</talentlabs>

CHAPTER 5

Automated Testing Basics

</talentlabs>

AGENDA

- Automated Testing Overview
- Automated Testing Types
- Writing Unit Tests in JavaScript

Automated Testing Overview

</talentlabs>

What is Software Testing?

Ensure:

Old feature are still functioning properly

New features are working as expected

Previous bugs are fixed

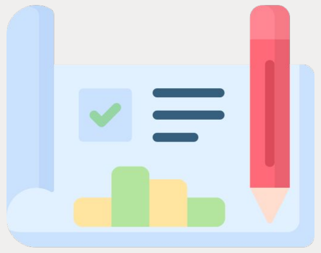
Can be done manually and/or automatically

</talentlabs>

Traditional Software Testing Process

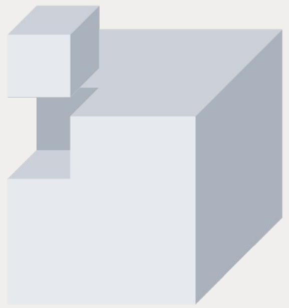
QA Team

01



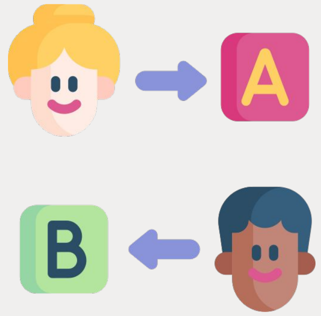
Create testing plan (test cases)

02



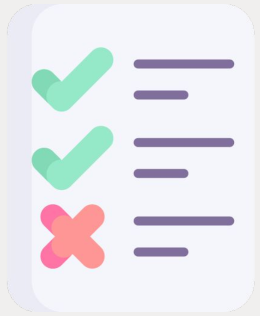
Break down test cases into smaller “packs”

03



Assign each team member a test packs and test the software manually

04



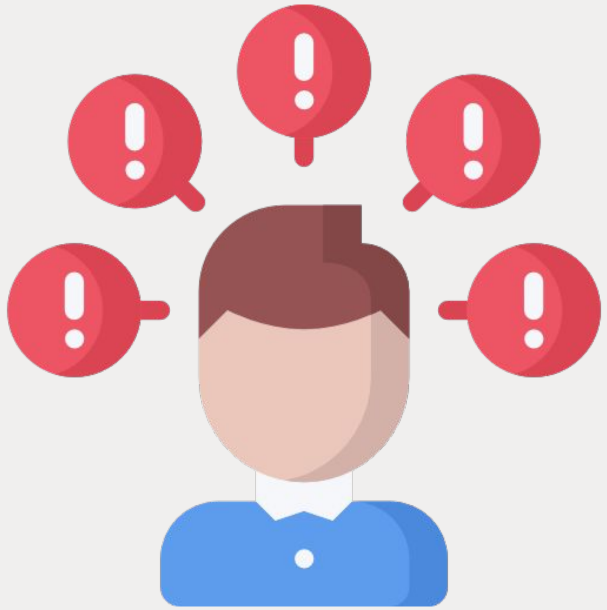
Share the testing report to software engineering team

Problems with Traditional Testing

- **Slow** - there are thousands of test cases that QA team need to manually run and click one by one

- **Low Coverage** - given the limit of time constraint, usually the team can only cover the popular features but not everything buttons in the software

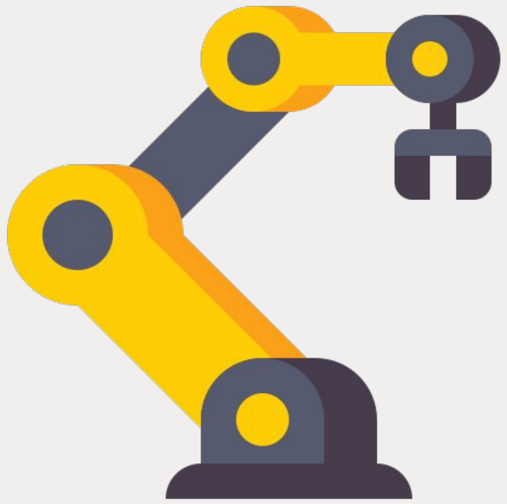
- **Expensive** - human resources are expensive and the manual process is very boring



</talentlabs>

Automated Testing as a Cure

- **Minimal Human Efforts** - Have the computer to click the buttons instead of human clicking the buttons
- **Short Testing Time** - Computers can "click" the buttons a lot faster than human
- **Reusable Test Cases** - Automated testing process can be "saved" and reuse every time



</talentlabs>

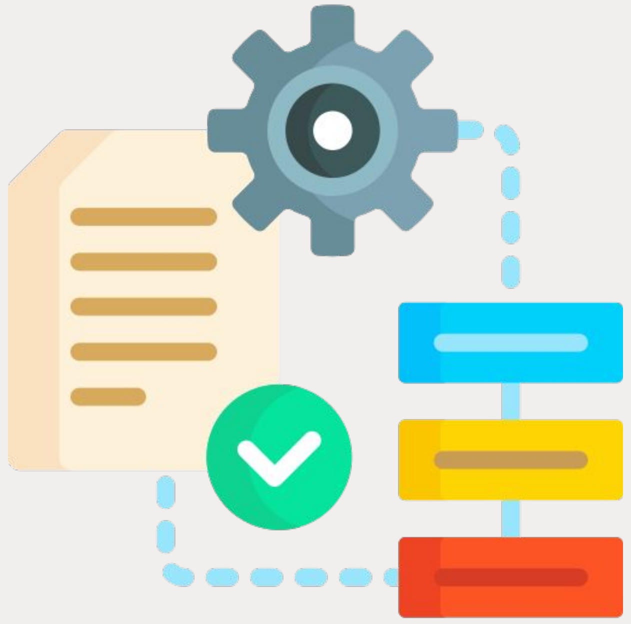
Automated Testing Types

</talentlabs>

Testing Types at Different Levels

Automated tests are categorised by the level of the testing

- Function level
- Code file level
- User interface level
- API level



</talentlabs>

Common Testing Types

1. Unit Testing
2. Integration Testing
3. Functional Testing
4. UI Testing
5. Smoke Testing
6. Regression Testing
7. Data Driven Testing

… and more

</talentlabs>

Common Testing Types

Commonly Used

1. Unit Testing
2. Integration Testing
3. Functional Testing
4. UI Testing

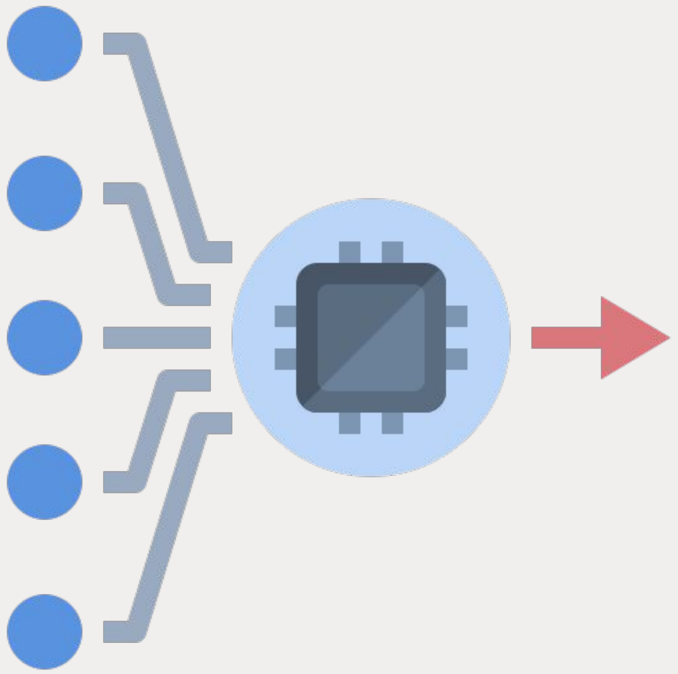
5. Smoke Testing
6. Regression Testing
7. Data Driven Testing

… and more

Unit Test

Testing Scope
Smallest logic unit of a piece of software (usually function)

Testing Method
Pass in different values to a function, and check if the output is the expected value



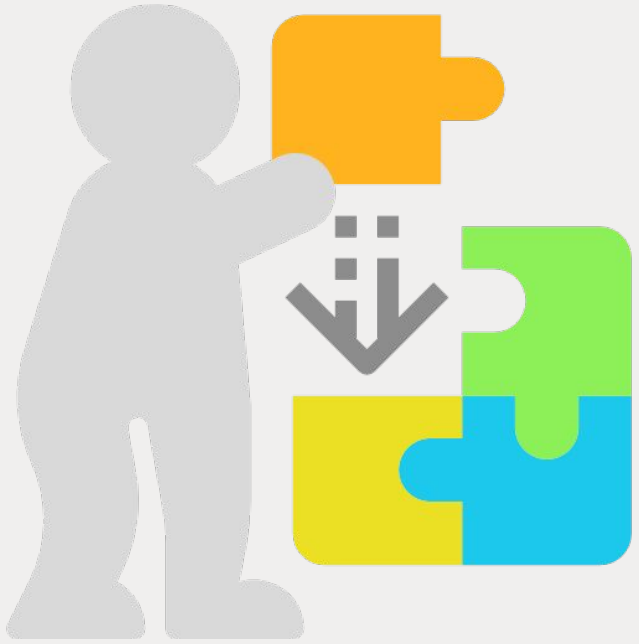
</talentlabs>

Integration Test

Testing Scope
Test multiple logical unit together
(e.g. testing multiple functions together, or testing an API)

Testing Method
Call an API or call some top level function, and see if the responses are correct

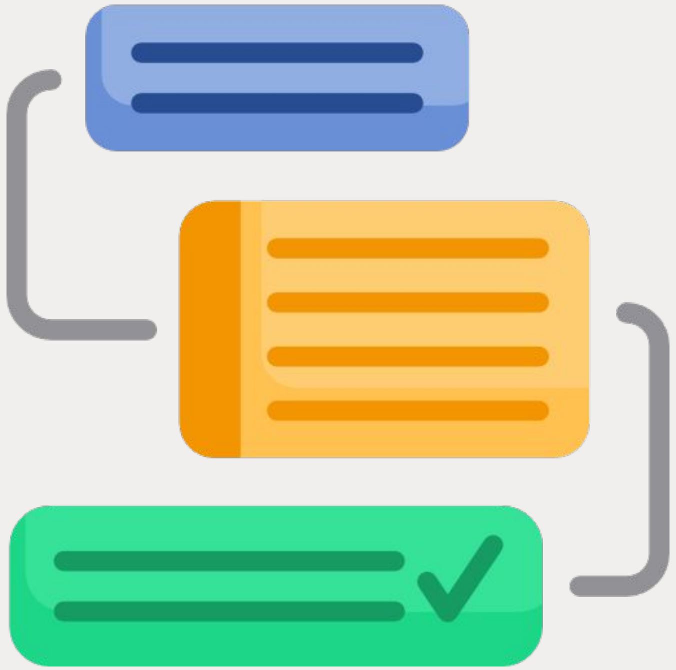
The API or top level function should be calling a few other functions.



Functional Testing

Testing Scope
Use case, from user’s perspective
(e.g. Test the whole flow of “register a new account” -> “sign in” -> “sign out”, which covered “registration API”, “Sign in API” and “Sign out API”)

Testing Method
Call multiple APIs or top level functions to simulate a users’ interaction with the software

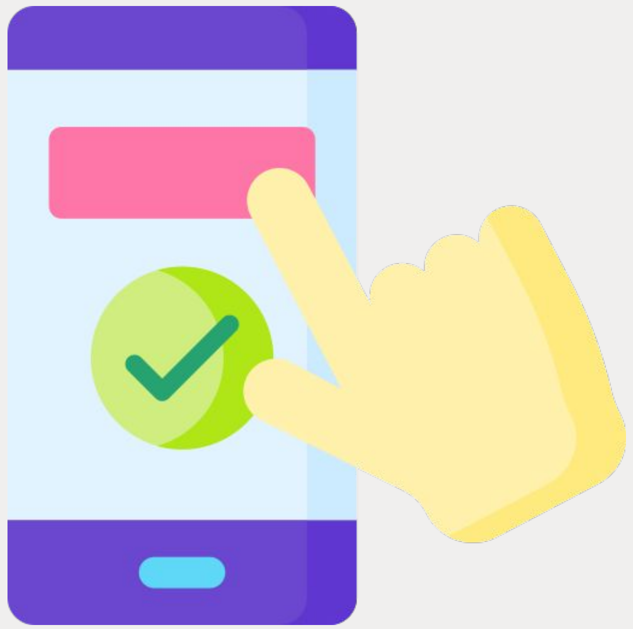


</talentlabs>

UI Testing

Testing Scope
User interface (interaction and information displayed)

Testing Method
Use UI automation tools to simulate clicking the buttons on the user interface



Writing Unit Tests in JavaScript

</talentlabs>

JavaScript Unit Test

- Most programming languages provides a built-in basic unit testing tool
- The built-in unit testing tool for JavaScript: assert
  - Provided by Node.js
  - No extra installation is needed

Create an Unit Test with Assert

01

Import the “assert” module

02

Write down the test cases

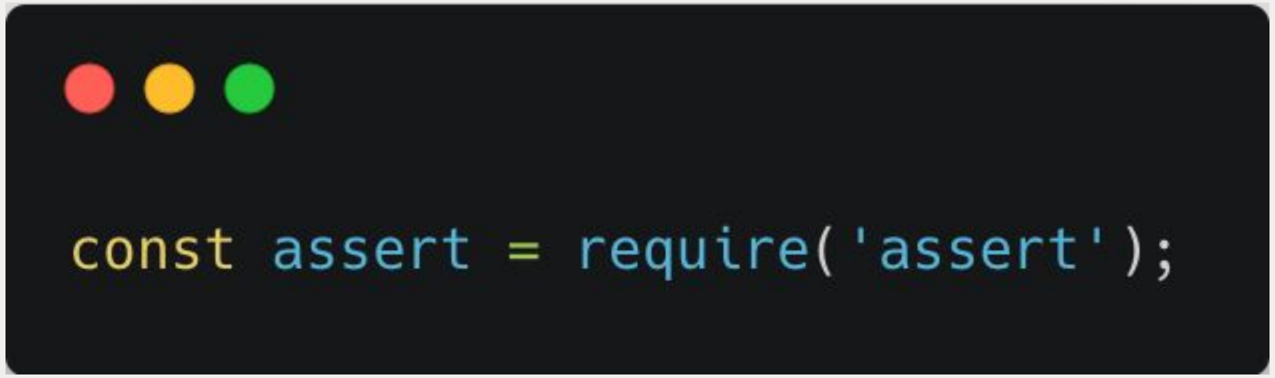
03

Code the test cases with JavaScript and assert

</talentlabs>

Create an Unit Test with Assert

Step 1: Import the “assert” module



```
const assert = require('assert');
```

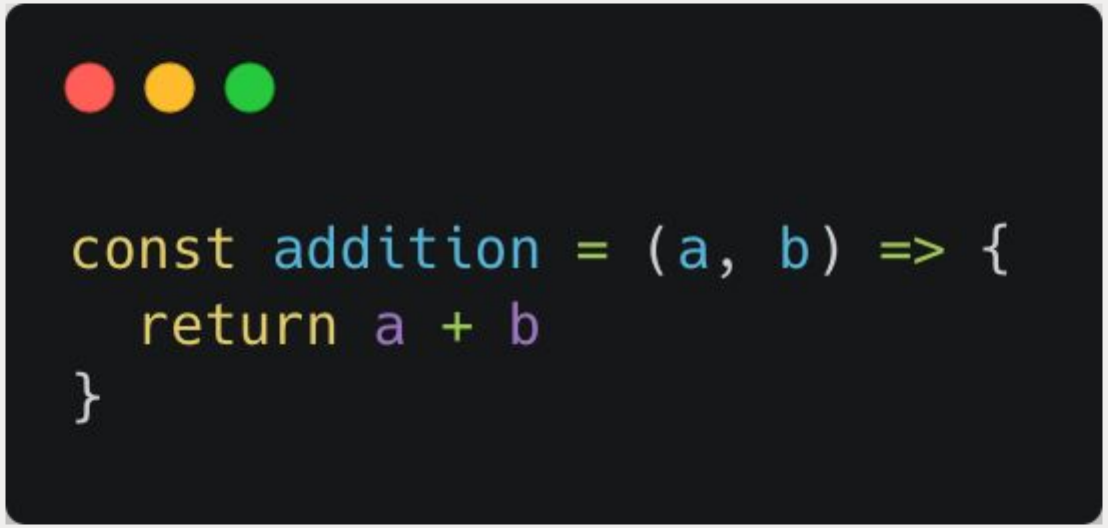
Same as other modules,
we need to import the Node.js module first.

</talentlabs>

Create an Unit Test with Assert

Step 2: Write down the test cases

Let’s say we are going to design test cases for the function “addition”



```
const addition = (a, b) => {
  return a + b
}
```

The function “addition”

The list of test cases we designed
(Each test case should cover a different type scenario)

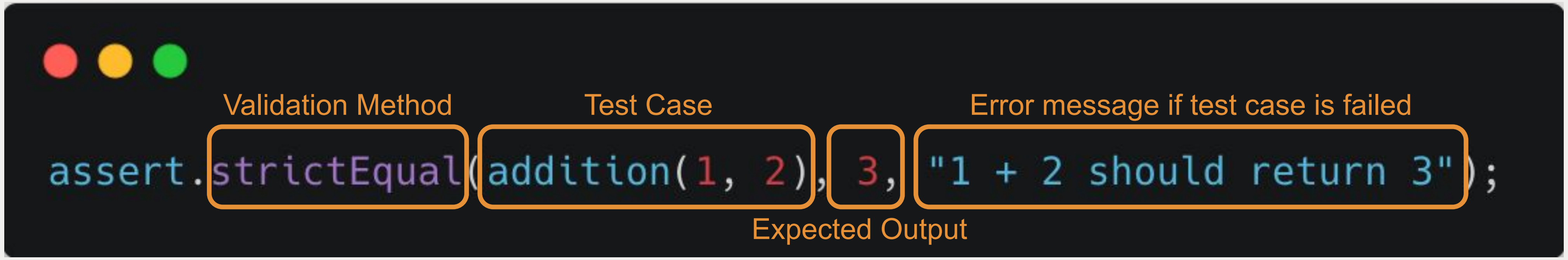
	a	b	Expected Output
+ve and +ve	1	2	3
using 0	0	3	3
-ve and +ve	-5	5	0
-ve and -ve	-1	-2	-3

</talentlabs>

Create an Unit Test with Assert

Step 3: Code the test cases with JavaScript and assert

Test Case Syntax



```
assert.strictEqual(addition(1, 2), 3, "1 + 2 should return 3");
```

</talentlabs>

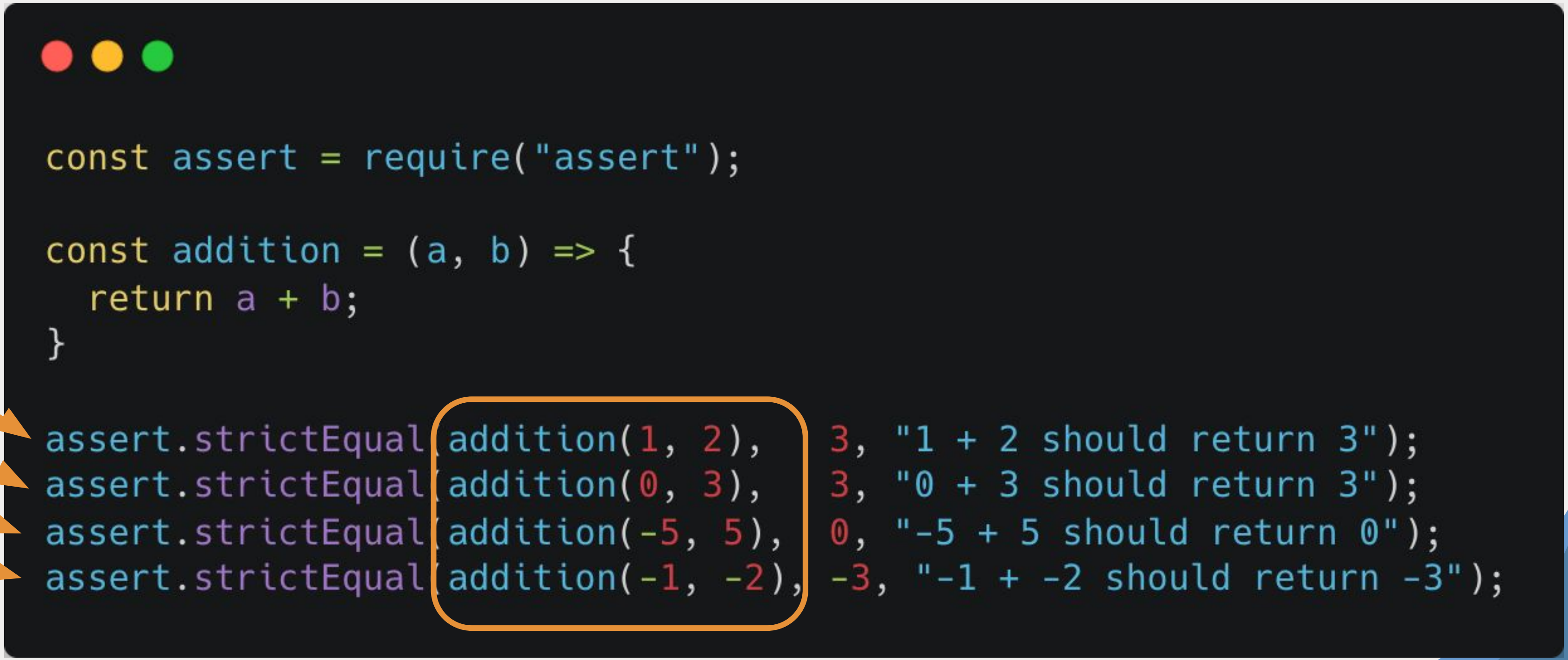
Create an Unit Test with Assert

Step 3: Code the test cases with JavaScript and assert

In this step, we are going to convert the test cases to code

The list of test cases we designed

a	b	Expected Output
1	2	3
0	3	3
-5	5	0
-5	-5	-10



```
const assert = require("assert");

const addition = (a, b) => {
  return a + b;
}

assert.strictEqual(addition(1, 2),   3, "1 + 2 should return 3");
assert.strictEqual(addition(0, 3),   3, "0 + 3 should return 3");
assert.strictEqual(addition(-5, 5),  0, "-5 + 5 should return 0");
assert.strictEqual(addition(-1, -2), -3, "-1 + -2 should return -3");
```

The test cases
Computer will execute this part and collect the return value

</talentlabs>

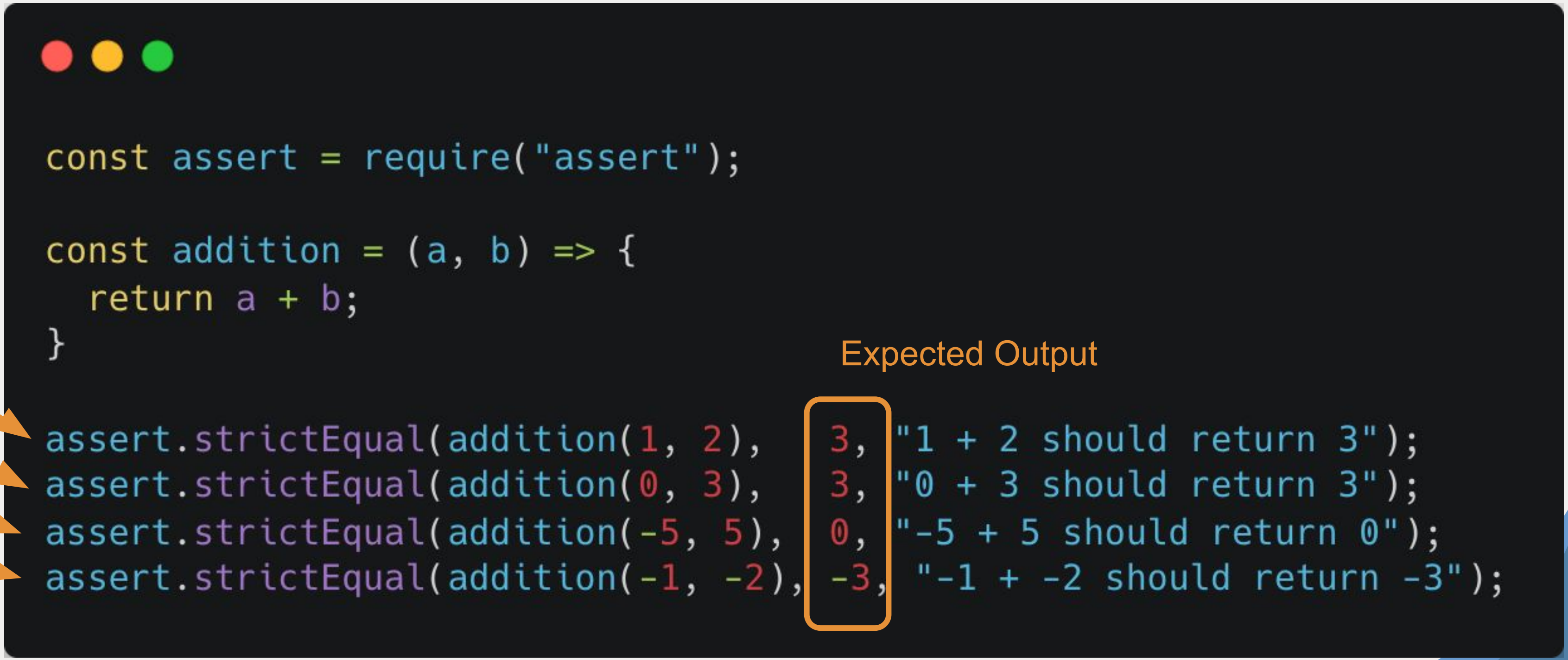
Create an Unit Test with Assert

Step 3: Code the test cases with JavaScript and assert

In this step, we are going to convert the test cases to code

The list of test cases we designed

a	b	Expected Output
1	2	3
0	3	3
-5	5	0
-5	-5	-10



```
const assert = require("assert");

const addition = (a, b) => {
  return a + b;
}

assert.strictEqual(addition(1, 2),   3, "1 + 2 should return 3");
assert.strictEqual(addition(0, 3),   3, "0 + 3 should return 3");
assert.strictEqual(addition(-5, 5),  0, "-5 + 5 should return 0");
assert.strictEqual(addition(-1, -2), -3, "-1 + -2 should return -3");
```

</talentlabs>

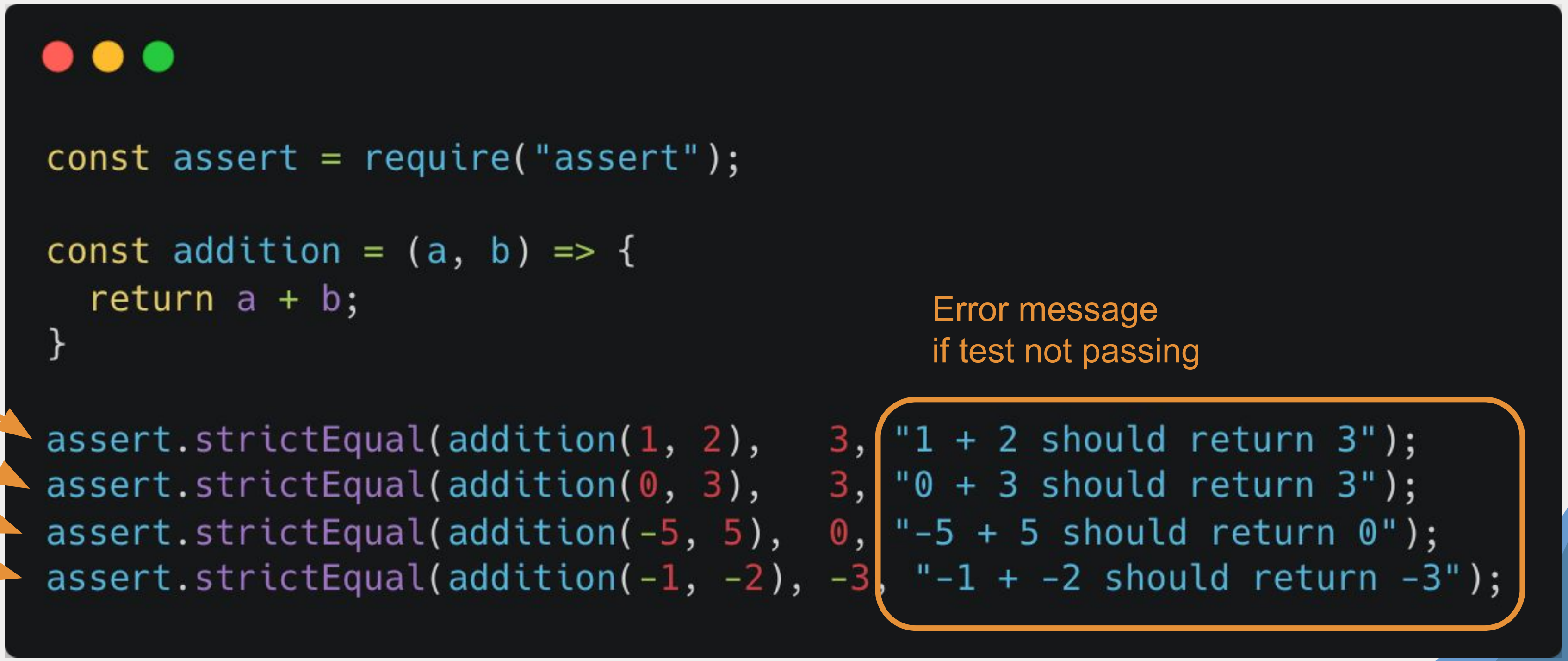
Create an Unit Test with Assert

Step 3: Code the test cases with JavaScript and assert

In this step, we are going to convert the test cases to code

The list of test cases we designed

a	b	Expected Output
1	2	3
0	3	3
-5	5	0
-5	-5	-10



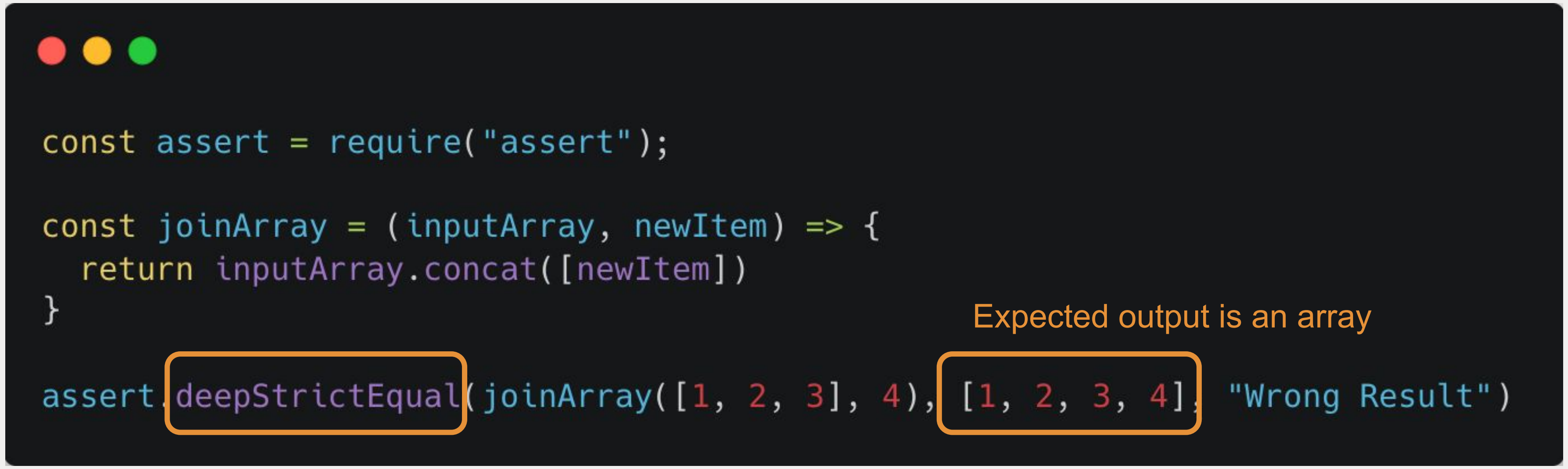
```
const assert = require("assert");

const addition = (a, b) => {
  return a + b;
}

assert.strictEqual(addition(1, 2),    3, "1 + 2 should return 3");
assert.strictEqual(addition(0, 3),    3, "0 + 3 should return 3");
assert.strictEqual(addition(-5, 5),   0, "-5 + 5 should return 0");
assert.strictEqual(addition(-1, -2), -3, "-1 + -2 should return -3");
```

4 Common Validation Methods

Assertion Function	Function
assert.StrictEqual()	Test for equals
assert.notStrictEqual()	Test for not equals (Note: Same value but different type would be passing the test, i.e. assert.notStrictEqual(1, '1') will pass the test)
assert.deepStrictEqual()	Test for arrays and objects results (Note: strictEqual **won’t work** for array and object)
assert.notDeepStrictEqual()	Test for not equal for arrays and objects

</talentlabs>

Asserting Objects and Arrays



```
const assert = require("assert");

const joinArray = (inputArray, newItem) => {
  return inputArray.concat([newItem])
}

assert.deepStrictEqual(joinArray([1, 2, 3], 4), [1, 2, 3, 4], "Wrong Result")
```

Expected output is an array

Make sure you use
“deepStrictEqual” to assert
for array and object results

</talentlabs>