Service Calls in React

- The `fetch()` call is same as without react
- React has `onXXX` event listeners
    - So we will make our calls there
- React has state setter functions
    - So we will use results and call setters
- React automatically re-renders on state change

Service Functions

I recommend the same structure for a `services.js`

- Functions that call fetch
- Translate results/errors
- Return a promise

No changes for React!

- Still a `.js` (No JSX)
- Entire `services.js` file = no changes
    - Separation of Concerns paying off!

Importing service functions

Just one way to do it

- but I recommend

import function as a **named import**

```
import { fetchTodos } from './services'
```

A `service.js` NOT a `.jsx` file

- Is plain JS, no JSX
- Same file as vanilla JS!

Calling services in React similar to vanilla JS

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

  function onSubmit(e) {
    e.preventDefault();
    fetchLogin(username)
      .then( session => {
        setIsLoggedIn(true);
        // Do something with session data
      })
      .catch( err => {
        // Do something with error state
      });
  }
  return ( // output simple for demo purposes
    <form onSubmit={onSubmit}>
      <input value={username}
        onChange={e => setUsername(e.target.value)}/>
      <button type="submit">Login</button>
    </form>
  );
}
```

What did we do there?

- Imported a function that calls `fetch()`
    - and returns promise of results
- Called `fetch` function in reaction to an event
- Updated state
    - React will automatically re-render

What was not in that example?

- Showing the new state in rendering
- Had to fit it on screen
- See the `samples/services-react-todo`

Organizing your code

With what we know so far:

- "top level components"
    - Hold application-wide state
    - Define functions that change that state
    - Pass state and functions to children
- Descendant ("lower") components
    - Hold their own temporary state
    - Use passed props and functions
        - WARNING: Don't set state from props
            - What do you do if props change?

State and State-Managing Functions get "heavy"

- Ex: A lot of code in `App.jsx`
- Resetting some state gets repetative
    - Ex: Always resetting error state on success
    - Feels "wrong" to need to remember so much

We will cover options to clean this up soon

- Focus on distinct state management functions
- Keep child components from getting too complex
    - They shouldn't need to know overall state
- Simplifying top level components comes later

Revised Login Example (still cramped for space)

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

  function onLogin(username) {
    fetchLogin(username)
      .then( session => {
        setIsLoggedIn(true);
        // Do something with session data
      })
      .catch( err => {
        // Do something with error state
      });
  }
  return ( // output simple for demo purposes
    <div>
      { !isLoggedIn && <Login onLogin={onLogin}/>}
      { isLoggedIn && <Content username={username} />}
    </div>
  );
}
```

Don't Forget Spinners (the biggest lie in webdev)

Also just like vanilla JS

- State says: show or not show spinner
- Set state to show spinner before service call
- Set state to not show spinner after call

React should not make **everything** different

- An easier way to render based on state

Your "Spinner" Can Be...

- Text
- Pure CSS
    - Such as found on **https://css.gg/**
- Image
- Image/Text + CSS

React has "Suspense" and "Transitions"

If you have a lot of loading situations

- React has added features to handle these
- But their use is finicky
- Generally intended for other libraries to use
- Not covered in this class
- But good for you to know

What state does a component assume?

If your component might render before data loads

- What do you show?
- How do you know if you are waiting?

This is **design**

- Is component responsible for loading indicator?
    - Is parent?
- What type are the props?
    - Ex: Should a value always be an array?
        - Empty/full? Never null/undefined?
- There is no universal "right" answer

Summary - Service calls like vanilla JS

- `fetch()` calls are best in outside .js files
    - Not in components
- Components should handle results
    - Success AND reporting errors
- UI for Error handling can be involved
    - Just like for vanilla JS

Summary - Service Call Results

- Define functions
  - Call `fetch()` wrapper function
  - Update state on success/error
- Pass state + function as props
- Cleaning state management to come later

Summary - Loading similar to vanilla JS

- Set state to indicate loading before service call
- Start async process/service call
- Set state to stop loading indicator after
- Suspense and Transitions available outside of this course
    - Group and optimize loading situations
    - Typically involve additional libraries