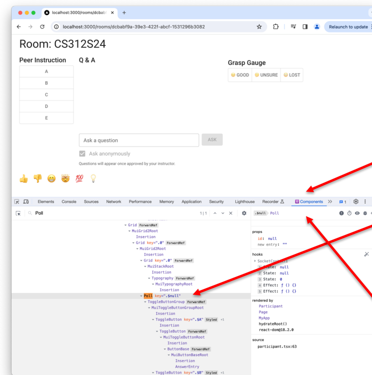
(Initial) due dates in the next week

- Tonight: practical 3, assignment 2
- Next Thursday: practical 4, assignment 3

Debugging our applications with React DevTools*



1. Open browser's developer tools
2. Select "Components" tab
3. Find component to inspect/modify props, state hooks, etc.
4. Use "Profiler" to see what rendered, why and when

*Install Chrome, Firefox extension

Recall when we talked about debugging, we noted one of the key steps was visualizing the errors. We can do so with the React developer tools (if you haven't installed that extension, now is a good time to do so). It allows you view the Component hierarchy, view hooks, view and modify props, etc. With the profiler tab, we can view what re-rendered and when. Like many tools we use, the DevTools are very powerful, and we will only scratch the surface. But if you find yourself chasing a bug in your React code it can be a powerful tool.

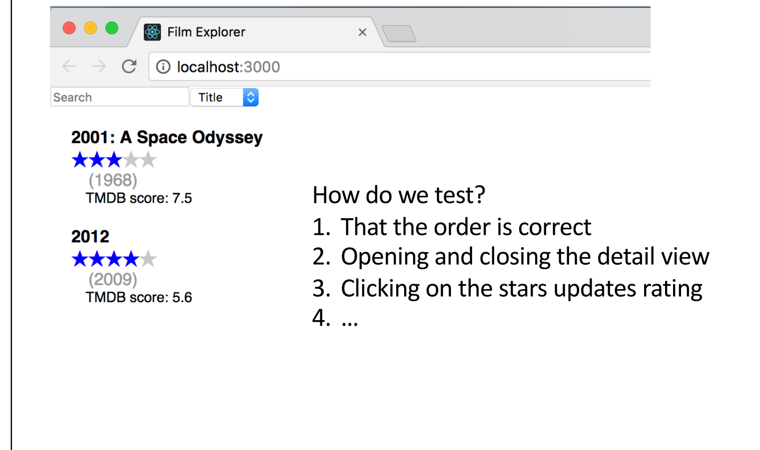
Testing a React application?

- 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

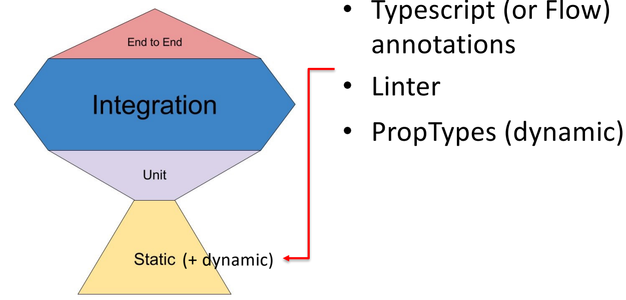
  A challenge is describing the relevant inputs and expected outputs



Here are some the properties we might want to verify in our UI… The latter two include both making assertions about what is shown in the browser and incorporating user interactions (i.e., our test involves “clicking”). How could we do so?

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 assertions about the DOM and 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.



As a quick reminder, we talk about tests in the context of this hierarchy. All these levels are still relevant to testing our React applications. But we also have some new static (or semi-static) tools.

- 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.

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 been largely displaced in favor of Typescript. As an aside, we considered introducing TS in the class, but decided against stuffing one more thing in an overful class. Instead, we will use PropTypes. PropTypes are more narrowly applied than either of the other; PropTypes allow us to specify the types of the props passed to

our React components and check those types during development. We will start there…

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

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://www.npmjs.com/package/prop-types

Here is an example of PropTypes for the slider in the ColorPicker. You see we are specifying the expected types for the props (and whether they are required). We will see warnings if props don’t match these specifications, [click]

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). That is, we can think of PropTypes as a form of “self-checking” documentation (i.e., it serves both the document the expected types and help enforce/check those types).

Note that PropTypes are less commonly used now in favor of Typescript. And as I mentioned we considered but decided not to use TS. Whether we use PropTypes or TS, part of the value is thinking through and clearly defining the expected types for our props. What do I mean by that? Do any of these types seem questionable to you?

Yes, the fact that value can be a string or a number, [click] That we (need to) allow two different types is a sign that we should probably revisit our design to make the implementation more consistent.

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>
 );
}
```

used like a function

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?

```
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>
 );
}


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

Optional object with at least name property

Required callback function

What are the full prop types for this component? <click> By reviewing this code, we can make inferences about the props and thus what types to specify. <click> Here we specify that record is an optional object and submit a required function. How did we make that inference?

- Record can be falsy and has a property name
- We set name with a string, suggesting that properties type
- Submit is used like a function and doesn't have a provision for being null/undefined.

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.

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.

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

What are the full prop types for this component? <click> By reviewing this code, we can make inferences about the props and thus what types to specify. <click> Here we specify that record is an optional object and submit a required function. How did we make that inference?

- Record can be falsy and has a property name
- We set name with a string, suggesting that properties type
- Submit is used like a function and doesn't have a provision for being null/undefined.

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.

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.

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 (i.e., the test can fail in many ways) but offer increased confidence the application works (because you are testing all the pieces). The ultimate mix of testing, and what “level” you call it doesn’t matter as much 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. TDD is equally applicable for testing our React applications as it was in testing "regular" JS code...

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

What do we need to test a React application?

1. Ability to 'render' components (and execute any hooks)
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

[after] The idea here is that tests should only perform user actions and only make assertions about content that is shown on the screen.

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 latter 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 a 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. If a component should be on the page, **get** is a good choice. The test will fail before you get to the assertion (but you will get a more helpful message showing what was actually present). If you are testing for non-existence, then use **query** as it doesn't throw an error if the query is not present. The **find** matcher is good for picking up on components that should appear based on some interaction, and particularly components that will appear some time in the future after an action/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. That can help decouple the test from the specific implementation, i.e., we care if there is something that can be used as a button, not whether it is specifically a button component.

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?

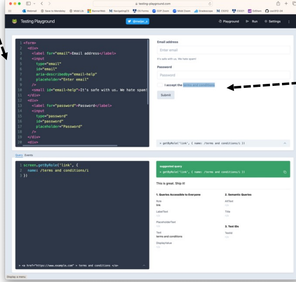
The link is to a very cool site that helps you define queries based on the DOM you have, i.e., you paste in HTML and it makes suggestions about the queries!

Figuring out the queries: An approach

1. Insert debug call to print DOM

```
test("…", () => {
    render(<Component … />);
    screen.debug(); // Print DOM to screen
})
```

2. Copy DOM into testing-playground.com



3. Pick element of interest

4. Suggested queries

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

Once we have the DOM element, we need to make an assertion. The testing library provides some custom matchers in the jest-dom package (an extension to Jest that adds matchers (assertions) relevant to the UI). In our previous example, we could use the toBeDisabled matcher to assert the button is disabled until we check the box to agree to the terms and conditions. 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…

We do that with…

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”-like behaviors.

Example from Simplepedia

1. Render the component (with mock function as prop)

2. Find the section

```
test('Clicking on a section displays titles', async () => {
    const selectFunction = jest.fn();
    render(<IndexBar collection={articles} setCurrentArticle={selectFunction} />);
    const section = await screen.findByText(sampleSections[0]);
    // Pre-condition assertions omitted…
    fireEvent.click(section);

    const titles = await screen.findAllByTestId('title');
    const expectedArticles = articles.filter(
      (article) => article.title.charAt(0).toUpperCase() === sampleSections[0]
    );
    expect(titles).toHaveLength(expectedArticles.length);
    expectedArticles.forEach((article) => {
      expect(screen.getByText(article.title)).toBeVisible();
    });
  });
```

3. “Click” on the section

4. Find all the titles

5. Assert expected titles are shown

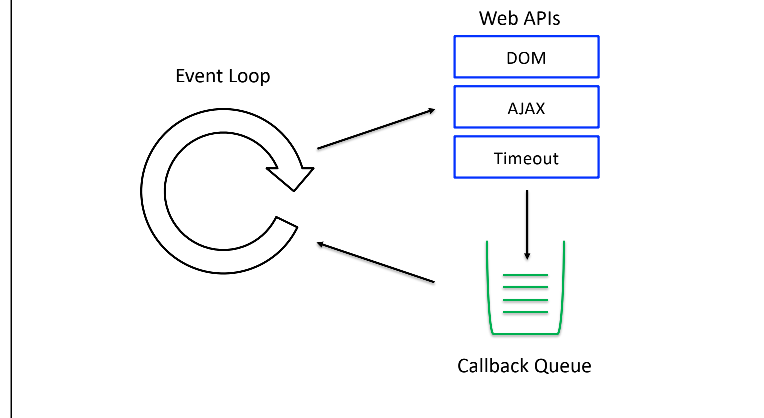
[at the end] What is async and await? These are tools for managing asynchronous computations, and particularly to enable an imperative style to working with Promises. Let’s talk about those more…

Example from Simplepedia

```
test('Clicking on a section displays titles', async () => {
    const selectFunction = jest.fn();
    render(<IndexBar collection={articles} setCurrentArticle={selectFunction} />);
    const section = await screen.findByText(sampleSections[0]);
    // Pre-condition assertions omitted…
    fireEvent.click(section);

    const titles = await screen.findAllByTestId('title');
    const expectedArticles = articles.filter(
      (article) => article.title.charAt(0).toUpperCase() === sampleSections[0]
    );
    expect(titles).toHaveLength(expectedArticles.length);
    expectedArticles.forEach((article) => {
      expect(screen.getByText(article.title)).toBeVisible();
    });
  });
```

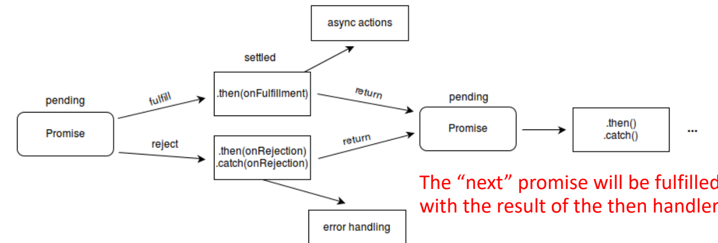
[at the end] What is async and await? These are tools for managing asynchronous computations, and particularly to enable an imperative style to working with Promises. Let's talk about those more…



As a reminder, asynchronous in this context means that actions may occur some indeterminate amount of time in the future, either because we are waiting on an external resource like the network, or we are waiting for the relevant callback to get executed by the event loop. In the case we want to use that value for a subsequent computation, how do we know when it is ”ready”?

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

A common action is to update state



MDN

One tool is a Promise.

A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate functions to be executed 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 (which isn’t yet known), the asynchronous method immediately returns a *promise* to supply the value at some point in the future. We can then pass that Promises around as needed.

We can think of a promise is being one of several states. Initially it is "pending", the value is not yet known. It can then fulfill with a value or reject due to an error. At that point, the relevant function is invoked (provided to `then` on success or provided to `catch` on failure).

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:

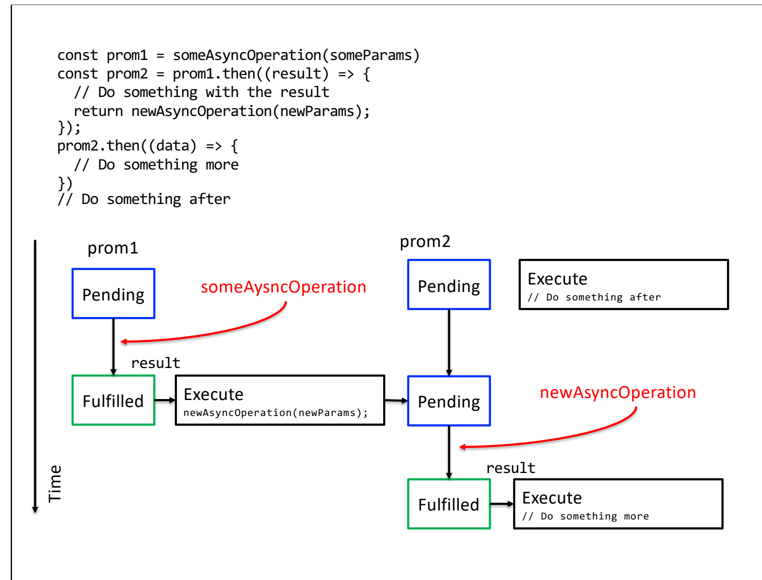
```
someAsyncOperation(someParams).then((result) => {
  // Do something with the result
  return newAsyncOperation(newParams);
}).then((result) => {
  // Do something more…
}).catch((error) => {  // Handle error});
```

The "traditional" way to handle asynchronous execution is via callbacks. We provide a function to execute when the result (or error) is available. If we want to perform a series of operations in sequence using results of previous asynchronous operations, we nest the callbacks.

[click]. Promises help us flatten a deeply nested set of callbacks into a linear chain of promises. someAsyncOperation returns a promise. We register a function with `then` to execute when it is fulfilled. What does `then` return? Also, a promise. Why? Recall, a promise is a proxy for a value not yet known. If the input to a `then` is not yet known, then its result must also not yet be known (and so on, down the chain). When that first value is fulfilled and the first handler is executed ("Do something with the result"), that second promise will be subsumed by the promise returned by newAsyncOperation. When it fulfills, ”Do something more” will execute. If at any point if there is an error, execution will skip the rest of the chain to execute the error handler.

Despite what it may seem like at the moment, the linear chain is much easier to reason about, especially in the case of errors (which can be picked up by single error handler at the end).

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



A Promise chain is a common source of confusion. Let's rewrite in into more discrete steps.

- prom1, prom2 are effectively defined immediately, that is someAsyncOperation and the `then` method return immediately with promises that will be resolved in the future.
- Thus, presumably before someAsyncOperation has completed, we start executing "Do something after"
- In the meantime, someAsyncOperation is executing. When it completes, the promise resolves with the result and we invoke the first `then` callback. It launches newAsyncOperation and returns a promise that will eventually resolve with its result. That newly returned promise subsumes the original prom2.
- When that second promise resolves we execute "// Do something more.

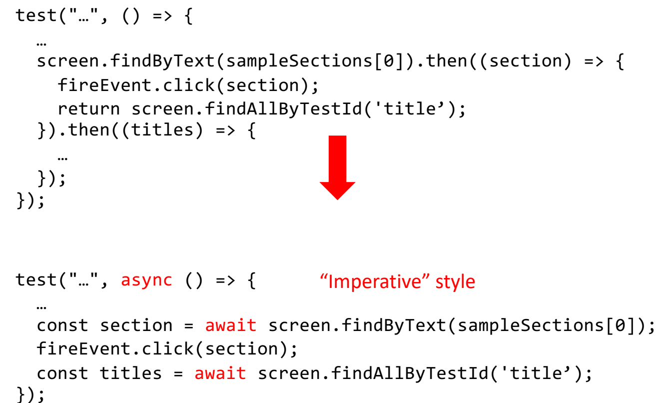
Assume the function `wait(sec)` returns a promise that resolves in `sec` seconds. What is the output of the following code?

```
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 don’t print Delay 2.



We noted that Promises "linearize" dependent actions. In that sense Promises seems more "imperative". In imperative code, the order of statements specifies the order of execution, i.e., each statement executes to completion before the next. The async and await keywords provide syntactic sugar for applying that style even more clearly to Promises. The "await" pauses execution until the promise returned by the await-ed expression has resolved, that is the Promise needs to have resolved before execution can proceed to the next statement (like imperative code). [the body of the then becomes the statements after await…]

We noted earlier that there is no way to "stop" a Promise chain and switch back to synchronous imperative code. That is true and still true (regardless of how it may appear). We should remember async/await are just syntactic sugar over Promises (i.e., async/await can be directly translated back to "raw" Promises) and that execution is still fundamentally asynchronous.

To that end, what is the return value of an async function? Always a Promise. Even if the returned value is not explicitly a Promise, i.e., the function returns a string, it will implicitly be wrapped in a Promise. Why? Because the operations are still fundamentally asynchronous, and we don't when in the future the function body will complete.

I suspect this is a 🤯 moment. I am with you. We will revisit Promises several times

and learn about different asynchronous operations. Our goal today is to introduce some of the tools and techniques we will need for testing out React applications.

Why async/await in our tests? The UI might not change immediately

The action may not immediately change the UI (need to update state, then re-render)

```
test('Clicking on a section displays titles', async () => {
    const selectFunction = jest.fn();
    render(<IndexBar collection={articles} setCurrentArticle={selectFunction} />);
    const section = await screen.findByText(sampleSections[0]);
    // Pre-condition assertions omitted…
    fireEvent.click(section);

    const titles = await screen.findAllByTestId('title');
    const expectedArticles = articles.filter(
      (article) => article.title.charAt(0).toUpperCase() === sampleSections[0]
    );
    expect(titles).toHaveLength(expectedArticles.length);
    expectedArticles.forEach((article) => {
      expect(screen.getByText(article.title)).toBeVisible();
    });
  });
```

“Pause” test to wait till the expected components appear in the DOM

Behind the scenes the “find(All)By” is a wrapper around another function, waitFor. It is repeatedly re-running the query, in this case for elements with a specific test-id, until either it succeeds, or timeouts. That way we can test for elements, in this case like the titles, that may not appear immediately due to asynchronous implementation (maybe it needs to fetch the relevant data).