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

  *The challenge is often getting “access” to the relevant props, components, etc.*
Recall: The hierarchy of testing (from “high” to “low” level)

• **System (or end-to-end) testing:** Testing the entire application

• **Integration testing:** Tests of combinations of units (i.e. integration of multiple units)

• **Unit testing:** Tests for isolated "units", e.g. a single function or object
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 with minimum effort!*
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 for “always have working code”*
What do we need to test a React application?

1. Ability to ‘render’ components (and execute any hooks or class lifecycle methods)
2. (Optionally) simulate user actions
3. Make assertions about what is rendered (after those actions)

*We will use the Enzyme library for these tasks*
Testing with Enzyme

- **shallow(<Component ... />)**
  
  *Renders component, but not the contents of any children (and only runs some lifecycle methods)*

- **mount(<Component ... />)**
  
  *Renders component and all children*

- **render(<Component ... />)**
  
  *Renders all components, but produces static HTML*

Warning: Support for hooks with shallow is incomplete
A (partial) list of kinds of tests we will implement

• “Smoke tests”
  Does it successfully render (or does it start “smoking”)

• Snapshot tests for regression testing
  Record JSON description of rendered components, check if subsequent renders are identical

• “Behavioral” tests
  Make assertions about rendered components/data (in response to user actions)
True or False? You can use snapshots for TDD.

A. True
B. False
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]
Example tests for FilmDetail/FilmSummary toggle

// Render component
const comp = mount(<FilmContainer {...film} setRatingFor={jest.fn} />);

// 1. Test precondition
expect(comp).not.toContainMatchingElement(FilmDetail);

// 2. Initiate an action
comp.find('FilmTitle').simulate('click');

// 3. Test the post-condition
expect(comp).toContainMatchingElement(FilmDetail);

// 4 & 5. Initiate action to return to original state
comp.find('FilmTitle').simulate('click');
expect(comp).not.toContainMatchingElement(FilmDetail);

Find component in rendered components
Action
Enzyme specific matchers
Testing (asynchronous) components with hooks is tricky

• Hooks trigger re-renders that are hard to capture
  Need to wrap `act()` function around code that will lead to state update
  Enzyme implements `act()` internally for us

• Asynchronous hooks (like fetching data) are particularly trick to test
Example with FilmExplorer

// Mock fetch to return test data (wrapped in Promises)
jest.spyOn(global, 'fetch').mockImplementation(mockFetch);

const comp = mount(<FilmExplorer />);

Asynchronous actions are still running! Component may not have update before assertions.

Need act because state change could occur
await act(async () => await flushPromises());
comp.update();

Wait until all promises resolved then ensure Enzyme updates component

// Assertions on FilmExplorer
expect(comp) ...
Imperative approach to asynchronous code with async/await

```javascript
useEffect(() => {
  const prom1 = fetch('/api/films/');
  const prom2 = prom1.then((response) => {
    return response.json();
  });
  prom2.then((data) => {
    setFilms(data);
  });
});

useEffect(() => {
  const runEffect = async () {
    const response = await fetch('/api/films/');
    const data = await response.json();
    setFilms(data);
  };
  runEffect();
});
```

“Synchronous” style