# React.3

▼ When is a promise used?

A Promise is used whenever some piece of code completes in the future.

▼ What is resolving a promise?

Resolving a promise means performing an action when the promise has completed its work.

▼ How to resolve a promise?

To resolve a Promise, you need to **call .then() on it and pass a callback function**. That callback function will be called when the promise has completed its work successfully.

```
functionThatReturnsPromise().then(result => {
    console.log(result);
});
```

▼ When a promise fails, how do you catch errors?

catch(callback)

```
functionThatReturnsPromise()
.then(result => {
    console.log(result); // when successful
})
.catch(error => {
    console.error(error); // when there's an error
})
```

▼ What is the fetch API?

a browser API (meaning that it's a functionality available in the browser), that lets you make a network request to a server

▼ What is the most common use case of the fetch API?

when building (web) apps, send and receive data from the backend/API

▼ What does the fetch API return?

a promise which is why we have to resolve it

▼ fetch API GET example

1. the `fetch()` call returns a promise, which is why we resolve it with `.then()`.
2. the `response` we get back from `fetch` is a generic response, which is why we want to convert it to a JSON object by calling `response.json()`.
3. `response.json()` also returns a promise, which is why we need to resolve it **again** with `.then()`.

```
fetch(URL)
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

- ▼ Why use fetch API with React?

  use the fetch API inside a React component and create a state variable out of the data that we get back from the backend/API

  If you forget to specify the effect dependency, and you have a setState call inside the effect, then you will end up with an infinite loop. Why? setState initiates component re-render.

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

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://website.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data); // keep it for debugging
            // only call setState here if there is a dependencies array as a control else get infinite loop
            setUsers(data);
        });
    }, []);

    return null;
}
```

- ▼ Why is calling fetch inside a component is considered a side effect?

  because fetch goes to the network (**outside of your component**), so it must be inside an effect

- ▼ How often do you want to call an effect that makes a fetch API call?

  most of the time it will be just once when the component mounts aka 1 call

```
useEffect(() => {
        fetch("https://website.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
        });
    }, []);
```

- ▼ How to handle error: cannot read property X of undefined?

  Add an if condition and return a different JSX (conditional rendering) to avoid this issue.

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

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://react-tutorial-demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
            setUsers(data);
        });
    }, []);

    if (!users) {
        return null; // return nothing
    }

    return <h1>There are {users.length} users</h1>
```

- ▼ How can you return nothing from JSX?

  return null

- ▼ How to handle rendering with JSX when data is undefined using logical && operator?

```
return <>
        <h1>Users</h1>
        <ul>
            {users && users.map(user => <li key={user.id}>{user.name}</li>)}
        </ul>
  </>
```

the expression users && users.map() will **short circuit** at users because users returns undefined

so before users has loaded, the App component will render:

```
<>
    <h1>Users</h1>
    <ul>
    </ul>
</>
```

▼ HTTP status codes 2xx

A status code of 200 (or any number between 200 and 299, often written as 2xx) is considered successful.

▼ HTTP status codes 4xx (client) and 5xx (server)

responses in the range of 400 to 499 (4xx) as well as 500 to 599 (5xx) mean that the fetch request did not complete successfully

▼ How to avoid setting state on an error object, assuming that the API returns an error object when things don't go as expected?

```
.then(data => {
    // only set the users when there's no error
    if (data && !data.error) {
        setUsers(data);
    }
});
```

▼ When do network errors happen?

when the connection breaks or when the user is offline

▼ How to handle network errors in fetch requests?

catch()

```
fetch("https://website.com/users.json")
.then(response => response.json())
.then(data => {
    console.log(data);
})
.catch(error => {
    console.log(error); // or console.error(error)
});
```

▼ How to handle showing a loading icon when fetching data from an API?

```
// index.js

import React, {useEffect, useState} from "react";
import {render} from "react-dom";
import Loader from './Loader.js';

function App() {
    const [users, setUsers] = useState();
    const [isLoading, setIsLoading] = useState(true);

    useEffect(() => {
        fetch("https://website.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
            if (data) {
                setUsers(data);
            }
        })
        .catch(error => {
            console.error(error);
        })
        .finally(() => {
            setIsLoading(false); // stop the loader
        });
    }, []);

    return <>
        {isLoading && <Loader />}
        {users && <h1>Users</h1>}
        <ul>
            {users && users.map(user => <li key={user.id}>{user.name}</li>)}
        </ul>
    </>;
}

render (<App />, document.querySelector("#react-root"));
```

```
// Loader.js

import React from "react";

export default function Loader() {
    return <svg className="spinner" width="65px" height="65px" viewBox="0 0 66 66" xmlns="http://www.w3.org/2000/svg">
   <circle className="path" fill="none" strokeWidth="6" strokeLinecap="round" cx="33" cy="33" r="30"></circle>
</svg>;
}
```

▼ Why disable a button while loading?

This is common practice to avoid the user from repeatedly clicking on a button that has already started a fetch request

```
<button disabled={isLoading}>Click me</button>
```

▼ If you need to run a fetch call based on a state value that can change, it's best to run the fetch request inside what?

a useEffect with a dependency on that state variable

```
import React, {useState, useEffect} from "react";
import {render} from "react-dom";

function CurrencyConversion() {
    const [currency, setCurrency] = useState("");
    const [rate, setRate] = useState("");
    const [isLoading, setIsLoading] = useState(false);

    useEffect(() => {
        if (currency) {   // fetch call won't worry if no currency
            setIsLoading(true);
            fetch(`https://website.com/currencies/${currency}.json`)
            .then(response => response.json())
            .then(data => {
                if (data) {
                    setRate(data);
                }
            })
            .catch(error => console.log(error))
            .finally(() => {
                setIsLoading(false);
            });
        }
    }, [currency]);

    function handleCurrencyChange(event) {
        setCurrency(event.target.value);
    }

    return <>
        <h2>Currency rates</h2>
        <select onChange={handleCurrencyChange} disabled={isLoading}>
            <option value="">Select a currency</option>
            <option value="usd">USD</option>
            <option value="eur">EUR</option>
            <option value="cad">CAD</option>
        </select>
        <h3>1 {currency.toUpperCase()} = {rate} USD</h3>
    </>;
}

render(<CurrencyConversion />, document.querySelector("#react-root"));
```

▼ What is the most common cause of stack overflow error?

Deep or infinite recursion, in which a function calls itself so many times that the space needed to store the variables and information associated with each call is more than can fit on the stack.

▼ How to use async/await (syntactic sugar on top of promises) when using useEffect? (example)

To use await, we need to be inside an async function.

We add an async immediately invoked function expression inside useEffect.

Use try/catch to handle errors

```
function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        (async () => {
          try {
            const response = await fetch("https://website.com/users.json")
            const data = await response.json()
            setUsers(data);
        } catch (error) {
            console.log(error);
        }
        })();
    }, []);

    return null;
}
```

▼ Async/await with events (example)

```
import React from "react";

function App() {
    async function handleButtonClick() {
        // use fetch with await here
        const response = await fetch("https://website.com/users.json")
        const data = await response.json();
    }

    return <button onClick={handleButtonClick}>Load data</button>;
}
```

▼ fetch POST is used for what?

to send data to an API

▼ fetch POST (example)

You need to specify the method: "POST" and often have to send the body: JSON.stringify(dataObjectHere).

It's also a best practice to set the header "Content-Type": "application/json".

```
fetch("https://website.com/grades.json", {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({grade: 50}) // converts JSON to string
})
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

▼ fetch POST in React (example)

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

function App() {
    const [number, setNumber] = useState(0);

    function handleFormSubmit(event) {
        event.preventDefault();

        fetch("https://website.com/grades.json", {
            method: "POST",
            headers: {
                "Content-Type": "application/json"
            },
            body: JSON.stringify({grade: number})
        })
        .then(response => response.json())
        .then(data => {
            console.log("Grade added");
            console.log(data);
        });
    }

    return <form onSubmit={handleFormSubmit}>
        <input type="number" value={number} name="grade" onChange={event => setNumber(event.target.value)} placeholder="Enter the grade
        <input type="submit" />
        </form>
    ;
}
```

Notice how {grade: number} is sending the state variable called number which is being set in the onChange handler with the setNumber(event.target.value) call.

▼ Why use PUT method?

to **update** an existing value

▼ What is a custom hook?

A custom hook is a JavaScript function whose name starts with use.

This function is able to call other hooks (for example useEffect, useState, etc.).

The goal of custom hooks is to be able to reuse them in several components.

It's also much cleaner to have them in a separate file.

▼ Why build a custom hook?

- easily reuse code in several components, which makes it easier to share common functionality.
- build abstractions which makes some complicated logic easier to read.

▼ Custom hook (example)

```
import {useEffect} from "react";

function useAnalyticsEvent() {
    useEffect(() => {
        gtag("event", "component_rendered");
    }, []);
}

function App() {

    useAnalyticsEvent(); // use hook for App Component

    return <h1>My App</h1>
}

render(<App />, document.querySelector("#react-root"));
```

This could be useful for sending an analytics event (for example using Google Analytics).

The gtag function is coming from Google Analytics library.

The point of this example is to show you that you can send the analytics event only once after the component has rendered the first time. That's because we're using useEffect(..., []).

▼ Custom hook file and import (example)

```
// useHelloWorld.js
import {useEffect} from "react";

export default function useHelloWorld() {
    useEffect(() => {
        console.log("Hello World!");
    });
}
```

```
import React from "react";
import useHelloWorld from "./useHelloWorld.js";

function App() {
    useHelloWorld();

    return <h1>My App</h1>;
}
```

▼ What does document represent?

The Document interface represents any web page loaded in the browser and serves as an entry point into the web page's content, which is the DOM tree.

The DOM tree includes elements such as <body> and <table>, among many others.

▼ Custom hook with state (example)

For example, for our supermarket web app, we will often need to have a counter that keeps track of how many items the user has added to their basket.

Now this custom hook needs to expose the counter state as well as the increment and decrement functions, so that they can be used by the outside component. How? Through returning an object and using object destructuring.

Why? When you have more than 2 items being returned by a custom hook is that the order won't matter as long as you use the same keys' names. (less error prone)

```
// useProductCounter.js
import {useState} from "react";

export default function useProductCounter() {
    const [counter, setCounter] = useState(0);

    function increment() {
        setCounter(prevCounter => prevCounter + 1);
    }

    function decrement() {
        setCounter(prevCounter => {
            if (prevCounter > 0) {
                return prevCounter - 1;
            }
            return 0;
        });
    }

    /* we need to return the counter and the 2 functions */
    return {
      counter: counter,
      increment: increment,
      decrement: decrement
    };
}
```

Inside the component (object destructuring).

```
function App() {
    const {counter, increment, decrement} = useProductCounter();

     return <>
        <h2>{counter}</h2>
        <button onClick={increment}>+</button>
        <button onClick={decrement}>-</button>
    </>;
}
```

Now that our custom hook is returning the increment and decrement functions, we need to update the JSX's onClick handlers to use the new increment and decrement functions.

▼ custom hook for fetch API (example)

```
// useFetch.js

import { useState } from "react";

export default function useFetch(baseUrl) {
  const [loading, setLoading] = useState(true);

  function get(url) {
    return new Promise((resolve, reject) => {
      fetch(baseUrl + url)
        .then(response => response.json())
        .then(data => {
          if (!data) {
            setLoading(false);
            return reject(data);
          }
          setLoading(false);
          resolve(data);
        })
        .catch(error => {
          setLoading(false);
          reject(error);
        });
    });
  }

  function post(url, body) {
    return new Promise((resolve, reject) => {
      fetch(baseUrl + url, {
          method: "post",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify(body)
      })
        .then(response => response.json())
        .then(data => {
          if (!data) {
            setLoading(false);
            return reject(data);
          }
          setLoading(false);
          resolve(data);
        })
        .catch(error => {
          setLoading(false);
          reject(error);
        });
    });
  }

  return { get, post, loading };
};
```

```
// index.js

import React, {useEffect} from "react";
import useFetch from "./useFetch.js";

function App() {
    const {get, loading} = useFetch("https://api.website.com/");

    useEffect(() => {
        get("users.json").then(data => {
            console.log(data);
        })
        .catch(error => console.error(error));
    }, []);

    return (<>
        <h2>{loading ? "Loading..." : ""}</h2>
    </>);
}
```

```
// POST example

// index.js

import React, {useState, useEffect} from "react";
import {render} from "react-dom";
import useFetch from './useFetch.js';

function GradeForm() {
    const [grade, setGrade] = useState(0);
    const {post} = useFetch("https://api.website.com/");

    function handleFormSubmit(event) {
        event.preventDefault();
        post("grades", {
            grade: grade
        }).then(data => {
            console.log(data);
        }).catch(error => console.error(error));
    }

    return (
        <form onSubmit={handleFormSubmit}>
            <input type="number" value={grade} name="grade" onChange={event => setGrade(event.target.value)} placeholder="Enter the gra
            <input type="submit" />
        </form>
    );
}

render(<GradeForm />, document.querySelector("#react-root"));
```

▼

▼

▼

▼

▼

▼

▼

▼

▼

▼

▼
▼
▼
▼
▼
▼
▼
▼
▼
▼
▼