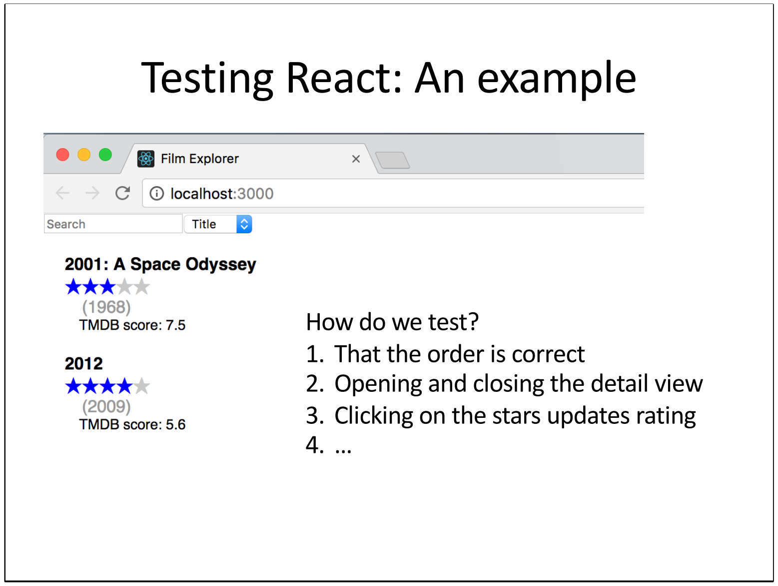
Testing a React application?

- Testing a React application isn't conceptually different than testing any other code

  Provide an input to the application and assert the output matches your expectation

- The difference is that some of those inputs are user actions, and the outputs are often UI

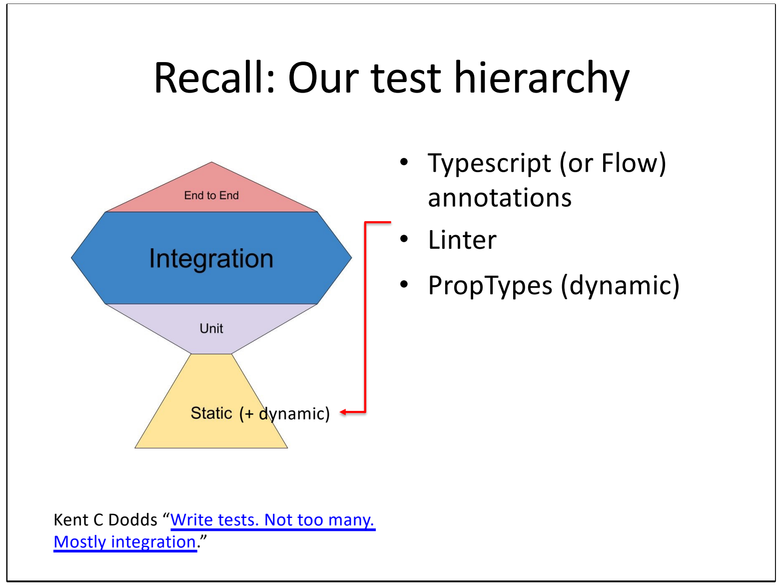
  A challenge is describing the relevant inputs and expected outputs



Here are some the properties we might want to verify in our UI... The latter depends on the what the user "sees" and interacts with the system.

When we talked about the need for tests to be "self-checking"", this is where that starts getting trickier. How do we test if it looks right? The simple answer is that we just open it up and look. We press buttons. But now we can't automate the process. A human must sit in front of a screen and sign off on it. This is slow, the tester must remember all the things that could be an issue with the interface, and test isolation is harder.

So, we need to automate interaction. To do so we will need more than the tools we have used so far. There are several different solutions, but here we will use the library recommended by the React team: React Testing Library.



All these levels are still relevant to testing our react applications:

- End-to-end testing will run our entire application (including server) and interact with the application just like a user would. There are number of tools designed for this purpose. These tools enable you to automatically "click" and make assertions about the results. These tools will often use a "headless" browser behind the scenes.
- Integration tests will render some or all of the front-end application and typically include some interaction, i.e., clicking on a button. We will typically mock network requests and other functionality to ensure our tests are F.I.R.S.T.
- Unit tests to verify helper functions or other tricky UI. The distinction between unit tests and integration tests is fuzzy. We might think of unit tests as those tests that only test a single component in isolation.

https://kentcdodds.com/blog/unit-vs-integration-vs-e2e-tests

Typescript is typed "version" of Javascript that is compiled (really transpiled) into standard Javascript. flow is a tool for annotating JavaScript with types that seems to have largely displaced in favor of Typescript.

PropTypes is a little more narrowly applied than either of the other, it allows us to specify the types of the props passed to our React components.

PropTypes in action

```
LabeledSlider.propTypes = {
  label: PropTypes.string.isRequired,
  value: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number,
  ]).isRequired,
  setValue: PropTypes.func.isRequired,
};
```

Bit of a "code smell"

Catch errors *and* document component "signature"

https://reactjs.org/docs/typechecking-with-proptypes.html

The more specific we can make these requirements the more likely we are to catch type errors (generally true for all kinds of validation). Note that validation isn't the only purpose for providing `PropTypes`. Doing so is also a way of documenting the "type signature" of the component (analogous to a function signature in a statically typed language). These won't change the behavior of your application, not even if you specify them incorrectly, but it will cause the component to complain if it is passed the wrong type a value, which we can pick up when we run our tests.

What is the expected type of the `submit` prop?

```
function NameForm({ record, submit }) {
 const [name, setName] = useState(record ? record.name : '');

 const handleName = (event) => { setName(event.target.value); };

 return (
  <div>
   <input type="text" value={name} onChange={handleName} />
   <button onClick={() => submit({...record, name: name})}>Submit</button>
  </div>
 );
}
```

A. String
B. object
C. Number
D. function

Answer: D

Since we are using submit like a function in a callback, it is likely a function.

Prop Types for this component?

Two props named record, submit

Record can be “falsy”, and has a property name

```
function NameForm({ record, submit }) {
 const [name, setName] = useState(record ? record.name : '');

 const handleName = (event) => { setName(event.target.value); };

 return (
  <div>
   <input type="text" value={name} onChange={handleName} />
   <button onClick={() => submit({...record, name: name})}>Submit</button>
  </div>
 );
}
```

The name property looks to be a string

submit used like a function

Optional object with at least name property

```
NameForm.propTypes = {
 record: PropTypes.shape({ name: PropTypes.string }),
 submit: PropTypes.func.isRequired,
};
NameForm.defaultProps = { record: null };
```

Required callback function

A couple bits of syntactic sugar at work:

- Destructuring to ”split” props object into its component properties in the function definition.
- Spread operator to create a new record object (the …record) part and then overwrite that with a new value for the name property.

By reviewing this code, we can make inferences about the props and thus what types to specify. Note that if a prop is optional, as record is here, we want to specify a default value (even if that default value) is just null. ESLint can warn you about missing default props.

Recall: Testing is ultimately about confidence

We test to build confidence:

- That our application works as intended, and
- Keeps working as intended, even when we make changes

Our goal is maximum confidence!

https://kentcdodds.com/blog/unit-vs-integration-vs-e2e-tests

Recall that each level of testing has tradeoffs, typically the higher levels have increased complexity and more points of failure but offer increased confidence the application actually works (because you are testing all of the pieces). The ultimate mix of testing, and what “level” you call it doesn’t matter that much. What matters is that you build confidence in your application.

Recall: Test-driven development (TDD)

- Think about one thing the code *should* do
- Capture that thought in a test, which fails
- Write the simplest possible code that lets the test pass
- Refactor: DRY out commonality w/other tests
- Continue with next thing code should do

Red – Green – Refactor

Aim to "always have working code"

Recall our focus is on agile development methods, which are all about short development cycles that improve working (but not yet complete) code. To that end we will practice test-driven development in which we write the tests first, then implement the code that passes those tests (I suspect this is very different from the way you typically work...)

What do we need to test a React application?

1. Ability to 'render' components (and execute any hooks or class lifecycle methods)
2. Simulate user actions
3. Find and make assertions about what is rendered (before and after those actions)

React Testing Library (RTL)

"The more your tests resemble the way your software is used, the more confidence they can give you."

-Kent C. Dodds

- Test DOM nodes (what is shown by browser), not components
- Tests should work the way the application is to be used

https://testing-library.com/docs/guiding-principles

To understand the contrast, there are other libraries, like Enzyme, which give us more control. They "know" React and we can test components (i.e., we can directly query their props and state — though hooks make the later more problematic). The problem is that users don't see props and state, they see the list of titles changing when the section changes.

Recall that Kent C. Dodds is the one who thought we should focus on integration testing (so maybe this view isn't that much a surprise).

The React Testing Library is built on top of the DOM Testing library. They realized when they stripped away all the low-level implementation details, that they essentially had a framework for testing dynamic websites, full stop. So, they now support six or seven different component frameworks. That "support" basically extends to a couple of functions that handle rendering the DOM virtually from components in the various libraries.

RTL: Rendering

`render(component)`

- Performs a virtual render of. React component
- Returns an object containing the rendered component, a rerender function, and query properties

`rerender(component)`

Returned by `render`, used to change props on a mounted component

`cleanup()`

Unmount React trees (this is handled for us by Jest)

`act()`

Wrapper around React `act()` function; makes sure React tasks are complete

Like cleanup, we won't need to use act very often. Most of the Testing Libraries helpers are already wrapped in an act() function.

RTL: Find components part 1, variants

`getBy*` or `getAllBy*`

Queries the DOM for the first matching node or array of matches, throwing error if none (or more than one for the singular variant) are found

`queryBy*` or `queryAllBy*`

Queries the DOM for the first matching node or array of matches, returning null or empty [] if none are found

`findBy*` or `findAllBy*`

Returns a Promise which resolves when a matching node(s) is found, throwing an error after 1000ms if none are found

A query is a variant + a type, e.g., `queryByText()` or `findAllByRole`

We have three variants of the queries we can run on the DOM. The star represents that are there of many of versions of the variants that differ based on what they are "getting", "querying" or "finding".

The behavior for each is subtly different. To some extent you can use whichever ones you like. However, if a component should be on the page, **get** is a good choice. The test will fail before you get to the assertion. If the component may or may not be on the page, or if you are testing for it not being present, then **query** is a good choice. The **find** matcher is good for picking up on components that should be appearing based on some interaction.

In general, the singular variant will throw an error is the query returns more than one component.

RTL: Find components part 2, types

- `*ByText`

  Search for an element based on the text contents of the node

- `*ByRole`

  Search based on the role of the component (e.g., listitem, button, textbox) and other properties

- `*ByTestId`

  Search for specific components based on `data-testid` property (basically the cheat code and not really in the spirit of the library)

A query is a variant + a type, e.g., `queryByText()` or `findAllByRole`

These are three completions to the queries. There are several others like ByLabel or ByTitle, but I have found that I stick primarily to these.

By role is an interesting one, because it taps into the accessibility features of the DOM. Certain DOM elements have a generic role, e.g., button. We can use elements for roles, and provide accessibility labels that communicate their intended role, which can be accessed by this query

```html
<div>
  <label for="email">Email address</label>
  <input type="email" id="email" placeholder="Enter email" />
</div>
<div>
  <label for="password">Password</label>
  <input type="password" id="password" placeholder="Password"/>
</div>
<div>
  <label for="terms">
    <input type="checkbox" id="terms"/>
    <span>
      I accept the terms and conditions
    </span>
  </label>
</div>
<div>
  <button type="submit">Submit</button>
</div>
```

How could we find this button?

`screen.getByText("Submit")`

`screen.getByRole('button', { name: /submit/I })`

https://testing-playground.com

Imagine our components generated the following DOM (HTML). We would want to make some assertions about the submit button (e.g., maybe it should be disabled until someone checks the box). To do so we first need to query it on the page. How could we do so? Let's start with the simplest variant, get. And then think about different types, e.g., ByText, ByRole or ByTestId. How could we use those types to find this button?

Assertions/matchers

- toBeDisabled
- toBeEnabled
- toBeEmpty
- toBeEmptyDOMElement
- toBeInTheDocument
- toBeInvalid
- toBeRequired
- toBeValid
- toBeVisible
- toContainElement
- toContainHTML
- toHaveAttribute
- toHaveClass
- toHaveFocus
- toHaveFormValues
- toHaveStyle
- toHaveTextContent
- toHaveValue
- toHaveDisplayValue
- toBeChecked
- toBePartiallyChecked
- toHaveDescription

https://github.com/testing-library/jest-dom

Of course, once we have the DOM element, we need to make an assertion. The testing library provides some custom matchers in the jest-dom package. In our previous example, we could use the toBeDisabled matcher to assert the button is disabled until we check the box. To do the latter we will need a way to simulate the user interaction.

RTL: Actions

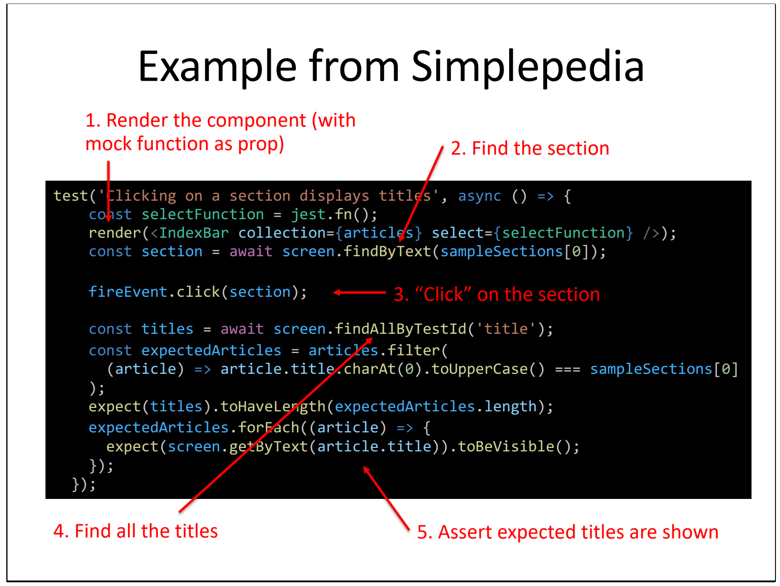
- `fireEvent.`*`type`*`(component, event properties)`

  Simulate user interaction where *type* is any kind of HTML event: click, change, drag, drop, keyDown, etc…

General behavioral testing pattern

1. Test that we are in the initial state
2. Initiate an action that should change state
3. Test that we are in the new state
4. [Initiate action to return state to original]
5. [Test that we are in original state]

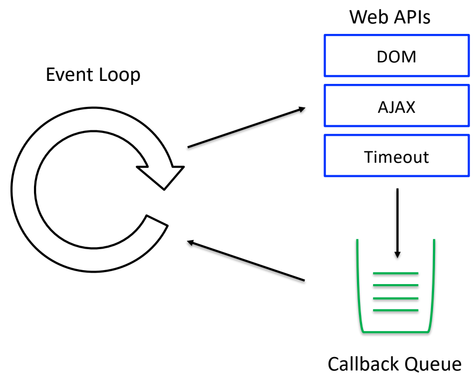
Why the first step? Without it how do we know the action caused any change. What if the component was previously in the expected state. While steps 4 and 5 are not strictly necessary, it is good practice, especially for ”toggling” behaviors.



What is missing here? The check on the state prior to the action.

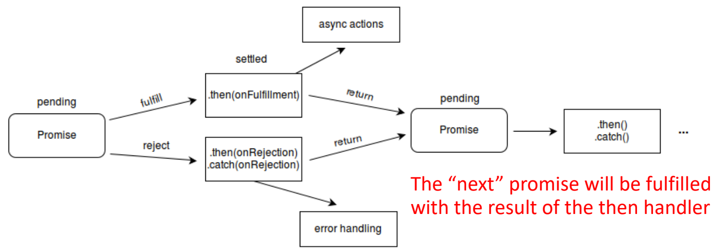
What is async and await? These are tools for managing asynchronous computations, and particularly to enable an imperative style to working with Promises.

Recall: The browser is asynchronous



A promise is a proxy for a value not yet available…

A common action is to update state



[MDN]

A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future.

One of the trickier aspects of Promises, is that the chains don’t ”stop” and become synchronous at some point. Instead, each invocation of then returns a promise. That promise may be fulfilled by the return value of the callback provided to then, or replaced by a promise return by that callabck.

Then can take both fulfillment and rejection handlers, although typically just used with fulfillment handler.

Promise vs. callbacks

```
someAsyncOperation(someParams, (result, error) =>
  // Do something with the result or error
  newAsyncOperation(newParams, (result, error) => {
    // Do something more...
  });
});
```

Flatten nested structure into a chain:

```
someAsyncOperation(someParams).then((result) => {
  // Do something with the result
  return newAsyncOperation(newParams);
}).then((result) => {
  // Do something more…
}).catch((error) => {  // Handle error});
```

One of the key advantages of Promises is flattening a deeply nested set of callbacks into a linear chain of promises. In our example here the first then (invoked on the Promise returns by someAsyncOperation) returns a Promise. That promise is eventually replaced by the Promise created by `newAsyncOperation in its handler.`

If instead of executing steps in sequence, you want to execute a set of synchronous operations in parallel, use:
Promise.all: If you care when they are all fulfilled
Promise.race: If you just care when the first Promise fulfills/rejects

Assume the function wait(sec) returns a promise that resolves in sec seconds. What is the output of the following code?

```
const current = Date.now();
wait(3).then(() => {
  console.log(`Delay 1: ${Date.now() - current) / 1000)}s`);
  return wait(4);
}).catch(() => {
   console.log(`Delay 2: ${Date.now() - current) / 1000)}s`);
});
console.log(`Delay 3: ${Date.now() - current) / 1000)}s`);
```

A	B	C	D	E
Delay 1: 3s Delay 2: 7s Delay 3: 7s	Delay 1: 3s Delay 3: 4s	Delay 1: 3s Delay 3: 7s	Delay 3: 0s Delay 1: 3s	Delay 3: 0s Delay 1: 3s Delay 2: 7s

Answer: D

The wait function returns immediately with a promise. Thus, the final console log executes first, and after 3 seconds the first promise resolves and we print "Delay 1". The original promise return by the `then` method is replaced by the promise return from wait(4), which will ultimately resolve 4 seconds in the future. However, nothing is "listening" for that promise to be fulfilled. The only listener is remaining is the catch. Since there is no error, we don't end up executing the catch statement (no error to handle), and thus printing Delay 2.

async/await: Imperative approach to asynchronous code

```
test("…", () => {
  …
  screen.findByText(sampleSections[0]).then((section) => {
    fireEvent.click(section);
    return screen.findAllByTestId('title’);
  }).then((titles) => {
    …
  });
});
```

```
test("…", async () => {
  …
  const section = await screen.findByText(sampleSections[0]);
  fireEvent.click(section);
  const titles = await screen.findAllByTestId('title’);
});
```

“Imperative” style

We noted earlier that there is not way to “stop” a Promise chain and switch back to synchronous imperative code. That is true. However, the async and await keywords provide syntactic sugar to make it appear that is happening. That is the following chain of Promises can be translated into to following imperative style code. The “await” specifies that the subsequent code should not execute until the promise has resolved.