Testing

CS 246

Objectives

- Students will be able to:
  - explain the importance of testing
  - describe the hierarchy of testing

Overview

- Time for a word cloud
  - write the adjectives to characterize good software
  - time for a word cloud - pollEv.com/mprogers145

Behold, a Word Cloud!

Testing

- Testing starts with **unit testing**, where we test individual functions to guarantee that they produce the intended results

- Once unit testing has confirmed that the individual pieces work, **integration testing** tests that the pieces, assembled, continue to work as expected

- **Usability testing** is designed to test that the product is intuitive and easy-to-use

- These notes focus on unit testing using Vitest, a popular and programmer-friendly testing framework

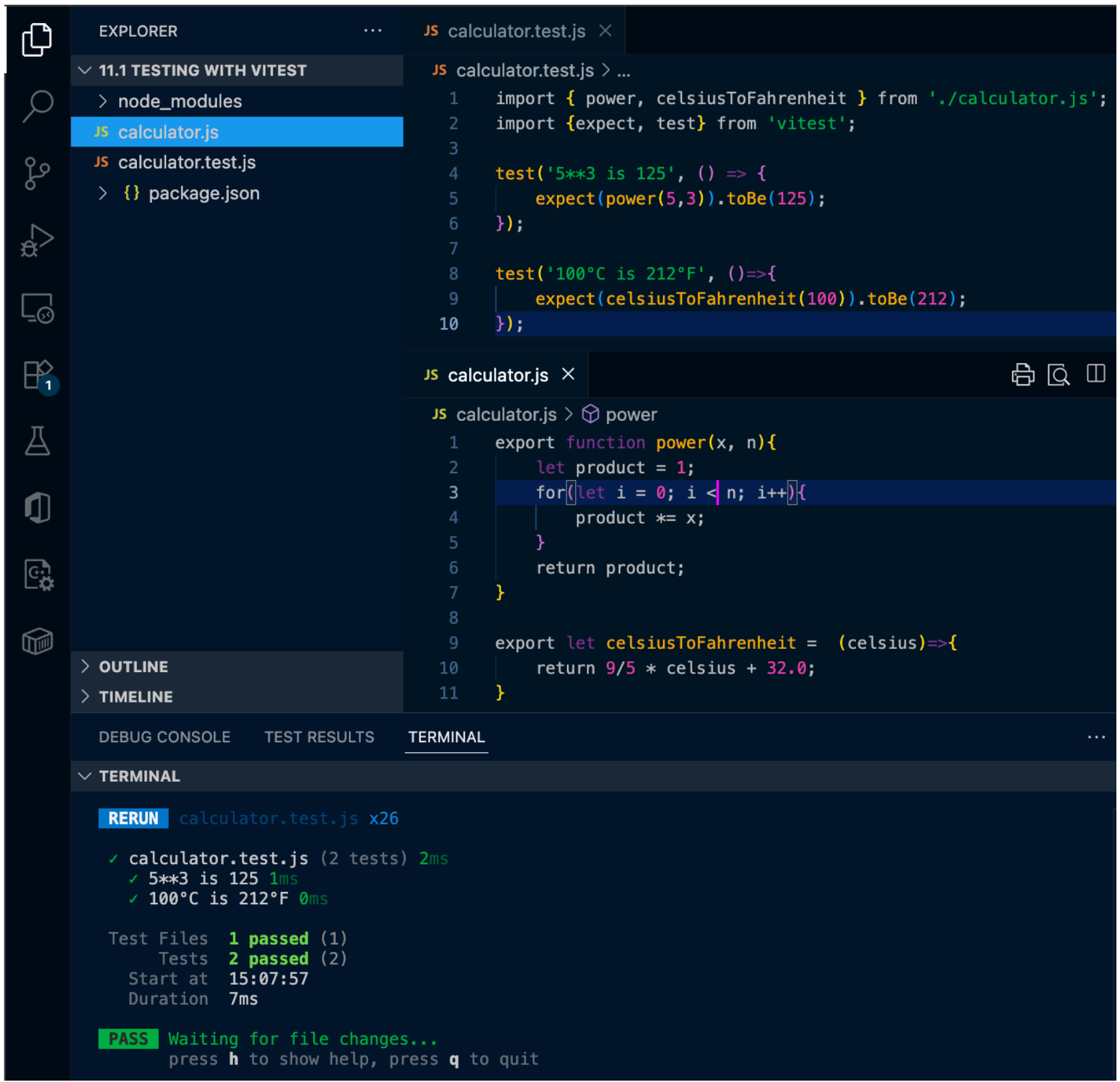
Unit Testing in 5 Easy Steps* Using Vitest

1. Create a Node.js project (npm init -y)
    1. configure your package.json file to use ES6 modules
    2. modify the test script to invoke vitest
2. npm install -D vitest # installs vitest as a dev dependency
3. Write code to do *something*
4. Write a series of tests using vitest
5. Run the tests using **npm test**
6. Inspect the terminal to see if your tests passed

```
{
  "name": "11.1-testing-with-vitest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "vitest"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "vitest": "^3.2.4"
  }
}
```

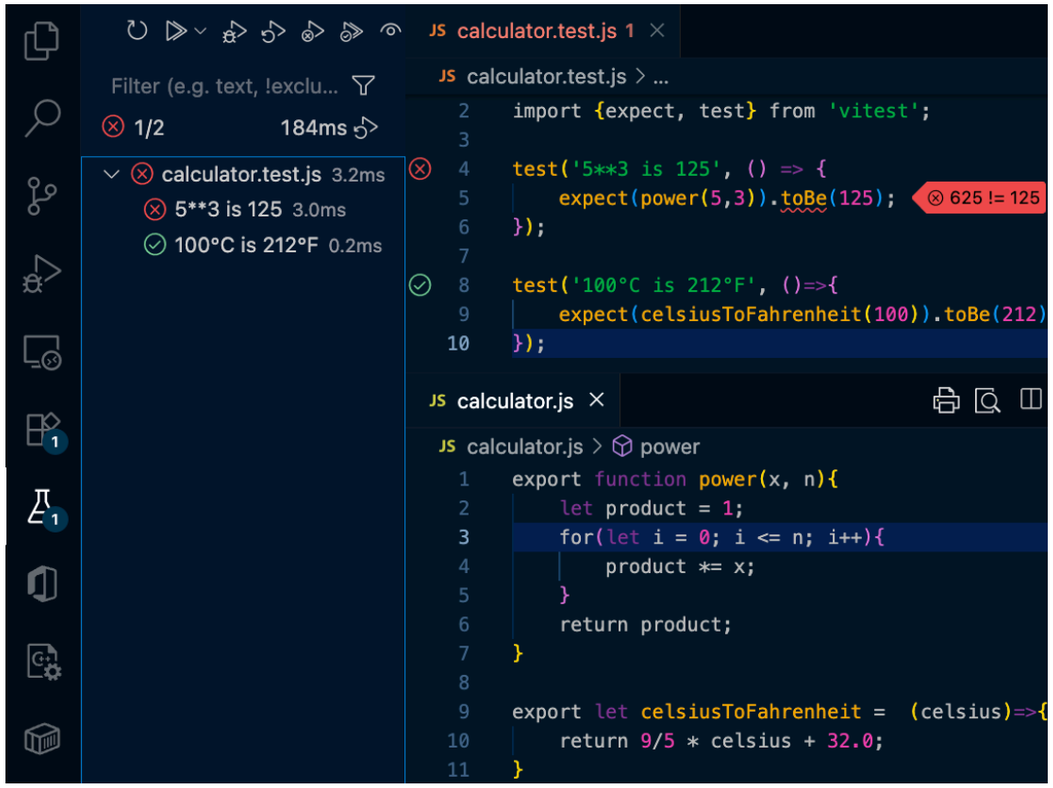
*and one hard one

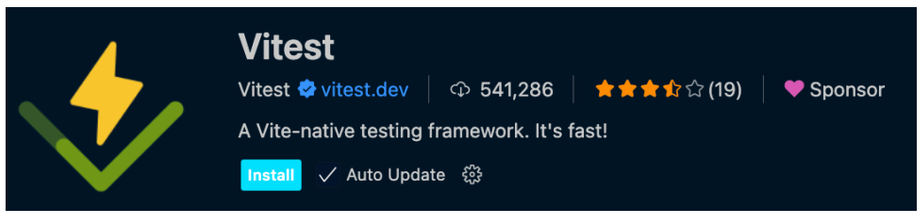
Vitest Example

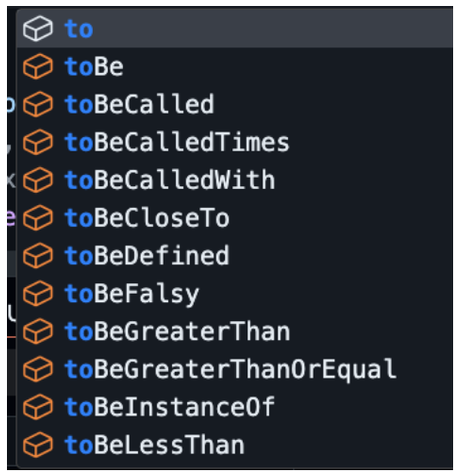


Test Details

- Tests must be in a file that contains .test. or .spec. in the file name
- As soon as you change your source code, the tests will be automatically rerun 😍
- The Vitest VS Code extension simplifies testing by displaying tests in the Testing view, but these tests won't rerun automatically when code changes
- test(description, testFunction), where testFunction contains a body with an assertion: () => expect(functionToTest).toBe (expected)
- Other "matcher" methods, like toEqual(), toBeGreaterThan(), etc., are possible







Designing Useful Tests

- Suppose a function is designed to work with values between 0.0-100.0

- What values should you test?
  - At the least, 0.0, 100.0, and 50.0

  - If you know that the function behaves fundamentally differently in different ranges [0.0,25.0), [25.0,50.0), [50.0, 75.0), and [75.0,100.0] you will need to design tests that check all the edge cases

  - How many tests are enough?

Test Coverage

- **Statement coverage**: Need to ensure that every statement gets tested
  - Otherwise, missed statements might contain bugs
- **Path coverage**: Need to ensure that every path through the code gets executed

Statement v. Path Coverage

- Call someFunNow() to achieve
  - full statement coverage
  - full path coverage

```
function someFunNow(input, cond1, cond2, cond3) {
  let x = input;
  let y = 0;

  if (cond1) {
    x++;
  } else {
    x--;
  }

  if (cond2) {
    x--;
  } else {
    x++;
  }

  if (cond3) {
    y = x;
  } else {
    y = -x;
  }

  return y;
}
```

Vitest Lives up to its Name: test.each()

- Clearly, we need someway to test a lot of different scenarios, and writing a test() all for each could be exhausting.

- Fortunately test.each() allows you to specify an array of test values

```
describe('power tests', () => {
    test.each([
        [2,3,8],
        [2,4,16]
    ])('%i ** %i = %i', (a, b, expected) => {
        expect(power(a,b)).toBe(expected);
    });
});
```