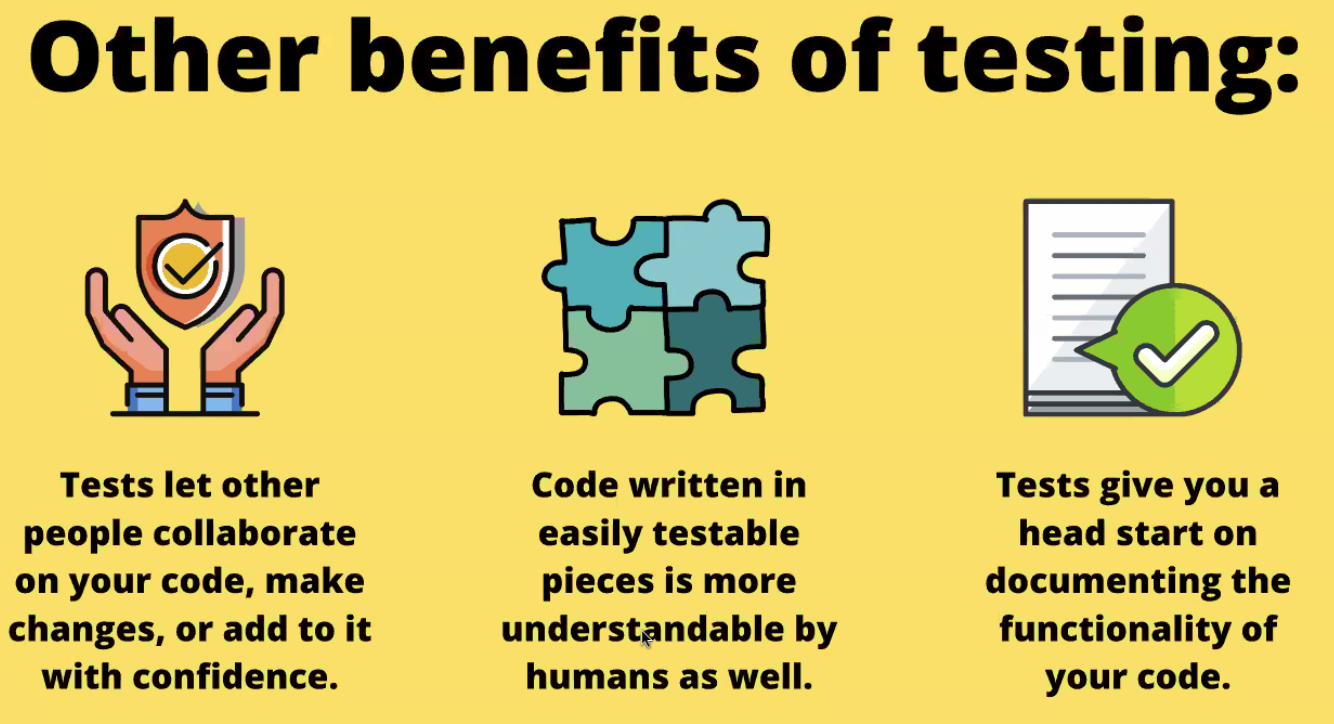
**Testing**

Imagine a hiker in the countryside, comes to a river that needs to get across. If you only need to solve it as a one-off problem then you might put a log across. But if lots of people are going to use it, we need a bridge! Sturdier, long term solution. This is the different between programming and software developing – using a log is just programming. But to ensure the bridge holds and is safe, it needs checking regularly and be maintained. Now imagine the golden gate bridge – this is your equivalent of Google or Twitter. If you’re replacing even just 1 bolt on the bridge, you need to ensure that you don’t break the bridge! This is why we have testing.

**Code testing**

A series of checks that tells us if our program does what we want it to do, every time! This involves asking it a lot of the same questions, especially as the code changes, to ensure it still gives you the correct answer. Helps ensure the code you’re writing doesn’t break another part of the program too!



“The more your tests resemble the way your software is used, the more confidence they can give you.” – Kent C. Dodds

So make sure the tests reflect the way a user would be using it!

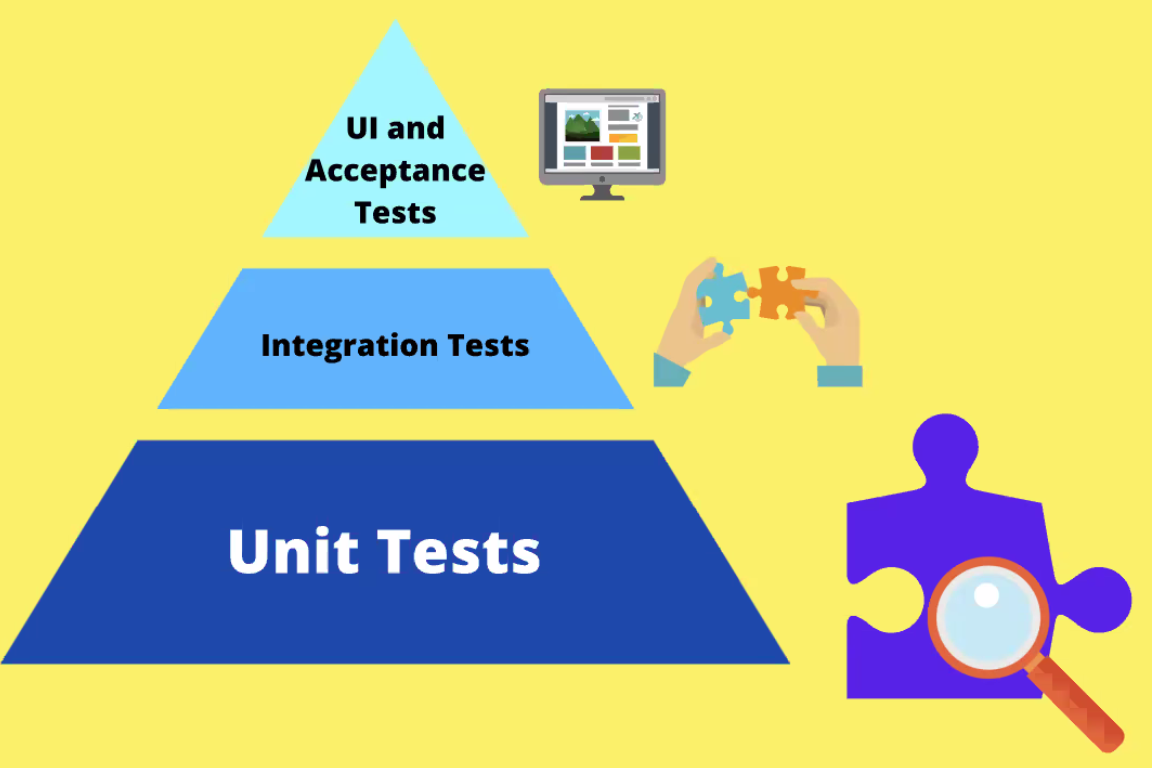
**The Testing Pyramid:**

Similar to what Nadeem showed us in his guest talk.

**Unit** means the smallest possible bit of your code that you can test in isolation (e.g. a function). These are the quickest and cheapest type of test to write and run, which also makes them cheap (in terms of time, people, resource etc)

**Integration Tests** test how different units of your code interact/integrate (e.g. how 2 functions might work together). Slower and more expensive so fewer tests than unit tests, but still important as need to ensure code works with other code

**UI and Acceptance Tests** test your user experience from end to end. Time-consuming, expensive and hard to automate, but do test the whole experience.



For today, we are focussing on Unit Tests only.

**Jest**

A library for testing JavaScript. Other frameworks are available! (always trade-offs). But Jest a good choice as common and in-built to many things.

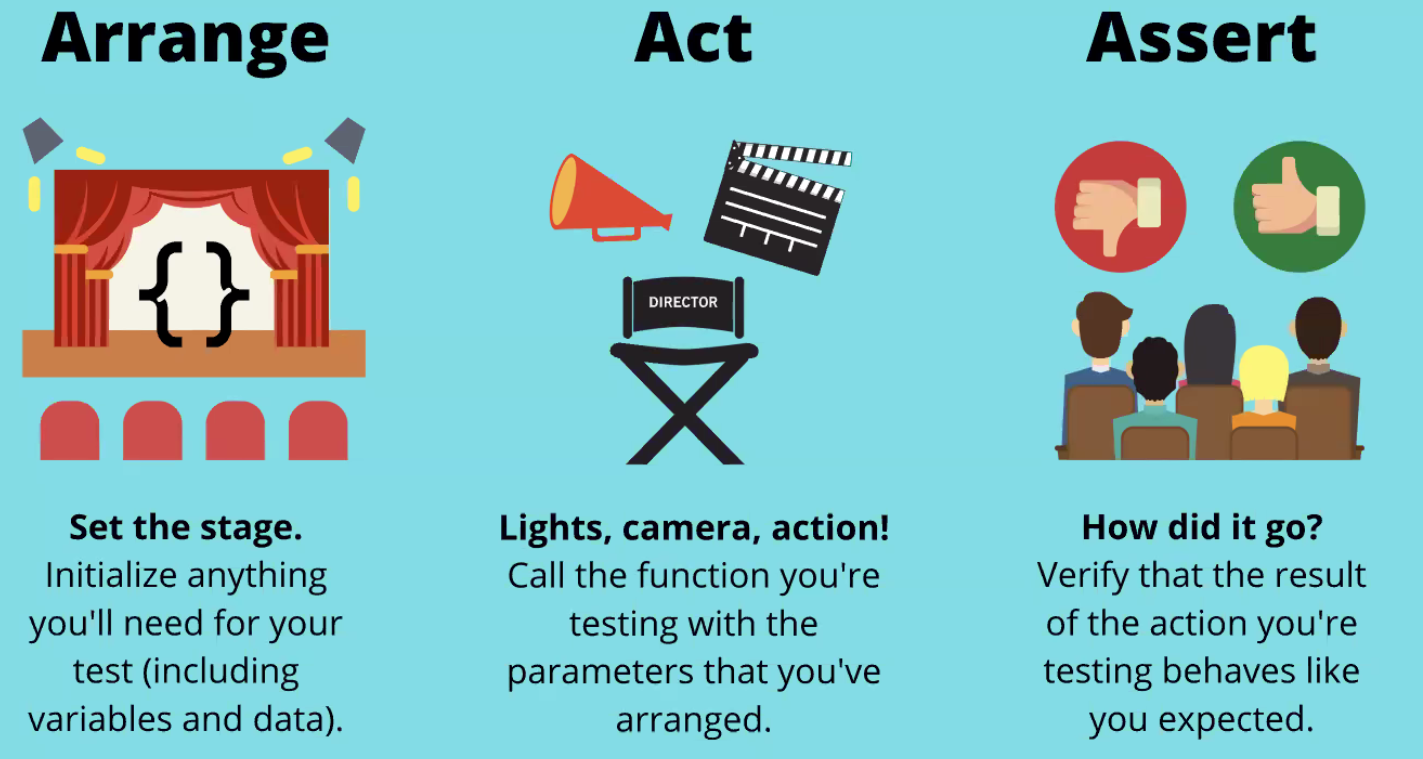
We were thrown into using Jest just by reading the docs (no syntax from Liz etc) and got it to work straight away! This is an example of what you will have to do in the industry! 😊

**Pattern for Testing:**

Arrange – Think of a theatre, this is “setting the stage”, actors in costume etc.

Act – just like the director calling “action”, this is actually calling the function we’re testing

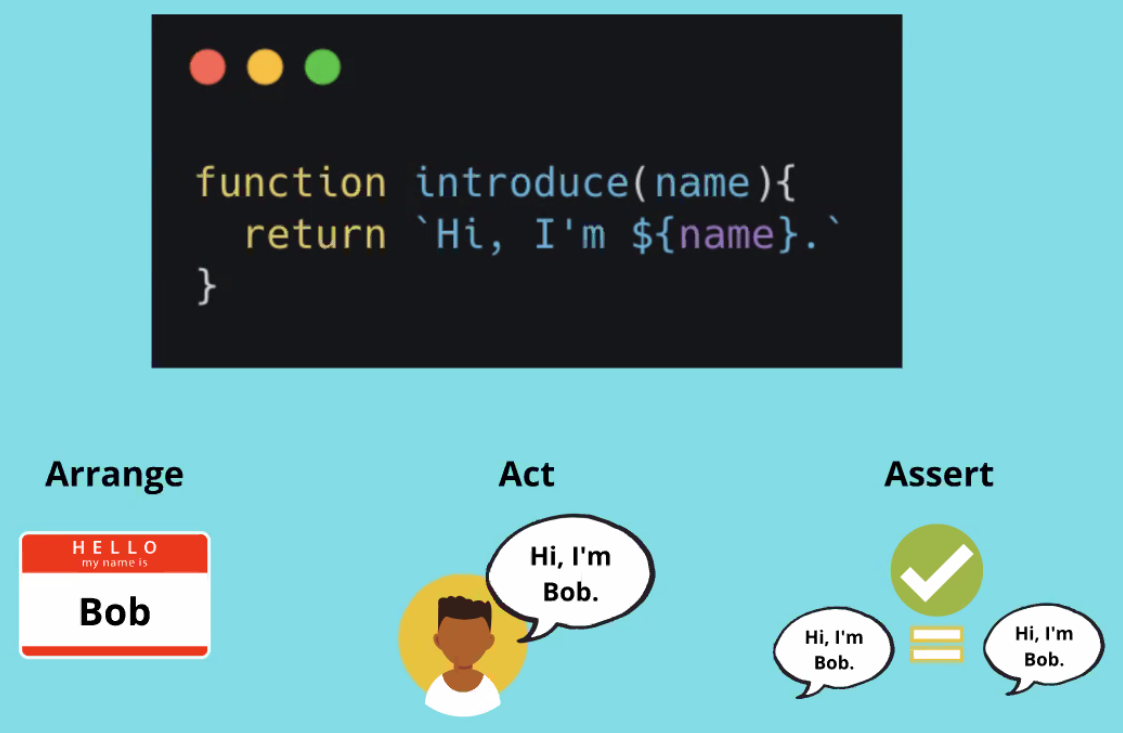
Assert – Did it work?

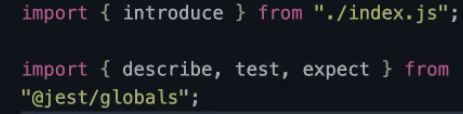


This pattern is language agnostic – use this basic layout of testing in any programming language.

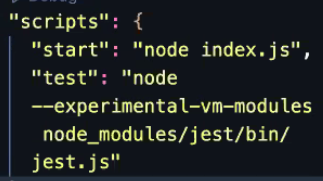
Tests are just like any other code – designed to be read by humans! So make sure they are clear.

Basic example of test:



/\*

THE PLAN

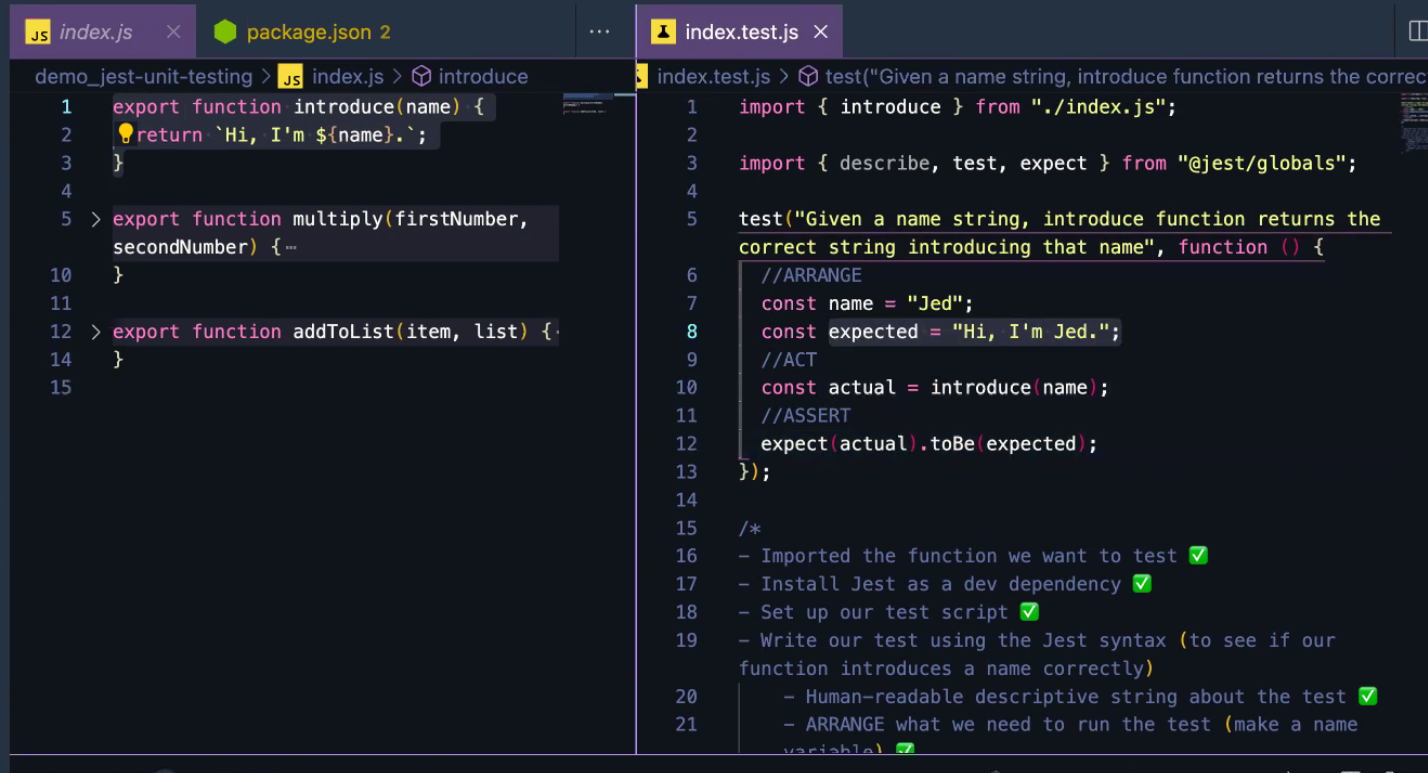
* Import the function we want to test
* Install Jest as a dev dependency
* Set up our test script (use the experimental bit for ESM)
* Write our test using the Jest syntax (to see if our function introduces a name correctly)
  + Human-readable descriptive string about the test
  + ARRANGE what we need to run the test (make a name variable)
  + ACT (run the function with the name)
  + ASSERT (see if we got the results we were looking for)

/\*

Not an exact science but the **descriptive string should use the GIVEN, WHEN, THEN framework**

Given *these variables*, When *I call the function* Then *This should happen*

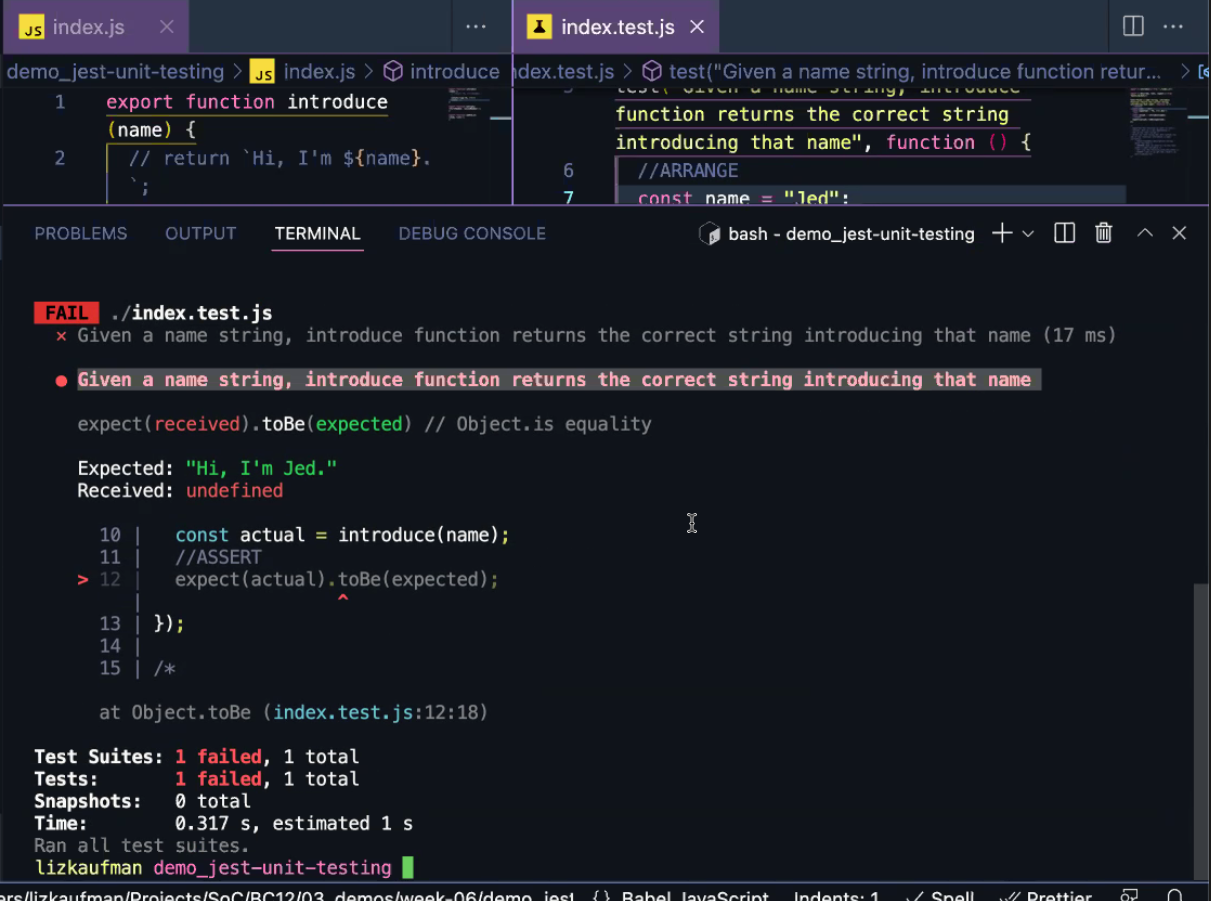
Finished code:



Run the file using npm test (if you’ve set the script in package.json).

Don’t need to do line 3 as Jest puts these into the global environment automatically (when it’s installed), but if you prefer explicit imports then you can use this (<https://jestjs.io/docs/api>). Trade-offs = Explicit imports make it more readable but not necessarily needed.

Here’s what a failed tests looks like (due to console.logging the statement in the introduce function instead of returning. It shows you what was expected and what it actually received!



Writing tests where you may get multiple results

export function multiply(firstNumber, secondNumber) {

if (typeof firstNumber !== “number” || typeof secondNumber !== “number”){

return “Try again with numbers!”;

}

return firstNumber \* secondNumber;

}

/\*

PLAN for if we give numbers to the function

- Test multiply function with two numbers, making sure it consistently returns the multiplied product of those

- Import the multiply function

- Structure our Jest test with the test function

- First arg: Human-readable string to describe what we’re testing

- Second arg: anonymous function

- ARRANGE

- two variables (that are numbers)

- What we expect the result to be

- ACT

- Call multiply function (store in variable)

- ASSERT

- Is it correct or not? (expect, toBe)

\*/

test(“Given two numbers, when calling multiply, then return multiplied product of those two numbers”, function () {

// ARRANGE

const firstNumber = 6;

const secondNumber = 7;

const expected = 42;

// ACT

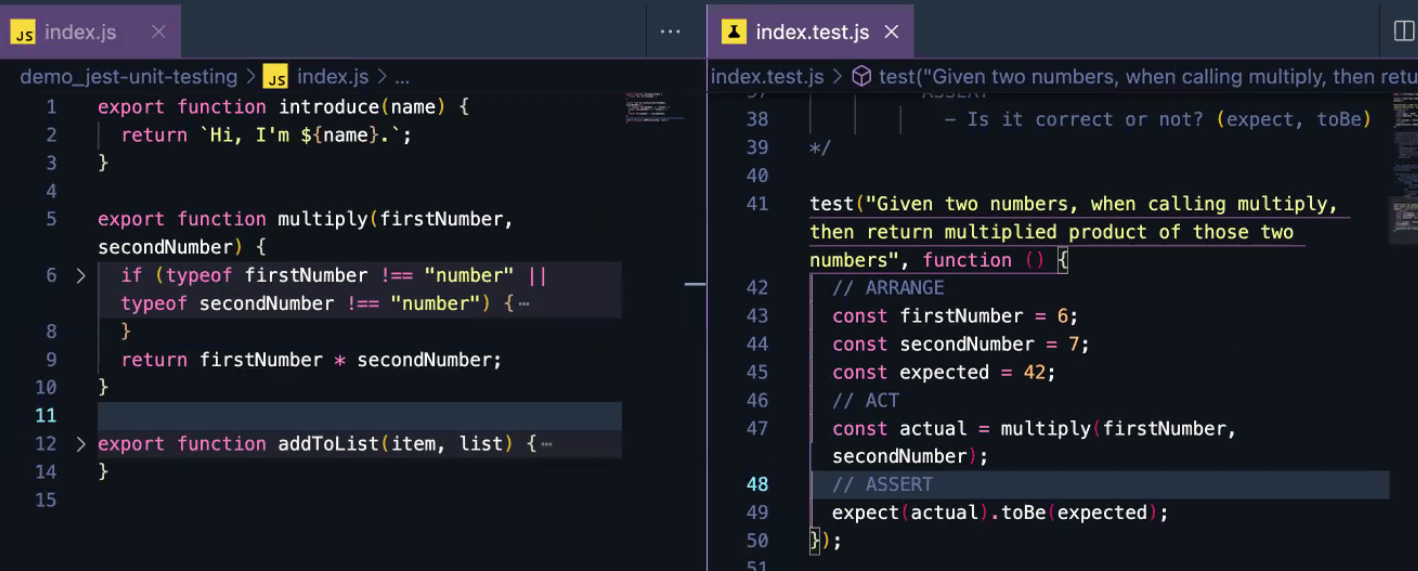
const actual = multiply(firstNumber, secondNumber);

// ASSERT

expect(actual).toBe(expected);

}

);

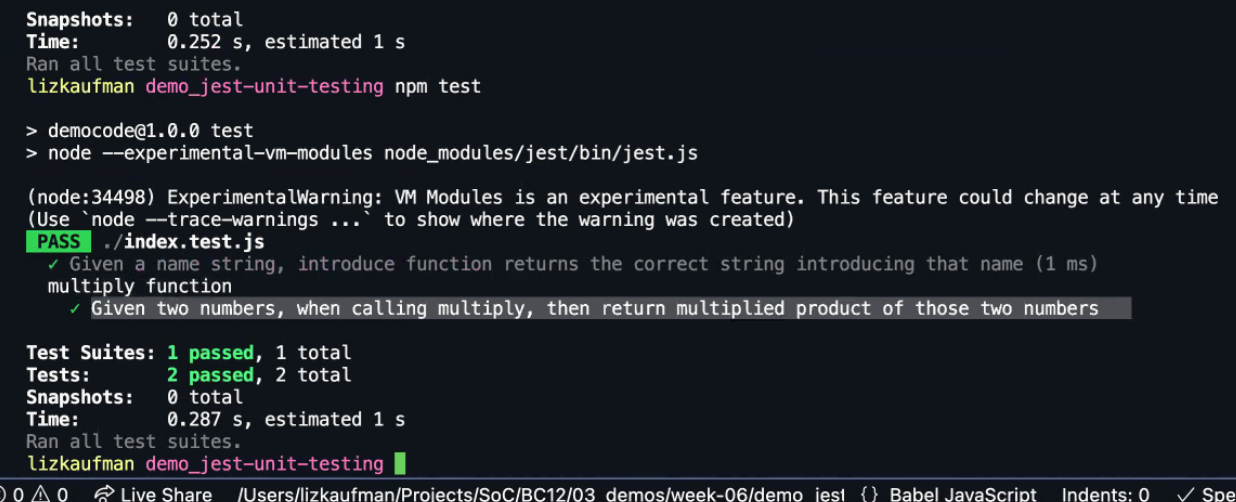


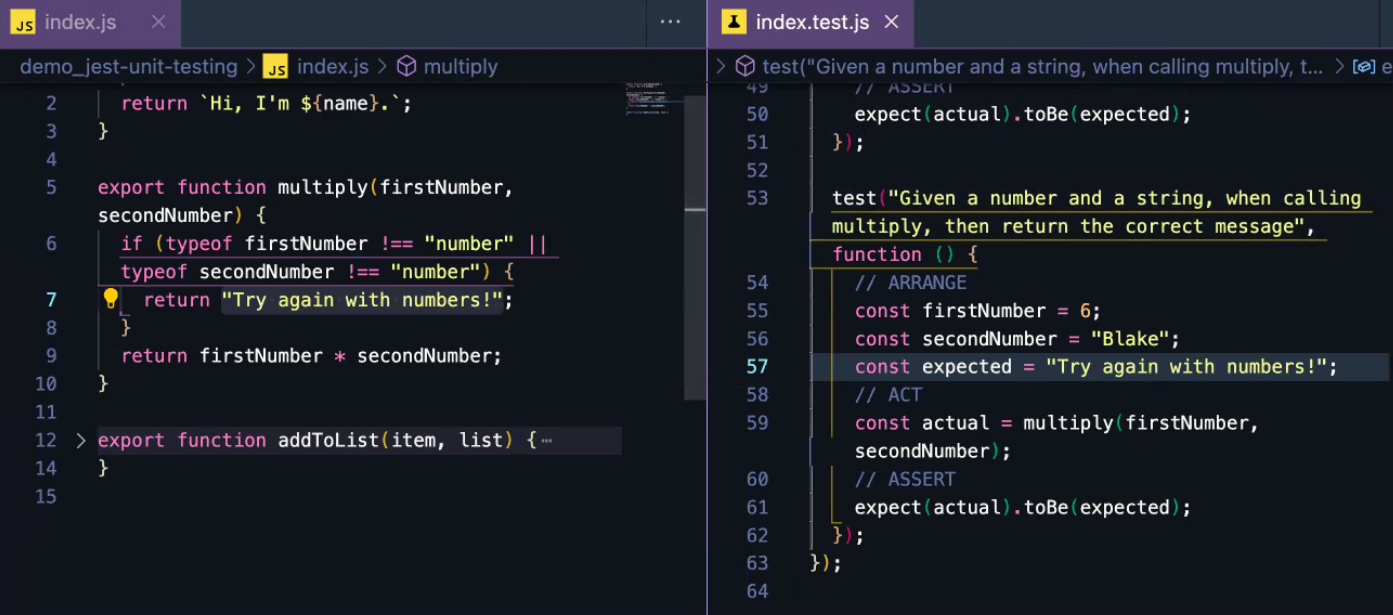
**Describe**

Used when you have multiple tests on the same unit. Gives a nice wrapper to the test result. (<https://jestjs.io/docs/api#describename-fn>)

Describe(‘multiply function’, function()){}

Gives a header of **multiply function** to the test results so you can easily see each unit





**With an array/object**

Export function addToList(item, list) {

Return […list, item];

}

…list is an immutable way of using array.push() method (not important for now!)

Describe(‘addToList’, function () {

test(‘Given a string and an array, addToList returns an array that now includes that string’, function () {

//ARRANGE – string, array, expected

const item = “banana”;

const list = [“apple”, “grapes”, “kiwis”];

const expected = [“apple”, “grapes”, “kiwis”, “banana”];

//ACT

const actual = addToList(item, list);

//ASSERT

Expect(actual).toBe(expected);

});

});

Primitive types (string, number etc) are always fixed and we are pointing telescopes at them.

Arrays and objects can be changed. So if you use actual and expected to test arrays/objects then it’s not exactly the same thing because they are 2 different arrays!

So hence the message: If things need to pass with “deep equality”, replace “toBe” with “toStrictEqual”

This checks each thing in the array to ensure they match each other.

(<https://jestjs.io/docs/expect#tostrictequalvalue>)

Loads of other matchers able to use: <https://jestjs.io/docs/using-matchers>

