Service functions

Service calls can be raw fetch() calls

- but I've shown a nice pattern of a service function
- returns a promise
- resolves with content
- or rejects with error

Logic vs Presentation

All of the fetch() logic isn't presentation

- Does not belong in JSX component
- handling the results CAN belong

Conclusion:

- don't call fetch() in component
- do call service function in component
- set state to reflect data/error

Importing service functions

Just one way to do it

- but I recommend

import function as a **named import**

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

Benefits

- Service logic off in a dedicated bit of code
- Only "presentation logic" in presentation
    - What to show
    - If to show it
- Service functions can be used anywhere in your code
    - reusable

Calling the functions

- Call in `useEffect` callback
- set state to reflect data and if any errors

Call demonstration

```
function App() {
  const [todos, setTodos] = useState({});
  useEffect( () => {
    fetchTodos()
    .then( results => {
      setTodos(results);
    })
    .catch( err => {
      // FIXME
    })
    },
    [setTodos]
  );

  return (
    <div className="app">
      <TodoList
        todos={todos}
      />
    </div>
  );
}
```

What happens

- First: `todos` state is set to default with `useState({})`
- Second: `useEffect()` callback is queued to run
- Third: App() renders `<TodoList>` (default todos)
- Fourth: `useEffect()` callback runs
    - puts `.then()` callback in queue
- Fifth: `.then()` callback runs
    - calls `setTodos()`
        - which queues a re-render of App()
- Sixth: App() rerenders, now with fetched `todos`
    - useEffect() does nothing as `setTodos` is same

What if error happens?

What do you want to show in this case?

- What component shows that?
- Based on what state? useEffect callback()
- should catch the error
- update state
    - You have to make the setters available
- can you redo fetch call if error handled?
    - easier to have same call outside of useEffect
    - Based on user action

Vital Lesson here!

Error handling involves a lot of work

- Often more than "happy path"
- Especially in presentation
- Don't delay it too long
- Don't expect it to be quick
    - rushed code is often bad code

Spinners are the biggest lie in webdev

- showing an animated image
- not a sign the computer is "thinking" or waiting
    - we code a sequence
        - show image
        - do thing
        - remove image
    - if we made mistake it will just lie

A Useful Lie

If we don't mess up, the spinner gives useful info

- "You should wait"
- "Don't click more buttons"
- "Don't refresh the page"

Finding a Spinner image

- See upcoming Licensing talk!
- Can be text-based
- Can be pure CSS
- Can be image + CSS
- Can be animated image

Combining the parts

In useEffect:

- show loading state
- make fetch call
- remove loading state

In JSX:

- If loading
    - show loading indicator
- If not loading
    - show content

Remember useEffect is AFTER first render!

Demo of Loading

```
const [todos, setTodos] = useState({});
const [isLoading, setIsLoading] = useState(true);

useEffect( () => {
    setIsLoading(true);
    fetchTodos()
    .then( results => {
      setTodos(results);
    })
    .finally( () => {
      setIsLoading(false);
    });
  },
  [setTodos, setIsLoading]
);
return (
  <div className="app">
    { isLoading && <div>Loading...</div> }
    { !isLoading && <div>{todos}</div> }
  </div>
);
```

Summary - Service calls

- 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

Summary - useEffect

useEffect() to load data

- happens AFTER first render
- updates state for success or error
- should indicate a loading status somehow
- should avoid infinite loops
    - from state updates
- May need to re-trigger loading if it failed

Showing Loading status

- Needs to show status starting BEFORE fetch call
- remove after fetch call
    - success OR error