Intro to UI unit/integration testing

Before digging into the what and how let's look into the why

Why do we need unit/integration testing?

- To make sure our application behaves and looks as expected
- To make sure new changes do not break older ones
- Ship code faster*
- Side effect- Can serve as documents for user flows

To sum it up -> To be as confident as her 👆 while pushing new changes

Types of testing

01 **Unit Test** - Single unit of code in isolation
Eg; A button component

02 **Integration Test** - Multiple units working together as expected
Eg; Search feature in a page

03 **End to End** - Like a user interacting with the real app
Eg; The entire login/logout flow

What to test?

-Hmmm, ideally everything 😅

What to test?

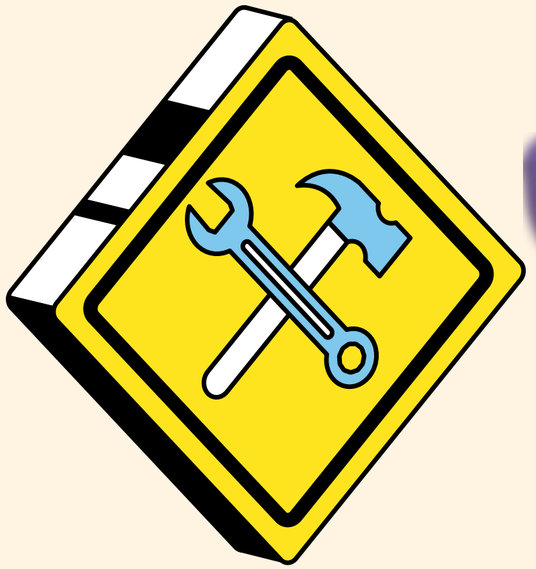
But sadly we don't live in an ideal world so we need to prioritise the tests

- High-value features, the ones driving the maximum business value and most actively used
- Things that are easy to break, like the parts with max cross-team contributions and complex features
- Parts of the application with a lot of spaghetti code

Okay enough of whats and whys, let's dive into the **how**

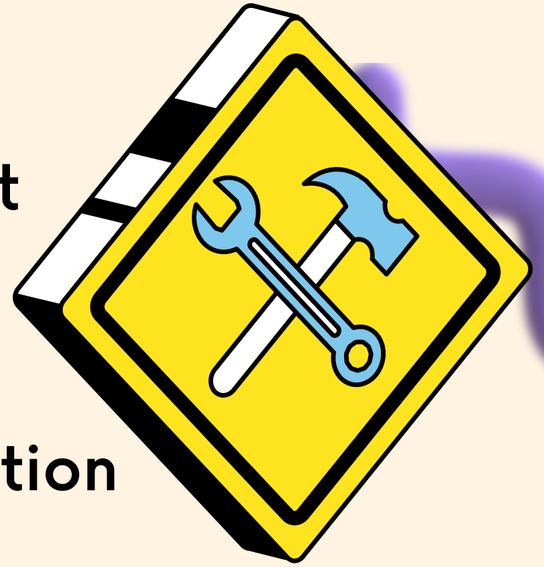
How to write unit/integration tests?

Tools for the job



Jest - It is a JS testing framework with a **test runner**, **assertion** and **mocking** capability built-in. It requires almost zero configuration to get started with. The best part, it is fully framework agnostic.

React Testing Library - It is a library for testing React components. The creator is none other than the OG **Kent C. Dodds.**



Instead of writing tests that are based on the **implementation details** of the component which are much more likely to change over time and hence make our tests brittle, RTL takes a totally different approach to writing test cases which is more inclined towards user's behavior rather than the implementation details.

How to put these tools into use?

Let's dig into the **structure** of a typical test block

```
test('counter increments and decrements when the buttons are clicked', async () => {
  render(<Counter />)

  const increment = screen.getByRole('button', {name: /increment/i})
  const decrement = screen.getByRole('button', {name: /decrement/i})
  const message = screen.getByText(/current count/i)

  expect(message).toHaveTextContent('Current count: 0')
  await userEvent.click(increment)
  expect(message).toHaveTextContent('Current count: 1')
  await userEvent.click(decrement)
  expect(message).toHaveTextContent('Current count: 0')
})
```

1. Render the component to test

2. Find the elements to interact with

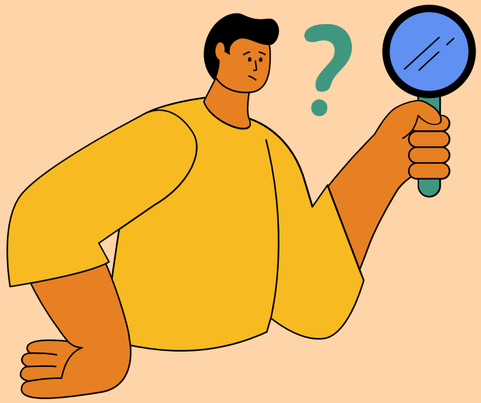
3. Simulate the interaction

4. Make the assertion

Related tests can be grouped together into a single **describe** block

```
describe("Todo",()=>{
    test("on adding a todo it should show up in the todo list",()=>{
        render(<MockTodo/>);

        addTask(["buy eggs"])

        const divElement = screen.getByText(/buy eggs/i)
        expect(divElement).toBeInTheDocument();
    })

    test("on adding multiple todos all should show up in the todo list",()=>{
        render(<MockTodo/>);

        addTask(["buy eggs","turn geyser off","wash car"])

        const divElements = screen.getAllByTestId("todo-item")
        expect(divElements.length).toBe(3);
    })
```

How to query/find an element to interact with?



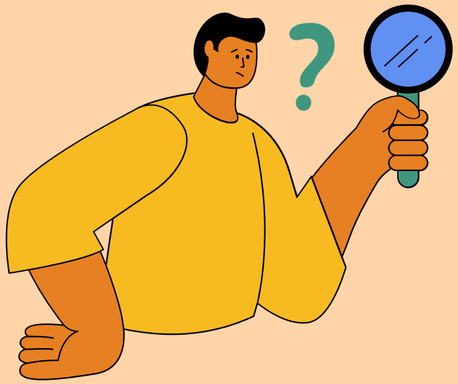
screen is a utility provided by RTL to easily interact with the component that we rendered and get elements from within that component.
There are different queries available for our use, the basic structure of a screen query looks like this

<find/get/query>.<allBy/By>.<[attribute]>

How to form the query?

allBy vs By

allBy returns an array with all the matching elements.
By returns an error if it finds more than one matching element



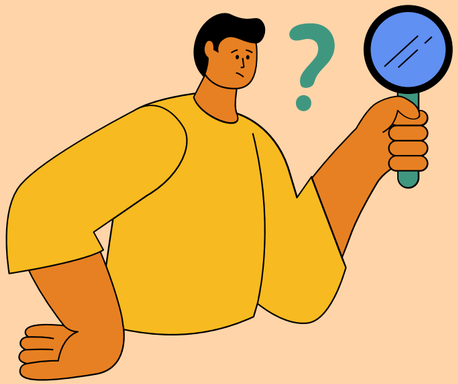
get/query vs find

find searches for an element asynchronously which can be useful when querying for elements which might get rendered asynchronously say after an API call

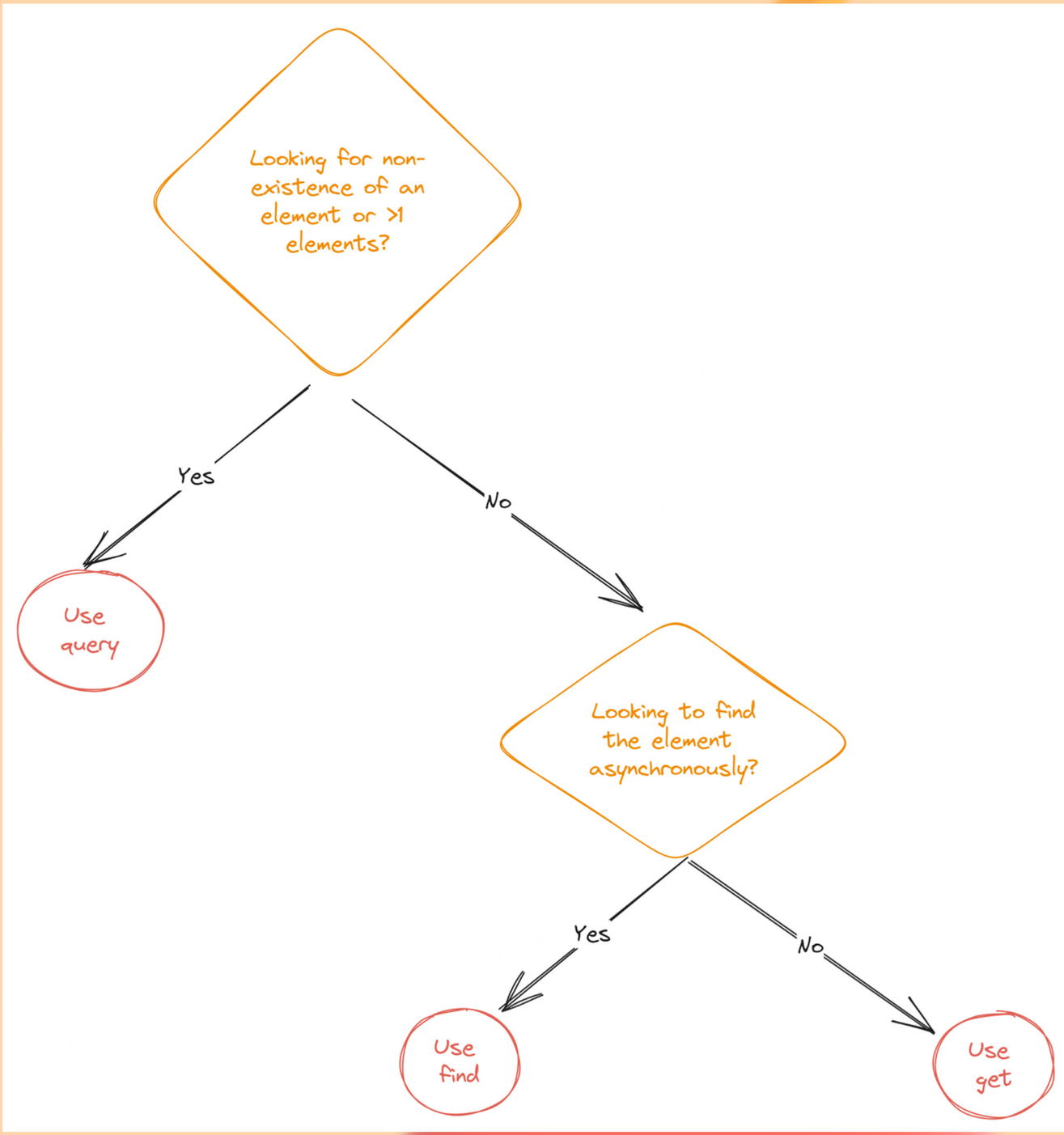
get/find vs query

get/find will throw an error if it finds no match or even on more than 1 match but **query** will only throw an error if there is more than one match, basically, if a match is not found query doesn't throw an error, unlike get/find. It is mostly used when we want to check for the non-existence of an element because using get here will always make the test fail

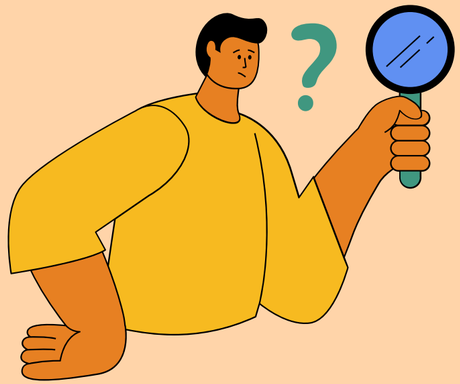
How to form the query? (cont.)



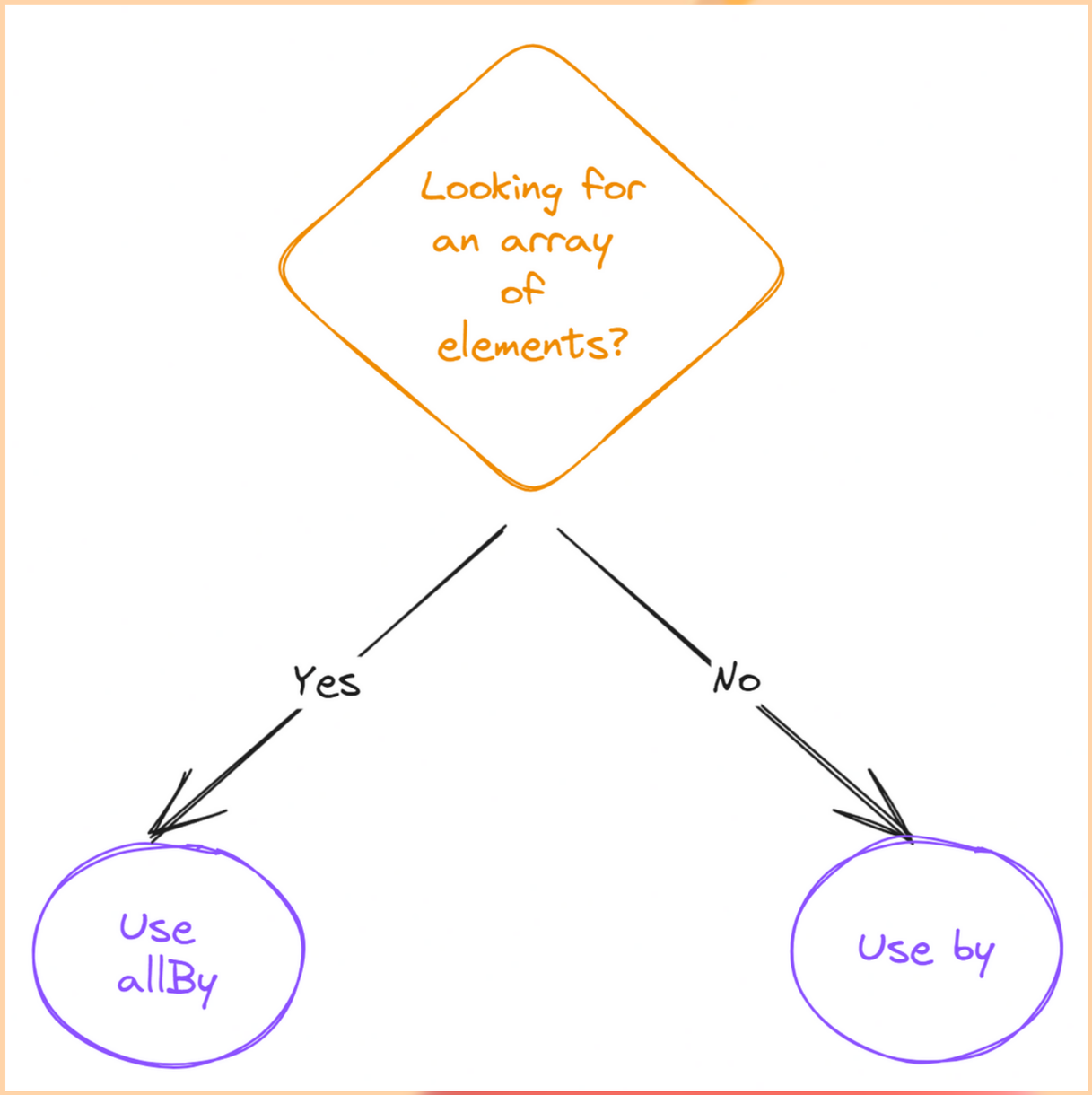
find vs get vs query



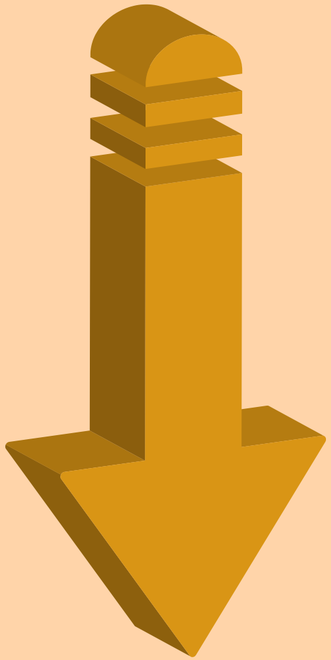
How to form the query? (cont.)



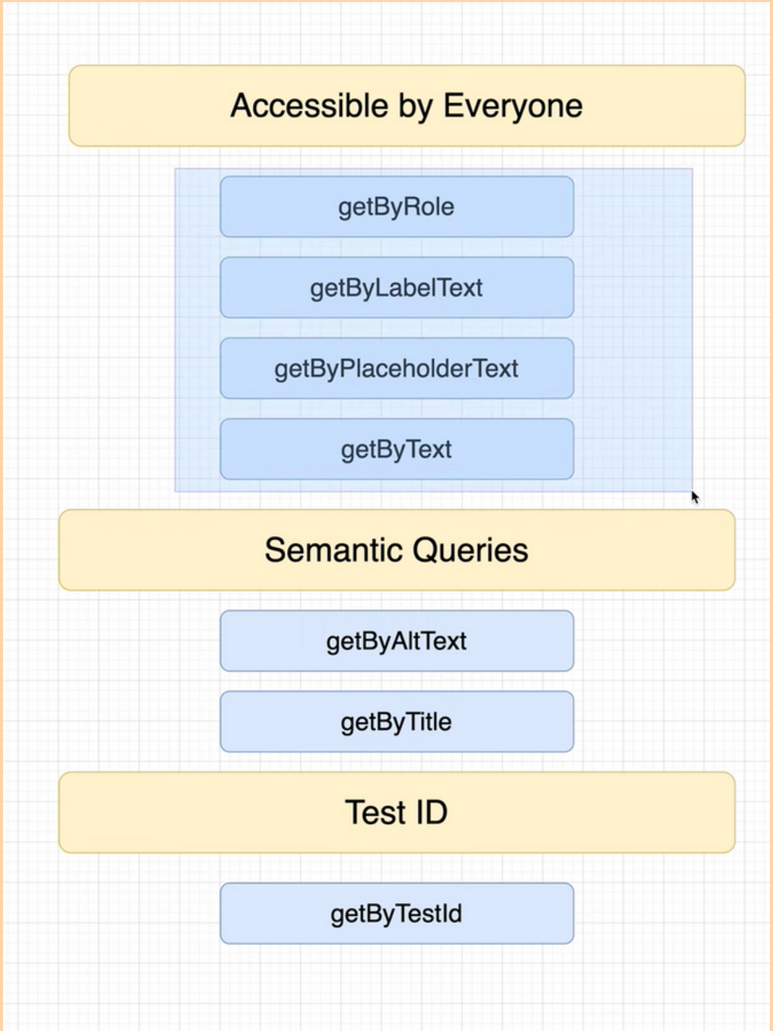
all vs allBy

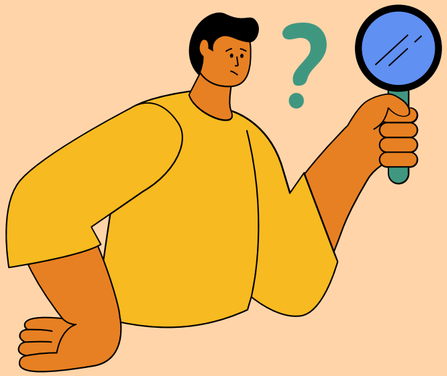


How to form the query (cont.)?



Attributes by priority





```
const divElements = screen.getAllByTestId("todo-item")
```

Simulating interaction

Once the element has been queried and found, the next step is to interact with it.
RTL provides a utility called **userEvent** for simulating common user interactions.
One thing to note here is that it is not a part of the core library so needs to be added separately but there is another utility called **fireEvent** , provided natively by the library which simulates native browser events. The difference is that a user-event could be a combination of multiple browser events and **userEvent** abstracts that for us so that we can focus only on the user interaction.

```
await userEvent.click(increment)
```

Making assertions

This is the main logical part of a test which decides whether a test was successful or a failure.

Some commonly used assertions

- toBeInTheDocument()
- toBeVisible()
- toContainHTML("p")
- toHaveTextContent("some text")
- toBeFalsy
- toBe

Making assertions(cont.)

Examples

```
expect(divElement).not.toHaveClass("todo-item-active")
```

```
expect(divElements.length).toBe(3);
```

```
expect(pElement).toBeInTheDocument();
```

Thank you!

learn, code & repeat.