

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/mprovers145

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

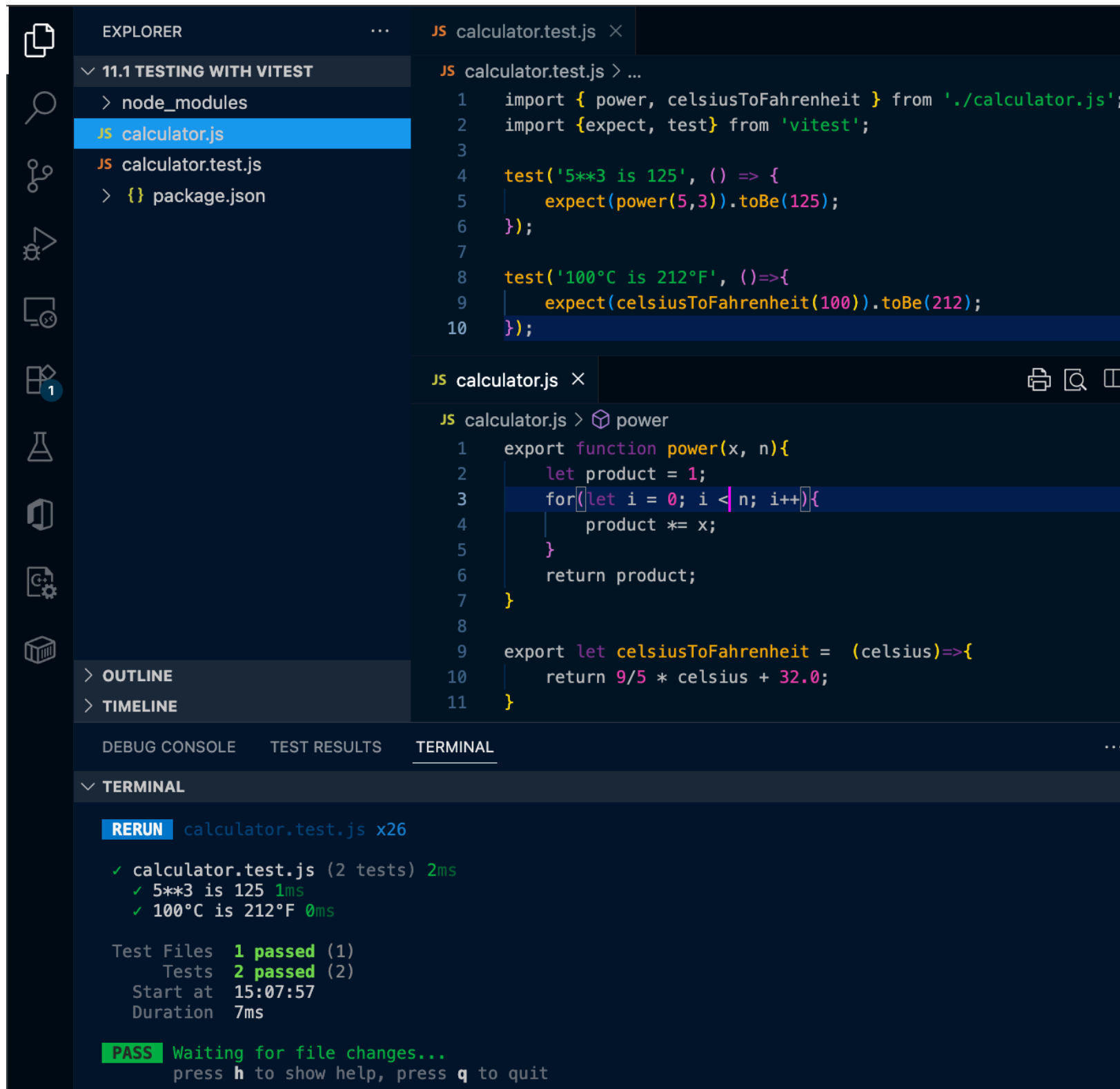
5. Run the tests using **npm test**

6. Inspect the terminal to see if your tests passed

*and one hard one

```
{  
  "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"  
  }  
}
```

Vitest Example



The screenshot displays the Visual Studio Code interface for a project named 'calculator'. The Explorer sidebar on the left shows the file structure under '11.1 TESTING WITH VITEST', including 'node_modules', 'calculator.js', 'calculator.test.js', and 'package.json'. The Editor window shows the 'calculator.test.js' file with the following code:

```
1 import { power, celsiusToFahrenheit } from './calculator.js';
2 import { expect, test } from 'vitest';
3
4 test('5**3 is 125', () => {
5   expect(power(5,3)).toBe(125);
6 });
7
8 test('100°C is 212°F', ()=>{
9   expect(celsiusToFahrenheit(100)).toBe(212);
10 });
```

Below the test file, the 'calculator.js' file is open, showing the implementation of the 'power' and 'celsiusToFahrenheit' functions:

```
1 export function power(x, n){
2   let product = 1;
3   for(let i = 0; i < n; i++){
4     product *= x;
5   }
6   return product;
7 }
8
9 export let celsiusToFahrenheit = (celsius)=>{
10   return 9/5 * celsius + 32.0;
11 }
```

The Terminal at the bottom shows the test results for 'calculator.test.js' (2 tests) in 2ms. The tests passed:

- ✓ 5**3 is 125 1ms
- ✓ 100°C is 212°F 0ms

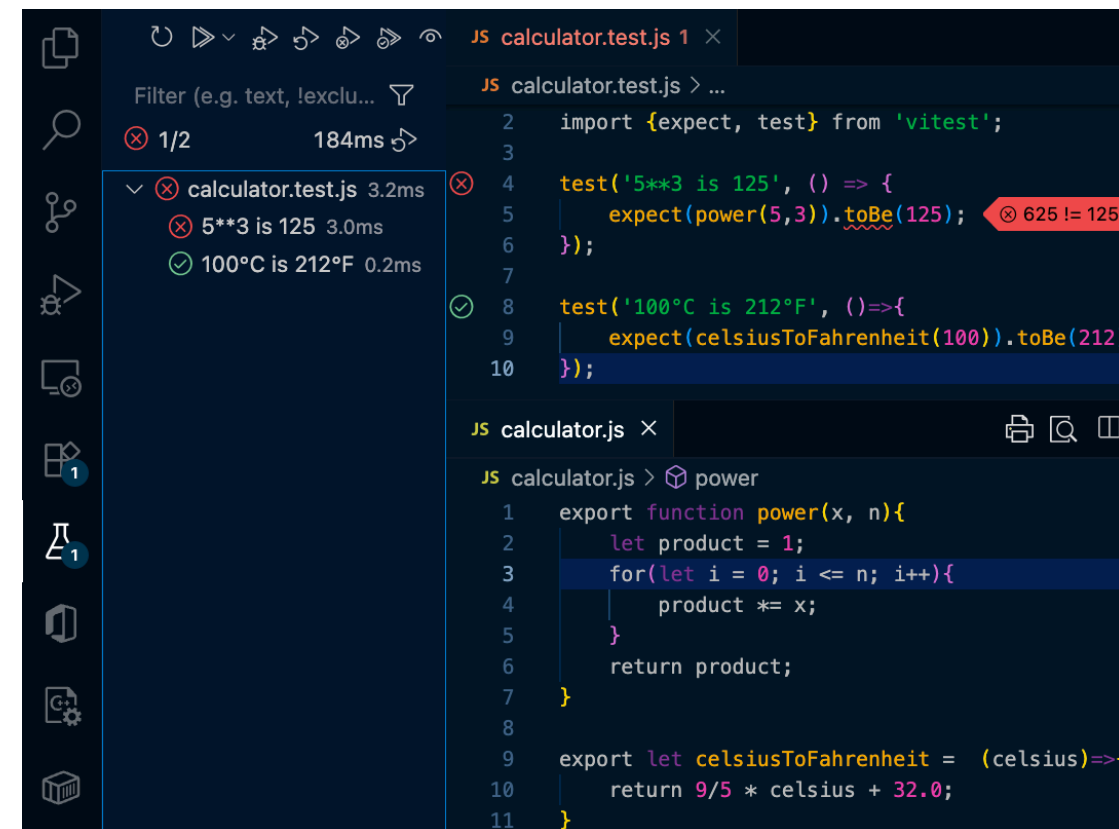
Summary statistics:

- Test Files: 1 passed (1)
- Tests: 2 passed (2)
- Start at: 15:07:57
- Duration: 7ms

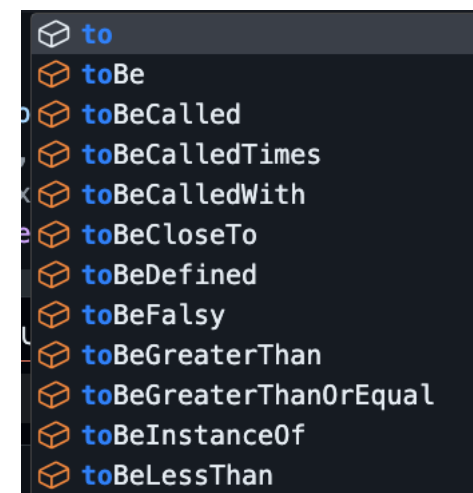
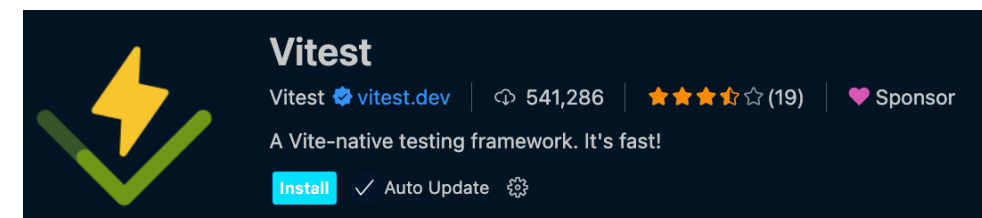
The terminal also shows a 'PASS' status and a message 'Waiting for file changes...'.

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



The screenshot shows the VS Code interface with the Vitest extension. On the left, the 'Testing' view displays a list of tests for 'calculator.test.js'. The first test, '5**3 is 125', is failed (indicated by a red 'x' and '3.0ms'). The second test, '100°C is 212°F', is passed (indicated by a green checkmark and '0.2ms'). The main editor shows the source code for 'calculator.test.js' and 'calculator.js'. The test function for '5**3 is 125' is highlighted, showing an assertion failure: `expect(power(5,3)).toBe(125);` with a red error message `625 !== 125`. The source code for 'calculator.js' shows the `power` function and the `celsiusToFahrenheit` function.



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);  
  });  
});
```