The Pragmatic Programmers

Test-Driven React 2

Find Problems Early, Fix Them Quickly, Code with Confidence

Trevor Burnham
edited by Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

The Tao of Test-Driven Development

Test-driven development (TDD) is sometimes defined as writing tests first. Although that's an important part of the methodology, it's not the essence. The essence of TDD is rapid iteration. You'll find that you learn more quickly from iterating—writing small, easy-to-understand pieces of code one at a time—than you would from trying to plan out a complex program from the ground up. You'll discover bad assumptions and potential pitfalls before you invest too much work. And you'll find the process more enjoyable, a smooth incremental progression rather than an alternation between bursts of inspiration and plateaus of "What do I do next?"

Our project for this chapter will be a solver for the classic programming challenge Fizz Buzz.[9] Here are the rules of Fizz Buzz:

> Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz." For numbers which are multiples of both three and five print "FizzBuzz."

If that sounds simple to you, congratulations: you're a programmer!

In this section, you'll apply the TDD process to implementing a function that takes a number and returns the appropriate Fizz Buzz output. First, you'll write a single test, knowing that it'll fail. Second, you'll write an implementation that satisfies the test. Once the test is passing, you'll use Git to save your progress.

Starting from Failure

Create an index.js with a placeholder implementation of fizzBuzz(), so that your tests will have a valid referent:

```
// index.js
module.exports = (num) => `${num}`;
```

Now add an index.test.js with a test for a single Fizz Buzz rule:

```
// index.test.js
const fizzBuzz = require('./index');

describe('fizzBuzz()', () => {
  it('returns "FizzBuzz" for multiples of 3 and 5', () => {
    expect(fizzBuzz(15)).toBe('FizzBuzz');
    expect(fizzBuzz(30)).toBe('FizzBuzz');
  });
});
```

9. http://wiki.c2.com/?FizzBuzzTest

Run the test:

```
$ npm test

 FAIL  ./index.test.js
  fizzBuzz()
    × returns "FizzBuzz" for multiples of 3 and 5 (2 ms)

  ● fizzBuzz() › returns "FizzBuzz" for multiples of 3 and 5

    expect(received).toBe(expected) // Object.is equality

    Expected: "FizzBuzz"
    Received: "15"

      3 | describe('fizzBuzz()', () => {
      4 |   it('returns "FizzBuzz" for multiples of 3 and 5', () => {
    > 5 |     expect(fizzBuzz(15)).toBe('FizzBuzz');
        |                          ^
      6 |     expect(fizzBuzz(30)).toBe('FizzBuzz');
      7 |   });
      8 | });

      at Object.toBe (index.test.js:5:26)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.178 s
Ran all test suites.
```

You may have cringed when you saw that glowing red FAIL. After all, having tests fail against production code is bad. But having tests fail during development can be a good thing! It means that you've anticipated some way your code *could* fail. Think of every failing test you see during development as a potential bug you've got a chance to preemptively squash.

Running Jest Tests Automatically

Jumping to the console every time you want to run some tests is a chore. Happily, Jest has a "watch mode" in which it automatically re-runs all tests when it detects any change to a test file, or to a source file depended on by a test.

To start Jest in watch mode, run it with the --watchAll flag:

```
$ npx jest --watchAll
```

Now you should see the same failure result as before. Try saving either index.js or index.test.js, and the test will re-run. (Blink and you might miss it!) You can press q at any time to quit. For now, leave Jest watch mode running.

Getting to Green

Since Jest is watching your project, see if you can tackle the "'FizzBuzz'" test case:

```
// index.js
module.exports = (num) => {
  if (num % 15 === 0) return 'FizzBuzz';
  return `${num}`
};
```

As soon as you hit save, your console output should change:

```
 PASS  ./index.test.js
  fizzBuzz()
    ✓ returns "FizzBuzz" for multiples of 3 and 5 (1 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.114 s, estimated 1 s
Ran all test suites.
```

Achievement unlocked: you've just completed a test-driven development cycle!

Measuring Test Coverage

A great feature of Jest is its built-in code coverage measurement. This shows you how much of the code being tested actually ran during tests. To compute code coverage, add the `--coverage` flag:

```
$ npx jest --coverage

PASS  ./index.test.js
  fizzBuzz()
    ✓ returns "FizzBuzz" for multiples of 3 and 5 (1 ms)

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |      75 |       50 |     100 |   66.66 |
 index.js |      75 |       50 |     100 |   66.66 | 3
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.133 s, estimated 1 s
Ran all test suites.
```

Here the report shows 75% of statements were covered, and 50% of branches. "Branches" refer to the possible outcomes of `if`/`else` statements. The 50% result reflects the fact that the current test only covers the case where the condition `num % 15 === 0` passes. Try adding a test to cover the case where it fails:

```
// index.test.js
const fizzBuzz = require('../index');

describe('fizzBuzz()', () => {
  it('returns "FizzBuzz" for multiples of 3 and 5', () => {
    expect(fizzBuzz(15)).toBe('FizzBuzz');
    expect(fizzBuzz(30)).toBe('FizzBuzz');
  });

➤  it('returns the given number for multiples of neither 3 nor 5', () => {
➤    expect(fizzBuzz(1)).toBe('1');
➤    expect(fizzBuzz(22)).toBe('22');
➤  });
});
```

Then run the test with code coverage again:

```
$ npx jest --coverage

 PASS  ./index.test.js
  fizzBuzz()
    ✓ returns "FizzBuzz" for multiples of 3 and 5 (1 ms)
    ✓ returns the given number for multiples of neither 3 nor 5

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 index.js |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.153 s, estimated 1 s
Ran all test suites.
```

Perfect! The report confirms that every possible code path was taken when the tests ran.

Like all metrics, code coverage is imperfect—projects with impressive code coverage numbers don't necessarily have the most *useful* tests—but the numbers can still guide you in the right direction. It's especially handy for identifying parts of your project with large gaps in test coverage.

Checking in Changes

Whenever you add a new test and get it to pass, that's a good time to get your project into source control. That way, no matter what you do to the project, you can always restore it to the all-green state later.

We'll use Git as our source control system in this book. If you're not familiar with Git, you might want to read through the "Git Basics" section of the excellent (and free) *Pro Git* by Scott Chacon and Ben Straub.[10]

The first step is initializing this project as a Git repository:

```
$ git init
Initialized empty Git repository in
/Users/tburnham/code/test-driven-fizzbuzz/.git/
```

Don't commit just yet. Although your project is brand new, there are a *staggering* number of files in it! Remember those "178 packages" npm mentioned when you installed Jest? They're all hanging out in the project's node_modules directory. Fortunately, we don't need to keep them in source control, because all of the information needed to re-create the node_modules tree is contained in package-lock.json. So tell Git to ignore the installed packages by creating a .gitignore file at the root of the project:

ch1/.gitignore
```
node_modules/
```

There. Now the project looks a lot more manageable, from Git's point of view:

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        index.js
        index.test.js
        package-lock.json
        package.json

nothing added to commit but untracked files present (use "git add" to track)
```

All of those files belong in source control, so stage them for commit:

```
$ git add .
```

Just for fun, this book uses gitmoji[11] for all of its commit messages. These are ASCII-friendly aliases that render as emoji on GitHub and in some other tools. For a project's first commit, the appropriate gitmoji is :tada:, which represents the "Party Popper" emoji:[12]

10. https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository#ch02-git-basics-chapter
11. https://gitmoji.carloscuesta.me
12. https://emojipedia.org/party-popper/

```
$ git commit -m ":tada: First commit"
[main (root-commit) dca2255] :tada: First commit
 5 files changed, 5893 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 index.js
 create mode 100644 index.test.js
 create mode 100644 package-lock.json
 create mode 100644 package.json
```

Congrats on completing your first feature! You wrote a test for the feature, made the test pass, and then checked that change into source control. Satisfying, isn't it?

As an exercise, see if you can repeat the TDD process for the remaining Fizz Buzz requirements. Namely, your fizzBuzz() function should return:

1. "Fizz" for multiples of 3,
2. "Buzz" for multiples of 5, and
3. The given number for multiples of neither 3 nor 5

For each of those requirements, add a test within the same suite (the describe() block), modify the implementation to make everything pass, then move to the next requirement. You can find an example solution at the end of the chapter.