25 July 2022

< Intro to Automated Software Testing />

Why test software in the first place?

Trough testing, we discover defects/bugs before we deliver to the customer! This is KEY if our goal is to deliver:

1. Quality
2. Reliable
3. Easy to use

Software.

Plainly speaking...

no tests

===

💩 SOFTWARE

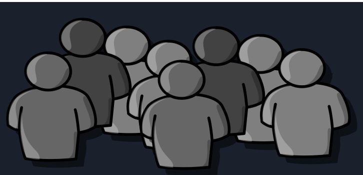
How QA’s used to do it in the old days?

Well… Manually

Often in Excel sheets… which included hundreds & sometimes thousands of test cases.

(Keep in mind each case required manual setup and teardown)

Relatively simple software required TEAMS of manual testers, TEAMS...



Common problems with manual testing?

As you may imagine, manual testing led to:

1. **Slow release process** (QA's manually re-did tests each iteration)
2. **Was prone to mistakes** (QA's are only human)
3. **Fundamentally prevents AGILE & CI CD**
4. **Eventually QA's cannot keep up and get burned out...**

Solution???

Get machines to do it for you!

IN COMES...

Automated Software Testing

Automated testing tools CHANGED THE GAME, forever…

1. QA's no longer solely responsible for testing.
   a. Now, dev's were writing scripts to test their code!

Doors open for TDD!

2. Entire complex scenarios could now be scripted. Including full
   a. Setup &&
   b. Teardown

DB / State / API Mocking

3. Large software projects could be tested & released in minutes, entire test suite ran dozens of times a day.

Doors open for CI&CD and AGILE

If testing can now be automated, what is the role of a QA in the team?

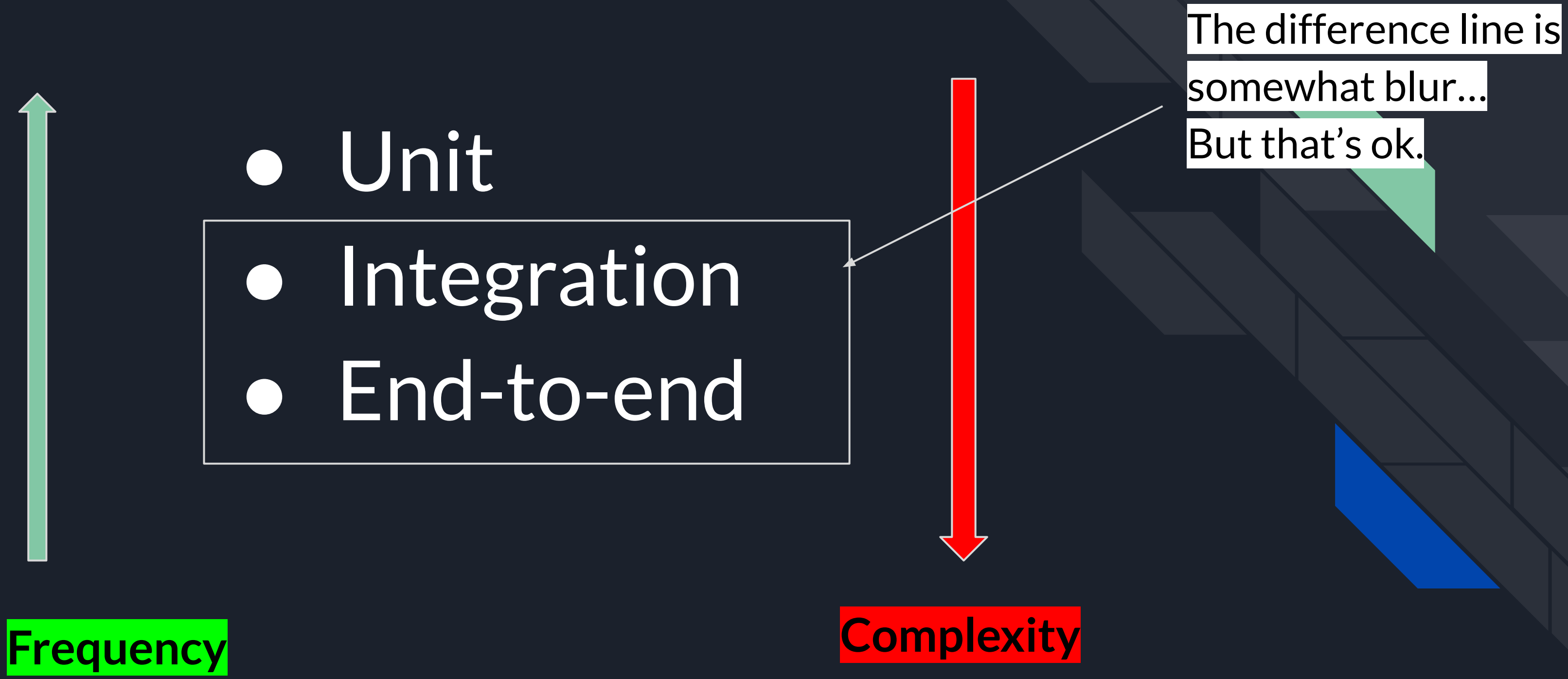
Automated testing software is exceptionally good at running tests.

But, Its at coming up with the actual test cases to run…

✓ QA specialists are exceptionally good at coming up with test cases and have re-skilled to be in-house experts at automated testing tools and best practices.

Automated testing tools evolved, FAST

Different levels / types of tests emerged



- Unit
- Integration
- End-to-end

The difference line is somewhat blur…
But that’s ok.

Frequency

Complexity

Unit Testing

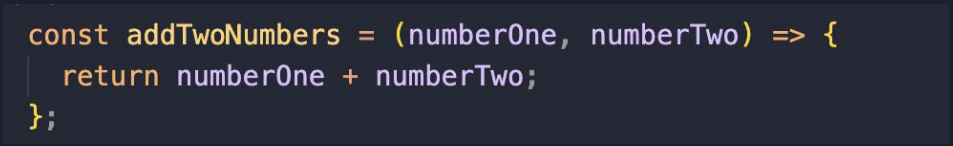
Wiki Definition

Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended.

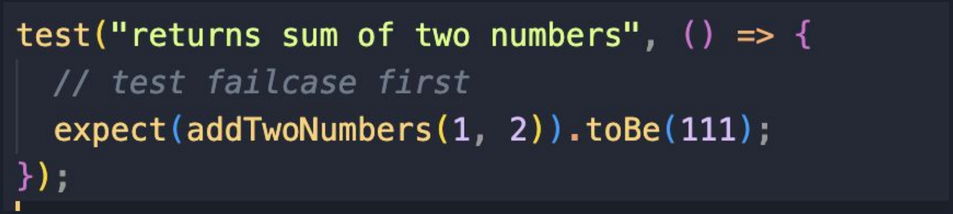
In plain words, a UNIT test is meant to test a small piece of functionality,
9 times out of 10 - *a function*, that does one thing.
To see if the output is what you expect.

Unit tests are written by developers.

Unit test example fail



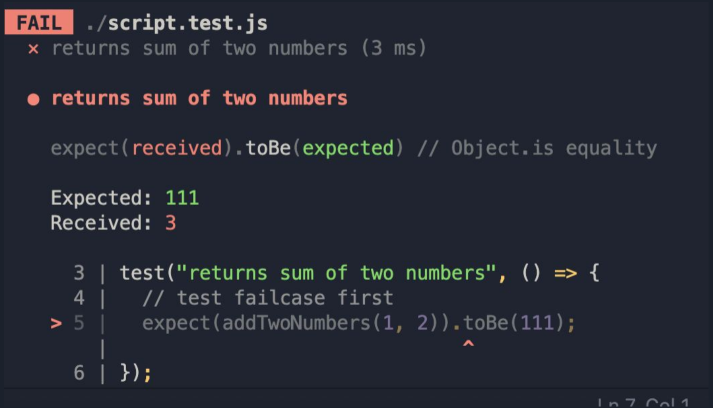
```
const addTwoNumbers = (numberOne, numberTwo) => {
  return numberOne + numberTwo;
};
```



```
test("returns sum of two numbers", () => {
  // test failcase first
  expect(addTwoNumbers(1, 2)).toBe(111);
});
```

Weapon of choice:

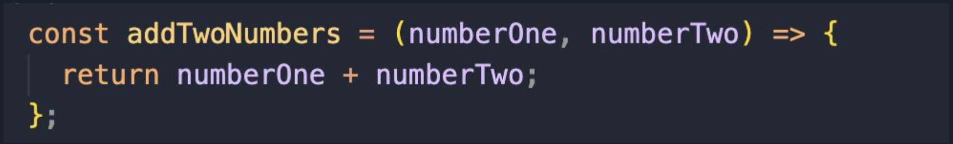
JEST JS



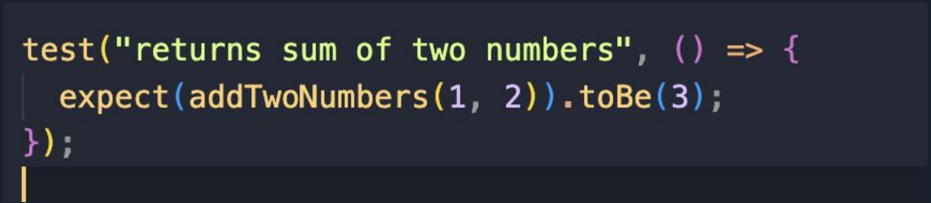
```
FAIL  ./script.test.js
  × returns sum of two numbers (3 ms)

  ● returns sum of two numbers

    expect(received).toBe(expected) // Object.is equality

    Expected: 111
    Received: 3

      3 | test("returns sum of two numbers", () => {
      4 |   // test failcase first
    > 5 |   expect(addTwoNumbers(1, 2)).toBe(111);
        |                               ^
      6 | });
                                                 Ln 7, Col 1
```

Testing failcase first

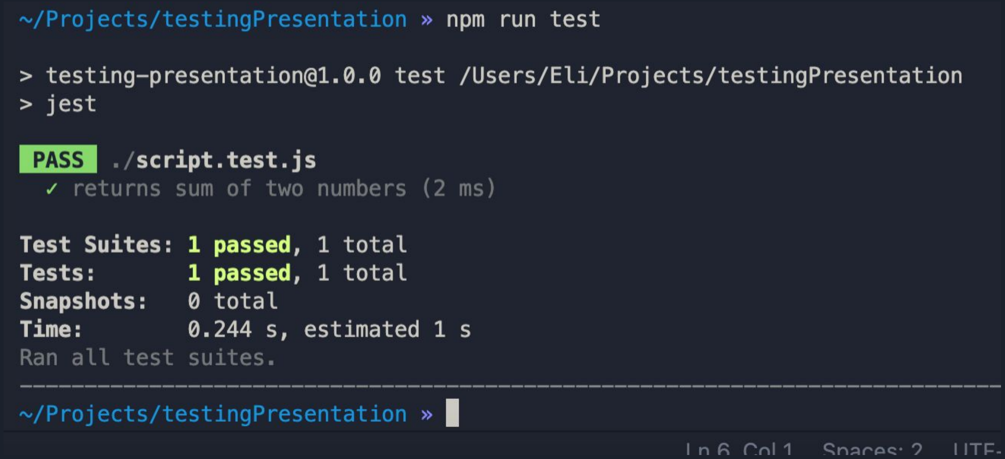
Unit test example pass



```js
const addTwoNumbers = (numberOne, numberTwo) => {
  return numberOne + numberTwo;
};
```



```js
test("returns sum of two numbers", () => {
  expect(addTwoNumbers(1, 2)).toBe(3);
});
```



```
~/Projects/testingPresentation » npm run test

> testing-presentation@1.0.0 test /Users/Eli/Projects/testingPresentation
> jest

 PASS  ./script.test.js
  ✓ returns sum of two numbers (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.244 s, estimated 1 s
Ran all test suites.
--------------------------------------------------------------------------
~/Projects/testingPresentation »
```

Unit test overview

ADVANTAGES

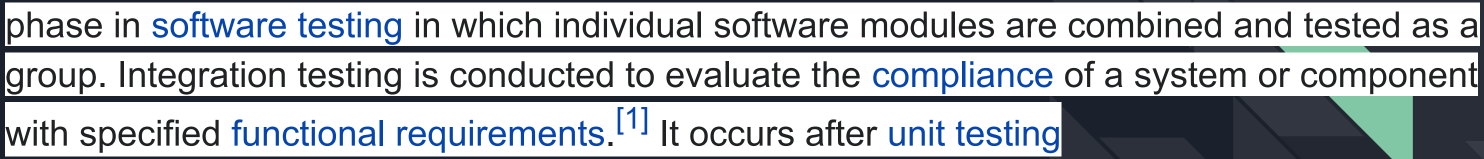
1. Easy to set up
2. Catch bugs early
3. Encourages you to write better code.
    a. Modular, testable…

Ahem… DISADVANTAGES

1. Initial investment requires double the code.
    a. Mostly an illusion, you end up with NET profit later.
2. Unit tests do not catch intricate errors
    a. They do not test how the function runs outside of its own scope or with other functions or modules.

Integration Testing

Wiki Definition



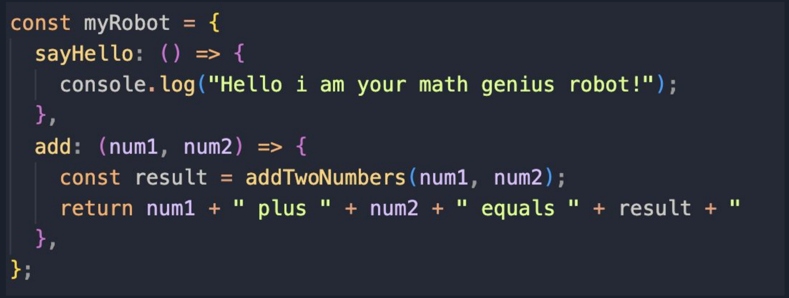
phase in software testing in which individual software modules are combined and tested as a group. Integration testing is conducted to evaluate the compliance of a system or component with specified functional requirements.[1] It occurs after unit testing

Again, to speak in plain words an integration test will usually test an interaction of two or more functions. To see if they cooperate as we would expect.

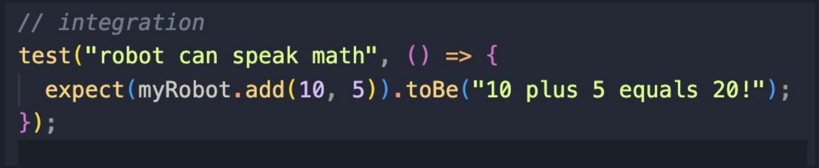
Catches bugs & <u>regressions</u> that are more intricate, this is something unit tests fell short on, since they are ISOLATED.

Integration tests are written by developers

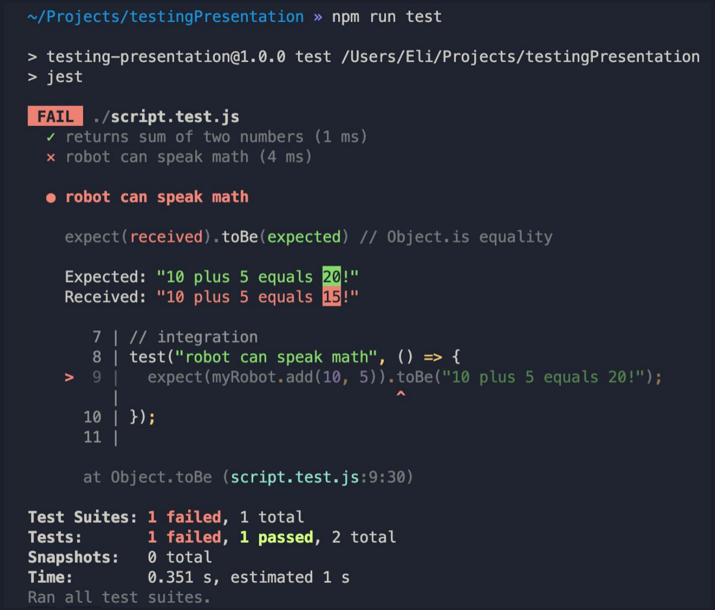
Integration test example fail



```
const myRobot = {
  sayHello: () => {
    console.log("Hello i am your math genius robot!");
  },
  add: (num1, num2) => {
    const result = addTwoNumbers(num1, num2);
    return num1 + " plus " + num2 + " equals " + result + "
  },
};
```



```
// integration
test("robot can speak math", () => {
  expect(myRobot.add(10, 5)).toBe("10 plus 5 equals 20!");
});
```



```
~/Projects/testingPresentation » npm run test

> testing-presentation@1.0.0 test /Users/Eli/Projects/testingPresentation
> jest

 FAIL  ./script.test.js
  ✓ returns sum of two numbers (1 ms)
  × robot can speak math (4 ms)

  ● robot can speak math

    expect(received).toBe(expected) // Object.is equality

    Expected: "10 plus 5 equals 20!"
    Received: "10 plus 5 equals 15!"

       7 | // integration
       8 | test("robot can speak math", () => {
    >  9 |   expect(myRobot.add(10, 5)).toBe("10 plus 5 equals 20!");
         |                              ^
      10 | });
      11 |

      at Object.toBe (script.test.js:9:30)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        0.351 s, estimated 1 s
Ran all test suites.
```

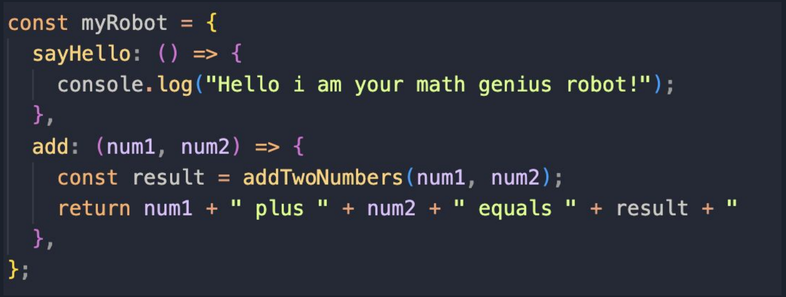
Testing failcase irst

Same tool:

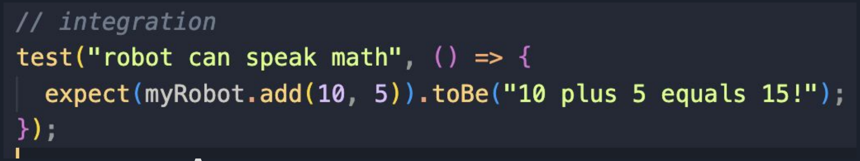
JEST JS

Notice how Jest is telling us that the sum function is all good, but the robot's output is not as expected, instantly identifying where the problem is at. This is key.

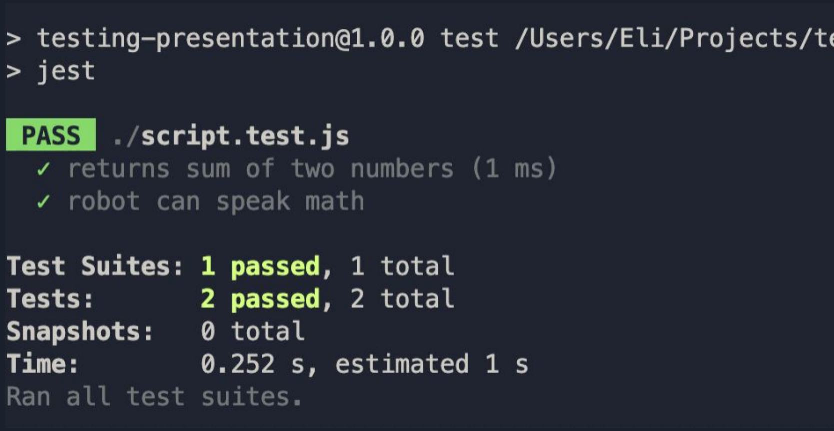
Integration test example pass



```
const myRobot = {
  sayHello: () => {
    console.log("Hello i am your math genius robot!");
  },
  add: (num1, num2) => {
    const result = addTwoNumbers(num1, num2);
    return num1 + " plus " + num2 + " equals " + result + "
  },
};
```



```
// integration
test("robot can speak math", () => {
  expect(myRobot.add(10, 5)).toBe("10 plus 5 equals 15!");
});
```



```
> testing-presentation@1.0.0 test /Users/Eli/Projects/te
> jest

 PASS  ./script.test.js
  ✓ returns sum of two numbers (1 ms)
  ✓ robot can speak math

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.252 s, estimated 1 s
Ran all test suites.
```

Same tool:

JEST JS

All good and green.

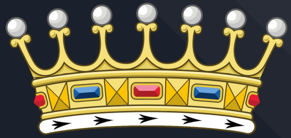
Integration test overview

ADVANTAGES

1. Tests intricacies unit tests could not see.
2. Catches regressions early.
3. Encourages you to write better code.
    a. Modular, testable…
4. Easy and confident code refactoring

CHALLENGES

1. More difficulty in setup & teardown. (State, DB, Mock API etc…)
2. Takes even longer to write than Unit tests.
3. Legacy code may not be written to be testable.



End-to-end Testing

Wiki Definition

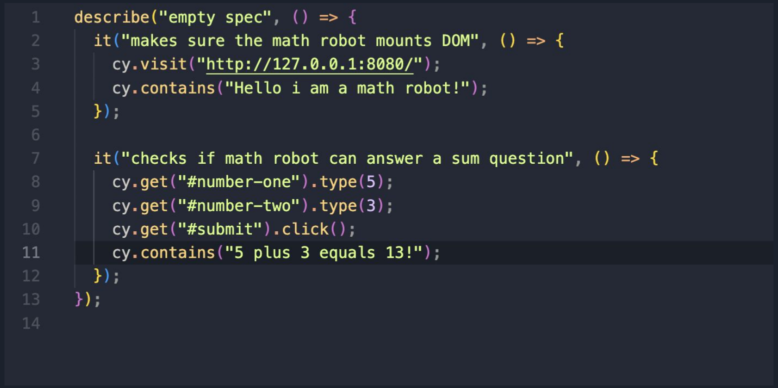
End-to-end testing is **a methodology that assesses the working order of a complex product in a start-to-finish process**. End-to-end testing verifies that all components of a system are able to run and perform optimally under real-world scenarios.

This is the big daddy, the alpha of the pack, the most complex, intricate and reward reaping type of automated software test.

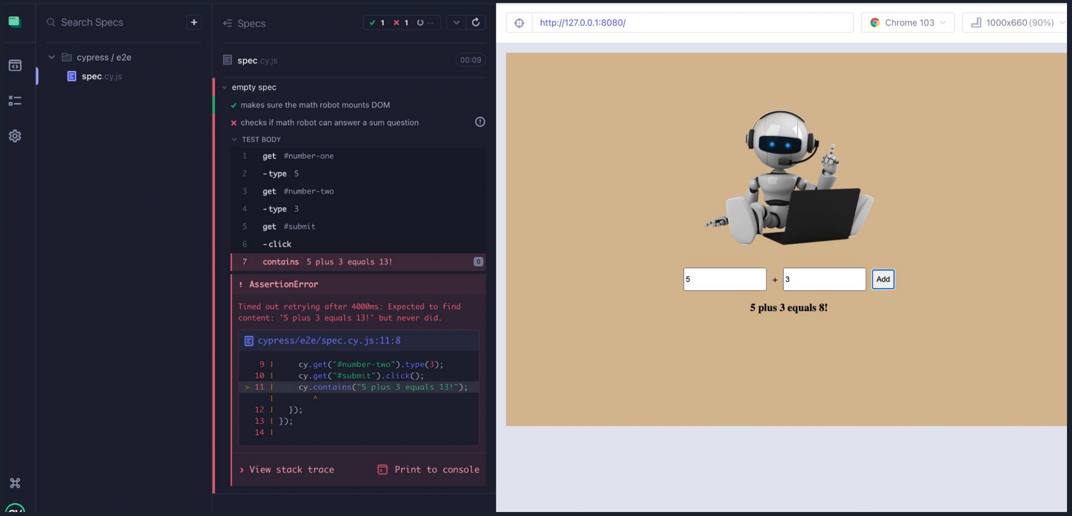
A more sophisticated tool is required to be used for this type of test, for example Cypress or Selenium.

- ***E2E tests are usually written by QA's AND Developers!***

end-to-end test example fail



```
describe("empty spec", () => {
  it("makes sure the math robot mounts DOM", () => {
    cy.visit("http://127.0.0.1:8080/");
    cy.contains("Hello i am a math robot!");
  });

  it("checks if math robot can answer a sum question", () => {
    cy.get("#number-one").type(5);
    cy.get("#number-two").type(3);
    cy.get("#submit").click();
    cy.contains("5 plus 3 equals 13!");
  });
});
```



spec.cy.js 00:09

empty spec

✓ makes sure the math robot mounts DOM

✕ checks if math robot can answer a sum question

TEST BODY

```
1   get #number-one
2   -type 5
3   get #number-two
4   -type 3
5   get #submit
6   -click
7   contains 5 plus 3 equals 13!
```

! AssertionError

```
Timed out retrying after 4000ms: Expected to find
content: '5 plus 3 equals 13!' but never did.
```

cypress/e2e/spec.cy.js:11:8

```
   9 |     cy.get("#number-two").type(3);
  10 |     cy.get("#submit").click();
> 11 |     cy.contains("5 plus 3 equals 13!");
     |        ^
  12 |   });
  13 | });
  14 |
```

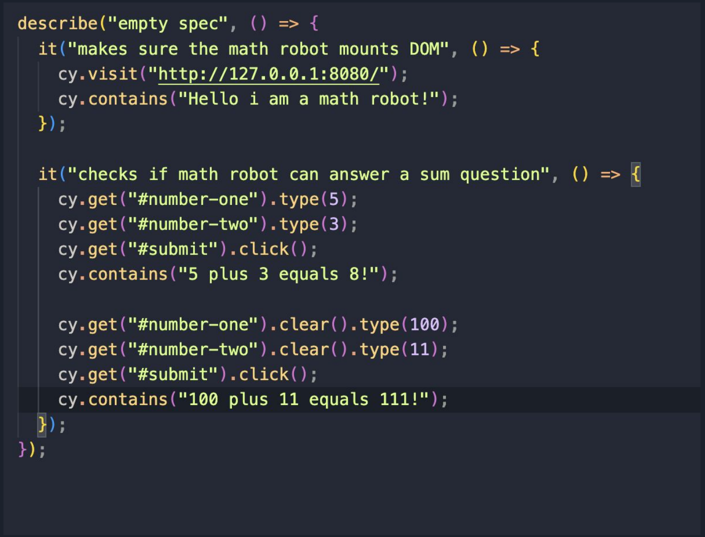
› View stack trace   Print to console

5 + 3 Add

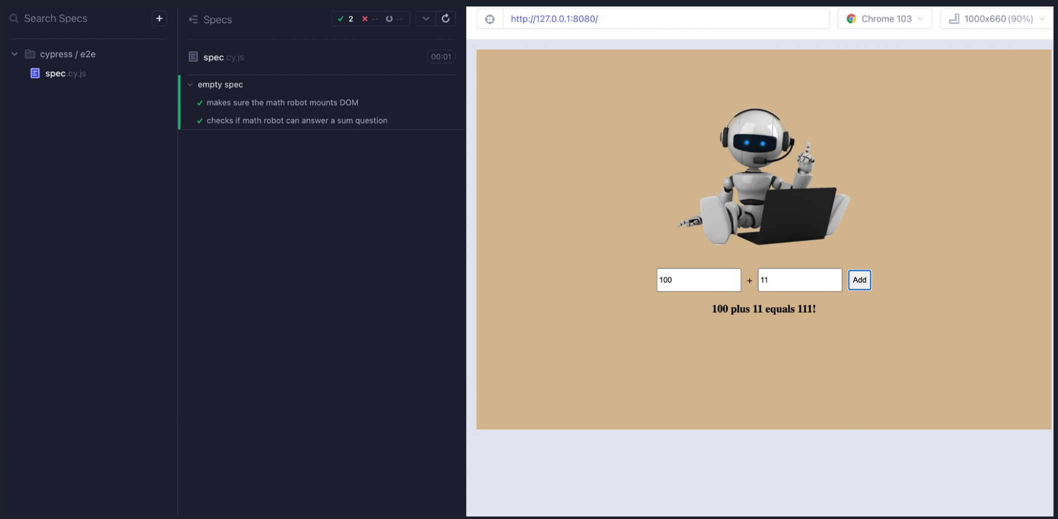
5 plus 3 equals 8!

cypress

end-to-end test example pass



```
describe("empty spec", () => {
  it("makes sure the math robot mounts DOM", () => {
    cy.visit("http://127.0.0.1:8080/");
    cy.contains("Hello i am a math robot!");
  });

  it("checks if math robot can answer a sum question", () => {
    cy.get("#number-one").type(5);
    cy.get("#number-two").type(3);
    cy.get("#submit").click();
    cy.contains("5 plus 3 equals 8!");

    cy.get("#number-one").clear().type(100);
    cy.get("#number-two").clear().type(11);
    cy.get("#submit").click();
    cy.contains("100 plus 11 equals 111!");
  });
});
```



empty spec

✓ makes sure the math robot mounts DOM

✓ checks if math robot can answer a sum question

100 plus 11 equals 111!

All good and green.

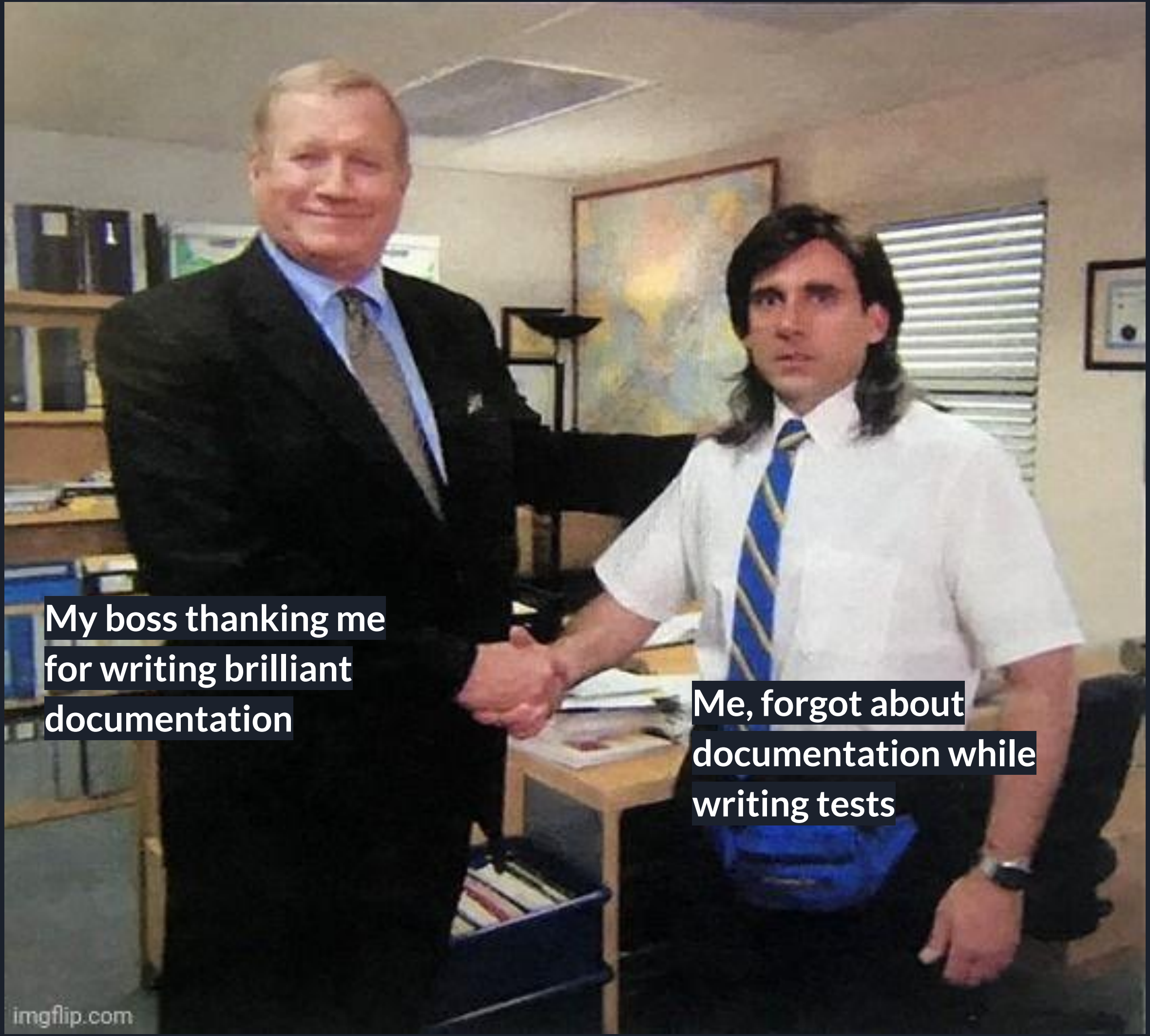
end-to-end test overview

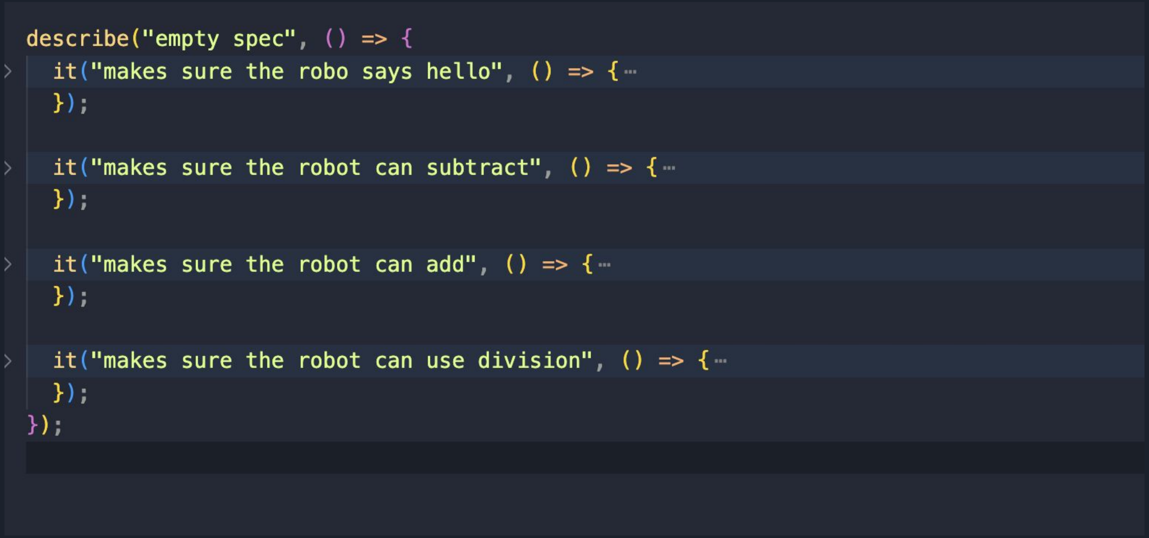
ADVANTAGES

1. Ensures business critical features are tested on all layers. (DB, logic, UI)
2. Massive confidence booster for the team
3. QA's can focus on what matters - identifying quality tests cases - not running manual tests
4. **You get documentation for FREE!!!**

CHALLENGES

1. Difficult to setup & teardown. (State, DB, Mock API etc...)
2. Time consuming at the start.
3. Your app may not be test-ready.
4. Known to be flaky.

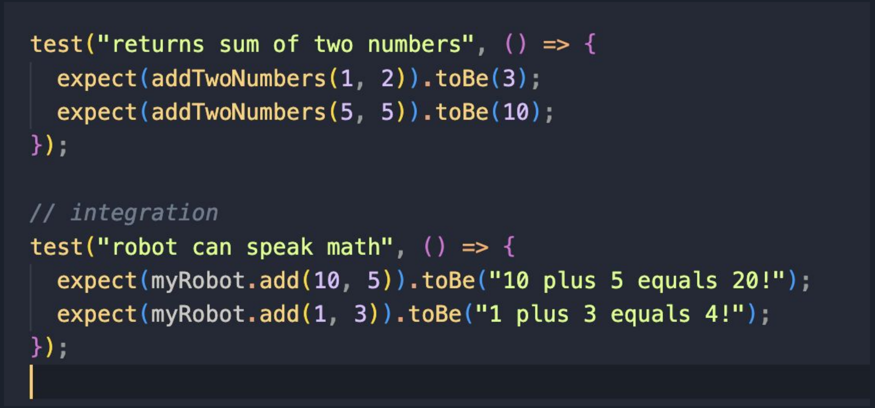

My boss thanking me for writing brilliant documentation
Me, forgot about documentation while writing tests
imgflip.com



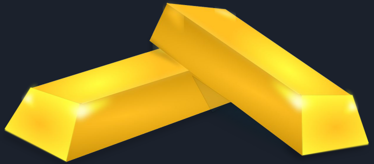
```
describe("empty spec", () => {
  it("makes sure the robo says hello", () => {⋯
  });

  it("makes sure the robot can subtract", () => {⋯
  });

  it("makes sure the robot can add", () => {⋯
  });

  it("makes sure the robot can use division", () => {⋯
  });
});
```

Not much tech knowledge is required to understand what the math robot can do.

Provides up to date usage examples



```
test("returns sum of two numbers", () => {
  expect(addTwoNumbers(1, 2)).toBe(3);
  expect(addTwoNumbers(5, 5)).toBe(10);
});

// integration
test("robot can speak math", () => {
  expect(myRobot.add(10, 5)).toBe("10 plus 5 equals 20!");
  expect(myRobot.add(1, 3)).toBe("1 plus 3 equals 4!");
});
```



The GOLD standard

An easy, fast route to get the first impression of how well an app is tested is running a code coverage report. Most automated testing tools do that very well for us.

Caution: a code coverage report can show 100% but if your test cases may be JUNK.

1. Have a mixture of UNIT, INTEGRATION & END-TO-END tests.
2. Have smart test cases.
3. Preferably a 100% coverage report or at least high 90’s.

Now, you may have noticed a <u>pattern</u> at this point..

Some challenges are repeating throughout ALL the test types.

Tests are time consuming to write & difficult to set up.

a. Usually double the code - physically demanding

b. Cognitively demanding

Codebase may not be ready to be - tested.

a. Ability to spin up environments

b. Engineering gaps

Let’s start by debunking the first point.

Tests are time consuming to write & difficult to set up.

a. Usually double the code - physically demanding

b. Cognitively demanding

Case study, no tests

An example we all may be familiar with- (INSERT HEAVY SARCASM)

1. We must deliver FEAT1 in 2 days = no time to write tests now, do it after.
2. FEAT1 gets delivered in 2 days. We can now go write our tests now, - but wait...
3. PO: FEAT2 must be done in 3 days, write tests for FEAT1 & FEAT2 after.
4. We write FEAT2 in 3 days & deploy.
5. FEAT2 introduces regressions with FEAT1 in prod env. Prod is now broken & FEAT1 must be fixed NOW - app is down.
6. Team spends 3 days refactoring FEAT1 during the weekend. Prod is ok again, we can go write our tests for FEAT1 & FEAT2 on monday. Oh wait
7. Monday comes - FEAT3 needs to happen in 3 days... and on it goes.

Total time spent on FEAT1 & FEAT2: 8 days

No tests OR documentation, morale is low, everyone is annoyed...

Let’s take a look at the same scenario, but this time we write our tests.

1. We must deliver FEAT1 in 2 days.
2. FEAT1 gets delivered in 3 days, we took an extra day to write tests.
3. FEAT2 must be done in 3 days.
4. We write FEAT2 in 4 days since we needed time to write integration and unit tests.
5. We deploy.
6. Team spends a nice weekend relaxing, sleeping well, knowing automated tests have us covered.
7. Monday comes - FEAT3 needs to happen in 3 days.
8. No problem, FEAT1 & FEAT2 have 100% code coverage, adding FEAT3 is a piece of cake.
9. FEAT3 gets delivered in 4 days etc…. You get the point.

Total time spent on FEAT1 & FEAT2: 7 days - LESS than example 1…

Test suites provide brilliant documentation, morale is high, everyone is enjoying their work. Team can go relax, during the weekend.

So again, In case you forgot...

no tests

===

SOFTWARE

In the end, writing tests have you end up with NET profit

Total time spent on FEAT1 & FEAT2: 8 days

No tests OR documentation, morale is low, everyone is annoyed...

VS

Total time spent on FEAT1 & FEAT2: 7 days - LESS than example 1…

Test suites provide brilliant documentation, morale is high, everyone is enjoying their work. Team can go relax, during the weekend.

Now the second, a very common, bigger problem…

Codebase may not be ready to be - tested.

a. Ability to spin up environments

b. Engineering gaps

The OATH - Thou shalt not write code that is not testable and without tests...

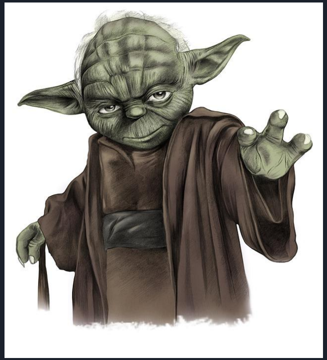
VS

Average software engineer

1. Procedural coding style
2. Lack of vision / bigger picture
3. Not modular
4. Hard to read / comprehend
5. Code is hard or even impossible to test.
6. No tests.

Senior software engineer

1. Conventional coding style FP / OOP
2. Has vision of structure and infrastructure
3. Writes modular, reusable code
4. Easy to read & comprehend
5. Testable from the start
6. Covered with tests.



Stakeholders don’t really care about tests, and why should they…?

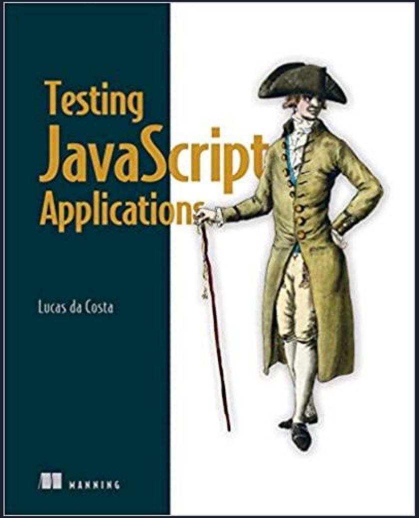
Don’t ask,
dont tell…

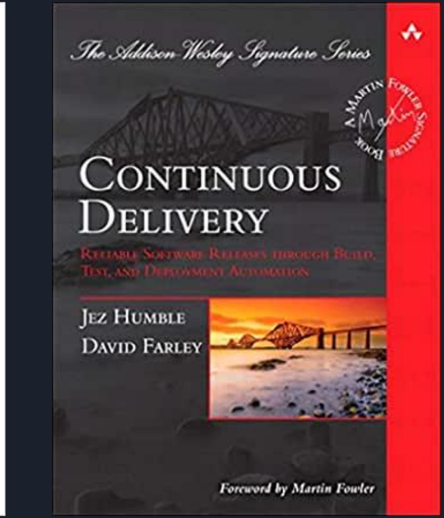
It’s all on us,
always has been…

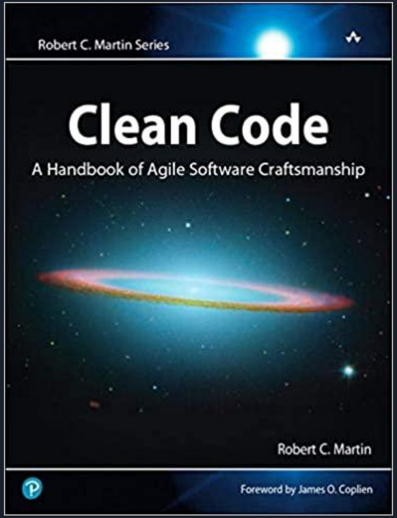
We must ALL commit to write testable code & tests. Whatever it takes.

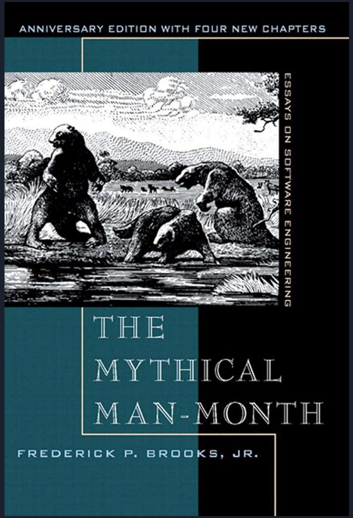
1. Do the research.
2. Learn.
3. Come up with a testing strategy.
4. Implement.
5. Do not tolerate average code - *(you took the oath)*
6. Help others do the same.

Knowledge base / learning resources




The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Continuous
Delivery
Reliable Software Releases through Build, Test, and Deployment Automation
Jez Humble
David Farley
Foreword by Martin Fowler


Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Robert C. Martin
Foreword by James O. Coplien


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE
MYTHICAL
MAN-MONTH
FREDERICK P. BROOKS, JR.

Git'it

https://github.com/ELiHimself/intro-to-automated-software-testing.git

Thank you