

# Unit Tests

In this lesson, we'll learn what Unit Tests are and how to build and run them.

Unit tests should be straightforward and answer, “given these inputs, what are the expected outputs.”

```
function runner({inputs, expectedOutputs, func}) {
  assert(inputs.length === expectedOutputs.length);
  for (let i = 0; i < inputs.length; i++) {
    assert(func(inputs[i]) === expectedOutputs[i]);
  }
}

function firstNameTest() {
  const invalidInputs = ["@", "", "blah$", "123"];
  const validInputs = ["asdf", "Alfred", "ALFRED"];

  runner({
    inputs: validInputs,
    expectedOutputs: validInputs.map(_ => true),
    func: isValidName
  });

  runner({
    inputs: invalidInputs,
    expectedOutputs: invalidInputs.map(_ => false),
    func: isValidName
  });
}

function emailTest() {
  const invalidEmails = ["@asdf.com", "what@what", "", ".."];
  const validEmails = ["asdf@asdf.com", "what@what.au", "a@a.c"];



  runner({
    inputs: validEmails,
    expectedOutputs: validEmails.map(_ => true),
    func: isValidEmail
  });

  runner({
    inputs: invalidEmails,
    expectedOutputs: invalidEmails.map(_ => false),
    func: isValidEmail
  });
}
```

`assert` is a Node thing, which, as its name indicates, checks that the argument evaluates to `true` and throws an error if it's not.

We've structured our tests such that there's a `runner` function, and every test feeds a set of inputs, expected outputs, and the function used. The line, `x.map(_ => true)`, is just creating an array of equal size with the values `true` for every element. Remember, `map` applies a function to every element, `_` is the variable we use when we want to convey that it's unused, and `=>` is just shorthand for `function() { return true }`.

Since `assert` isn't available to us, we can display test results on the client instead. We can create a new HTML page, import the source code JavaScript, and the test code, and instead of `assert`, we'll display the results on the page. (This is similar to what a testing framework called [QUnit](#) does.)

Output
JavaScript
HTML
<h2>Test Results</h2> <p>isValidName(asdf) === true: <b>Fail</b> isValidName(Alfred) === true: <b>Fail</b> isValidName(ALFRED) === true: <b>Fail</b></p> <p>isValidName(@) === false: <b>Fail</b> isValidName() === false: <b>Fail</b> isValidName(blah\$) === false: <b>Fail</b> isValidName(123) === false: <b>Fail</b></p> <p>isValidEmail(asdf@asdf.com) === true: <b>Fail</b> isValidEmail(what@what.au) === true: <b>Fail</b> isValidEmail(a@a.c) === true: <b>Fail</b></p> <p>isValidEmail(@asdf.com) === false: <b>Fail</b> isValidEmail(what@what) === false: <b>Fail</b> isValidEmail() === false: <b>Fail</b> isValidEmail(..) === false: <b>Fail</b></p>
 

This should hopefully be a familiar code to you. A fun way of testing, right? It's not perfect by any means. For example, we're feeding in strings as input but don't have double quotes around it. That's easy to add, though: check the type of the input when formatting the test display.

See if you can make the name tests pass! Just fill in the `//TODO` in the JavaScript tab. Remember the rule: name must be a nonempty alpha string.

Now you can change the source code, and as you refresh the test runner page, it'll refresh the results of the tests.

Now that you're familiarized with unit tests, let's move on to integration tests.