Unit Testing Workshop

Ali Piccioni
apiccion@gmail.com
2019-11-19

Initial Setup - NodeJS

Install NodeJS and NPM

Verify the following commands work:

```
node -v
```

```
npm -v
```

If not, install from: https://nodejs.org/en/download/

Initial Setup - Download Example Code

Clone the example code from GitHub

```
git clone https://github.com/apiccion/PowerCoders.git
```

Calculator App - Running

```
cd PowerCoders/calc/

npm install

node app
```

Open your browser to http://localhost:1337

To shut-down the app use ctrl+c.

Calculator App - Structure

```
public/index.html    # HTML file sent to your browser
app.js               # NodeJS server that serves requests
math_lib.js          # Calculator math logic
```

Calculator App - Testing Evolution

How could a programmer check that the calculator works?

Calculator App - Testing Evolution

How could a programmer check that the calculator works?

Manual testing

Calculator App - Testing Evolution

How could a programmer check that the calculator works?

Manual Test

- Requires manual human interaction
- Takes a long time
- Prone to human error

Calculator App - Testing Evolution

How could a programmer check that the calculator works?

`node test_demo/demo1.js`

- Write a script that does some sample calculations
- Need to remember to run the script
- Output needs human-verification

Calculator App - Testing Evolution

How could a programmer check that the calculator works?

```
node test_demo/demo2.js
```

Improvement: Script shows “PASSED” or “FAILED”

Code is a bit messy.

Calculator App - Testing Evolution

How could a programmer check that the calculator works?

```
node test_demo/demo3.js
```

Code is cleaner and easier to understand.

This isn’t a bad solution.

Calculator App - Testing Evolution

Often better to use a testing framework.

We'll be using the Mocha framework.

```
    npm test
```

```
Tests are in tests/math_lib.js
```

Calculator App - Testing Evolution

Often better to use a testing framework.

- Consistent output formatting
- Features you'll need [e.g. mocking]

Calculator App - Deploying

Now before we deploy our webapp, we can run some tests.

```
npm test
```

```
node app
```

Calculator App - Deploying

Always run tests before deploying!

```
npm run deploy
```

Humans are forgetful. Better to deploy and test in one command, so someone won’t forget to run the tests.

Calculator App - Committing Code

Always run tests before you commit code!

Well built automated systems will enforce certains tests pass when committing or pushing code.

Git supports pre commit hooks.

Calculator App - Fixing Bugs

There are several bugs in the add and multiply implementations.

Play around with the webapp a bit see if you can find them.

Hint: Think of special kinds of numbers.

Calculator App - Fixing Bugs

Addition

Doesn’t work properly with numbers numbers. E.g. [1.5 + 3.2]

Calculator App - Fixing Bugs

Addition

- Doesn’t work properly with numbers numbers. E.g. [1.5 + 3.2]

Multiplication

- Doesn’t work with some negative numbers [-5 * 5, -5 * -5]

Calculator App - Fixing Bugs

Test driven approach to fixing bugs.

1. Write failing tests first

Calculator App - Fixing Bugs

Test driven approach to fixing bugs.

1. Write failing tests first
2. Attempt to fix the code

Calculator App - Fixing Bugs

Test driven approach to fixing bugs.

1. Write failing tests first
2. Attempt to fix the code
3. Repeat #2 until all tests pass

Calculator App - Fixing Bugs

Test driven approach to fixing bugs.

1. Write failing tests first
2. Attempt to fix the code
3. Repeat #2 until all tests pass
4. Run the app and test locally

Calculator App - Fixing Bugs

Test driven approach to fixing bugs.

1. Write failing tests first
2. Attempt to fix the code
3. Repeat #2 until all tests pass
4. Run the app and test locally
5. Commit your changes

Calculator App - Fixing Addition

Go ahead and write some new test cases in test/math_lib.js.

Make sure the tests fail with

```
npm test
```

Calculator App - Fixing Addition

Go ahead and write some new test cases in test/math_lib.js.

```
it('1.5 + 3.0 = 4.5', function() {
     assert.equal(math.add("1.5", "3.0"), 4.5);
 })
```

Calculator App - Fixing Addition

Try to fix the problem by changing the code in math_lib.js

Everytime you make a change, run the tests. See if your change works.

Calculator App - Fixing Addition

Solution #1

```
function add(a, b) {
    return a + b;
};
```

Calculator App - Fixing Addition

Solution #1

```
function add(a, b) {
    return a + b;
};
```

This breaks tests. Why? Good example of specifications in test.

Calculator App - Fixing Addition

Solution #2

```
function add(a, b) {
    return Number(a) + Number(b);
};
```

All tests should pass.

Calculator App - Fixing Multiplication

Again, start by writing failing tests.

Account App - Running

```
cd PowerCoders/account

npm install

node app
```

Open browser to http://localhost:1337

Account App - Structure

Similar structure to calculator app

accounts.js

Account creation logic

tests/accounts.js

Tests account creation

Account App - Bugs!

We have many bugs in account creation webapp.

Try to find them all.

Write tests for each bug you find.

Account App - Bugs!

What bugs have you found?

Account App - Bugs!

What bugs have you found?

- Duplicate account names

Account App - Bugs!

What bugs have you found?

- Duplicate account names
- List accounts doesn’t work

Account App - Bugs!

What bugs have you found?

- Duplicate account names
- Password limits
- List accounts doesn’t work

Account App - Fixing Bugs

Step 1: Write tests for all bugs

Step 2: Try to fix the code.

Unit Tests

The kinds of tests we've written are called unit tests.

Unit Tests

The kinds of tests we've written are called unit tests.

Test individual components, not the whole system.

Unit Tests

Benefits:

- Very fast to run tests, can quickly try new code.
- Can catch many common mistakes.
- Can confidently change very old code [that maybe you didn’t write].
- Reading the tests can help engineers understand the code.

Unit Tests

Benefits:

- Very fast to run tests, can quickly try new code.
- Can catch many common mistakes.

Unit Tests

Benefits:

- Very fast to run tests, can quickly try new code.
- Can catch many common mistakes.
- Can confidently change very old code [that maybe you didn’t write].

Unit Tests

Benefits:

- Very fast to run tests, can quickly try new code.
- Can catch many common mistakes.
- Can confidently change very old code [that maybe you didn’t write].
- Reading the tests can help engineers understand the code.

Unit Tests

Limitations:

- Won’t catch all mistakes [see integration tests]

Unit Tests

Limitations:

- Won’t catch all mistakes [see integration tests]
- Takes time to write: Sacrifices the present for the future.

Unit Tests

Limitations:

- Won’t catch all mistakes [see integration tests]
- Takes time to write: Sacrifices the present for the future.
- Not a good tradeoff for unknown or quickly changing requirements

Unit Tests

Limitations:

- Won’t catch all mistakes [see integration tests]
- Takes time to write: Sacrifices the present for the future.
- Not a good tradeoff for unknown or quickly changing requirements
- [e.g. if a website layout changes every week, not worth testing the layout]

Unit Tests - Coverage

The examples we've covered today are very small.

Unit Tests - Coverage

The examples we've covered today are very small.

Even very basic systems can have at least 10,000 lines of code [tiny].

Large systems may have millions of lines of code [Windows estimated at ~50 million].

Unit Tests - Coverage

Unit testing is critical for maintaining a reliable large scale codebase.

Example: Bug whack-a-mole

Unit Tests - Coverage

Unit testing is critical for maintaining a reliable large scale codebase.

Use a metric called “coverage” to determine how well tested a codebase is.

Unit Tests - Coverage

Unit testing is critical for maintaining a reliable large scale codebase.

Use a metric called “coverage” to determine how well tested a codebase is.

> lines of code executed by at least one test / total lines of code

Unit Tests - Coverage

Unit testing is critical for maintaining a reliable large scale codebase.

Use a metric called “coverage” to determine how well tested a codebase is.

> lines of code executed by at least one test / total lines of code

Output of npm test should coverage at the bottom table.

Try adding a new function to math_lib.js or accounts.js, see how it changes coverage numbers.

Unit Tests - Coverage

I’ve found 80-90% coverage is good rule of thumb for unit testing on projects I’ve worked on.

Other projects may require more.

Integration Tests

Integration tests cover the system end-to-end.

E.g. Start up a real database to read and write data.

These can sometimes take a lot of time to setup and run [e.g. minutes to hours].

End to End Tests

Tries to test everything from end-to-end

Often very difficult to set-up, and very time-consuming [e.g. hours].

Example: Can simulate a browser, and user clicking at areas on the screen.

Example: Video playing services, can verify the output video.

Bonus

Use what you've learned to add a new feature to the accounts webapp.

Can be as complicated or simple as you like.

Simple example: When a user creates an account, let them enter a favorite color. Make the list account button list their favorite color next to the username.

You'll want to make changes to:

`public/index.html, app.js, accounts.js, test/accounts.js`