Introducing React

- Can feel daunting
- Benefits aren't immediate
    - But are real and helpful!
- Tried to use similar patterns

React makes render easier

- Closer to HTML
- Like how render() could embed other functions
    - No more functions in strings in functions

Common Points of Confusion

- Importing React
- Folder Structure
- State immutable
    - During Component function
- State Management
- Form Issues

No need to import React

- Not sure where this came from
    - I didn't show it!
    - A tutorial people looked up?
        - Tell me what I'm missing!
    - VSCode auto fill-in?

We are running React 18

- React 17 removed `import React from 'react';`
    - Unless you use `React.XXX` somewhere

Folder/Directory Structure

- I saw lots of `/components`, `/pages`, `/helpers`, etc
  - Not uncommon
    - Not the only way to do it
  - At this stage of learning
    - Don't overcomplicate
    - So much `../../`!
  - Personally not a fan of `/components`
    - Doesn't add value
    - When editing a feature, how many directories involved?

State is Constant

```
function Component() {
  const names = [ 'Jorts', 'Jean', 'Nyan', 'Floofa' ];
  const [name, setName] = useState('');

  return (
    <button
      onClick={ () => {
        // name = "bob"  // Bad! No longer state!
        setName(
          names[Math.floor(Math.random*names.length)]
        );
        console.log( `name after click is ${name}` ); // Old!
      }
    >{name}</button>
  );
}
```

Key Lessons

- State should be IMMUTABLE
    - automatic with const primatives
    - NOT automatic with object/array
- Actual state is outside your function
    - useState() gives you a COPY
    - Calling the setter updates real state
        - Does not alter copy until next render
        - If you need altered state now
            - You have value you set it to!

Why didn't my state update?

Very Common confusion

- Misleading
- State did update
    - Just checking a stale copy

Calling setter queues a render

Render = component function will be called

- useState() will return NEW state

This is NOT so different than vanilla JS

- We never updated state variables in render
- We did update state variables in listeners
    - Rarely used new values before render
- React has explicit state variables and setters

state can "sprawl" and couple

```
function App() {
  const [username, setUsername] = useState('');
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  const [error, setError] = useState('');

  return (
    <div>
      { isLoggedIn && <p>Stuff here</p>}
      { !isLoggedIn && <Login
          username={username}
          setUsername={setUsername}
          setIsLoggedIn={setIsLoggedIn}
          error={error}
          setError={setError}
      />}
    </div>
  );
}
```

What's undesireable there?

- Login is highly coupled to App state
- App has state it doesn't actually use
- Passing a log of variables feels tedious/heavy
    - Easy to pass too much/miss one

Separate state

- What state ISN'T a top level state
    - State should be at lowest common ancestor
- What state is actually different?
    - App username = logged in username
    - Login username = username as typing

Separated States

```
function App() {
  const [username, setUsername] = useState(''); // Logged in
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  return (
    <div>
      { isLoggedIn && <p>Stuff here</p>}
      { !isLoggedIn && <Login
          setUsername={setUsername}
          setIsLoggedIn={setIsLoggedIn}
      />}
    </div>
  );
}
```

```
function Login({ setUsername, setIsLoggedIn }) {
  const [tempUsername, setTempUsername] = useState('');
  const [error, setError] = useState(''); // only used here
  // ...
}
```

Encapsulate changes

- Login has a few "actions"
    - Typing
        - Could be handled inside Component
    - Report error
        - Might be inside Component?
        - Depends which Component reports
    - Login
        - Sets App `username`
        - Sets App `isLoggedIn`

Passing actions reduces coupling

```
function App() {
  const [username, setUsername] = useState(''); // Logged in
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  function onLogin (username) {
    setUsername(username); // username is the one passed in!
    setIsLoggedIn(true);
  }

  return (
    <div>
      { isLoggedIn && <p>Stuff here</p>}
      { !isLoggedIn && <Login onLogin={onLogin} /> }
    </div>
  );
}
```

```
function Login({ onLogin }) {
  const [tempUsername, setTempUsername] = useState('');
  const [error, setError] = useState(''); // only used here
  // ...
}
```

Decoupled

- Login no longer gets ANY state from App
- Login no longer gets ANY setters from App
- Login just gets an action function
    - App knows little of how Login works
        - Just the rules for `onLogin()`
    - App can change considerably
        - No/few changes needed to Login

Can't always get complete separation/decoupling

- Reduced coupling is still better

Derived State

- Values based solely on state and constants
    - **derived state**
- Tempting to add to state!
    - But problematic!
    - What is the source of truth?
    - Need to remember to redetermine
        - Every time origin state changes
- Instead calculate every render

Example Derived State

Bad Temporary Username Message

- IF Login has username-as-typed state
    - Separate from App username-logged-in state

```
import { validateUsername } from './validations';

function Login({ onLogin }) {
  const [username, setUsername] = useState(''); // as typed
  // REMOVED error message state!
  // const [error, setError] = useState('');
  const error = validateUsername(username); // Still renders
  // No more setError(...) anywhere

  // ...
}
```

Derived State Notes

- NOT Derived State when
    - Based on more than state and constants
- If calculation is "expensive"
    - Still calculate vs storing in state
    - "memoize" function return
- May need new state values
    - Such as "have they ever typed?"
        - To avoid immediate error message
        - Reduced complexity of managing state
            - Usually worth extra state values

State Can Be/Have Collections

```
function App() {
  const [userInfo, setUserInfo] = useState({
    username: '',
    isLoggedIn: false,
  });

  function onLogin (username) {
    setUserInfo({
      username,
      isLoggedIn: true;
    });
  }

  return (
    <div>
      { userInfo.isLoggedIn && <p>Stuff here</p>}
      { !userInfo.isLoggedIn && <Login onLogin={onLogin} /> }
    </div>
  );
}
```

`Login.jsx` has no changes!

When to Have State as Collection

Pros:

- Easy to pass if you have related state fields
- Easier to see all of "state"
    - Easier to know which variables are "state"

Cons:

- Remember no mutation of state objects!
    - More involved to change only part of state

Next week: More options for state management

Forms Involve Common Mistakes

```
function Form() {
  const [name, setName] = useState('');
  const [tempName, setTempName] = useState('');
  return (
    <form>
      <p>Hello {name}</p>
      <label>New Name:
        <input
          value={tempName}
          onChange={ e => {
            setTempName(e.target.value);
          }}
        />
      </label>
      <button
        onClick={ () => {
          setTempName('');
          setName(tempName);
        }}
      >Replace</button>
    </form>
  );
}
```

Why does the page reset?

- Check the URL
    - The page is reloading
    - We navigated on form submit
        - `<form>` defaults to same page
        - button defaults to type 'submit'
- We need to stop the form from submitting
    - We could set the button to `type="button"`...

Changing the button to not submit

```
function Form() {
  const [name, setName] = useState('');
  const [tempName, setTempName] = useState('');
  return (
    <form>
      <p>Hello {name}</p>
      <label>New Name:
        <input value={tempName}
          onChange={ e => {
            setTempName(e.target.value);
          }}
        />
      </label>
      <button
        type="button"
        onClick={ () => {
          setTempName('');
          setName(tempName);
        }}
      >Replace</button>
    </form>
  );
}
```

That works...until

- Button click no longer submits
- But "Enter" in sole input field still submits

We can put `onSubmit` on form to `e.preventDefault`

- A better option is to do everything onSubmit
  - Button is type "submit"
  - Enter OR click will trigger submit event
  - `onSubmit` stops actual navigation
  - `onSubmit` processes instead of `onClick`

onSubmit version

```
function Form() {
  const [name, setName] = useState('');
  const [tempName, setTempName] = useState('');
  return (
    <form
      onSubmit={ e => {
        e.preventDefault();
        setTempName('');
        setName(tempName);
      }}
      >
      <p>Hello {name}</p>
      <label>New Name:
        <input value={tempName}
          onChange={ e => {
            setTempName(e.target.value);
          }}
        />
      </label>
      <button type="submit" >Replace</button>
    </form>
  );
}
```

Key Submit Lessons

- These were HTML issues, not React!
- Navigation resets page state
    - Can look like state changes!
        - Waste time working wrong problem
- UX important
    - Consider all interactions
        - Ex: Enter vs Click, keyboard vs mouse
- Not all interactions are the same
    - Forms for data submit
    - Buttons controls for state
    - Small forms vs big forms

Labels and `for`

- HTML `<label for="">` attribute
  - Needs to be `htmlFor` prop in JSX
  - Same reason as `className`
    - `for` is reserved word in JS
    - `htmlFor` is the DOM Node property
- Not needed if `<label>` around labeled element
  - Then browser automatically knows
- **Value must be id of labeled element**
  - not `name`, `class`, or `className`
- Labels are important (a11y)
  - But only count if you get them right