Day 5: Testing with Mocha & Chai

Learning Objectives (5 Minutes)

1. Provide an introduction to unit testing with JavaScript.
2. Experience Test Driven Development and Behavior Driven Development first-hand through the use of Mocha and Chai.
3. Practice TDD and BDD concepts via in-class pair challenges.

Overview / TT (35 Minutes)

Definitions

> **Test**
>
> 1. **noun** - a procedure intended to establish the quality, performance, or reliability of something, especially before it is tak
> 2. **verb** - take measures to check the quality, performance, or reliability of (something), especially before putting it into wi

TDD and BDD

Test-Driven Development is a popular trend in software development. TDD uses software tests as a baseline metric to deterr
wreck themselves, their teams, and the reputation of the product.

> ***Test-Driven Development* (TDD) is a software development process that relies on the repetition of a very short dev
> software is improved to pass the new tests, only. This is opposed to software development that allows software to**

In short, TDD follows this cycle:

1. **Add a Test**: New features are added by first writing a test that fails.
2. **Run All Tests**: Thus validating that the test is working as it should not pass.
3. **Write Code**: Write code that will cause the test to pass.
4. **Run Tests**: If all tests pass, congratulations! You can be confident your code meets the requirements of the test cases. If
5. **Refactor Code**: As the code base grows, it must be cleaned and improved. Tests will help to find problems, errors, or bu
6. **Repeat**: With each cycle, the functionality of the product, and the reliability/usefulness of the tests, should increase.

TDD offers some advantages. It can increase productivity and help avoid the writing of unnecessary code. It also avoids erro

- **Wikipedia: Test-Driven Development**

TDD does NOT solve all of the your problems for you. Some things are difficult to test, UI interactions for example. **Some te**
large amounts of time writing tests, code which doesn't go directly into your finished product.

For best results, everyone on the team must support TDD, and tests must be carefully planned.

It has been suggested that testing first produces better results than writing tests after implementation.

BDD (*Behavior Driven Development*) is an extension of TDD that focuses on communication and collaboration. Imagine puttir
sentences starting with a conditional verb "should" for example. Taken further, these sentences should describe adding value

> **It displays a notification when a new post is created.**

Some software libraries incorporate BDD into their interfaces. For example:

```
user.should.have.properties("first", "last", "age");
```

You can see this is code but it also is clearly read in english describing important product features. BDD will manifest for us around BDD syntax like: `should.have.properties()` and `expect().to.be()` or similar.

Testing with Mocha

The goal of testing is to define software features that the app will have to meet, and test those features to confirm the code Mocha **runs tests and displays the results**. Mocha **doesn't** handle assertions. Instead, it **allows you** to **define your own a**

Defining an Assertion

> **Assertion**
>
> 1. a confident and forceful statement of fact or belief.
> 2. the action of stating something or exercising authority confidently and forcefully.

In software testing, an `assertion` states what the programmer *expects*, failing when the results do not match the program

Types of Assertions

Should vs Expect

```
const chai = require('chai')
  , expect = chai.expect
  , should = chai.should();
```

Notice that the `expect` require is just a reference to the `expect` function, whereas with the `should` require, the functior

- The `expect` interface provides a function as a starting point for chaining your language assertions.
- The `should` interface extends Object.prototype to provide a single getter as the starting point for your language assertic

What Should Be Tested

Everything! (that *can* be tested)

Okay — there are things that are difficult to test, which doesn't necessarily mean that you shouldn't test them! Use common

If you think of a feature, begin thinking of a test for that feature. **Any function that takes input and provides output can ar**

Good vs. Bad Tests

A test should focus on **one** thing. 1 test = 1 feature / code path.

Tests should validate a **unique** feature and not be repeated.

Tests should be short. If you find you are writing a long test case, step back and ask if this case should really be broken dov

Final Thoughts Before the Challenges

Writing tests for code *before* you have the code to test is a great mental exercise. You may find that you will have to write co

BREAK (10 Minutes)

In Class Activity II - TDD Practice (70 Minutes)

Clone the **starter repo** for this activity and open the code in your preferred code editor.

Your instructor will walk you through writing the first test.

Then, work with a partner to finish the rest. The goal is to make each test pass!

Level 1 Challenges (25 Minutes)

Overview

Imagine you just got a job with a MeasureIt.com. They want to create an app that measures everything. You'll need some me

- Area should return the area of a rectangle.
- Perimeter should return the perimeter of a rectangle.
- Should return the area of a circle with radius.

You'll start by writing pending tests for these methods. Then write functions that make the tests pass.

Level 2 Challenges (30 Minutes)

Overview

The product is a shopping cart. The cart will track products added to a cart. The cart needs to add new products, remove pr

Start with these test cases and write code to answer test case. Note: there is no code yet that does any of the things the tes

From a TDD perspective, you start with failing tests and build the application to meet the requirements of the tests.

In terms of **BDD**, the test descriptions are written to describe what the product should be capable of doing.

```
it('Should create a new item with name and price');
it('Should return an array containing all items in cart');
it('Should add a new item to the shopping cart');
it('Should return the number of items in the cart');
it('Should remove items from cart');

// Stretch challenges
it('Should update the count of items in the cart');
it('Should remove an item when its count is 0');
it('Should return the total cost of all items in the cart');
```

Your goal is to write code that meets the above test cases.

To help visualize the how the cart behaves, you can picture the cart as a table. Imagine the tables below were drawn up by t

Imagine your shopping cart is empty:

name	price	qty	cost

Imagine you add an apple to the cart

name	price	qty	cost
apple	0.99	1	0.99
total		1	0.99

What if you add a banana?

name	price	qty	cost
apple	0.99	1	0.99
banana	1.29	1	1.29
total		2	2.28

What if you add another apple?

name	price	qty	cost
apple	0.99	2	1.98
banana	1.29	1	1.29
total		3	3.27

Hints

While you won't be making a functional shopping cart, you will have to create some of the system. Think about how shoppir item is.

"Items" in the cart will be JavaScript Objects, and the "cart" system will hold them in an array.

- Set up your tests run your code. All tests should be pending.
- Solve each test case one at a time by following the TDD pattern.
  - Write functions that handle the test case. The functions should return a value the test case can evaluate.
  - Run your tests. If the first case passes move on to the next, if not revise your code and test again.
- Solving one test case may break a previously working case. In this case refactor and test again.

After Class

Submit your TDD/BDD Challenge repo to **Gradescope** by EOD Thursday.

Start studying for the **midterm assessment**.

Additional Resources

- **Step by Step Setup** - Quick documentation on how to get started with TDD and BDD in Node.
- **Chai.js Cheatsheet** - Awesome cheat sheet for implementing TDD and BDD!