

Important Points :-

- Video on Critical Rendering Path :- <https://www.youtube.com/watch?v=PkBnYxqj3k>
- Code for shimmer effect loading for cards :- <https://codepen.io/andru255/pen/wvBdxbb>
- Article for data binding in React :- <https://www.joshwcomeau.com/react/data-binding/>
- Best way for understanding Formik :- <https://formik.org/docs/tutorial#the-basics>

The best way to understand Formik as a beginner, refer to the official documentation.

1. Install Formik : `npm i formik`
2. Code along with this tutorial (<https://formik.org/docs/tutorial#the-basics>, from "The Basics" through "Wrapping Up") at the end of the section the code is highly abstracted removing all the repetitive code(eg: 30lines to 10lines reduce). So, don't skip.

N.B: Before going through this official doc I try to learn from YouTube tutorials & some random blogs, it didn't click to me. Lastly, understand from this tutorial. (edited)

Nested Paths :-

A path inside another path. It can be created by making a children array for an already existing children. Ex :-

```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
    errorElement: <Error />,
    children: [
      {
        path: "/",
        element: <Body />
      },
      {
        path: "/about",
        element: <About />,
        children: [
          {
            path: "profile",
            element: <Profile />
          }
        ]
      },
    ],
  },
]);
```

In the 1st layer of children, we are specifying path with the suffix "/", which denotes root (like -> "/about"). We could have also written "about" instead and it would work just fine because the parent path is already specified as root (since it's written path: "/").

However, in the nested path, always write just the name in the path argument (like :- path: "profile"), and not path: "/profile". This is because in the latter one, the actual path will be treated as "localhost:1234/profile", since as stated earlier, "/" denotes root path. However, in the former case, the actual path will be "localhost:1234/about/profile".

But, this syntax will just show the About component only and not the Profile component.



This is because, children are always rendered within an Outlet component and Outlet should be created inside the parent.

```
App.js          About.js ×       Profile.js
src > components > About.js > About
1   import { Outlet } from "react-router-dom";
2
3   const About = () => {
4     return (
5       <div>
6         <h1>About Us Page</h1>
7         <p>This is a part of the Namaste React Day 07 Code</p>
8         <Outlet />
9       </div>
10    );
11  };
12
```



This was done just for the sake of showing an example. We actually want our profile component to be always present in our About component, so we will just import our Profile component and use it instead of the Outlet.

```
import { Outlet } from "react-router-dom";
import Profile from "./Profile";

const About = () => {
  return (
    <div>
      <h1>About Us Page</h1>
      <p>This is a part of the Namaste React Day 07 Code</p>
      <Profile />
    </div>
  );
};
```

Class Based Components :-

When react was launched, class based components was the norm. Just like a functional component, a class component is also very similar to a normal JS class, but it extends the Component class of React to denote that it's not a normal JS class. Let's create a component named "ProfileClass".

You cannot create a class-based component without a **render()** method. Just like a functional component, returns a JSX using a keyword return, a class-based component (CBC) returns JSX using the **render()**. In the below image, try to compare a class component with a functional one :-

```
ProfileClass.js
src > components > ProfileClass.js > [o] default
1  import React from "react";
2
3  class ProfileClass extends React.Component {
4    render() {
5      return (
6        <h1>
7          Profile Class
8          Component
9        </h1>
10   );
11 }
12 export default ProfileClass;
```

```
About.js
src > components > About.js > ...
1  import ProfileClass from "./ProfileClass";
2
3  const About = () => {
4    return (
5      <div>
6        <h1>About Us Page</h1>
7        <p>This is a part of the Namaste React Day 07 Code</p>
8        <ProfileClass />
9      </div>
10 );
11 };
```

```
Profile.js
src > components > Profile.js > [o] default
1  const Profile = () => {
2    return (
3      <div>
4        <h2>Profile Component</h2>
5      </div>
6    );
7 }
8
9 export default Profile;
```

Passing props :- Here, we pass props using the keyword "this".

```
ProfileClass.js
src > components > ProfileClass.js > [o] default
1  import React from "react";
2
3  class ProfileClass extends React.Component {
4    render() {
5      return (
6        <div>
7          <h2>
8            Profile Class
9            Component
10           </h2>
11          <h2>Name: {this.props.name}</h2>
12        </div>
13   );
14 }
15 export default ProfileClass;
```

```
About.js
src > components > About.js > [o] About
1  import ProfileClass from "./ProfileClass";
2  import Profile from "./Profile";
3
4  const About = () => {
5    return (
6      <div>
7        <h1>About Us Page</h1>
8        <p>This is a part of the Namaste React Day 07 Code</p>
9        <Profile name="Arpan" />
10       <ProfileClass name="ArpanClass" />
11     </div>
12   );
13 }
14
15 export default About;
```

```
Profile.js
src > components > Profile.js > [o] Profile
1  const Profile = (props) => {
2    return (
3      <div>
4        <h2>Profile Functional Component</h2>
5        <h3>Name: {props.name}</h3>
6      </div>
7   );
8 }
9
10 export default Profile;
```

Using state (creating the state variable):-

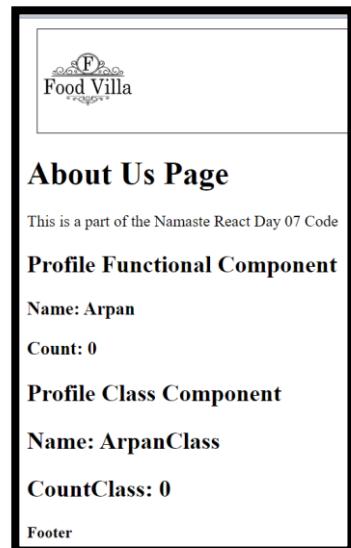
In functional comp. we used to import the useState hook. We know that classes have a constructor and constructor is a special member function used for initialisation and hence we create the state variables inside a class constructor. Now, we use useState to create state variables, but earlier we used **this.state** object ,given to us by React, just like this.props. In that object we can write the state variables with their initial value in the form of key-value pairs. Again showing the difference between FC and CBC.

```

src > components > ProfileClass.js > ProfileClass > render
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4   constructor(props) {
5     super(props)
6     this.state = {
7       count: 0
8     }
9   }
10  render() {
11    return [
12      <div>
13        <h2>
14          Profile Class Component
15        </h2>
16        <h2>Name: {this.props.name}</h2>
17        <h2>CountClass: {this.state.count}</h2>
18      </div>
19    ]
20  }
21}

src > components > Profile.js > Profile
1 import { useState } from "react";
2
3 const Profile = (props) => {
4   const [count] = useState(0);
5   return (
6     <div>
7       <h2>Profile Functional Component</h2>
8       <h3>Name: {props.name}</h3>
9       <h3>Count: {count}</h3>
10    </div>
11  );
12}
13
14 export default Profile;

```



We can also destructure both the this.props and this.state object :-

```

render() {
  const { name } = this.props;
  const { count } = this.state;
  return (
    <div>
      <h2>
        Profile Class Component
      </h2>
      <h2>Name: {name}</h2>
      <h2>CountClass: {count}</h2>
    </div>
  )
}

```

Now, to create multiple set variables, in FC, we used to create them separately, but here in CBC, we will create them in the same this.state object like :-

```

src > components > Profile.js > Profile
1 import { useState } from "react";
2
3 const Profile = (props) => {
4   const [count] = useState(0);
5   const [count2] = useState(0);
6   return (
7     <div>
8       <h2>Profile Functional Component</h2>
9       <h3>Name: {props.name}</h3>
10      <h3>Count: {count}</h3>
11    </div>
12  );
13}
14
15 export default Profile;

```

Using state (using the set function):-

We know that if we use the set function for updating the value of a set variable, instead of doing normally like ->

“setVar = newValue”,

React will trigger the reconciliation cycle on encountering it and keep our UI in sync with it. Similarly, for CBC, React gives access to a function called **setState**, where we pass the modified object as parameter. Never mutate state directly. In the below example we have shown how to update even 2 state variables in both FC and CBC :-

```

src > components > ProfileClass.js > ProfileClass > render > <button>
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4   constructor(props) {
5     super(props)
6     this.state = {
7       count: 0,
8       count2: 0
9     }
10  }
11  render() {
12    const { name } = this.props;
13    const { count, count2 } = this.state;
14    return [
15      <div>
16        <h2>
17          Profile Class Component
18        </h2>
19        <h2>Name: {name}</h2>
20        <h2>CountClass: {count}</h2>
21        <h2>CountClass2: {count2}</h2>
22        <button onClick={() => {
23          this.setState({
24            count: 1,
25            count2: 2
26          })
27        }}>ClassButton</button>
28      </div>
29    ]
30  }
31}

src > components > Profile.js > Profile
1 import { useState } from "react";
2
3 const Profile = (props) => {
4   const [count, setCount] = useState(0);
5   const [count2, setCount2] = useState(0);
6   return (
7     <div>
8       <h2>Profile Functional Component</h2>
9       <h3>Name: {props.name}</h3>
10      <h3>Count1: {count}</h3>
11      <h3>Count2: {count2}</h3>
12      <button onClick={
13        () => {
14          setCount(1);
15          setCount2(2);
16        }
17      }>Button</button>
18    </div>
19  );
20}
21
22 export default Profile;

```

About Us Page

This is a part of the Namaste React Day 07 Code

Profile Functional Component

Name: Arpan
Count1: 0
Count2: 0

Profile Class Component

Name: ArpanClass
CountClass: 0
CountClass2: 0

Footer

After clicking both the buttons :-

About Us Page

This is a part of the Namaste React Day 07 Code

Profile Functional Component

Name: Arpan
Count1: 1
Count2: 2

Profile Class Component

Name: ArpanClass
CountClass: 1
CountClass2: 2

Footer

- Important Point regarding Class Component :-**

Every React Class component is called a React Life Cycle. Everytime, we go into a life cycle, firstly the constructor is called and then the component is rendered.

```

<code>
<div>
<h1>Food Villa</h1>
<h2>About Us Page</h2>
<p>This is a part of the Namaste React Day 07 Code</p>
<b>Profile Functional Component</b>
<p>Name: Arpan<br/>Count1: 0<br/>Count2: 0<br/><input type="button" value="Button"/></p>
<b>Profile Class Component</b>
<p>Name: ArpanClass<br/>CountClass: 0<br/>CountClass2: 0<br/><input type="button" value="ClassButton"/><br/>Footer</p>
</div>
</code>

```

Everytime I refresh the page and click on the button "ClassButton" :-

Profile Class Component

Name: ArpanClass
CountClass: 1
CountClass2: 2

Console

Message
6 messages
4 user mess...
No errors
2 warnings
Console was cleared
Constructor
render
render

So, the constructor is called only once, but everytime the UI is updated, the render() function is definitely called.

Doing API call in CBC :-

In FC, we used to do API Call in the useEffect() because it got called after the first render, so that at first we can render whatever we can in the default state and later on we update the state. On updating, the component is rerendered. So, in CBC too, first of all we have to render and then update. For this purpose, there is a member function called **componentDidMount()**, which will be called after my render.

Whenever a CBC is loaded onto a page, it has some Life-Cycle methods that are called. The constructor, render(), componentDidMount() are those Life-Cycle Methods.

The sequence in which they are called is :-

- First the constructor is called
- Then the component is rendered
- Then the function componentDidMount() is called

```

js ProfileClass.js X
src > components > js ProfileClass.js > default
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4     constructor(props) {
5         super(props)
6         this.state = ...
7     }
8     console.log("Constructor");
9
10    componentDidMount() {
11        // Do API calls here
12        console.log("API class based")
13    }
14    render() {
15        const { name } = this.props;
16        const { count, count2 } = this.state;
17        console.log("render");
18        return (
19            ...
20        )
21    }
22}
23
24
25
26

```



```

js Profile.js X
src > components > js Profile.js > Profile
1 import { useEffect, useState } from "react";
2
3 const Profile = (props) => {
4     const [count, setCount] = useState(0);
5     const [count2, setCount2] = useState(0);
6     useEffect(() => {
7         // API Call
8         console.log("API functional");
9     });
10    console.log("render functional");
11    return (
12        <div>
13            <h2>Profile Functional Component</h2>
14            <h3>Name: {props.name}</h3>
15            <h3>Count1: {count}</h3>
16            <h3>Count2: {count2}</h3>
17            <button onClick={...}>Button</button>
18        </div>
19    );
20}
21
22
23
24
25
26

```

Console Recorder

Filter

- render functional
- constructor
- render
- API class based
- API functional

The Output is in this order because, in the About component, where the Profile and ProfileClass components are actually rendered, the Profile component is mentioned first and then the latter, which is why “render functional” getting printed before “Constructor”.

Converting the About Page from FC to CBC :-

The previous one

```

import ProfileClass from "./ProfileClass";
import Profile from "./Profile";

const About = () => {
    return (
        <div>
            <h1>About Us Page</h1>
            <p>This is a part of the Namaste React Day 07 Code</p>
            <Profile name={"Arpan"} />
            <ProfileClass name="ArpanClass" />
        </div>
    );
}

export default About;

```

The CBC one

```

import ProfileClass from "./ProfileClass";
import Profile from "./Profile";
import { Component } from "react";

class About extends Component {
    render() {
        return (
            <div>
                <h1>About Us Page</h1>
                <p>This is a part of the Namaste React Day 07 Code</p>
                <Profile name={"Arpan"} />
                <ProfileClass name="ArpanClass" />
            </div>
        );
    }
}

export default About;

```

Also, notice that in the new CBC code, we have imported the Component Class directly from react library.

Output 1 :- An Output to show the flow of how CBC components are called :-

First we render only the ProfileClass component in the About component and remove the Profile component. Then see the below code and output :-

```

js About.js X
src > components > js About.js > About > render
1 import ProfileClass from "./ProfileClass";
2 import { Component } from "react";
3
4 class About extends Component {
5     constructor(props){
6         super(props);
7         console.log("Parent Constructor");
8     }
9
10    componentDidMount() {
11        // Do API calls here
12        console.log("Parent componentDidMount");
13    }
14
15    render() {
16        console.log("Parent render");
17        return (
18            <div>
19                <h1>About Us Page</h1>
20                <p>This is a part of the Namaste React Day 07 Code</p>
21                <ProfileClass name="ArpanClass" />
22            </div>
23        );
24    }
25}

js ProfileClass.js X
src > components > js ProfileClass.js > ProfileClass > render
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4     constructor(props) {
5         super(props)
6         this.state = ...
7     }
8     console.log("Child Constructor");
9
10    componentDidMount() {
11        // Do API calls here
12        console.log("Child componentDidMount");
13    }
14
15    render() {
16        const { name } = this.props;
17        const { count, count2 } = this.state;
18        console.log("Child render");
19        return (
20            ...
21        )
22    }
23
24
25
26
27
28
29
29
30
31
32
33
34
35
36
37
38
39
39
40
41
42 export default ProfileClass;

```

```

Parent Constructor
Parent render
Child Constructor
Child render
Child componentDidMount
Parent componentDidMount

```

Output 2 :- Showing how multiple child CBCs are rendered from 1 parent CBC

```

About.js
src > components > About.js > About > render
1 import ProfileClass from "./ProfileClass";
2 import { Component } from "react";
3
4 class About extends Component {
5   constructor(props) {
6     super(props);
7     console.log("Parent Constructor");
8   }
9
10  componentDidMount() {
11    // Do API calls here
12    console.log("Parent componentDidMount")
13  }
14
15  render() {
16    console.log("Parent render");
17    return (
18      <div>
19        <h1>About Us Page</h1>
20        <p>This is a part of the Namaste React Day 07
21        Code</p>
22        <ProfileClass name="ArpanClassChild 1" />
23        <ProfileClass name="ArpanClassChild 2" />
24      </div>
25    )
26  }
}

ProfileClass.js
src > components > ProfileClass.js > default
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4   constructor(props) {
5     super(props)
6     this.state = {...}
7   }
8   console.log("Child Constructor " + this.props.name);
9
10  componentDidMount() {
11    // Do API calls here
12    console.log("Child componentDidMount " + this.props.name)
13  }
14
15  render() {
16    const { name } = this.props;
17    const { count, count2 } = this.state;
18    console.log("Child render " + this.props.name);
19    return (
20      ...
21    )
22  }
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42 export default ProfileClass;

```

Log Message	File	Line Number
Parent Constructor	About.js	7
Parent render	About.js	16
Child Constructor ArpanClassChild 1	ProfileClass.js	10
Child render ArpanClassChild 1	ProfileClass.js	21
Child Constructor ArpanClassChild 2	ProfileClass.js	10
Child render ArpanClassChild 2	ProfileClass.js	21
Child componentDidMount ArpanClassChild 1	ProfileClass.js	15
Child componentDidMount ArpanClassChild 2	ProfileClass.js	15
Parent componentDidMount	About.js	12

When React is rendering things up i.e. when reconciliation happens, it does that in 2 phases :-

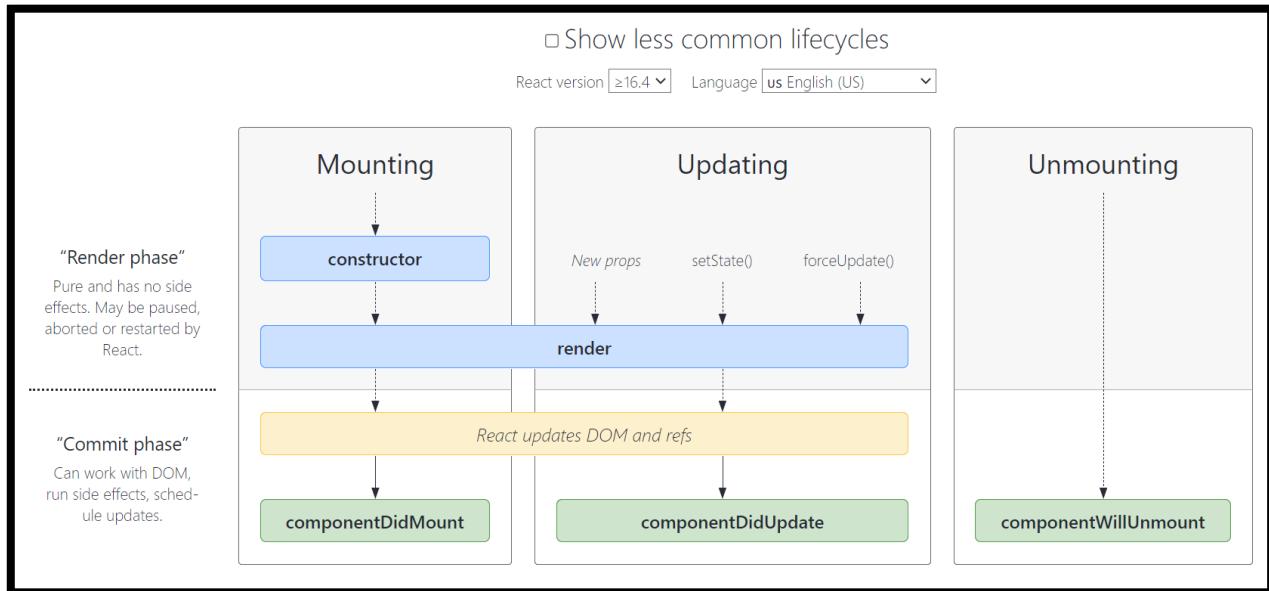
1. **Render phase** :- This phase includes the calling of constructor and render().
2. **Commit phase** :- The phase where React is actually modifying our DOM, and we know that the componentDidMount() is called after we have updated the DOM (That is why we see the Skimmer effect)

So, the steps involved in the previous example :-

- So, in the above example, first the constructor of About CBC is called, then it's render().
- React will generate the core HTML that needs to be put in the DOM and we already know that Babel helps us to convert the given JSX into HTML.
- Now in About page's JSX, there are 2 child CBC which also needs to be rendered. So, the React also calls the constructor of 1st child, calls the render() to generate its corresponding HTML from the given JSX.
- Now, you might think that after this, the render phase of the first child CBC is over, so commit phase will be done and then the render phase of the 2nd child CBC will start
- But, the commit phase takes more time than the render phase because in the former, React needs to update the DOM. That's why React tells that it should first complete the render phase of all children i.e. it batches up the render phase of all children
- So, the 2nd child's render phase starts and it gets completed first

- These are the steps in the Render phase.
- After that the commit phase of 1st and 2nd child gets completed i.e. DOM is updated for children.
- Then React will update the initial parent DOM and then the Parent componentDidMount() gets called

Refer to this diagram (link :- <https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>)



**Just pay attention to the Mounting Phase.

Let's make an API call from componentDidMount():-

We will use a github API to fetch an users called the Users API. Go to this link :-

<https://docs.github.com/en/rest/users?apiVersion=2022-11-28> and then click on "Get a user" link :-

The screenshot shows the GitHub REST API documentation for the Users endpoint. The left sidebar includes links for Quickstart, Overview, Guides, REST API reference, Actions, Activity, and Apps. The main content area displays the Users endpoint details, including a summary: "Use the REST API to get public and private information about authenticated users." Below this, there are several actions listed under the "Users" section:

- Get the authenticated user
- Update the authenticated user
- List users
- Get a user

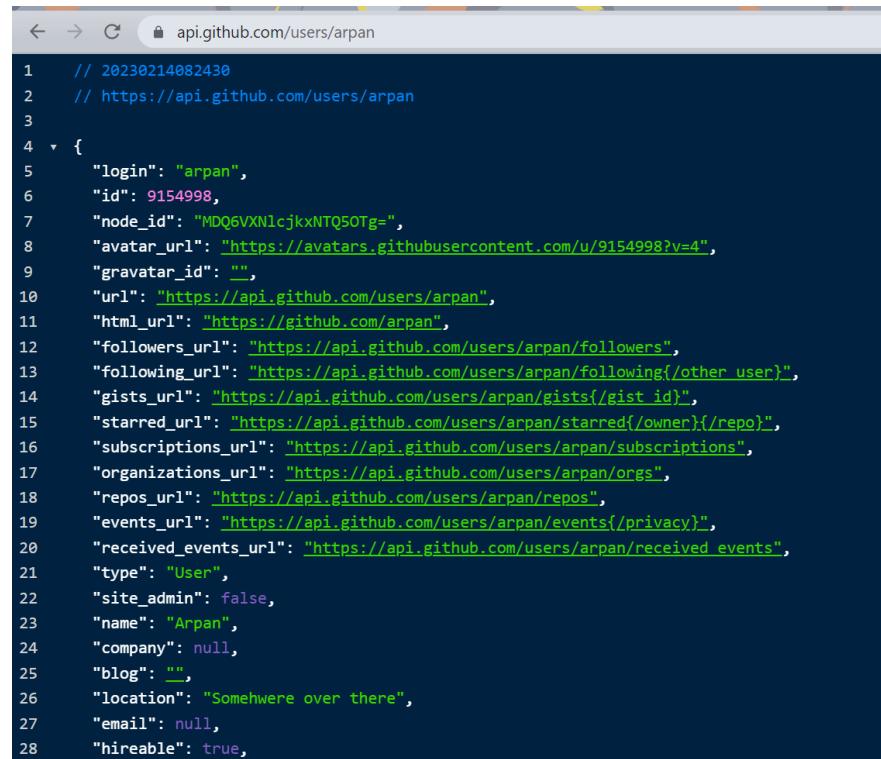
The screenshot shows the GitHub REST API documentation for the Get a user endpoint. The left sidebar lists various endpoints under the "Users" category, including "Get the authenticated user", "Update the authenticated user", "List users", "Get a user", "Get contextual information for a user", "Blocking users", and "Emails". The main content area is titled "Get a user" and includes the following details:

- Works with GitHub Apps**: A note indicating that GitHub Apps with the Plan user permission can use this endpoint.
- Description**: Provides publicly available information about someone with a GitHub account.
- Code samples**: Includes examples for curl, JavaScript, and GitHub CLI.
- curl Example**:


```
curl \
-H "Accept: application/vnd.github+json" \
-H "Authorization: Bearer <YOUR-TOKEN>" \
-H "X-GitHub-Api-Version: 2022-11-28" \
https://api.github.com/users/USERNAME
```

The link highlighted with blue is the API link. Just write any name in place of the “USERNAME” and it will give the github data.

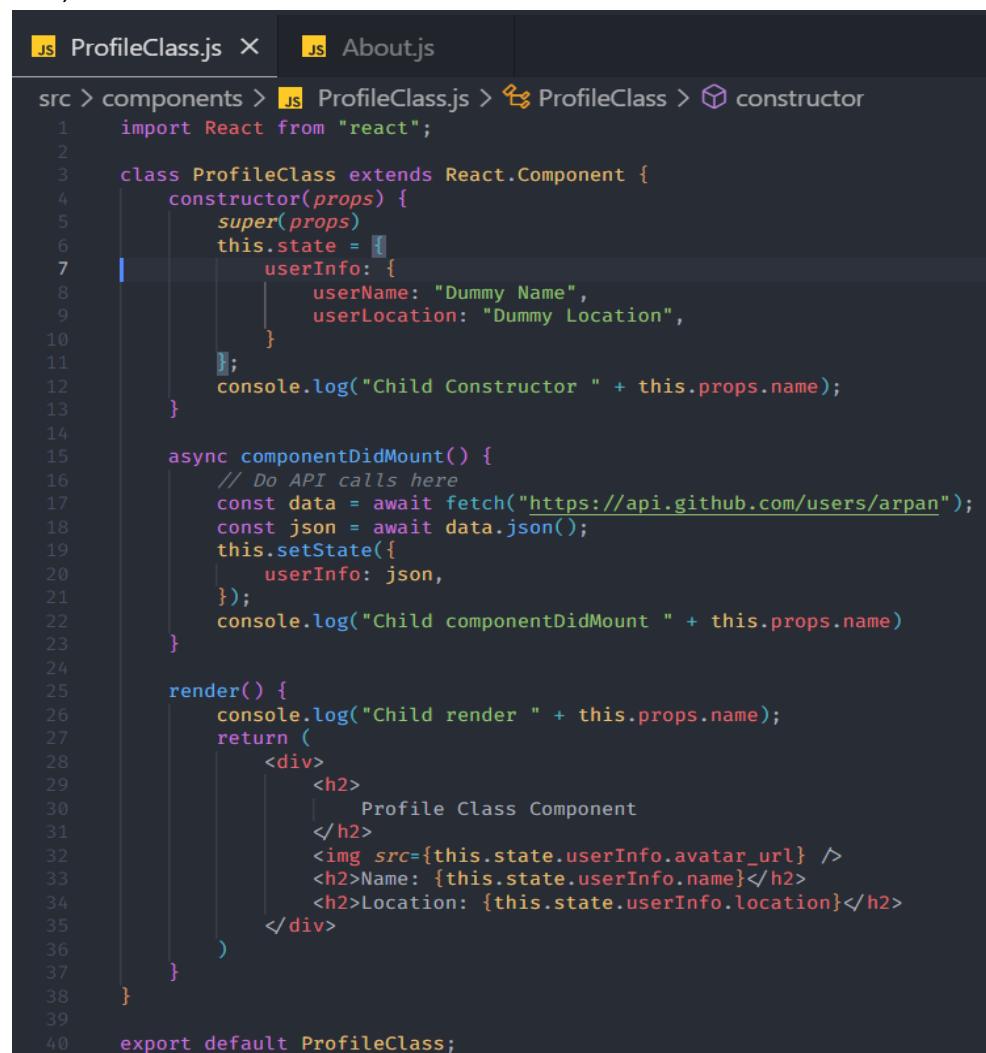
Ex :-



A screenshot of a browser window displaying the JSON response from the GitHub API for the user 'arpan'. The URL in the address bar is 'api.github.com/users/arpan'. The JSON object contains various user details and their corresponding URLs, such as login, id, node_id, avatar_url, gravatar_id, url, html_url, followers_url, following_url, gists_url, starred_url, subscriptions_url, organizations_url, repos_url, events_url, received_events_url, type, site_admin, name, company, blog, location, email, and hireable. Most of these URLs are highlighted in blue, indicating they are links.

To get a response from an API, we are using the `fetch()` function here, axios can also be implemented. Now, we know that making an API call requires an `async` function. If you remember, in our `RestaurantMenu.js`, where we used `useEffect()`, we did not make the callback function `async`. The reason is written at the bottom of this assignment, as to why we should not write an `async` function as a callback in `useEffect`.

However, we can make the `componentDidMount()` an `async` function and fetch API data using `fetch()`/axios. To store the API data, we have created another state variable named `userInfo`.



A screenshot of a code editor showing the `ProfileClass.js` component. The code defines a class `ProfileClass` that extends `React.Component`. It has a constructor that initializes the `userInfo` state with a dummy name and location. It logs the child constructor and then performs an `async componentDidMount()` to fetch data from the GitHub API for the user 'arpan'. It sets the fetched JSON as the `userInfo` state and logs the child componentDidMount message. Finally, it renders a `div` containing an `h2` for 'Profile Class Component', an `img` tag for the user's avatar, and `h2` tags for 'Name' and 'Location' with their respective values.



See the flow of output too :- (Left side is the About.js page i.e the Parent CBC and ProfileClass.js is the child CBC)

```

src > components > js About.js > About > componentDidMount
1 import ProfileClass from "./ProfileClass";
2 import { Component } from "react";
3
4 class About extends Component {
5   constructor(props) {
6     super(props);
7     console.log("Parent Constructor");
8   }
9
10  componentDidMount() {
11    // Do API calls here
12    console.log("Parent componentDidMount")
13  }
14
15  render() {
16    console.log("Parent render");
17    return (
18      <div>
19        <h1>About Us Page</h1>
20        <p>This is a part of the Namaste React Day 07 Code</p>
21        <ProfileClass name="ArpanClassChild 1" />
22        <ProfileClass name="ArpanClassChild 2" />
23      </div>
24    )
25  }
26}
27
  
```

Line Number	File	Message
7	About.js:7	Parent Constructor
16	About.js:16	Parent render
12	ProfileClass.js:12	Child Constructor ArpanClassChild 1
26	ProfileClass.js:26	Child render ArpanClassChild 1
12	ProfileClass.js:12	Child Constructor ArpanClassChild 2
26	ProfileClass.js:26	Child render ArpanClassChild 2
12	About.js:12	Parent componentDidMount

Here the “Parent componentDidMount” gets printed and after that the children’s componentDidMount because the latter one is inside an async function.

We know that React finishes our Render phase/cycle first and then goes to the commit cycle. That is why all things “Child render ArpanClassChild 2” is printed first. Then, the componentDidMount() of the parent will get printed first because, that of the children’s are async and they will take some data to load. Similarly, if we console log the json object too that stores our response, in the componentDidMount of a child in ProfileClass.js and just render one child component instead of two in About.js like :-

```

src > components > js ProfileClass.js > ProfileClass > componentDidMount
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4   constructor(props) {
5     super(props)
6     this.state = {
7       userInfo: ...
8     }
9   }
10  console.log("Child Constructor " + this.props.name);
11
12  async componentDidMount() {
13    // Do API calls here
14    const data = await fetch("https://api.github.com/users/arpan");
15    const json = await data.json();
16    console.log(json);
17    this.setState({
18      userInfo: json,
19    });
20    console.log("Child componentDidMount " + this.props.name)
21
22  }
23
24  render() {
25    console.log("Child render " + this.props.name);
26    return (
27      <div>
28        <h1>About Us Page</h1>
29        <p>This is a part of the Namaste React Day 07 Code</p>
30        <ProfileClass name="ArpanClassChild 1" />
31      </div>
32    )
33  }
34}
35
36 export default ProfileClass;
  
```

Line Number	File	Message
7	About.js:7	Parent Constructor
16	About.js:16	Parent render
12	ProfileClass.js:12	Child Constructor ArpanClassChild 1
27	ProfileClass.js:27	Child render ArpanClassChild 1
12	About.js:12	Parent componentDidMount

We will get output in the console like :-

Line Number	File	Message
7	About.js:7	Parent Constructor
16	About.js:16	Parent render
12	ProfileClass.js:12	Child Constructor ArpanClassChild 1
27	ProfileClass.js:27	Child render ArpanClassChild 1
12	About.js:12	Parent componentDidMount
	ProfileClass.js:19	{login: 'arpan', id: 9154998, node_id: 'MDQ6VXNlcjkxNTQ5OTg=', ava tar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4', gr avatar_id: '' , ...}
	ProfileClass.js:23	Child componentDidMount ArpanClassChild 1
	ProfileClass.js:27	Child render ArpanClassChild 1

Now, if there are again 2 child CBC in the About.js, which is the parent CBC, with names “ArpanClassChild 1” and “ArpanClassChild 2”, then below will be the output :-

```

Parent Constructor
Parent render
Child Constructor ArpanClassChild 1
Child render ArpanClassChild 1
Child Constructor ArpanClassChild 2
Child render ArpanClassChild 2
Parent componentDidMount
▶ {login: 'arpan', id: 9154998, node_id: 'MDQ6VXNLcjkxNTQ5OTg=', avatar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4', gravatar_id: '', ...}
Child componentDidMount ArpanClassChild 1
Child render ArpanClassChild 1
▶ {login: 'arpan', id: 9154998, node_id: 'MDQ6VXNLcjkxNTQ5OTg=', avatar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4', gravatar_id: '', ...}
Child componentDidMount ArpanClassChild 2
Child render ArpanClassChild 2
>

```

Now, again we just use the “ArpanClassChild 1” and remove the 2nd one in the About.js, which is the parent CBC. However, if we write the code of console logging the “Child componentDidMount” before making the API call like below, then we will get a different output :-

```

ProfileClass.js X
src > components > ProfileClass.js > ProfileClass > componentDidMount
1   import React from "react";
2
3   class ProfileClass extends React.Component {
4     constructor(props) {
5       super(props)
6       this.state = {
7         userInfo: ...
8       }
9     };
10    console.log("Child Constructor " + this.props.name);
11
12  }
13
14
15  async componentDidMount() {
16    console.log("Child componentDidMount " + this.props.name)
17    // Do API calls here
18    const data = await fetch("https://api.github.com/users/arpan");
19    const json = await data.json();
20    console.log(json);
21    this.setState({
22      userInfo: json,
23    });
24  }
25

```

Console Recorder Performance insights »

Filter Default levels ▾ 1 Issue

```

Parent Constructor
Parent render
Child Constructor ArpanClassChild 1
Child render ArpanClassChild 1
Child componentDidMount ArpanClassChild 1
Parent componentDidMount

{login: 'arpan', id: 9154998, node_id: 'MDQ6VXNLcjkxNTQ5OTg=', avatar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4', gravatar_id: '', ...}
Child render ArpanClassChild 1
>

```

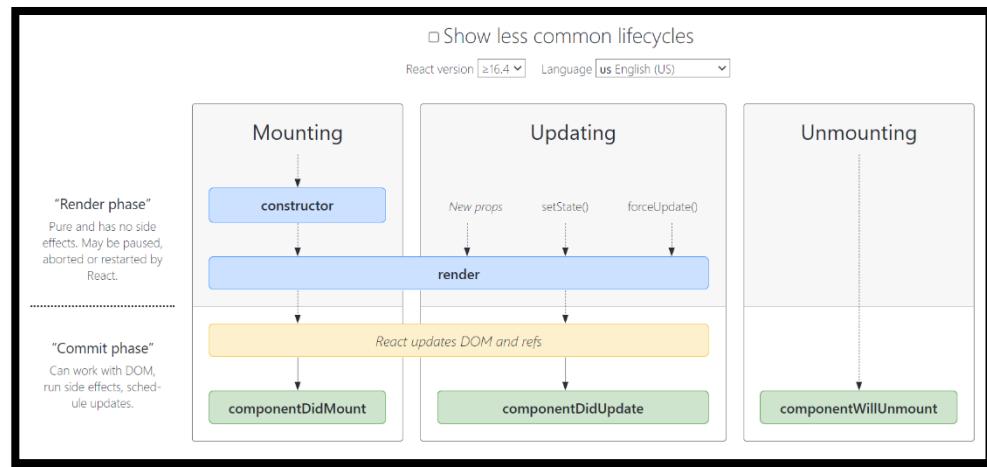
Now, if there are again 2 child CBC in the About.js, which is the parent CBC, with names “ArpanClassChild 1” and “ArpanClassChild 2”, then below will be the output :-

```

Parent Constructor
Parent render
Child Constructor ArpanClassChild 1
Child render ArpanClassChild 1
Child Constructor ArpanClassChild 2
Child render ArpanClassChild 2
Child componentDidMount ArpanClassChild 1
Child componentDidMount ArpanClassChild 2
Parent componentDidMount
▶ {login: 'arpan', id: 9154998, node_id: 'MDQ6VXNLcjkxNTQ5OTg=', avatar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4', gravatar_id: '', ...}
▶ {login: 'arpan', id: 9154998, node_id: 'MDQ6VXNLcjkxNTQ5OTg=', avatar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4', gravatar_id: '', ...}
Child render ArpanClassChild 1
Child render ArpanClassChild 2
>

```

Remember this diagram :-



Here in the last few examples, when we are doing `this.setState()` inside the child's `componentDidMount()`, we are actually updating the component. So, looking at the diagram you can say that, at first the Mounting phase happens. Here, the `constructor` is called, then the `render()`. This concludes the render phase of Mounting. Then commit phase starts where React updates the DOM with the default component and `componentDidMount()` is called. In our

example, in that function we have made an API call.

So, after the data is fetched from the API, the commit phase of the Mounting is also over.

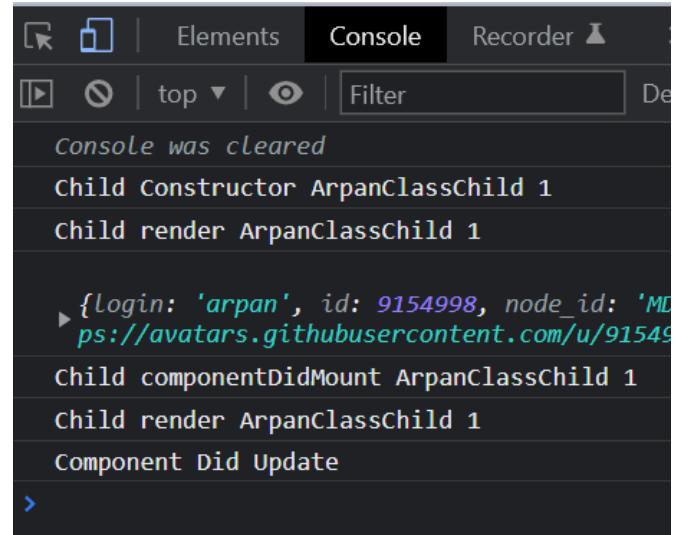
Now, we actually use that data to update an existing component i.e. rendering the component with the help of `setState()`. So, we enter the updating phase of the diagram.

In this, again the `render()` is called, which concludes the render phase of Updating. Then, the DOM is updated and `componentDidUpdate()` is called (if present, in our examples it was not there).

ComponentDidUpdate() Life-Cycle Method :-

As seen from the above diagram, it is called when the rerendering of an existing component and its updation in DOM has already been done. In simple words, this method will be called after every next render.

```
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4   constructor(props) {
5     super(props)
6     this.state = { ... }
7     console.log("Child Constructor " + this.props.name);
8   }
9
10  >  async componentDidMount() { ... }
11
12    componentDidUpdate() {
13      console.log("Component Did Update");
14    }
15
16    render() {
17      console.log("Child render " + this.props.name);
18      return ( ... )
19    }
20  }
21
22  export default ProfileClass;
```



ComponentWillUnmount() Life-Cycle Method :-

This method will be called once our component is unmounted i.e. our component is no longer there in the webpage. In our project it can be seen by writing the below piece of code, which will give the 1st output on entering the URL "localhost:1234/about" or "localhost:1234/about/profile". Now, in this page, the About component, along with its child components (here ProfileClass component) is displayed in the webpage.

The moment we navigate to the Home/Contacts component from the navbar, we will see that the console log inside the `componentWillUnmount()` gets printed.

```

js ProfileClass.js X js About.js
src > components > js ProfileClass.js > ProfileClass > componentWillUnmount
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4   constructor(props) {
5     super(props)
6     this.state = { ... };
7   }
8   console.log("Child Constructor " + this.props.name);
9 }
10
11 > async componentDidMount() { ... }
12
13
14
15 > componentDidUpdate() {
16   console.log("Component Did Update");
17 }
18
19
20 componentWillUnmount() {
21   console.log("Component Will Unmount");
22 }
23
24
25
26
27
28
29
30
31
32
33
34 render() {
35   console.log("Child render " + this.props.name);
36   return ( ...
37 )
38 }
39
40
41
42
43
44
45
46
47 }
48

```

localhost:1234/about

Dimensions: Responsive ▾ 656 x 522 100% ▾ No throttling ▾

Console was cleared runtime-058dc400e1b24991.js:135

- Child Constructor ArpanClassChild 1 ProfileClass.js:12
- Child render ArpanClassChild 1 ProfileClass.js:35
- Component Will Unmount ProfileClass.js:31
- Component Did Update ProfileClass.js:22

```

{login: 'arpan', id: 9154998, node_id: 'MDQ6VXNLcjkxNTQ5OTg=', avatar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4', gravatar_id: '', ...}
  ↵ Child componentDidMount ArpanClassChild 1 ProfileClass.js:23
  Child render ArpanClassChild 1 ProfileClass.js:35
  Component Did Update ProfileClass.js:27
  Component Will Unmount ProfileClass.js:31
  >

```

Dimensions: Responsive ▾ 656 x 522 100% ▾ No throttling ▾

Console was cleared runtime-058dc400e1b24991.js:135

- Child Constructor ArpanClassChild 1 ProfileClass.js:12
- Child render ArpanClassChild 1 ProfileClass.js:35
- Component Will Unmount ProfileClass.js:31
- Component Did Update ProfileClass.js:27

```

{login: 'arpan', id: 9154998, node_id: 'MDQ6VXNLcjkxNTQ5OTg=', avatar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4', gravatar_id: '', ...}
  ↵ Child componentDidMount ArpanClassChild 1 ProfileClass.js:23
  Child render ArpanClassChild 1 ProfileClass.js:35
  Component Did Update ProfileClass.js:27
  Component Will Unmount ProfileClass.js:31
  >

```

NEVER EVER COMPARE REACT LIFE CYCLE METHODS TO FUNCTIONAL COMPONENT HOOKS

How to mimic the functionality of a dependency array in useEffect(), here in CBC?

The elements of the dependency array in useEffect() used to denote that whenever there is a change in any one of those state variables, the callback function of useEffect() will be called. In CBC, we know that the life-cycle method that gets called after every update is the componentDidUpdate().

We know that normally, that method is called just after the initial render. So, it is like putting an empty dependency array. But what if I want it to be called after a state variable is changed?

We pass 2 arguments :- prevProps and prevState.

In the below screenshot on the left side is the CBC named ProfileClass and on the left side is the FC named Profile. In the FC, we have a dependency array with 2 state variables :- count and count2. To mimic this functionality, we write that if-else piece

of code in the `componentDidUpdate()` -> it means that when the state of `count` or `count2` changes, React has to execute the given statements written inside the if block.

The screenshot shows two code editor panes. The left pane displays `ProfileClass.js` with the following content:

```
src > components > ProfileClass.js > ProfileClass > componentDidUpdate
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4     constructor(props) {
5         super(props)
6         this.state = {
7             count: 0,
8             count2: 0,
9             userInfo: { ...
10        }
11    };
12    console.log("Child Constructor " + this.props.name);
13 }
14
15 > async componentDidMount() { ...
16 }
17
18 componentDidUpdate(prevProps, prevState) {
19     if(this.state.count !== prevState.count ||
20         this.state.count2 !== prevState.count2
21     ){
22         // Do something
23     }
24
25     console.log("Component Did Update");
26 }
27
28 }
```

The right pane displays `Profile.js` with the following content:

```
src > components > Profile.js > Profile
1 import { useEffect, useState } from "react";
2
3 const Profile = (props) => {
4     const [count, setCount] = useState(0);
5     const [count2, setCount2] = useState(0);
6     useEffect(() => {
7         // API Call
8         console.log("API functional");
9     }, [count, count2]);
10    console.log("render functional");
11    return (
12        <div>
13            <h2>Profile Functional Component</h2>
14            <h3>Name: {props.name}</h3>
15            <h3>Count1: {count}</h3>
16            <h3>Count2: {count2}</h3>
17            <button onClick={(
18                () => {
19                    setCount(1);
20                    setCount2(2);
21                }
22            )}>Button</button>
23        </div>
24    );
25
26
27 export default Profile;
```

Now, if we wanted to execute 2 different tasks for changes in the state variable count and count2, then in CBC, we have to write 2 different if blocks and in the FC, we have to write 2 different useEffect()s like :-

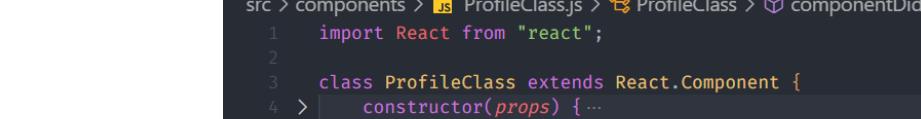
```
src > components > ProfileClass.js > ProfileClass
16
17 >     async componentDidMount() { ... }
18
19
20     componentDidUpdate(prevProps, prevState) {
21         if(this.state.count !== prevState.count){
22             // Do something
23         }
24
25         if(this.state.count2 !== prevState.count2){
26             // Do something else
27         }
28
29         console.log("Component Did Update");
30     }
31
32
33
34
35
36
37
38 }

src > components > Profile.js > Profile
1
2 const Profile = (props) => {
3     const [count, setCount] = useState(0);
4     const [count2, setCount2] = useState(0);
5
6     useEffect(() => {
7         // API Call
8         console.log("API functional");
9         // Do something
10    }, [count]);
11
12    useEffect(() => {
13        // API Call
14        console.log("API functional");
15        // Do something else
16    }, [count2]);
17
```

Use Case of componentWillUnmount() :-

It is called when a component is unmounted. It means when we move from one component to another component, or rather one page to another page (not to be confused with React is a single page application. Here we are just using page to mean a component rather than a webpage). So, when we navigate to another component/page, sometimes we need to clear something up. For ex :-

Lets we write a setInterval in the componentDidMount() method, which prints a statement “Namaste React OP” after every sec. Now, when we open the url “localhost:1234/about” in the browser, we will be seeing that statement getting printed multiple times



```
src > components > ProfileClass.js > ProfileClass > componentDidMount
1   import React from "react";
2
3   class ProfileClass extends React.Component {
4     >     constructor(props) { ...
5       }
6
7       async componentDidMount() {
8         setInterval(() => {
9           console.log("Namaste React OP");
10        }, 1000);
11
12        console.log("Child componentDidMount " + this.props.name)
13      }
14
15
16
17
18
19
20
21
22
23 }
```

The screenshot shows a browser window with the URL `localhost:1234/about`. The page content is the 'About Us Page' for 'Food Villa'. It features a logo with a stylized 'F', a navigation bar with links to Home, About, Contact, and Cart, and a 'Login' button. Below the navigation, there is a search bar with two input fields labeled 'Search' and a placeholder 'Search'. Two small images of food are displayed below the search bar. The browser's developer tools are open, specifically the 'Console' tab, which displays several log entries. These logs include messages like 'Child Constructor ArpanClasschild 1', 'Child render ArpanClasschild 1', 'Child componentDidMount ArpanClasschild 1', and 'Namaste React OP'. The count of these logs increases as the page is interacted with.

Now, when we will navigate to the Home/Contacts page, we would live this `setInterval` execution to stop. But, that won't happen because this is a single page application where only the component is changed by React through its reconciliation process. So, we will still see that statement getting printed in the console.

This screenshot is similar to the previous one, showing the 'About Us Page' of the 'Food Villa' application. The page layout is identical, including the logo, navigation bar, and search functionality. The developer tools' console tab shows a continuation of the log entries from the previous screenshot. The logs now include additional entries such as 'Initial render', 'Component Will Unmount', and 'Namaste React OP'. The count of these logs continues to increase over time, demonstrating the persistent execution of the `setInterval` function even after navigating away from the page.

Now, if we navigate back to the About Page, we will be again calling the `componentDidMount()` method which has the `setInterval`. So, now there will be 2 `setIntervals` running and if you notice the count of the statement in the console, you will see that the count is increasing by 2 in every 1 sec.

This screenshot shows the 'About Us Page' again, with the same visual elements and developer tools setup. The logs in the console now clearly show the count increasing by 2 every second, indicating that two `setInterval` functions are running simultaneously. This visualizes the 'major downside of single page application' mentioned in the text below.

This is major downside of single page application

So, when we are changing our page, instead of reloading the page, React is changing the components. So, if we again move to the Contacts/Home page and again navigate back to the About page, the count of that statement getting printed will be 3 in every sec.

To avoid this, we need to cleanup this functionality. We know that when a component unmount, the `componentWillUnmount()` will be called and also that to clear a `setInterval()`, we use a `clearInterval()`. But, how can we reference the `clearInterval()` in the `componentWillUnmount()` to the `setInterval()` in the `componentDidMount()`?

To do that, while writing the `setInterval()`, store it in a variable using the "this" keyword and reference that variable in the `clearInterval()` too like :-

JS ProfileClass.js

```

src > components > JS ProfileClass.js > ProfileClass > componentWillUnmount
1 import React from "react";
2
3 class ProfileClass extends React.Component {
4   constructor(props) { ... }
5
6
7   async componentDidMount() {
8     this.timer = setInterval(() => {
9       console.log("Namaste React OP");
10    }, 1000);
11
12   console.log("Child componentDidMount " + this.props.name)
13 }
14
15   componentDidUpdate(prevProps, prevState) { ... }
16
17   componentWillUnmount() {
18     clearInterval(this.timer);
19     console.log("Component Will Unmount");
20   }
21
22 }
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41

```

Doing this, the statement will get printed only when we are in the About page and not any other page because as soon as we navigate away from the About Page, that setInterval() will be cleaned up.

localhost:1234/about

Dimensions: Responsive 656 x 522 100% No throttling

Console

```

Child Constructor ArpanClassChild 1
Child render ArpanClassChild 1
Child componentDidMount ArpanClassChild 1
13 Namaste React OP
>

```

About Us Page

This is a part of the Namaste React Day 07 Code

Profile Class Component

localhost:1234

Dimensions: Responsive 656 x 522 100% No throttling

Console

```

Child Constructor ArpanClassChild 1
Child render ArpanClassChild 1
Child componentDidMount ArpanClassChild 1
24 Namaste React OP
Initial render
Component Will Unmount
Body.js:25
{
  statusCode: 0,
  data: {...},
  tid: '406226be-c69e-4763-8f2e-76fb5f01376f',
  sid: '5ew157c4-5813-4605-a291-be6c60f11aec',
  deviceId: 'e040c99c-48ec-ec77-6296-dee77014ad22',
  ...
}
Initial render
Body.js:29

```

Performing cleanup in the Functional Component :-

Just like in the CBC, if we write a setInterval() in useEffect() of a FC, then it will get executed and never stop. The 1st image is the code, the 2nd is the output in console when we reload the About page. The 3rd image is when we navigate to the Home Page and you can see that the statement is still getting printed. Then in the 4th image, it shows the output in console when we navigate back to the About Page, where you will see that in one second, the statement is getting printed twice just like in the case of CBC.

The screenshot shows two code editors side-by-side. The left editor contains `About.js` with the following code:

```

src > components > About.js > About > render
1 import Profile from "./Profile";
2 import { Component } from "react";
3
4 class About extends Component {
5   constructor(props) {
6     super(props);
7     // console.log("Parent Constructor");
8   }
9
10  componentDidMount() {
11    // Do API calls here
12    // console.log("Parent componentDidMount")
13  }
14
15  render() {
16    // console.log("Parent render");
17    return (
18      <div>
19        <h1>About Us Page</h1>
20        <p>This is a part of the Namaste React Day 07 Code</p>
21        <Profile name="ArpanClassChild 1" />
22      </div>
23    );
24  }
25}
26
27 export default About;

```

The right editor contains `Profile.js` with the following code:

```

src > components > Profile.js > Profile
1 import { useEffect, useState } from "react";
2
3 const Profile = (props) => {
4   const [count, setCount] = useState(0);
5   const [count2, setCount2] = useState(0);
6   useEffect(() => {
7     setInterval(() => {
8       console.log("Namaste React OP");
9     }, 1000);
10   }, []);
11
12   return (
13     <div>
14       <h2>Profile Functional Component</h2>
15       <h3>Name: {props.name}</h3>
16       <h3>Count1: {count}</h3>
17       <h3>Count2: {count2}</h3>
18       <button onClick={...}>Button</button>
19     </div>
20   );
21
22 export default Profile;

```

The browser window displays the `About` page of the Food Villa website. The header includes a logo, navigation links (Home, About, Contact, Cart), and a Login button. Below the header, there is a heading "About Us Page" and a subtext "This is a part of the Namaste React Day 07 Code". The browser's developer tools are open, showing the Network tab with one request to "Profile.js:8" and the Console tab with logs related to the component.

The browser window displays the `About` page of the Food Villa website. The header includes a logo, navigation links (Home, About, Contact, Cart), and a Login button. Below the header, there is a search bar and two small images. The browser's developer tools are open, showing the Network tab with one request to "Profile.js:8" and the Console tab with logs related to the component.

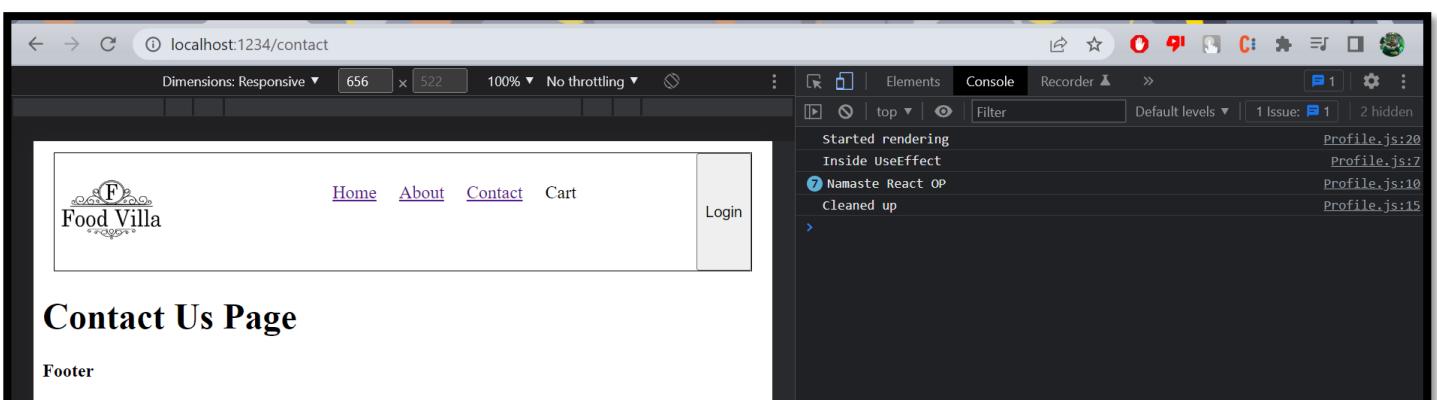
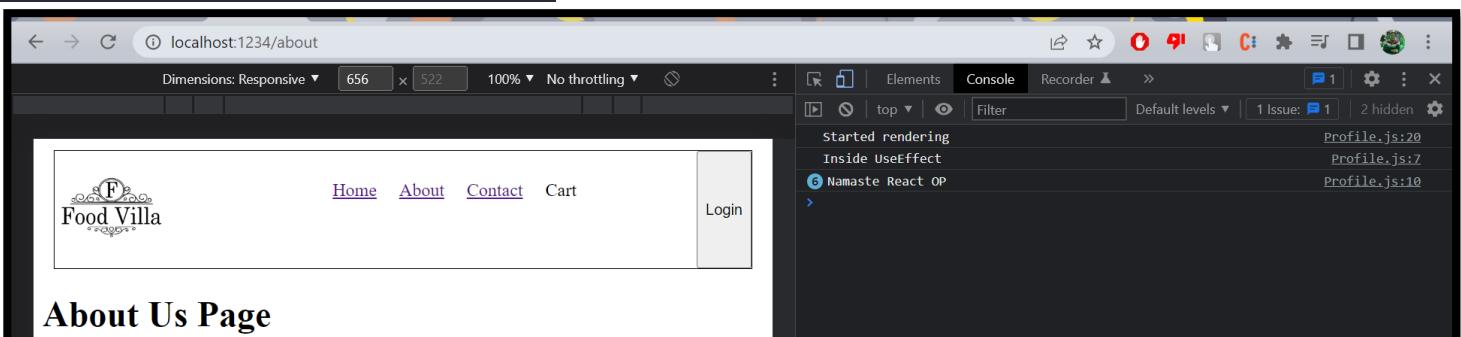
The browser window displays the `About` page of the Food Villa website. The header includes a logo, navigation links (Home, About, Contact, Cart), and a Login button. Below the header, there is a heading "About Us Page" and a subtext "This is a part of the Namaste React Day 07 Code". The browser's developer tools are open, showing the Network tab with one request to "Profile.js:8" and the Console tab with logs related to the component.

Till now we know that the callback of the `useEffect` function does not return any statement. However, it can return only 1 type of function that will get executed once the component is unmounted. That function is called a [cleanup function](#).

So, it can be implemented in this way (We also created a variable named timer to store the setInterval()) :-

```
3 const Profile = (props) => {
4     const [count, setCount] = useState(0);
5     const [count2, setCount2] = useState(0);
6     useEffect(() => {
7         console.log("Inside UseEffect");
8
8         const timer = setInterval(() => {
9             console.log("Namaste React OP");
10        }, 1000);
11
12         return () => {
13             clearInterval(timer);
14             console.log("Cleaned up");
15         }
16     }, []);
17
18     console.log("Started rendering");
19     return (
20     );
21 }
22 >
23 >
24 >
25 >
26 >
27 export default Profile;
```

The 1st image below shows the output when we are in the About page and the 2nd image shows the output when we navigate to the Contacts page.



Why shouldn't we make the useEffect()'s callback an async function ?

Before knowing the reason, lets make an useEffect()'s callback an async function and see what error or warning React throws us. For this example, I am just making the useEffect()'s callback in the Profie FC as an async function.

```
JS Profile.js X
src > components > JS Profile.js > default
1   import { useEffect, useState } from "react";
2
3   const Profile = (props) => {
4       const [count, setCount] = useState(0);
5       const [count2, setCount2] = useState(0);
6       useEffect(async () => {
7           console.log("Inside UseEffect");
8       }, []);
8
9       console.log("Started rendering");
10  >
11  >
12  >
13  >
14  >
15  >
16  >
17  >
18  >
19  >
20  >
21  >
22  >
23  >
24  >
25  >
26  >
27  export default Profile;
```

Also we have removed the setInterval and the returning of the cleanup function statement from the useEffect()'s callBack. In the below image, we can see the warning React throws.

The screenshot shows a browser developer tools console with the URL `localhost:1234/about`. The page title is "Food Villa". The console has tabs like Elements, Console, Recorder, Performance insights, Sources, Network, Performance, Memory, Application, Security, Lighthouse, and AdBlock. The Console tab is active. A warning message is displayed:

```

Warning: useEffect must not return anything besides a function, which is used for clean-up.

It looks like you wrote useEffect(async () => ...) or returned a Promise. Instead, write the async function inside your effect and call it immediately:

useEffect(() => {
  async function fetchData() {
    // You can await here
    const response = await MyAPI.getData(someId);
    // ...
  }
  fetchData();
}, [someId]); // Or [] if effect doesn't need props or state

Learn more about data fetching with Hooks: https://reactjs.org/link/hooks-data-fetching
at Profile (http://localhost:1234/index.7271efb6.js:32918:51)
at div
at About (http://localhost:1234/index.7271efb6.js:32857:9)
at RenderedRoute (http://localhost:1234/index.7271efb6.js:28741:11)
at Outlet (http://localhost:1234/index.7271efb6.js:29034:28)
at AppLayout
at RenderedRoute (http://localhost:1234/index.7271efb6.js:28741:11)
at RenderErrorBoundary (http://localhost:1234/index.7271efb6.js:28695:9)
at Routes (http://localhost:1234/index.7271efb6.js:29103:11)
at Router (http://localhost:1234/index.7271efb6.js:29052:21)
at RouterProvider (http://localhost:1234/index.7271efb6.js:28927:11)

```

The solution is to just write an `async` function inside the callback of the `useEffect()` hook and call that `async` function inside the `useEffect()`'s callback only. You can see the solution in this [Link](#) (Also, read it till the end of the 1st part i.e. before :How to Trigger a Hook Manually").

When should we use the `useEffect` cleanup?

Let's say we have a React component that fetches and renders data. If our component unmounts before our promise resolves, `useEffect` will try to update the state (on an unmounted component) and send an error that looks like this:

```

Warning: Can't perform a React state update on an unmounted component. This index.js:1
is a no-op, but it indicates a memory leak in your application. To fix, cancel all
subscriptions and asynchronous tasks in a useEffect cleanup function.
in Post (at App.js:13)

```

To fix this error, we use the cleanup function to resolve it.

When you go through the below links, you might see this warning mentioned in those articles, or videos. However, you might not encounter this warning maybe because React has removed it.

See this video on cleanup too :- https://www.youtube.com/watch?v=5gCtW7RQtQA&ab_channel=ThapaTechnical
 Then this :- https://www.youtube.com/watch?v=aKOQtGLT-Yk&ab_channel=TheNetNinja and
https://www.youtube.com/watch?v=Wu0rVQuawLU&ab_channel=WebDevCody

Then see these 2 links too :- [Link1](#), [Link2](#), [Link3](#)

Mistakes every React developer should avoid (Also covers the cleanup) :-

https://www.youtube.com/watch?v=QQYeipc_cik&ab_channel=LamaDev

React follows a one-way data binding i.e. data flow only from parent to child, not reverse, at least in theory. But how to do the reverse action? See this [Link4](#)

Why do we write `super(props)` and is there a problem in writing just `super()` ? -> [Link5](#), [Link6](#) (See the 1st answer)