

Another approach is to create the test *before* you create the software solution. This can seem unintuitive at first because *of course* any test you write before you write the solution is doomed to fail. That turns out to be the point.

RED, GREEN, REFACTOR

This cycle of testing is most often referred to as Test-Driven Development (TDD). The core of this method is summarized as "Red, Green, Refactor". It goes like this:

1. Write a failing test - Red
2. Make that test pass - Green
3. Clean up the code - Refactor



RED, GREEN, REFACTOR

RED, GREEN, REFACTOR

What's with the colors?

You'll see in a minute. The short answer is it has become a convention that our test frameworks will mark failing tests as red and passing tests as green. So Red = failing test, Green = passing test.

Refactor has been made blue because who knows...

WHY REFACTOR?

When we write tests using the TDD method, we write in very small steps. Then we only write enough code to make that test pass. *Then* we write another test for the next small step we can take. We pass that test without failing our previous test. If we did nothing else, our solution would quickly devolve into a real mess. That's why after each time we get to green we stop to refactor. We look for ways to improve our solution *without changing its behavior*.

EFFECTIVE REFACTORING

We already know it works (we're green) so after we refactor we re-run our test to make sure we're still passing. If we are, then we know we haven't changed our code's behavior. We've only changed it's readibility or refactored away a messy or less optimal part of the solution.

TEST FRAMEWORKS

TEST FRAMEWORKS

In order to run our tests we need a test framework to exercise our code and check the results of our tests. Most popular languages have several popular testing frameworks. Which you prefer largely has to do with personal preference.

JAVASCRIPT TESTING FRAMEWORKS

The big testing frameworks in JavaScript are

- Jasmine
- Mocha
- QUnit

There are others but these are the ones you're likely to run into.

JASMINE

The testing framework we will be using is Jasmine. It's a very nice testing framework that is easy to use and complete out of the box. It's not as configurable as some of the other frameworks, but we think it's completeness makes up for that.

Jasmine is what gives us the `describe`, `it` and `expect` functions.

MOCHA

Mocha was originally created to test NodeJS code, but it has been adapted to easily be able to test browser code as well. It is less complete as a standalone product but it is much easier to customize. It is also a very solid choice.

QUNIT

Originally created to test jQuery code, it's a slightly older framework and is not as popular as it used to be. I don't have as much experience with this framework as some of the others, but it looks like it might make it easier to test UI which is actually kind of difficult. So there is definitely some value in learning how it works.

JAVASCRIPT TEST RUNNER

These test frameworks can help us define tests. But often they are used in conjunction with *test runners* to actually run the tests.

In our case *Karma*.

KARMA

Karma will take care of several things for us...

- Find our JavaScript files and tests
- Run them in a browser
- Print out the results on the terminal
- Run the tests automatically every time we save a file

TESTING ANGULAR

TESTING SERVICES

Services are the easiest thing to test in Angular. The trick is getting a handle on them in our tests.

How can we get a handle on a service, say, from a controller?

DEPENDENCY INJECTION!

We're still using dependency injection to get the service, but it looks a little different in tests.

1. Create test file with "describe"
2. Specify the module name
3. Inject the service
4. Write your tests!

STEP 1: CREATE TEST FILE WITH "DESCRIBE"

```
describe("whatToWearService", function() {
    // Everything else goes in here.
});
```

STEP 2: SPECIFY MODULE NAME

```
describe("whatToWearService", function() {
    // This tells it which module we're testing.
    beforeEach(module("testingDemoModule"));
});
```

STEP 3: INJECT THE SERVICE

```
// Declare variables here that need to be shared between all the funcitons below.
var whatToWearService;

// use inject() to grab angular dependencies
beforeEach(inject(function(_whatToWearService_) {
    whatToWearService = _whatToWearService_; // this is the real whatToWearService
}));
```

STEP 4: WRITE YOUR TESTS

```
it("returns coat at top of temp. range", function() {
    expect(whatToWearService.getOuterwear(53)).toBe("a coat");
});

it("returns something comfy for casual", function() {
    expect(whatToWearService.getOutfit("casual")).toBe("something comfy");
});
```

TESTING CONTROLLERS

Controllers are a bit different to test.

1. Controllers usually have $scope.
2. Controllers don't get called by your code like services do. Angular creates them automatically.

TESTING CONTROLLERS

1. Create test file with "describe"
2. Specify the module name
3. Inject $controller and any dependencies you need
4. Create an empty $scope
5. Initialize the controller with $controller
6. Write your tests

STEP 1: CREATE TEST FILE WITH "DESCRIBE"

```
describe("whatToWearController", function() {
    // Everything else goes in here.
});
```

STEP 2: SPECIFY MODULE NAME

```
describe("whatToWearController", function() {
    // This tells it which module we're testing.
    beforeEach(module("testingDemoModule"));
});
```

STEP 3: INJECT $CONTROLLER & DEPENDENCIES

```
// use inject() to grab angular dependencies
beforeEach(inject(function(_whatToWearService_, $controller) {
    whatToWearService = _whatToWearService_; // this is the real whatToWearService
}));
```

STEP 4: CREATE EMPTY $SCOPE

```
// use inject() to grab angular dependencies
beforeEach(inject(function(_whatToWearService_, $controller) {
    whatToWearService = _whatToWearService_; // this is the real whatToWearService

    $scope = {}; // Scope starts out empty
}));
```

STEP 5: INITIALIZE THE CONTROLLER

Call `$controller()` with the name of the controller. The second parameter is an object of the dependencies to inject.

```
// use inject() to grab angular dependencies
beforeEach(inject(function(_whatToWearService_, $controller) {
    whatToWearService = _whatToWearService_; // this is the real whatToWearService
    $scope = {}; // Scope starts out empty

    // Create the controller. Specify the dependencies to inject.
    whatToWearController = $controller('whatToWearController', {
        $scope: $scope,
        whatToWearService: whatToWearService
    });
}));
```

STEP 6: WRITE YOUR TESTS

Each test runs after the controller has run and initialized the $scope.

```
it("starts out with default", function() {
    expect($scope.output).toBe("Wear something comfy and no jacket.");
});

it("updates output when temperature changes", function() {
    $scope.temperature = 60;
    $scope.updateTemperature();
    expect($scope.output).toBe("Wear something comfy and a jacket.");
});
```

TESTING WITH MOCKS & SPIES

MOCKS & SPIES

- Mocks are fake versions of real services and other dependencies.
- Spies are fake functions with special abilities to track how they are called.

SPIES

You can test whether a spy function has been called and what arguments were passed to it.

```
var fakeFunction = jasmine.createSpy("fakeFunction");

fakeFunction("Hello");

expect(fakeFunction).toHaveBeenCalled();
expect(fakeFunction).toHaveBeenCalledWith("Hello");
expect(fakeFunction).not.toHaveBeenCalledWith("Goodbye");
```

SPIES

You can also tell a spy what it should do when it is called.

```
var fakeFunction = jasmine.createSpy("fakeFunction").and.returnValue("Boo!");

var result = fakeFunction("Hello");

expect(fakeFunction).toHaveBeenCalled();
expect(fakeFunction).toHaveBeenCalledWith("Hello");
expect(result).toBe("Boo!");
```

SPIES

There's a lot you can do with spies. See docs.

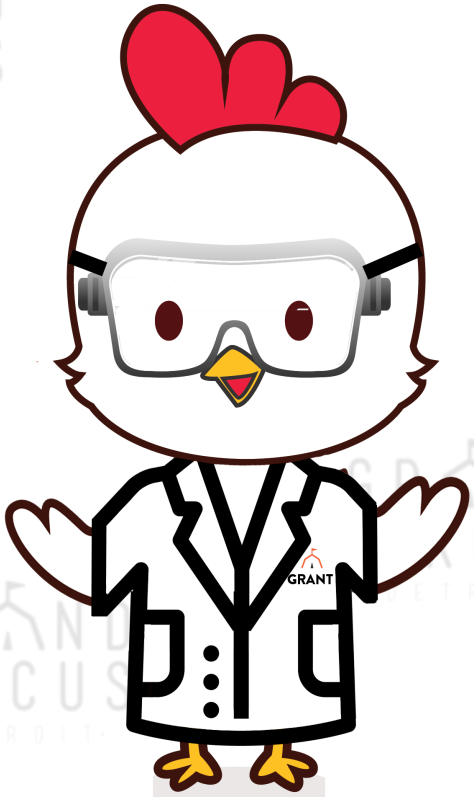
MOCKS

In order to test the controller without messing with the real service dependencies, we need to make a temporary "mock" versions of the service that acts like the real one. We can fake the behavior using spies for the methods.

```
var mockWhatToWearService = {
    // this mock method will always return "Wear this."
    describeAppropriateClothing: jasmine.createSpy("describeAppropriateClothing")
                                 .and.returnValue("Wear this.")
};
```

LAB T3

ANGULAR TESTING



INSTRUCTIONS

1. Test your lab 18 palindrome, fizz buzz or primes service.
    - Get back in your groups and make the changes together.
    - If you did not originally implement this as a service, change it into a service for this.
2. Test your lab 15 "Mad Libs" service.
    - Do this one on your own.
    - Test storing and retrieving values from it.

BONUS

- Test the controllers in Lab 15 "Mad Libs"
- You may choose whether or not you'd like to try it with spies and mocks.