Working with Data

Web Design 2

23 Feb 24

React Hooks

- Hooks are utility functions that are part of React
- Hooks exposes APIs to local state, lifecycle events, reference to JSX elements etc
- Hooks follow a naming pattern. All hooks are prefixed with `use{hook_name}`. E.g. useState, useRef, useEffect etc

Local State

With useState

- State is an object to store values in the component
- Components (and all children components) are automatically re-rendered whenever the state changes
- In React, we can store state data with useState hook

Local State

With useState

```
function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Local State

With useState

```
function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Initial value

Getter

Get the state value

Setter

Set/update the state value

Updating state

Using the Setter function on event listener

Local State

With useState

```
export default function App() {
  const [count, setCount] = useState(0)
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}
```

Updating Previous Value

With useState

```
const [counter, setCounter] = useState(0);

const incrementHandler = () => {
  setCounter(counter + 1);
}
```

```
const [counter, setCounter] = useState(0);

const incrementHandler = () => {
  setCounter((prevValue) => prevValue + 1);
}
```

State Setter function accepts a value <u>or</u> a function that returns a value

Exercise

- Using React
  - Create a Counter app
  - Using a Button component, create an Increment and Decrement buttons that pass the state up the tree
  - Conditionally update an image/text on screen based on the state value

Asynchronous JavaScript

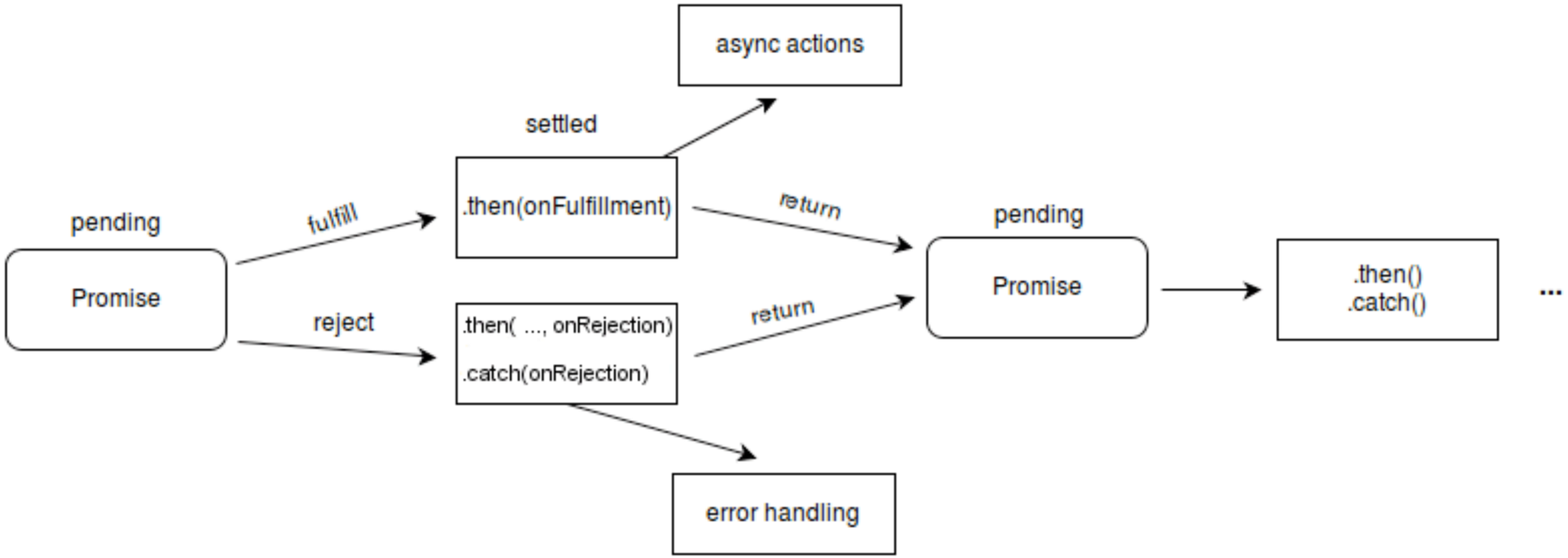
- By default, JavaScript is a synchronous, single threaded programming language. This means that instructions can only run one after another, and not in parallel.
- Asynchronous means that things can happen independently of the main program flow
- For example, functions like fetching a remote API or accessing user's camera can potentially take a longer time to execute. Which is where asynchronous programming is helpful.

Asynchronous JavaScript

Promises

- Promise is a JavaScript object that represents the evaluation state of an asynchronous function
- A Promise object can be
  - Pending // Still working. Result is undefined
  - Fulfilled // Done evaluating. Result is an error
  - Rejected // Done evaluating. Result is an error

Promises



Source: MDN

Writing Async JavaScript

- Functions prefixed with the async keyword turn into asynchronous functions and return a Promise object

```
async function getMyData(param) {
  // ... some evaluation code ...
  return myValue;
}
```

Writing Async JavaScript

Resolving Promise Objects with .then

```
1  async function f() {
2    return 1;
3  }
4
5  f().then(alert); // 1
```

Writing Async JavaScript

Resolving Promise Objects with async-await

```javascript
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

Fetch API

- Fetch API is an interface for fetching resources from a server
- Fetch returns a Promise object that can be resolved into a Response object

The syntax

```
let myPromise = fetch(url)
```

Fetch API

Requesting Remote APIs with fetch

```
1  async function showAvatar() {
2
3    // read our JSON
4    let response = await fetch('/article/promise-chaining/user.json');
5    let user = await response.json();
6
```

Fetch API

Requesting Remote APIs with fetch

```
1  async function showAvatar() {
2
3    // read our JSON
4    let response = await fetch('/article/promise-chaining/user.json');
5    let user = await response.json();
6
```

Fetch API

- The default HTTP method for fetch API is GET
- The API also accepts POST, PUT, DELETE etc methods in the options param
- Authorised requests are made using the headers object and passing in the API token/access_key (but may vary depending on the API)
- Always go through the APIs documentation to make sure how they expect the requests

Fetch API

Complete list of optional options object

```
let promise = fetch(url, {
  method: "GET", // POST, PUT, DELETE, etc.
  headers: {
    // the content type header value is usually auto-set
    // depending on the request body
    "Content-Type": "text/plain;charset=UTF-8"
  },
  body: undefined, // string, FormData, Blob, BufferSource, or URLSearchPara
  referrer: "about:client", // or "" to send no Referer header,
  // or an url from the current origin
  referrerPolicy: "strict-origin-when-cross-origin", // no-referrer-when-dow
  mode: "cors", // same-origin, no-cors
  credentials: "same-origin", // omit, include
  cache: "default", // no-store, reload, no-cache, force-cache, or only-if-c
  redirect: "follow", // manual, error
  integrity: "", // a hash, like "sha256-abcdef1234567890"
  keepalive: false, // true
  signal: undefined, // AbortController to abort request
  window: window // null
});
```

Source: javascript.info

Fetch API

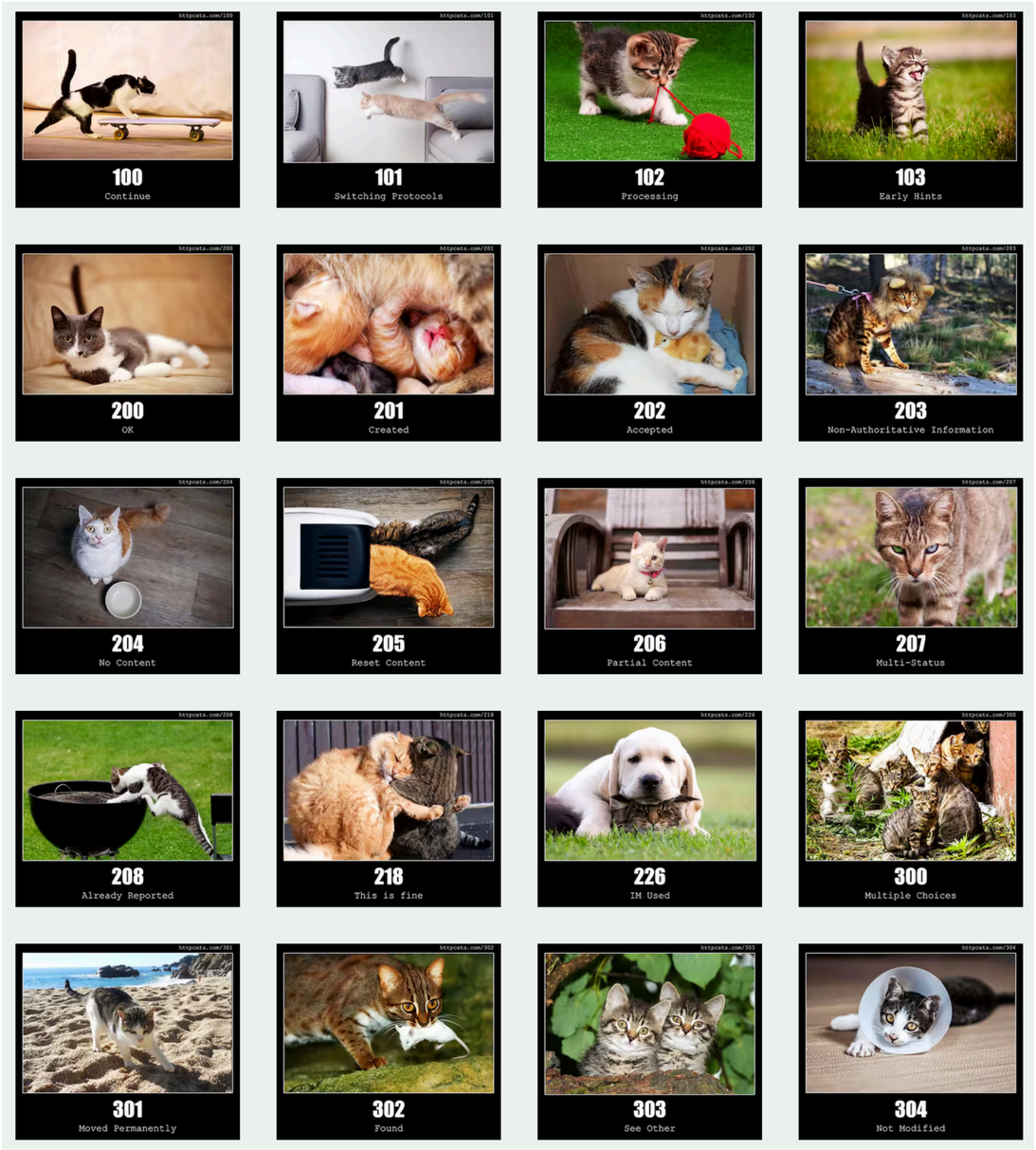
HTTP Status Codes

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirection messages (300 – 399)
- Client error responses (400 – 499)
- Server error responses (500 – 599)

Fetch API

HTTP Status Codes

httpcats.com



Error Handling

With try…catch

- Errors in code are common, and they can occur because of various reasons. E.g., server down, error in the URL, unauthorised requests etc (refer: HTTP Status Codes)

- Try…catch construct allows ‘catching’ of errors, so they can be handled without breaking the program

Error Handling

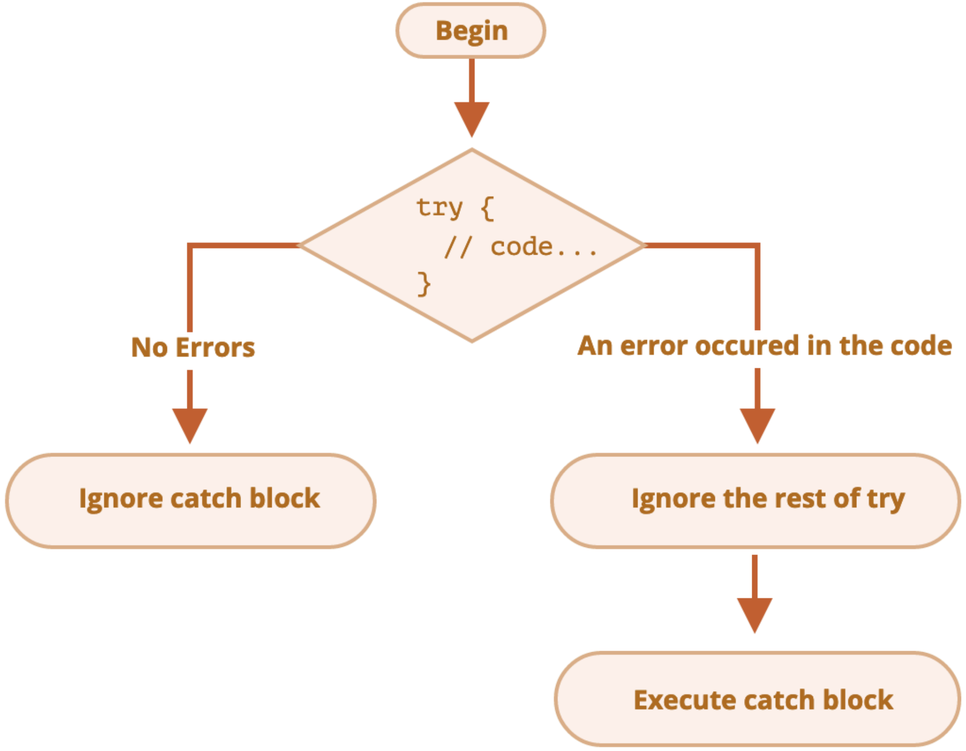
With try…catch

```
try {

  // code...

} catch (err) {

  // error handling

}
```

Source: javascript.info

Error Handling

With try…catch



Source: javascript.info

Error Handling

With try…catch

```
try {
  const response = await fetch('https://website');
} catch (error) {
  console.log('There was an error', error);
}
```

Error Handling

With try…catch

```
try {
  const response = await fetch('https://website');
} catch (error) {
  console.log('There was an error', error);
}
```

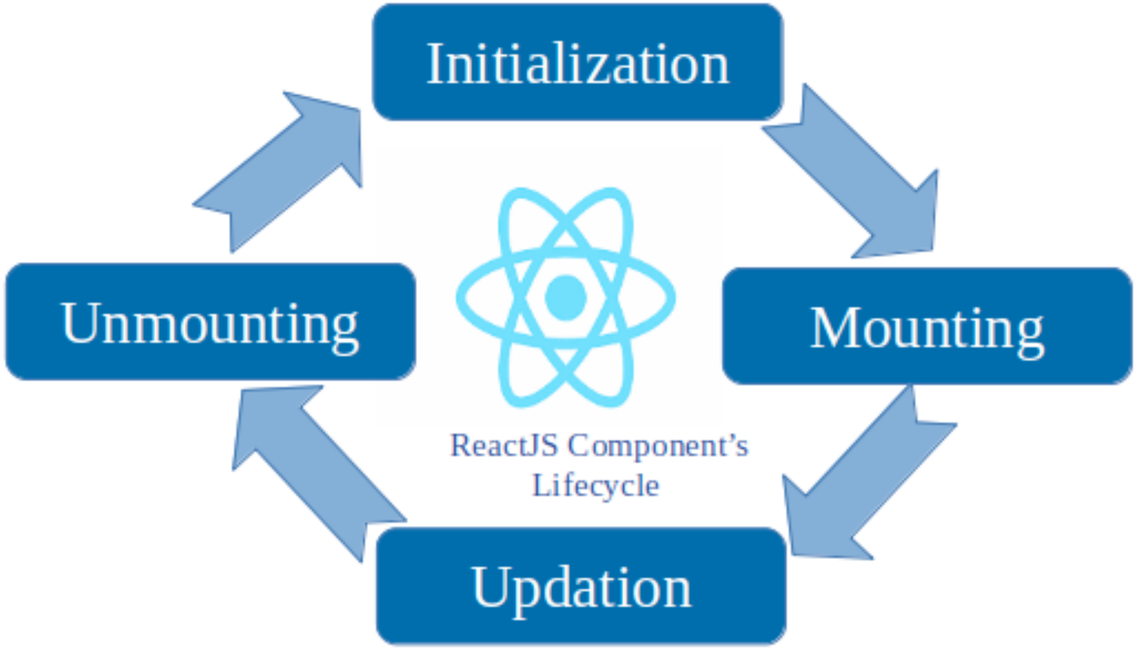
Logs out if there is any error within the try block

Combining with ReactJS

Component Lifecycles

- Components in React (and all other UI frameworks) have a lifecycle process which can be categorised into three main phases
  - Mounting
  - Unmounting
  - Updating

Component Lifecycles



Source: freecodecamp

useEffect hook in React

Running side-effects in the code

- The useEffect hook in React allows performing side-effects in React components
- A ‘side-effect’ is anything that affects something outside the scope of the component. Example, a network request, timeOut functions etc

useEffect hook in React

Running side-effects in the code

- useEffect hook accepts two arguments

```
useEffect(callback[, dependencies]);
```

`callback` is a function that contains the side-effect logic. `callback` is executed right after the DOM update.

`dependencies` is an <u>optional</u> array of dependencies. `useEffect()` executes `callback` only if the dependencies have changed between renderings.

useEffect hook in React

Managing Dependencies

- The dependencies array object in the useEffect hook can be run in three ways that ties into working with the component lifecycle

1. No dependency passed

```
useEffect(() => {
  // Runs on every render
});
```

Runs every time the component is rerendered

2. An empty array

```
useEffect(() => {
  //Runs only on the first render
}, []);
```

Runs the first time

3. Dependency values

```
useEffect(() => {
  //Runs on the first render
  //And any time any dependency value changes
}, [prop, state]);
```

Runs every time the dependency values change (prop, state)

useEffect + fetch

- The callback function argument can be turned into asynchronous function by defining an async function and calling in within useEffect hook
- Passing an empty array as the dependency argument because we want the data to be loaded once <u>before</u> mounting the component

Other Network Request Libraries

Axios

- Axios is a popular library for working with network requests
- Docs available at https://axios-http.com/docs

```
import React, { useEffect, useState } from "react";
import axios from "axios";

export default function App() {
  const [data, setData] = useState([]);  const getData = async () => {
    const { data } = await axios.get(`https://yesno.wtf/api`);
    setData(data);
  };

  useEffect(() => {
    getData();
  }, []);

  return <div>{JSON.stringify(data)}</div>;
}
```

Other Network Request Libraries

SWR

- SWR (Stale While Revalidate) is a HTTP data fetching library that also allows caching, revalidation, along with built-in error-handling functions
- SWR requires a fetcher function that can be built using either the default fetch API, or axios etc
- Docs available at https://swr.vercel.app/

```
import useSWR from 'swr'
import Pokemon from './Pokemon'

const url = 'https://pokeapi.co/api/v2/pokemon'

const fetcher = (...args) => fetch(...args).then((res) => res.json())

export default function App() {
    const { data, error } = useSWR(url, fetcher)

    if (error) return <h1>Something went wrong!</h1>
    if (!data) return <h1>Loading...</h1>

    return (
        <main className='App'>
            <h1>Pokedex</h1>
            <div>
                {result.results.map((pokemon) => (
                    <Pokemon key={pokemon.name} pokemon={pokemon} />
                ))}
            </div>
        </main>
    )
}
```

Lab Exercise

Using JSON Placeholder API -

Design a Blog with reusable components with ReactJS