# Testing, Chapter 1

</>

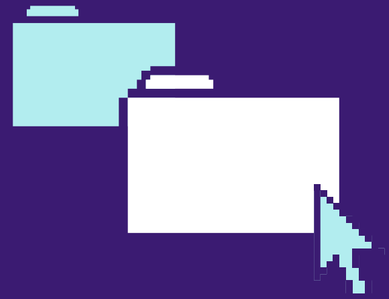
Written by
**Denis Tokarev**

Canva

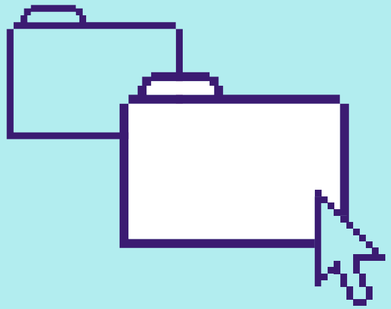
# I'VE HEARD ABOUT TESTING, BUT...

- What exactly is testing?
- What types of tests exist?
- Why bother writing automated tests?

BASICALLY...
TESTING IS A PROCESS OF
VALIDATING THE SOFTWARE TO
ENSURE THAT **IT'S NOT BROKEN.**



# BUT WHY DOES SOFTWARE BREAK?



## SOFTWARE TENDS TO GET MORE AND MORE COMPLEX

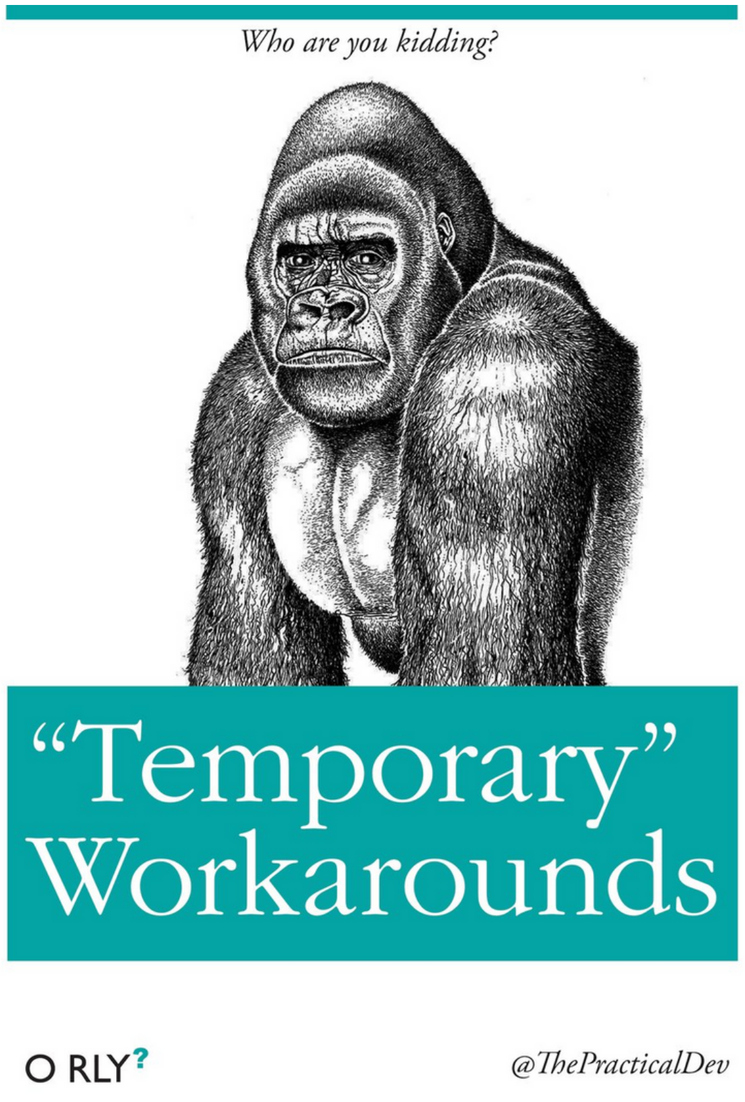
Software helps solve problems, and that's why we people keep building and using it.

Real-world problems may be difficult and operate vague entities. Software too.

## ANY LARGE ENOUGH SYSTEM HAS HACKS

Sometimes, hacks are just okay. A nice hack can dramatically improve performance, or save some RAM, or help you ship a complex feature earlier and make your customers happy.

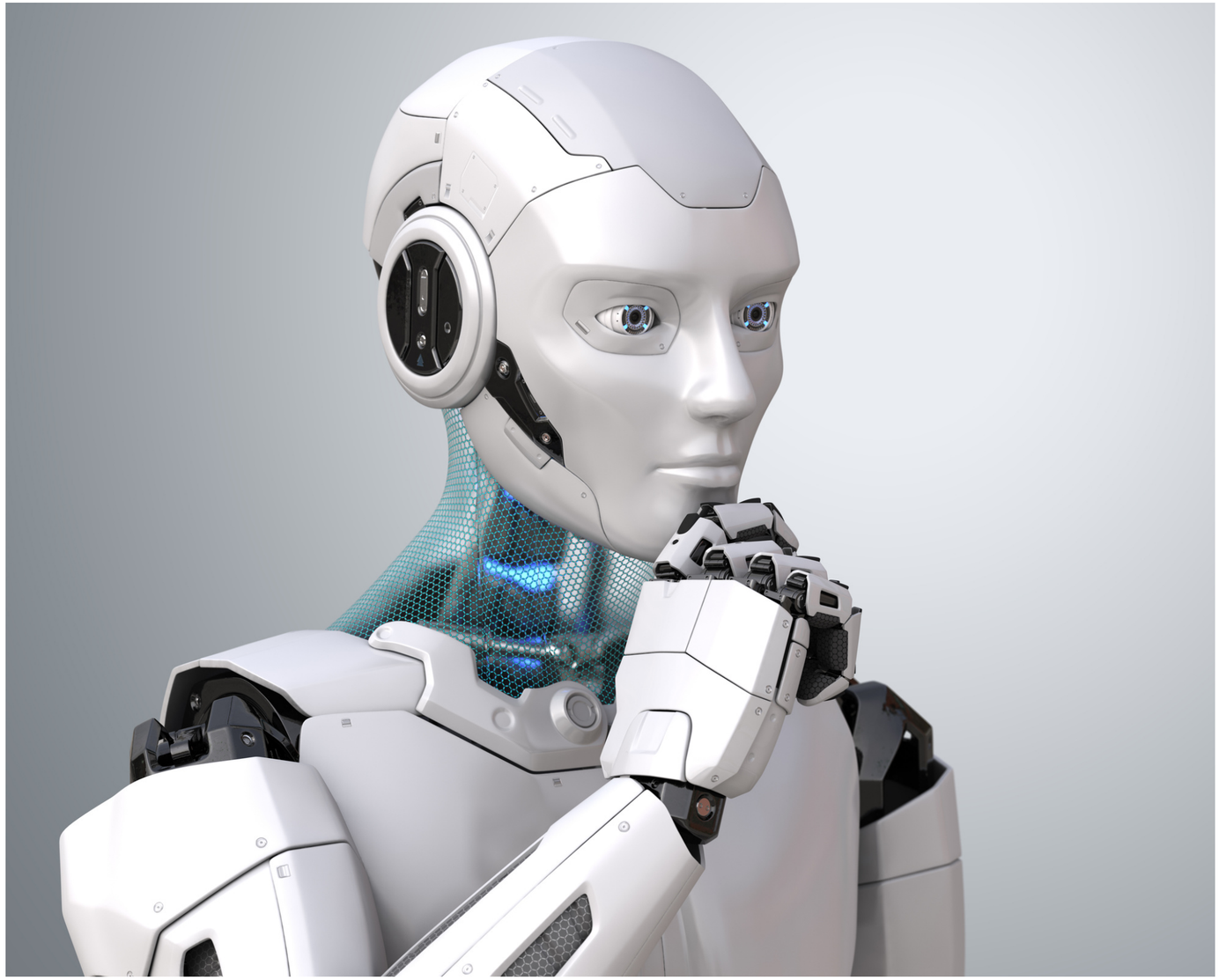
Yet, hacks often make use of something in a totally unexpected and unsafe way.



Did you know you can use your seat belt to open beers while driving?

# COMPUTERS DON'T UNDERSTAND US AND THE WORLD AROUND

Computers don't solve people's problems just by existing. We tell computers what to do with all these weird instructions written in special languages. Sometimes, we can't translate our thoughts into these languages correctly.



# PEOPLE MAKE MISTAKES

People are smart, but there's always a chance of mistake.

Didn't sleep well?
Your code will crash.

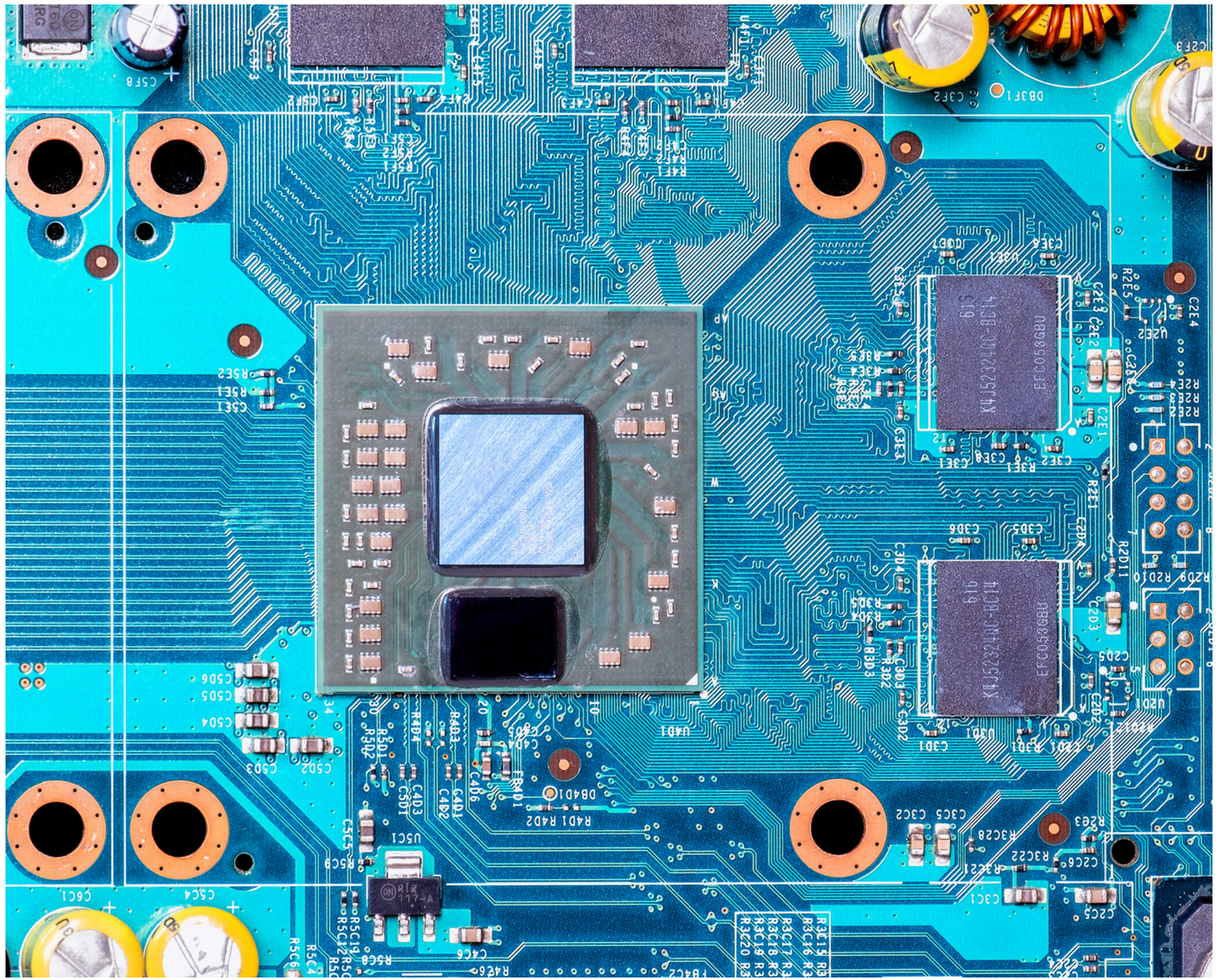
Slept well?
Your code still will crash.

Maybe, you simply haven't considered another use case.

# *YOUR* CODE IS NOT THE ONLY THING THAT HAS ERRORS

The lines of code that you write don't get executed on the CPU directly.

There's also a framework, a compiler, runtime libraries, an operating system, HAL, and hardware that consists of billions of tiny transistors.



## SOMETIMES, ERRORS CAN COST TOO MUCH

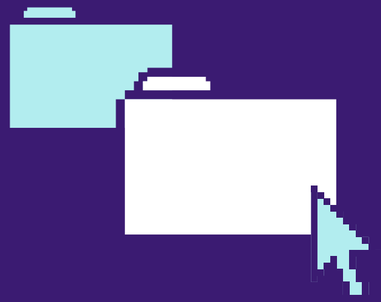
Some software bugs resulted in the loss of **billions** dollars and even **human deaths**.

Not handling a null pointer properly is one of billion dollar mistakes.

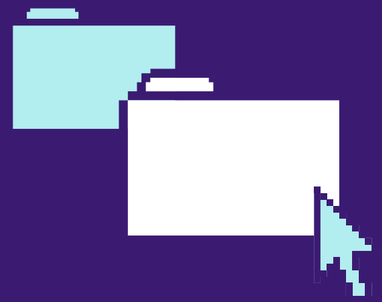
HENCE, WE HAVE TO MAKE SURE THAT OUR CODE DOES EXACTLY WHAT WE EXPECT IT TO DO, AND THE ENTIRE PRODUCT BEHAVES IN THE WAY WE WANT IT TO BEHAVE.
THIS IS CALLED TESTING.

TESTING ISN'T A SPONTANEOUS AND RANDOM PROCESS.

IT IS A SYSTEMATIC AND DELIBERATE ACTIVITY THAT AIMS TO DETECT AND DOCUMENT **BUGS** AND **REGRESSIONS** TIMELY BEFORE THEY GET RELEASED TO USERS.



DEVELOPING RELIABLE HIGH-QUALITY SOFTWARE IS **NOT** ABOUT ALWAYS WRITING **FLAWLESS** CODE, IT'S ABOUT DETECTING THE POSSIBLE ISSUES **IN TIME** AND MAKING SURE THEY'RE FIXED BEFORE THEY WOULD EVER AFFECT CUSTOMERS.



RELIABLE SOFTWARE =

WRITING CODE

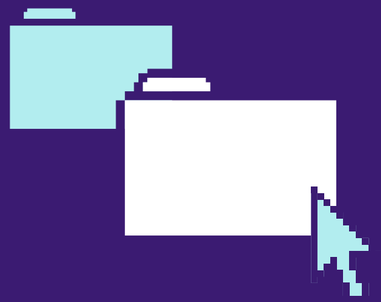
+

PROPERLY TESTING THE FEATURES

+

FIXING THE ISSUES TIMELY

FROM THE TESTING PERSPECTIVE, USE CASES AND USAGE SCENARIOS IMPLEMENTED IN THE SOFTWARE PRODUCT ARE CALLED **TEST CASES, OR TEST SCENARIOS.**

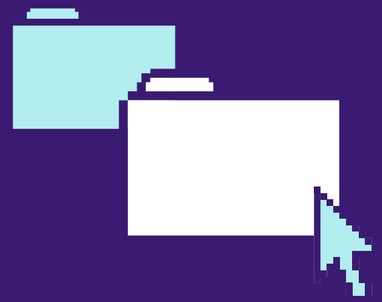


A **TEST CASE** IS AN ACTION THAT CHECKS FOR A SPECIFIC OUTCOME OF THIS ACTION.

*'WHEN I'VE ENTERED MY USERNAME AND PASSWORD, THE SUBMIT BUTTON GETS ENABLED'*

A **TEST SCENARIO** IS A SET OF FUNCTIONALITY OR FEATURES TO VERIFY.

*'LOGGING IN WITH A USERNAME AND A PASSWORD WORKS'*



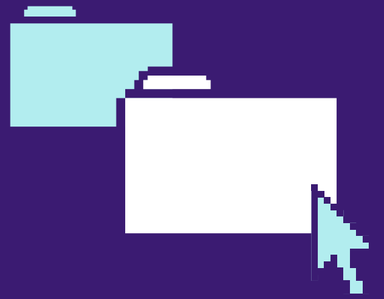
IF WE CHECK A TEST CASE OR TEST SCENARIO AND THE SOFTWARE WORKS AS WE'VE EXPECTED, WE SAY THAT IT '*PASSES*' THE TEST CASE OR TEST SCENARIO.

OTHERWISE, WE SAY THAT IT '*FAILS*' IT.

TOGETHER, TEST CASES AND TEST SCENARIOS DEFINE A **FUNCTIONAL SPECIFICATION** OF A SOFTWARE PRODUCT.

THUS, WHILE BEING A FEEDBACK INSTRUMENT, TESTING ISN'T LIMITED TO JUST A BUG DISCOVERY TOOL.

IT ALSO HELPS UNDERSTAND AND FORMALIZE THE IMPLEMENTED PRODUCT'S BEHAVIOUR AND BRINGS THE IMPLEMENTATION CLOSER TO THE EXPECTATIONS.



# HOW DO WE TEST?

# MANUAL TESTING

It's the most obvious way of testing.

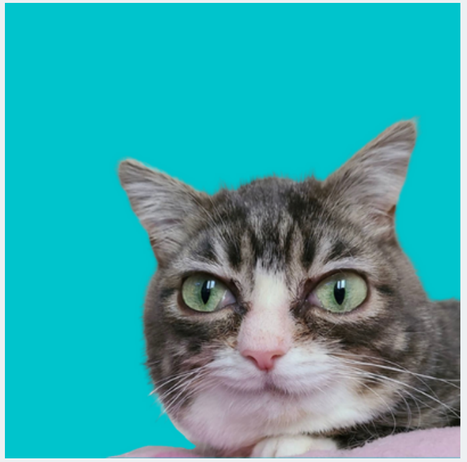
You get the software running somewhere*, and then you start checking each test scenario by hand, ticking off the ones that pass and reporting the ones that fail.

https://staging.yourcoolproject.com/

## VERY IMPORTANT SOFTWARE

Click the button below to see a funny cat.

Show!

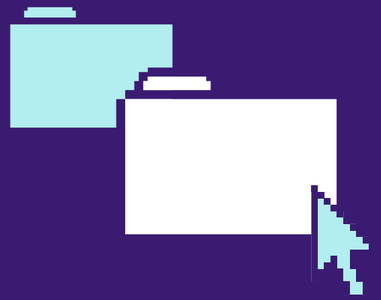


Manual Tests

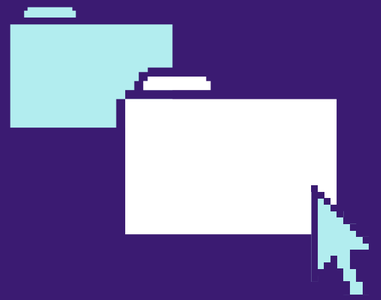
File Edit View Insert Format Data Tools Add-ons Help Last edit was

| Scenario / Case | Pass? |
|---|---|
| Page opens | ✅ |
| Header is there | ✅ |
| Description is there | ✅ |
| Button is there | ✅ |
| Clicking a button shows a cat photo | ✅ |
| Cat is funny | ✅ |
| "Hi" text arrears next to the cat after 5 seconds | ❌ |

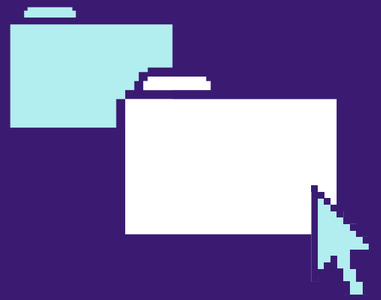
FOR MANUAL TESTING, BETTER DEPLOY THE **RELEASE CANDIDATE** (A SOFTWARE VERSION THAT IS PLANNED TO BE RELEASED) ONTO AN ENVIRONMENT CLOSE TO THE PRODUCTION. SUCH ENVIRONMENTS ARE USUALLY CALLED 'STAGING.'



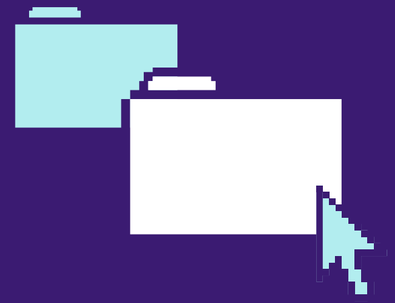
MANUAL TESTING IS CONVENIENT FOR CHECKING THAT A FEATURE YOU'VE BUILT *KINDA WORKS*.



WHEN A SOFTWARE PRODUCT GROWS AND BECOMES MORE COMPLEX, THE NUMBER OF TEST CASES BECOMES TOO BIG, AND QA PROCESS BECOMES A BLOCKER FOR NEW RELEASES.

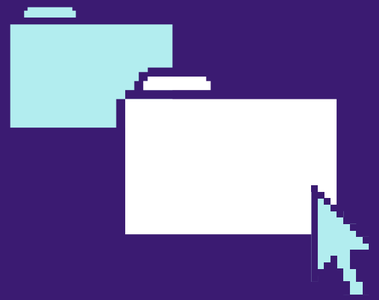


WHEN YOU HAVE TO SUPPORT MULTIPLE DEVICES OR PLATFORMS, THE NUMBER OF TEST CASES GETS MULTIPLIED BY NUMBER OF THE DEVICE GROUPS OR PLATFORMS. IT ALSO BECOMES A BLOCKER FOR NEW RELEASES, OR LEADS TO REGULARLY ROLLING OUT AN UNDER-TESTED PRODUCT THAT BREAKS ON CUSTOMER DEVICES.



THE MAIN ISSUE WITH MANUAL TESTING IS THAT IT DOESN'T SCALE.

MANUAL WORK USUALLY DOESN'T SCALE WELL AND NEEDS AUTOMATION.



LUCK THAT SOFTWARE ENGINEERS ARE GOOD AT AUTOMATION.

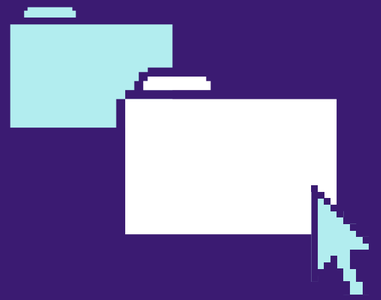
# UNIT TESTING

Automated tests that are written for every unit separately.

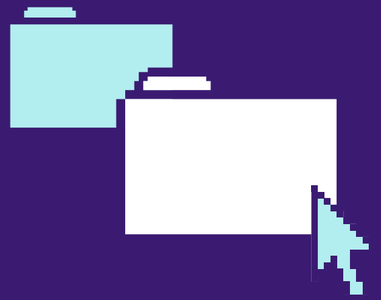
Unit tests **don't** test the big picture and how all the pieces work together. Instead, in unit tests, we focus on making sure that every piece perfectly works in separation.

```
yourcoolprooject - user@dev - -zsh - 100x40
user@dev ~/projects/yourcoolproject (master) $ npm run test

PASS  src/components/components.test.js (0.492s)
  Your Cool project
    Components
      <Header />
        ✓ Has customizable text
      <Paragraph />
        ✓ Has customizable text
      <Button />
        ✓ Has customizable caption
        ✓ Clicking triggers onClick
      <ImageBox />
        ✓ Renders empty area when no image URL is provided
        ✓ Renders image by URL when it can be loaded
        ✓ Falls back to empty area when image can't be loaded

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        1.227s
Ran all test suites.
✨  Done in 1.35s.
user@dev ~/projects/yourcoolproject (master) $
```

UNIT TESTING HELPS SPLIT THE CODE INTO INDEPENDENT DECOUPLED PIECES AND MAKE SURE THEY *JUST WORK*™.



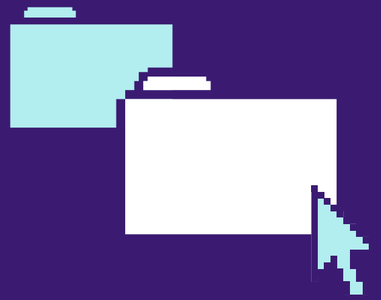
IT HELPS BE CONFIDENT THAT EVERY CLASS, MODULE OR COMPONENT IS STABLE ENOUGH AND HAS A PREDICTABLE AND WELL-DEFINED BEHAVIOUR.



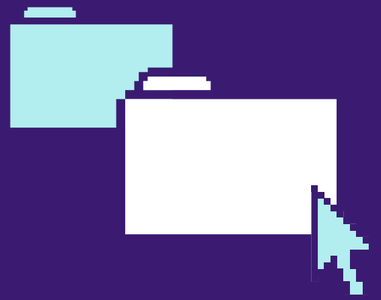
UNIT TESTS ARE BASICALLY A PROGRAMMED SPECIFICATION OF A COMPONENT'S BEHAVIOUR, AND THEY WILL FAIL IF COMPONENT BEHAVIOUR ACCIDENTALLY CHANGES.

IT GETS EXTREMELY CONVENIENT WHEN WE WANT TO REFACTOR A MODULE AND MAKE SURE ITS PUBLIC API AND REMAINS UNCHANGED.

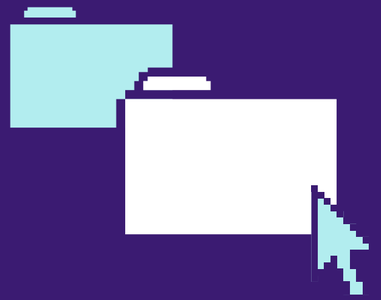
WHEN SOFTWARE MODULES AND COMPONENTS ARE PROPERLY UNIT TESTED, IT REALLY FEELS EASIER AND MORE CONFIDENT TO BUILD SOFTWARE OUT OF THESE PIECES, KNOWING THAT THEY (MOST LIKELY) WON'T ACT UNEXPECTEDLY.



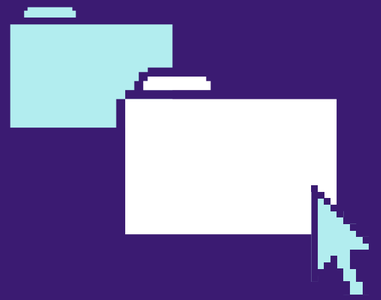
UNIT TESTING **DOES** SCALE VERY WELL.



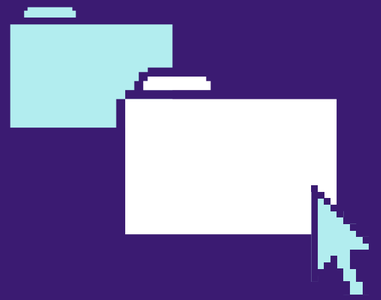
UNIT TESTS RUN VERY FAST, DON'T HAVE SPECIAL REQUIREMENTS FOR THE ENVIRONMENT, CAN BE LAUNCHED FROM EVERYWHERE, INCLUDING A DEVELOPER'S LAPTOP.



UNIT TESTS EVEN CAN BE AUTO-RUN FOR EVERY COMMIT THAT GETS PUSHED IN THE REPO, AND CAN BLOCK PULL REQUEST FROM GETTING MERGE IF THERE IS ANY REGRESSION.



UNIT TESTS HELP DISCOVER REGRESSIONS VERY QUICKLY AND ENCOURAGE FIXING UP REGRESSIONS IN A CONCISE TIME.

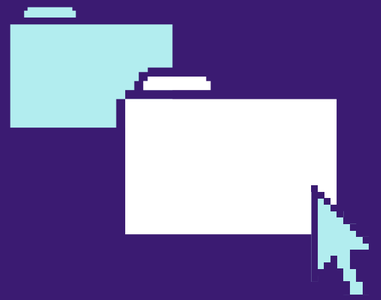


# PLEASE WRITE UNIT TESTS

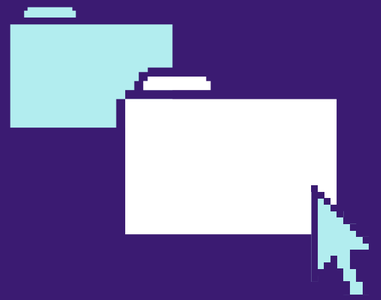
It is a skill a professional software engineer **must** master.

It might feel difficult or unnecessary, and it does have a learning curve, but in long term, it's such a **timesaver**.

UNIT-TESTING A MODULE THAT HAS TOO MANY IMPLICIT DEPENDENCIES AND CONTAINS TOO MUCH LOGIC CAN BE DIFFICULT. THAT'S WHY UNIT TESTING ENCOURAGES **TIMELY REFACTORING.**



IT HELPS SPLIT COMPLEX LOGIC INTO SMALLER PIECES, DECOUPLE THE CODE AND REDUCE IMPLICIT DEPENDENCIES. IT MAKES THE **ARCHITECTURE OF AN ENTIRE SOFTWARE** PRODUCT BETTER.



WE'LL SPEAK A BIT MORE ABOUT HOW TO WRITE UNIT TESTS, THE TDD, AND BEST PRACTICES OF UNIT TESTING IN THE DEMO SECTION OF THIS PRESENTATION.


UNIT TESTING!
UNIT TESTING!
UNIT TESTING!

# INTEGRATION TESTING

Unit testing helps check every component or module separately; however, it doesn't guarantee the correctness of the application.

Even though our 'legos' work well separately, the code that binds them together can still be wrong.

```
yourcoolprooject - user@dev - -zsh - 100x40
user@dev ~/yourcoolproject (master) $ npm run test:integration

 PASS  src/main.test.js (6.522s)
  Main Page
    ✓ Renders header, description, and gray area
    ✓ Shows the image after clicking the button
    ✓ Shows "Hi" text next to image after 5 seconds
    ✓ Doesn't show the image when can't load it

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        7.371s
Ran all test suites.
✨  Done in 8.655s.
user@dev ~/projects/yourcoolproject (master) $
```

# INTEGRATION TESTING

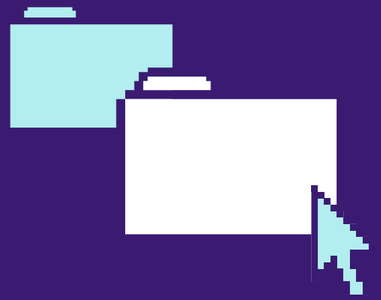
Manual testing solves that sort of issues, but as we proved earlier, it doesn't scale well.

Integration testing implies running a web browser or a browser-like runtime and programmatically triggering functionality.

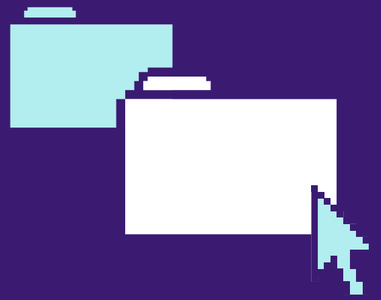
```
yourcoolprooject - user@dev - -zsh - 100x40
user@dev ~/yourcoolproject (master) $ npm run test:integration

PASS  src/main.test.js (6.522s)
  Main Page
    ✓ Renders header, description, and gray area
    ✓ Shows the image after clicking the button
    ✓ Shows "Hi" text next to image after 5 seconds
    ✓ Doesn't show the image when can't load it

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        7.371s
Ran all test suites.
✨  Done in 8.655s.
user@dev ~/projects/yourcoolproject (master) $
```

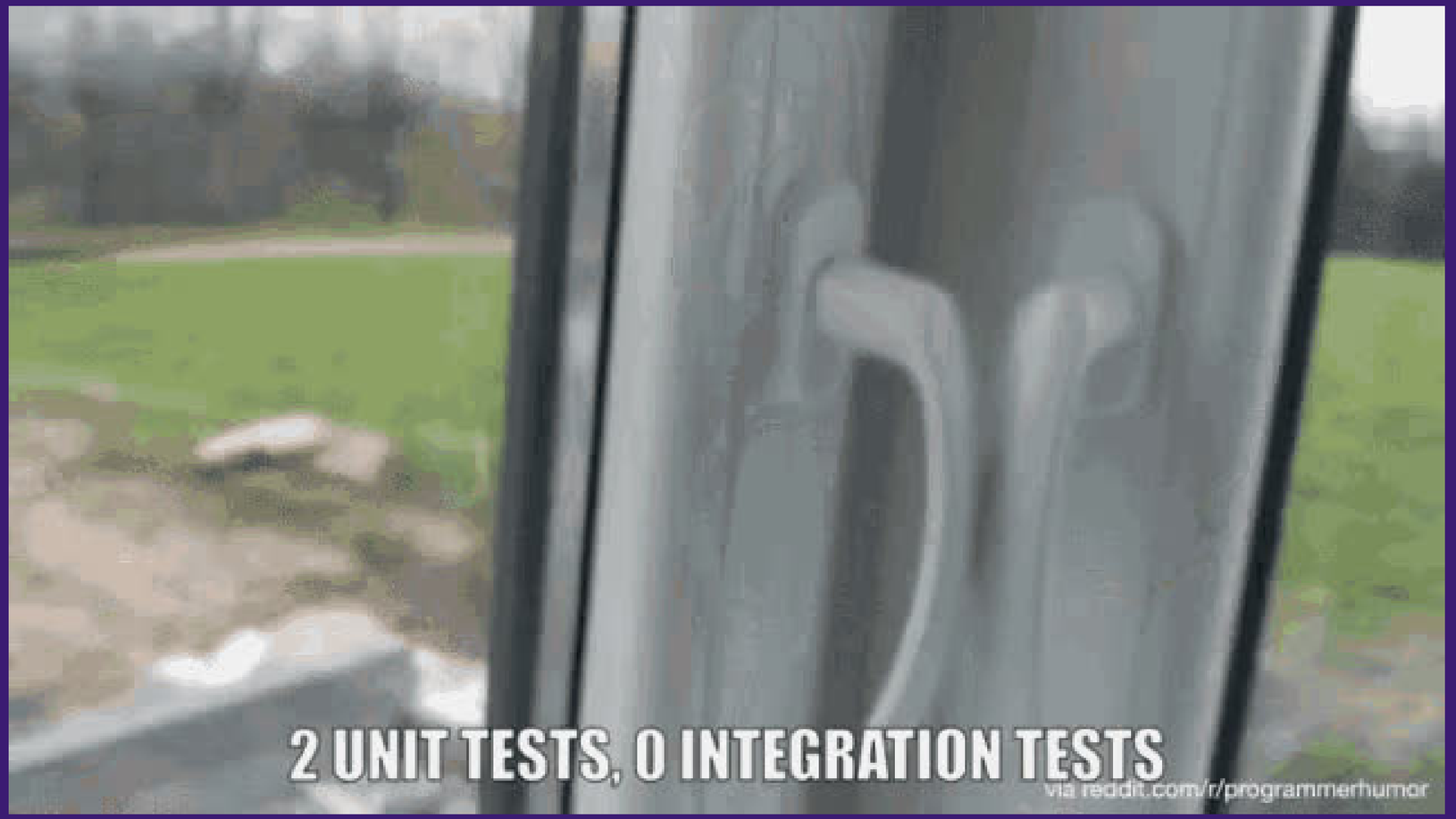
INTEGRATION TESTING IS USUALLY MOVE 'HEAVY' AND TIME-CONSUMING THAN UNIT TESTING.

THEY'RE SLOWER TO RUN, THEY'RE A BIT MORE DIFFICULT TO WRITE, AND SOMETIMES, THEY MAY BE A BIT MORE *'FLAKY'*.



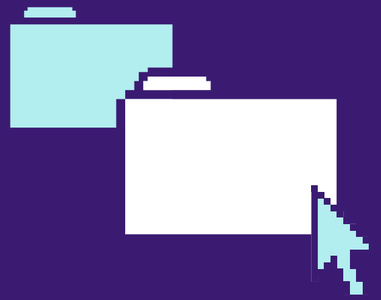
HOWEVER, YOU CAN'T GUARANTEE THAT THE ENTIRE SYSTEM WORKS, WITHOUT RUNNING IT AND CHECKING THAT ALL THE PIECES INTEROP PROPERLY.



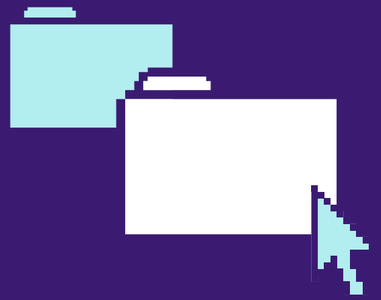

2 UNIT TESTS, 0 INTEGRATION TESTS
via reddit.com/r/programmerhumor

INTEGRATION TESTING HELPS MAKE SURE A SOFTWARE PRODUCT IS CONSISTENT AND FEATURES ARE PROPERLY IMPLEMENTED.

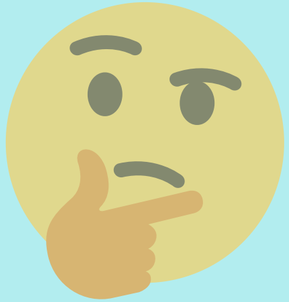
INTEGRATION TESTING CAN USE A REAL DOM (IN A REAL WEB BROWSER) OR A FAKE/PROGRAMMATIC DOM (LIKE JSDOM).



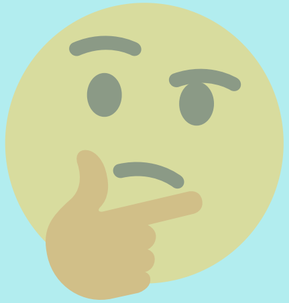
REAL DOM TESTING PROVIDES A MORE ACCURATE RESULT. HOWEVER, BROWSERS ARE HEAVY, SLOWER TO RUN, AND SOMETIMES, SUCH TESTS MAY BE FLAKIER.

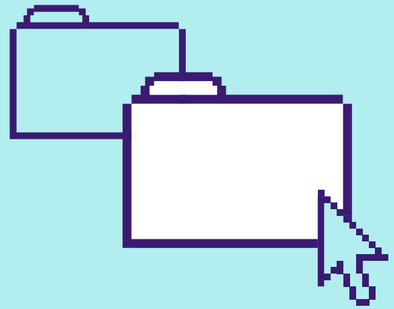


TESTING WITH A FAKE DOM IS USUALLY EASIER, AND SUCH TESTS ARE A LOT FASTER TO RUN. HOWEVER, FAKE DOM IMPLEMENTATIONS DON'T SUPPORT ALL THE DOM FEATURES THAT BROWSERS DO.



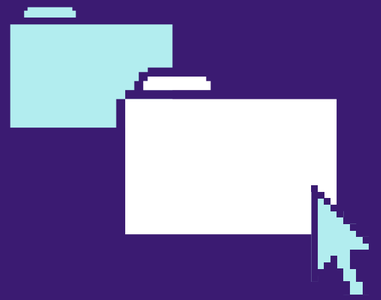
# WHAT TO CHOOSE FOR INTEGRATION TESTING? REAL DOM OR FAKE DOM?





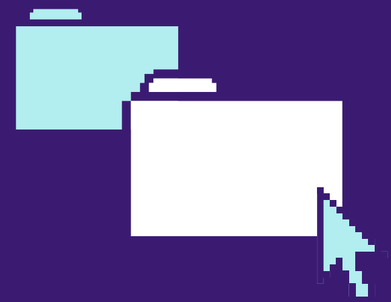
IT DEPENDS.

IN MOST CASES, USING A FAKE DOM IS SUFFICIENT.



WITH A FAKE DOM, YOU DON'T NEED A WEB BROWSER. YOU JUST NEED NODE.JS, AND A TEST LIBRARY WITH DOM SUPPORT (E.G. *JEST+JSDOM*).

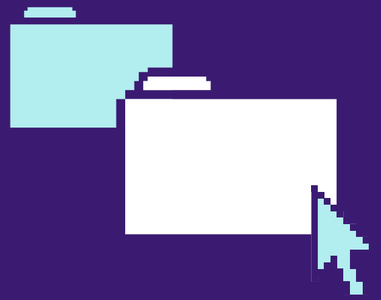
THESE TESTS ARE GENERALY MORE RELIABLE, AND IT'S EASIER TO MOCK NETWORK REQUESTS AND EXTERNAL DEPENDENCIES.



A FAKE DOM

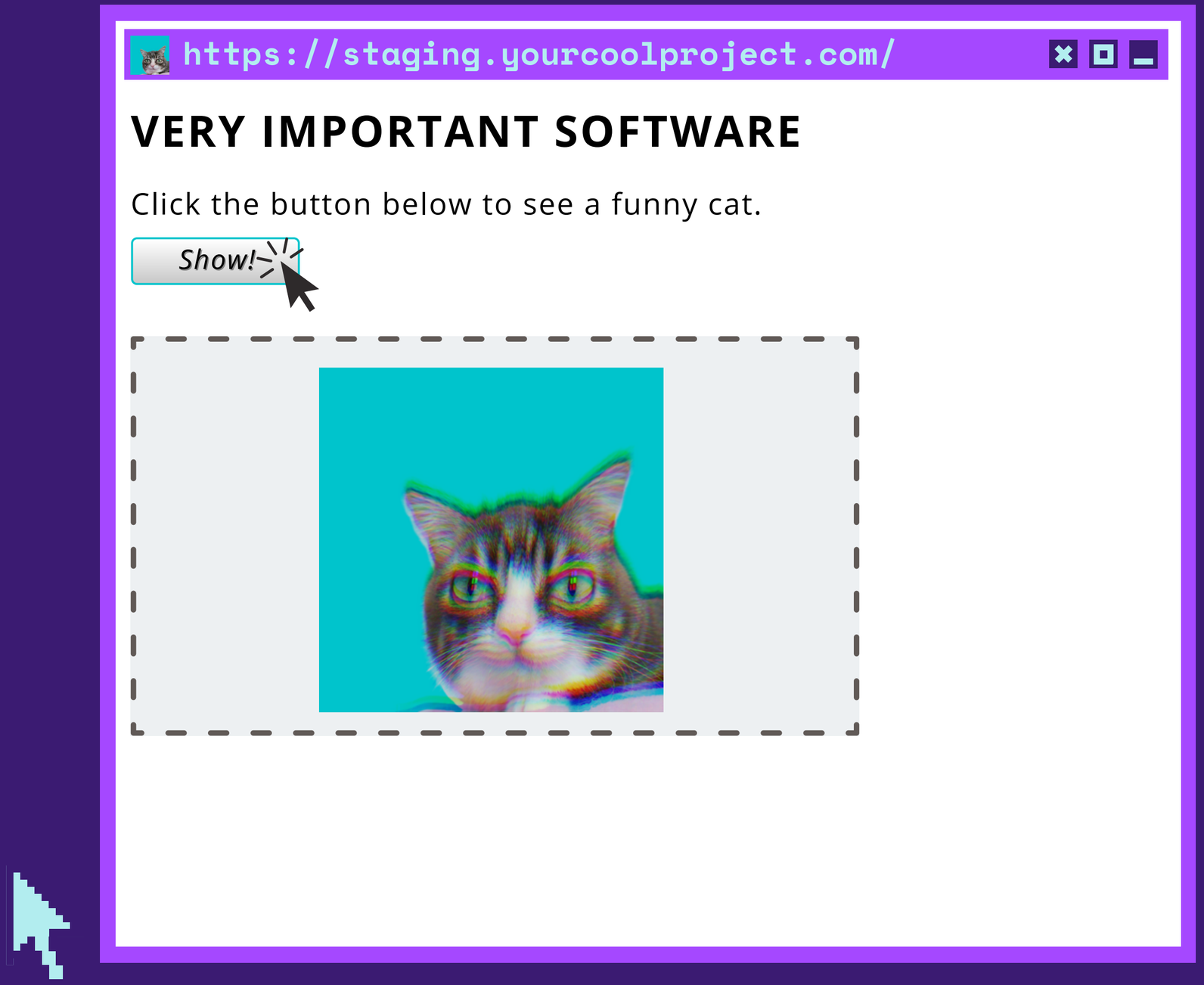
"~~64~~KB OUGHT TO BE ENOUGH FOR ANYBODY"

WE'LL DISCUSS THE INTEGRATION TESTING IN MORE DETAIL IN THE DEMO SECTION.



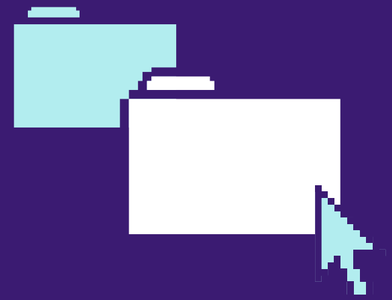
# UI TESTING

UI testing simulates a user's activity. A UI testing suite runs a full-featured web browser like Chrome (sometimes, a headless Chrome), and interacts with the elements on the screen. These interactions are triggered programmatically.

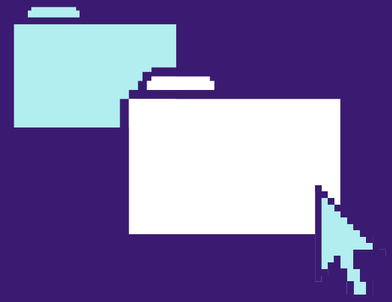


UI TESTING IS ABOUT RUNNING REAL USER FLOWS IN A REAL WEB BROWSER.

IT'S PRETTY SIMILAR TO HOW MANUAL TESTING IS DONE... EXCEPT FOR BEING AUTOMATED.

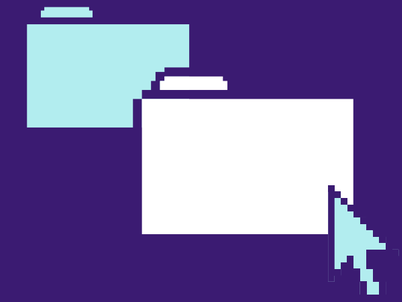


UI TESTING MAY BE DIFFICULT, AND SOMETIMES, UI TESTS ARE HARD TO MAINTAIN.



YOU HAVE TO FIND AN ELEMENT ON THE PAGE BY ITS XPATH OR CSS SELECTOR AND TRIGGER AN EVENT ON THIS ELEMENT.

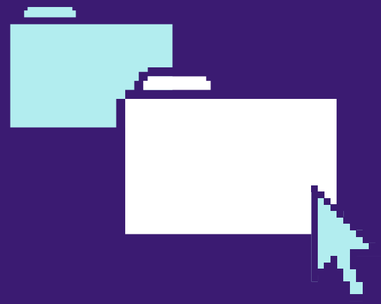
THEN, YOU HAVE TO WAIT FOR THE UI TO UPDATE OR REACT, AND CHECK THAT WHATEVER WAS SUPPOSED TO HAPPEN HAS HAPPENED.



TOOLS LIKE *CYPRESS* OR *SELENIUM* HELP ORGANIZE PROPER UI TESTING AND PROVIDE HELPERS FOR MAKING THE TESTING PROCESS EASIER.

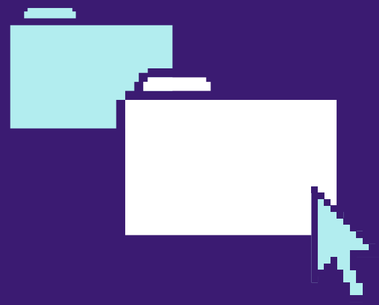
HOWEVER, SOMETIMES, UI TESTING IS STILL A PRETTY PAINFUL THING.

ONE OF THE REASONS FOR THAT IS HAVING TO USE CSS SELECTORS.



GENERATED HTML STRUCTURE/DOM IS A 'PRIVATE' THING THAT CAN BE CHANGED AT ANY TIME, WITH NO VISUAL CHANGES NOTICEABLE TO USERS.

HOWEVER, EVEN THOUGH UI TESTING IMPLIES USER FLOW TESTING, WE HAVE TO RELY ON THESE 'PRIVATE' IMPLEMENTATION DETAILS DUE TO THE PROGRAMMATIC NATURE OF UI TESTING.



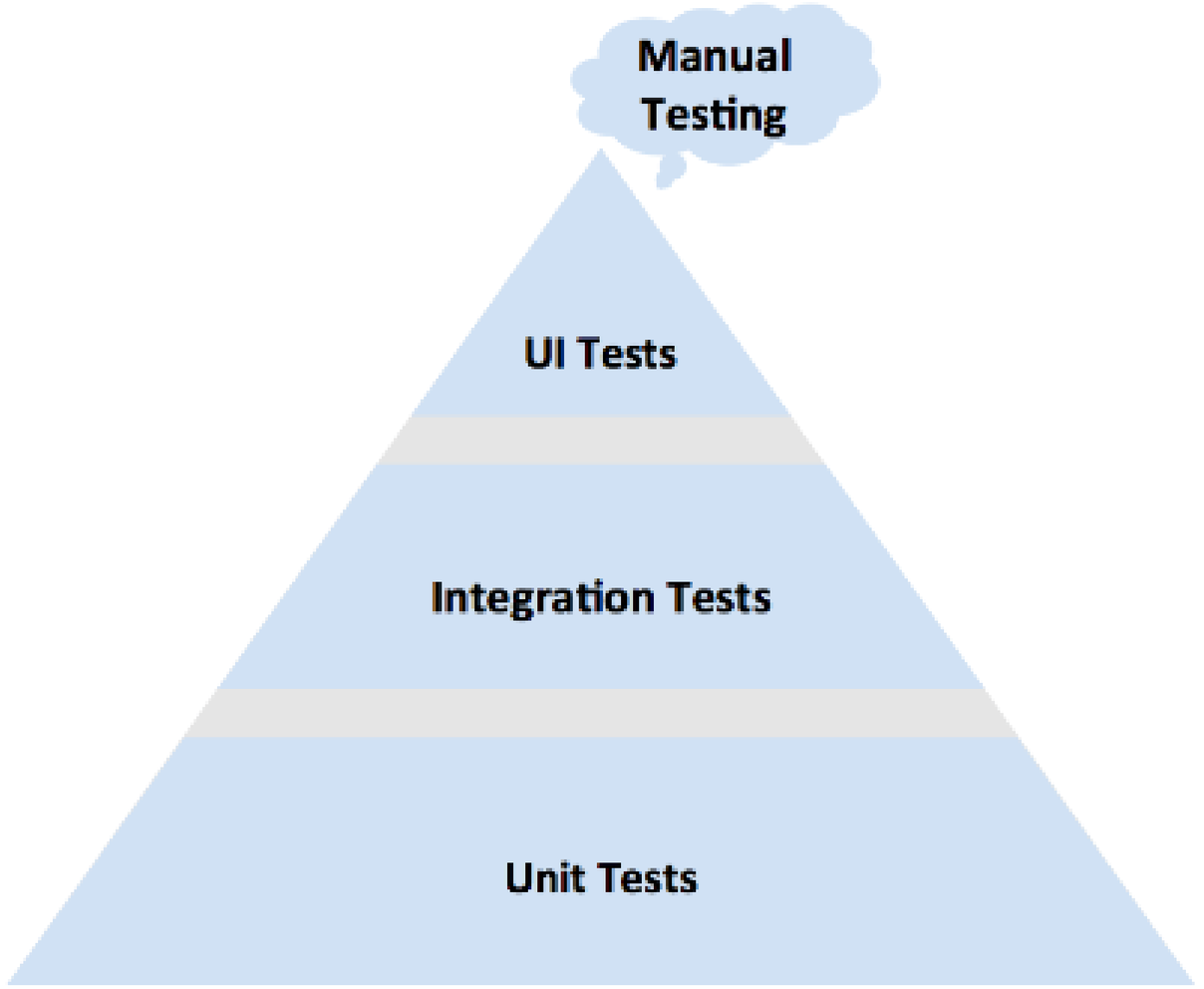
EVEN THOUGH UI TESTING MAY BE HARD,
IT IS EXTREMELY IMPORTANT.

WE SHIP SOFTWARE TO BE USED BY
PEOPLE, AND WE MUST MAKE SURE THEY
CAN USE IT IN THE DESIGNED WAY.

WE'LL DISCUSS UI TESTING AND HOW TO SET IT UP IN THE NEXT LECTURE.

## TESTING PYRAMID

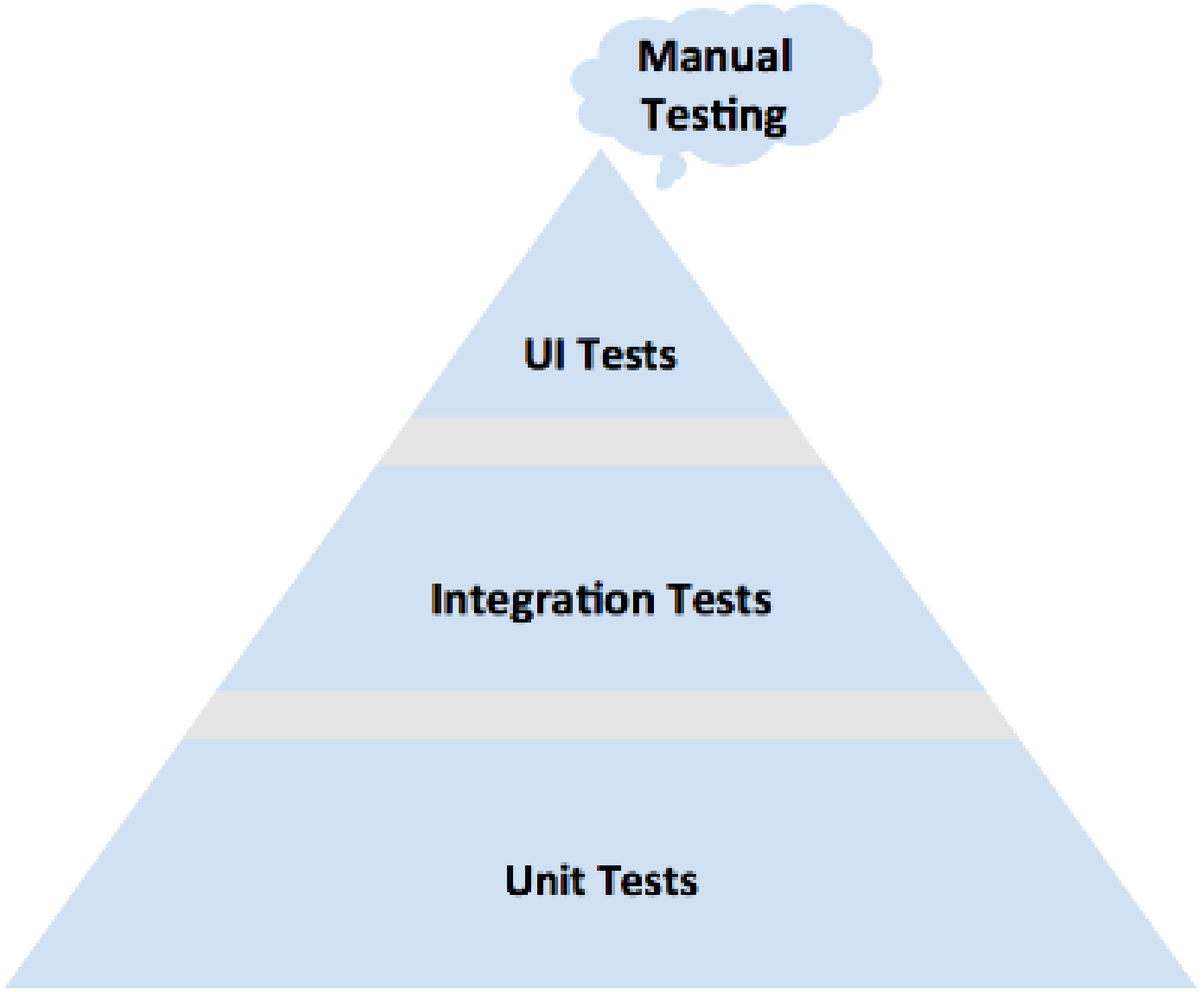
The classic '*testing pyramid*' looks like this →



# TESTING PYRAMID

The basis of the pyramid is unit tests.

They are easy to write and are fast to run, and they help keep modules stable and the software architecture neat.

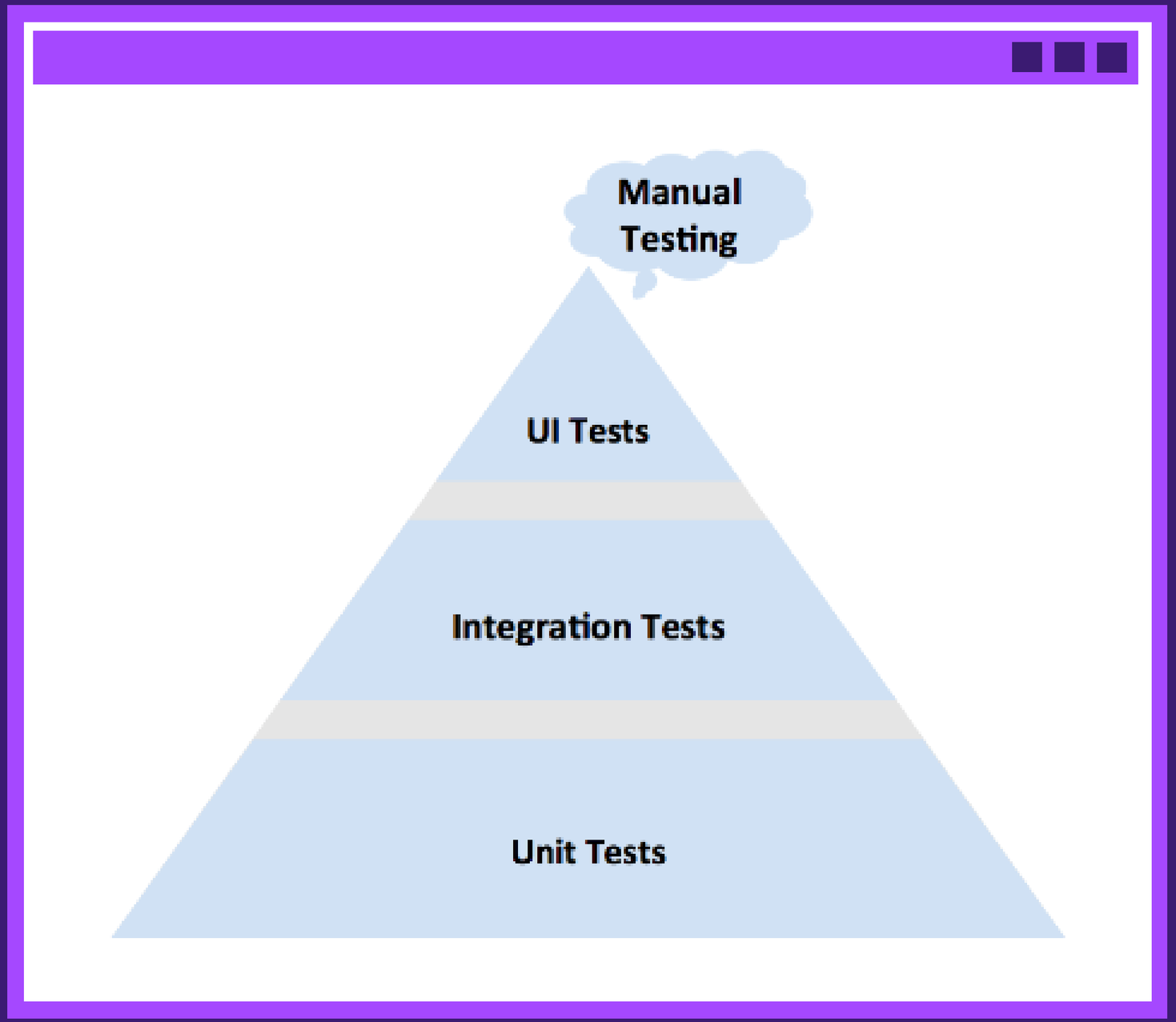


## TESTING PYRAMID

In the middle, there are integration tests.

They are a bit harder to write but they are still pretty fast to run (especially with fake DOM).

They can help ensure that modules work perfectly together.

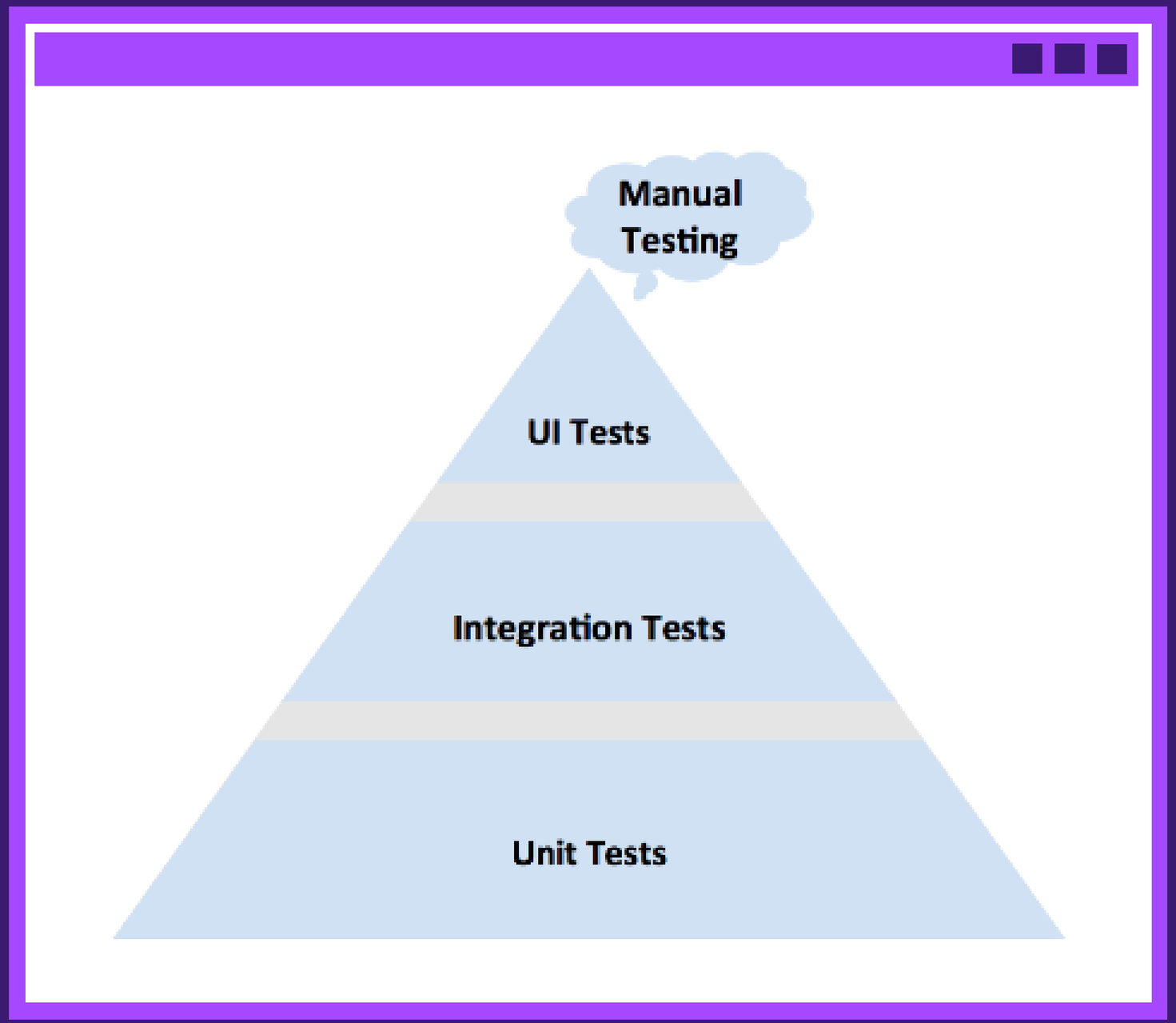


# TESTING PYRAMID

On the top, there are UI tests.

They are UI-focused and user flow-focused, and they help make sure that from the user's perspective, everything will work exactly as we want it to work.

They're slower, harder to maintain, and require running a real web browser.

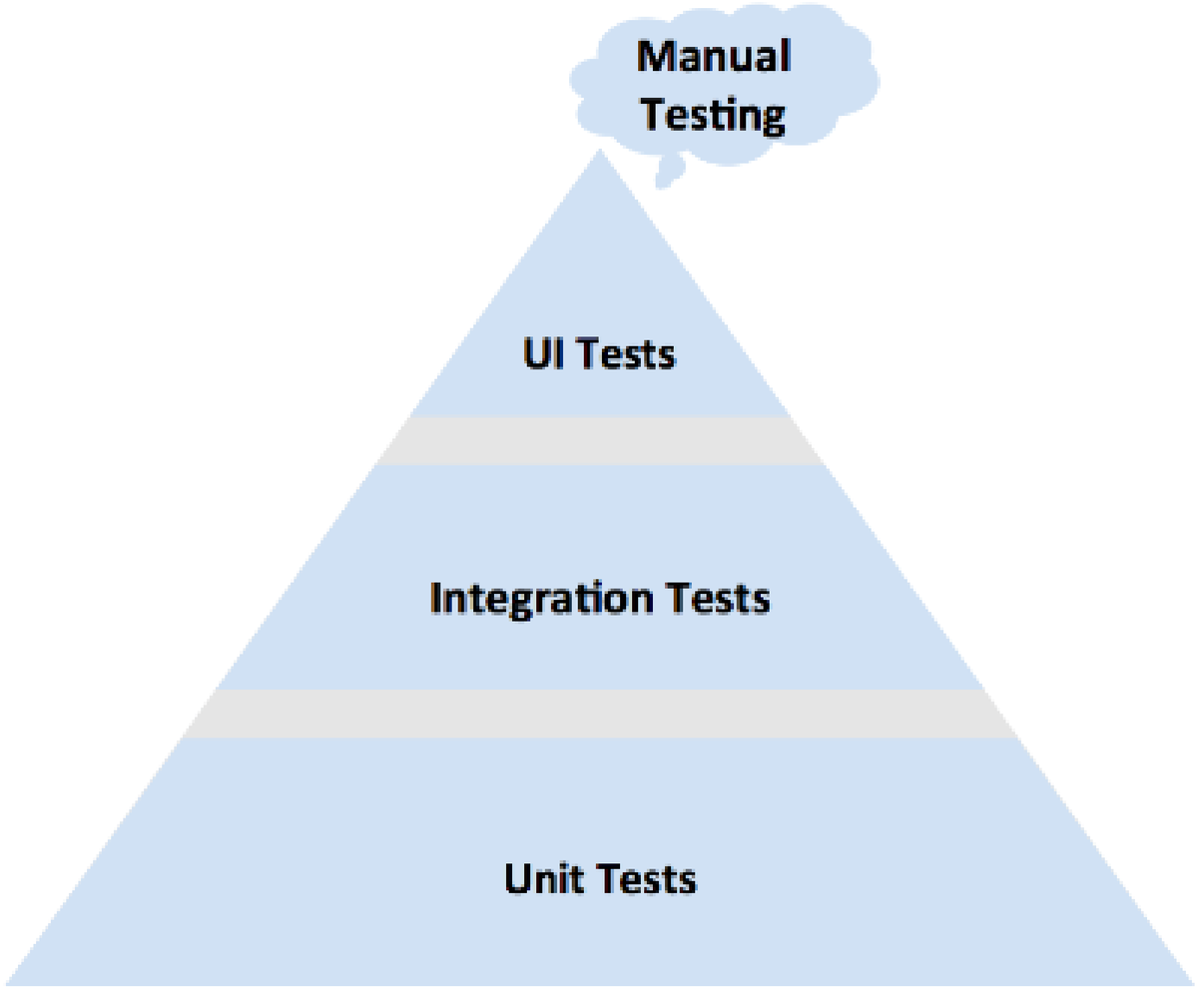


# TESTING PYRAMID

Separately and higher than the pyramid itself, there are manual tests.

Sometimes, you just have to test the software manually.

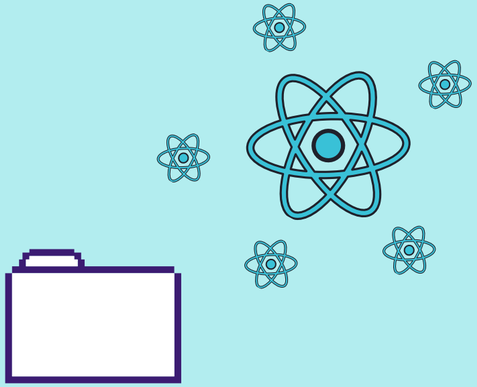
The main point is to maintain enough unit, integration, and UI tests to keep the manual testing to the minimum.



# TESTING IS IMPORTANT

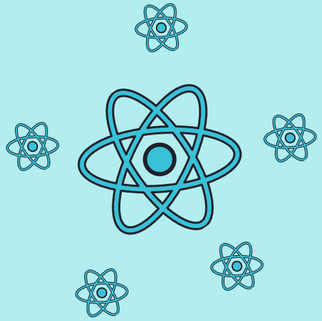
Testing saves time on debugging and helps avoid or reduce the user suffering.

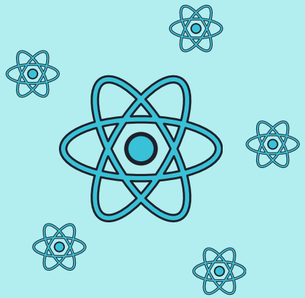
Software may be hard to use, and it's even harder when it doesn't work properly.



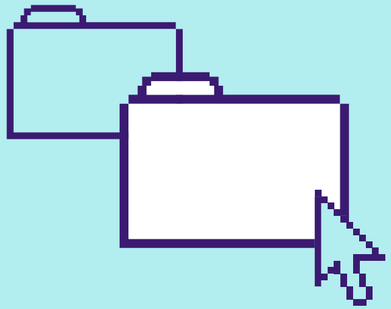
# DEMO

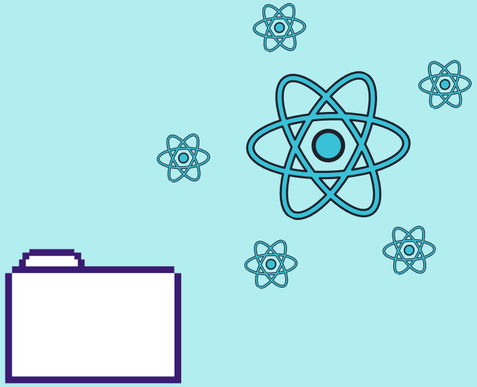
# *LET'S UNIT TEST A REACT COMPONENT WITH JEST AND REACT TESTING LIBRARY*





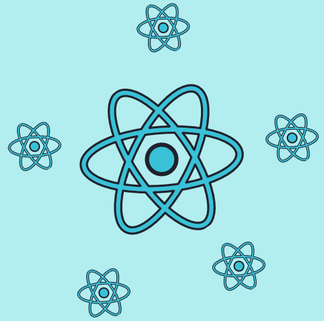
*HTTPS://JESTJS.IO/*
*HTTPS://TESTING-LIBRARY.COM*

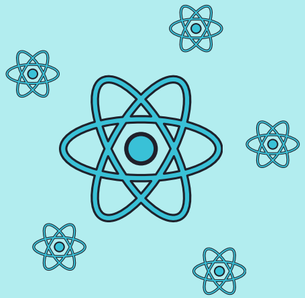




# DEMO

## *LET'S WRITE SOME INTEGRATION TESTS FOR THE PAGE WITH A CAT IMAGE*





*HTTPS://JESTJS.IO/*
*HTTPS://GITHUB.COM/JSDOM/JSDOM*