Important Points :-

- Video on Critical Rendering Path :- https://www.youtube.com/watch?v=PkOBnYxqj3k
- 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

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

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

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.

```jsx
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 Namas
            <Profile />
        </div>
    )
};
```

Class Based Components :-

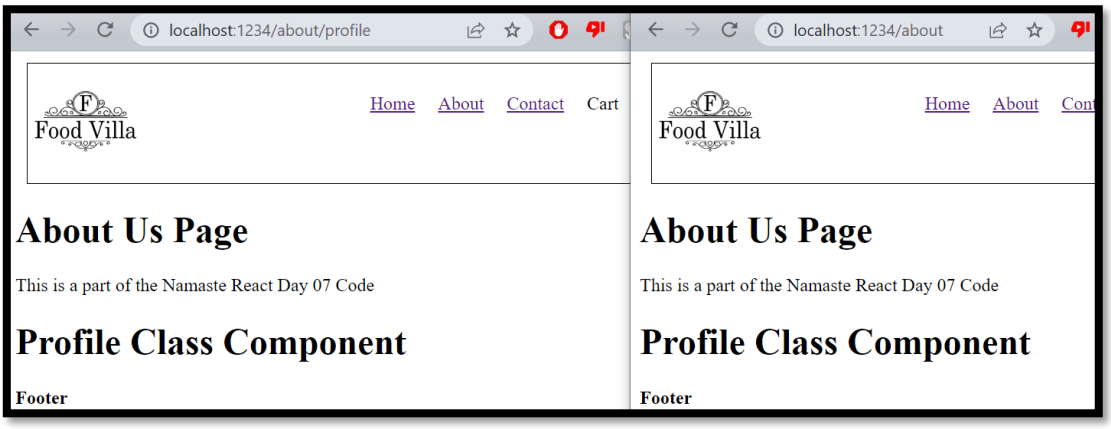
When react was launched, class based components was the norm. Just like a functional component, a class component is also very similar to a normal JS class, but it extends the Component class of React to denote that it's not a normal JS class. Let's create a component named "ProfileClass".

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
```jsx
import React from "react";

class ProfileClass extends React.
Component {
    render() {
        return (
            <h1>
                Profile Class
                Component
            </h1>
        )
    }
}

export default ProfileClass;
```

About.js
```jsx
import ProfileClass from "./
ProfileClass";

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

Profile.js
```jsx
const Profile = () => {
    return (
        <div>
            <h2>Profile
            Component</h2>
        </div>
    )
};

export default Profile;
```



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

ProfileClass.js
```jsx
import React from "react";

class ProfileClass extends React.
Component {
    render() {
        return (
            <div>
                <h2>
                    Profile Class
                    Component
                </h2>
                <h2>Name: {this.props.
                name}</h2>
            </div>
        )
    }
}
```

About.js
```jsx
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;
```

Profile.js
```jsx
const Profile = (props) => {
    return (
        <div>
            <h2>Profile Functional
            Component</h2>
            <h3>Name: {props.name}</h3>
        </div>
    )
};

export default Profile;
```

Using state (creating the state variable):-

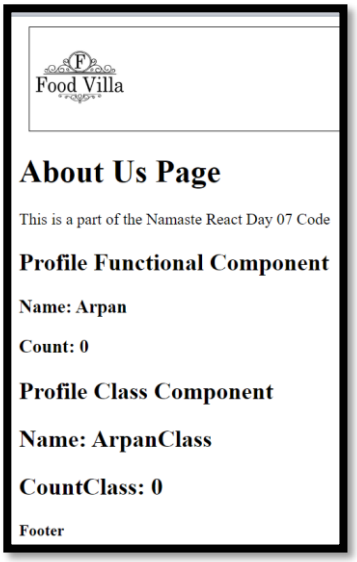
In functional comp. we used to import the useState hook. We know that classes have a constructor and constructor is a special member function used for initialisation and hence we create the state variables inside a class constructor. Now, we use useState to create state variables, but earlier we used **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.

ProfileClass.js

```
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            count: 0
        }
    }
    render() {
        return (
            <div>
                <h2>
                    Profile Class Component
                </h2>
                <h2>Name: {this.props.name}</h2>
                <h2>CountClass: {this.state.count}</h2>
            </div>
        )
    }
}
```

Profile.js

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

const Profile = (props) => {
    const [count] = useState(0);
    return (
        <div>
            <h2>Profile Functional Component</h2>
            <h3>Name: {props.name}</h3>
            <h3>Count: {count}</h3>
        </div>
    )
};

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 :-

```
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            count: 0,
            count2: 0,
        }
    }
```

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

const Profile = (props) => {
    const [count] = useState(0);
    const [count2] = useState(0);
    return (
        <div>
            <h2>Profile Functional Component</h2>
            <h3>Name: {props.name}</h3>
            <h3>Count: {count}</h3>
```

Using state (using the set function):-

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

"setVar = newValue",

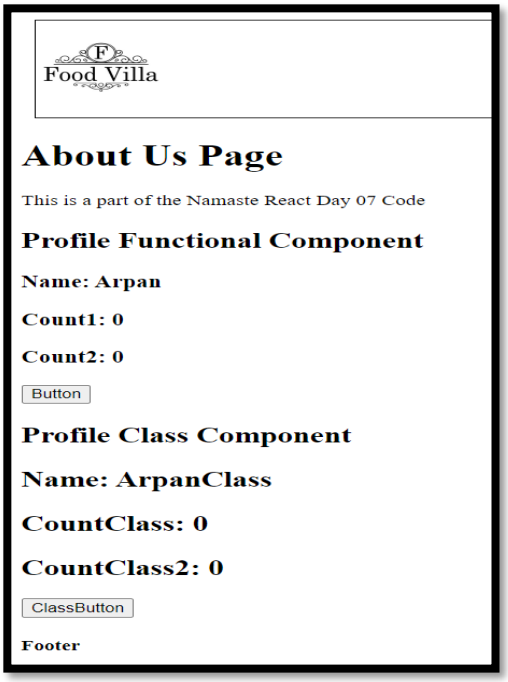
React will trigger the reconciliation cycle on encountering it and keep our UI in sync with it. Similarly, for CBC, React gives access to a function called **setState**, where we pass the modified object as parameter. Never mutate state directly. In the below example we have shown how to update even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

ProfileClass.js

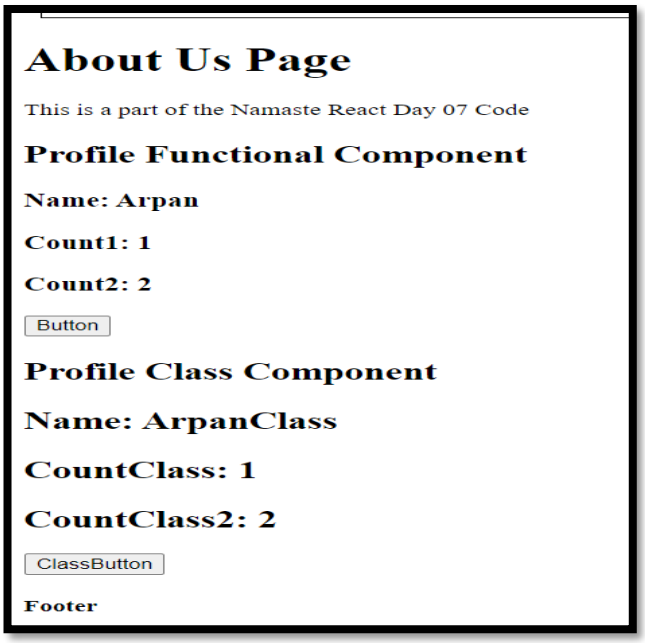
```
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            count: 0,
            count2: 0
        }
    }
    render() {
        const { name } = this.props;
        const { count, count2 } = this.state;
        return (
            <div>
                <h2>
                    Profile Class Component
                </h2>
                <h2>Name: {name}</h2>
                <h2>CountClass: {count}</h2>
                <h2>CountClass2: {count2}</h2>
                <button onClick={() => {
                    this.setState({
                        count: 1,
                        count2: 2
                    })
                }}>ClassButton</button>
            </div>
        )
    }
}
```

Profile.js

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

const Profile = (props) => {
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(0);
    return (
        <div>
            <h2>Profile Functional Component</h2>
            <h3>Name: {props.name}</h3>
            <h3>Count1: {count}</h3>
            <h3>Count2: {count2}</h3>
            <button onClick={
                () => {
                    setCount(1);
                    setCount2(2);
                }
            }>Button</button>
        </div>
    )
};
```



After clicking both the buttons :-

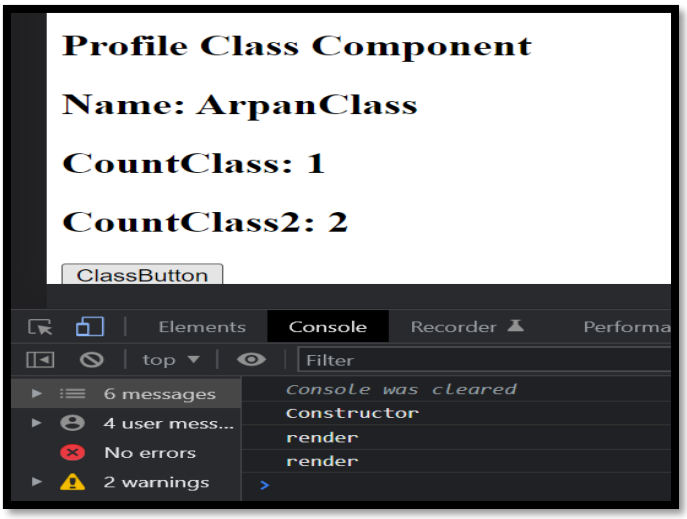


- **<u>Important Point regarding Class Component :-</u>**

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.

```
class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            count: 0,
            count2: 0
        }
        console.log("Constructor");
    }
    render() {
        const { name } = this.props;
        const { count, count2 } = this.state;
        console.log("render");
        return ( ···
        )
    }
}
```

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



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

<u>Doing API call in CBC :-</u>

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

ProfileClass.js

```
src > components > ProfileClass.js > default
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {···
        }
        console.log("Constructor");
    }

    componentDidMount() {
        // Do API calls here
        console.log("API class based")
    }
    render() {
        const { name } = this.props;
        const { count, count2 } = this.state;
        console.log("render");
        return (···
        )
    }
}
```

Profile.js

```
src > components > Profile.js > Profile
import { useEffect, useState } from "react";

const Profile = (props) => {
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(0);
    useEffect(() => {
        // API Call
        console.log("API functional");
    });
    console.log("render functional");
    return (
        <div>
            <h2>Profile Functional Component</h2>
            <h3>Name: {props.name}</h3>
            <h3>Count1: {count}</h3>
            <h3>Count2: {count2}</h3>
            <button onClick={···
            }>Button</button>
        </div>
    )
};
```

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 Rea
            <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 Re
                <Profile name={"Arpan"} />
                <ProfileClass name="ArpanClass" />
            </div>
        )
    }
}
```

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 :-

About.js

```
src > components > About.js > About > render
import ProfileClass from "./ProfileClass";
import { Component } from "react";

class About extends Component {
    constructor(props){
        super(props);
        console.log("Parent Constructor");
    }

    componentDidMount() {
        // Do API calls here
        console.log("Parent componentDidMount")
    }

    render() {
        console.log("Parent render");
        return (
            <div>
                <h1>About Us Page</h1>
                <p>This is a part of the Namaste React Day 07
                Code</p>
                <ProfileClass name="ArpanClass" />
            </div>
        )
    }
}
```

ProfileClass.js

```
src > components > ProfileClass.js > ProfileClass > render
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {···
        }
        console.log("Child Constructor");
    }

    componentDidMount() {
        // Do API calls here
        console.log("Child componentDidMount")
    }

    render() {
        const { name } = this.props;
        const { count, count2 } = this.state;
        console.log("Child render");
        return (···
        )
    }
}

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

class About extends Component {
    constructor(props) {
        super(props);
        console.log("Parent Constructor");
    }

    componentDidMount() {
        // Do API calls here
        console.log("Parent componentDidMount")
    }

    render() {
        console.log("Parent render");
        return (
            <div>
                <h1>About Us Page</h1>
                <p>This is a part of the Namaste React Day 07
                Code</p>
                <ProfileClass name="ArpanClassChild 1" />
                <ProfileClass name="ArpanClassChild 2" />
            </div>
        )
    }
}
```

ProfileClass.js

```
src > components > ProfileClass.js > default
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {···
        }
        console.log("Child Constructor " + this.props.name);
    }

    componentDidMount() {
        // Do API calls here
        console.log("Child componentDidMount " + this.props.name)
    }

    render() {
        const { name } = this.props;
        const { count, count2 } = this.state;
        console.log("Child render " + this.props.name);
        return (···
        )
    }
}

export default ProfileClass;
```

```
Console  Recorder  Performance insights  1
Filter  Default levels  1 Issue: 1  2 hidden
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 :-

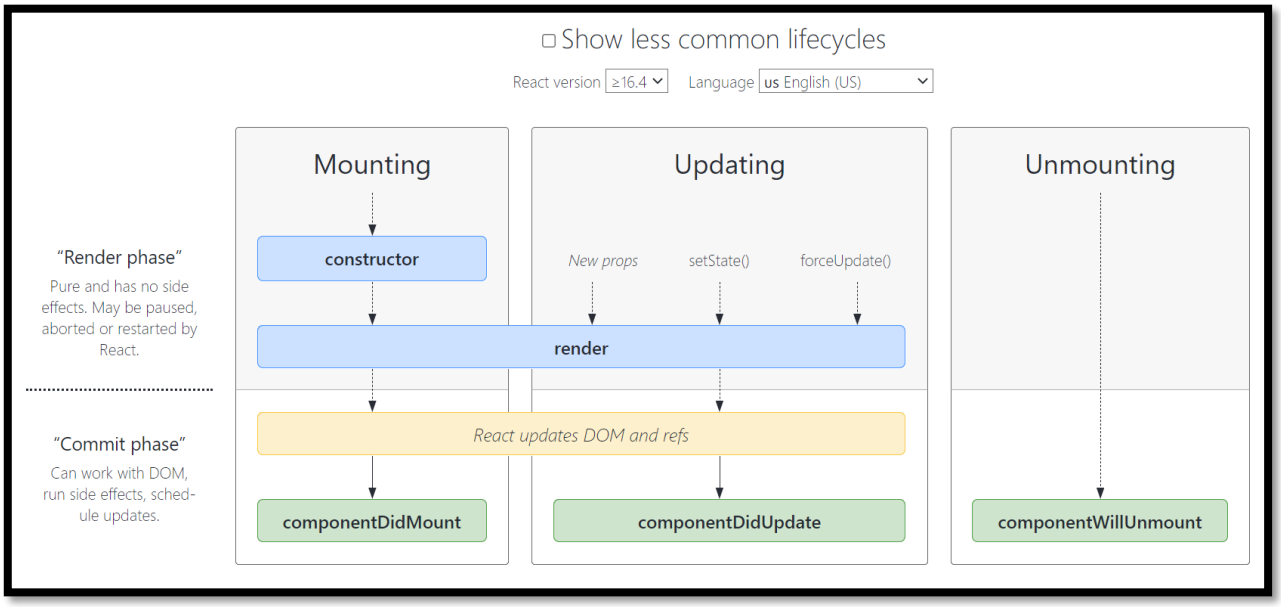
1. **Render phase** :- This phase includes the calling of constructor and render().
2. **Commit phase** :- The phase where React is actually modifying our DOM, and we know that the componentDidMount() is called after we have updated the DOM (That is why we see the Skimmer effect)

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 it's 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

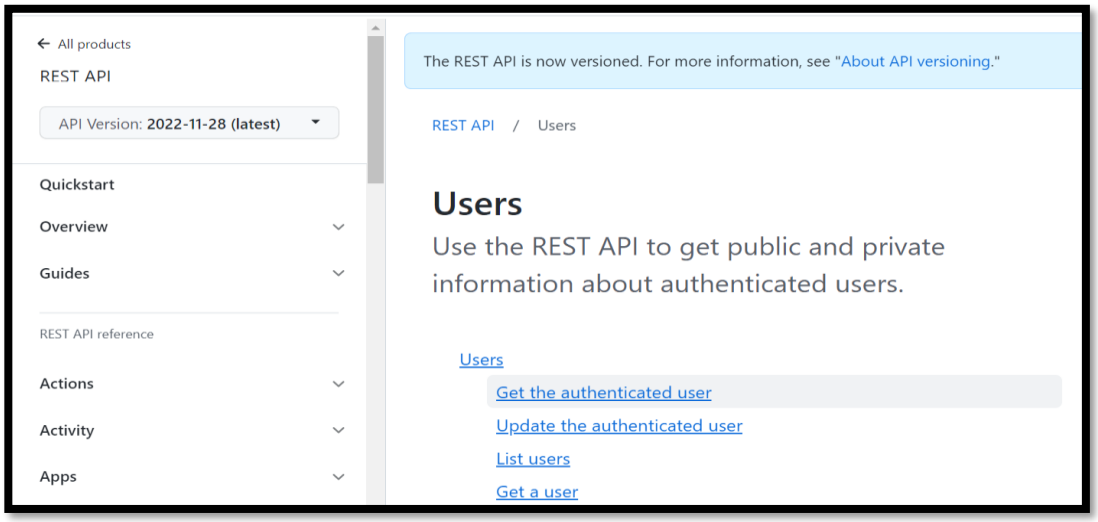
Refer to this diagram (link :- https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)

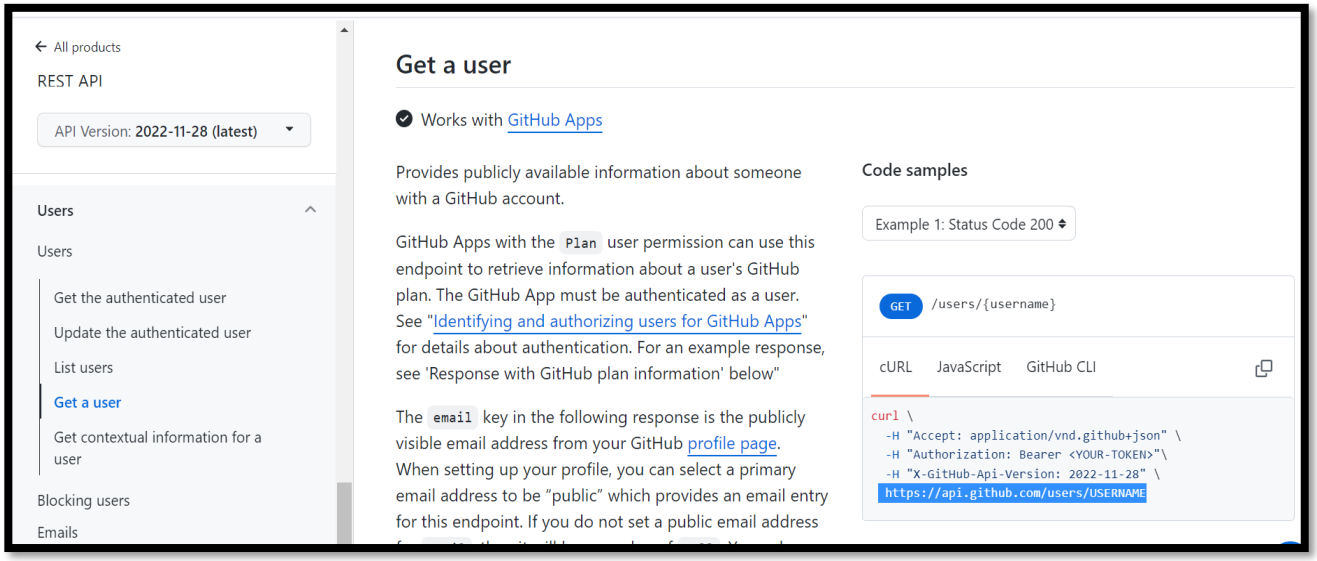


**Just pay attention to the Mounting Phase.

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 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 :-

api.github.com/users/arpan

```
// 20230214082430
// https://api.github.com/users/arpan

{
  "login": "arpan",
  "id": 9154998,
  "node_id": "MDQ6VXNlcjkxNTQ5OTg=",
  "avatar_url": "https://avatars.githubusercontent.com/u/9154998?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/arpan",
  "html_url": "https://github.com/arpan",
  "followers_url": "https://api.github.com/users/arpan/followers",
  "following_url": "https://api.github.com/users/arpan/following{/other_user}",
  "gists_url": "https://api.github.com/users/arpan/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/arpan/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/arpan/subscriptions",
  "organizations_url": "https://api.github.com/users/arpan/orgs",
  "repos_url": "https://api.github.com/users/arpan/repos",
  "events_url": "https://api.github.com/users/arpan/events{/privacy}",
  "received_events_url": "https://api.github.com/users/arpan/received_events",
  "type": "User",
  "site_admin": false,
  "name": "Arpan",
  "company": null,
  "blog": "",
  "location": "Somehwere over there",
  "email": null,
  "hireable": true,
```

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.

ProfileClass.js | About.js

src > components > ProfileClass.js > ProfileClass > constructor

```
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            userInfo: {
                userName: "Dummy Name",
                userLocation: "Dummy Location",
            }
        };
        console.log("Child Constructor " + this.props.name);
    }

    async componentDidMount() {
        // Do API calls here
        const data = await fetch("https://api.github.com/users/arpan");
        const json = await data.json();
        this.setState({
            userInfo: json,
        });
        console.log("Child componentDidMount " + this.props.name)
    }

    render() {
        console.log("Child render " + this.props.name);
        return (
            <div>
                <h2>
                    Profile Class Component
                </h2>
                <img src={this.state.userInfo.avatar_url} />
                <h2>Name: {this.state.userInfo.name}</h2>
                <h2>Location: {this.state.userInfo.location}</h2>
            </div>
        )
    }
}

export default ProfileClass;
```

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)

ProfileClass.js | About.js

src > components > About.js > About > componentDidMount

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

class About extends Component {
    constructor(props) {
        super(props);
        console.log("Parent Constructor");
    }

    componentDidMount() {
        // Do API calls here
        console.log("Parent componentDidMount")
    }

    render() {
        console.log("Parent render");
        return (
            <div>
                <h1>About Us Page</h1>
                <p>This is a part of the Namaste React Day 07 Code</p>
                <ProfileClass name="ArpanClassChild 1" />
                <ProfileClass name="ArpanClassChild 2" />
            </div>
        )
    }
}
```

Console | Recorder | Performance insights | 1 Issue: 1 | 2 hidden

Filter | Default levels

Parent Constructor	About.js:7
Parent render	About.js:16
Child Constructor ArpanClassChild 1	ProfileClass.js:12
Child render ArpanClassChild 1	ProfileClass.js:26
Child Constructor ArpanClassChild 2	ProfileClass.js:12
Child render ArpanClassChild 2	ProfileClass.js:26
Parent componentDidMount	About.js:12
Child componentDidMount ArpanClassChild 1	ProfileClass.js:22
Child render ArpanClassChild 1	ProfileClass.js:26
Child componentDidMount ArpanClassChild 2	ProfileClass.js:22
Child render ArpanClassChild 2	ProfileClass.js:26

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 :-

ProfileClass.js

src > components > ProfileClass.js > ProfileClass > componentDidMount

```
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            userInfo: {...
            }
        };
        console.log("Child Constructor " + this.props.name);
    }

    async componentDidMount() {
        // Do API calls here
        const data = await fetch("https://api.github.com/users/arpan");
        const json = await data.json();
        console.log(json);
        this.setState({
            userInfo: json,
        });
        console.log("Child componentDidMount " + this.props.name)
    }

    render() {
        console.log("Child render " + this.props.name);
        return (...
        )
    }
}

export default ProfileClass;
```

About.js

src > components > About.js > About > render

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

class About extends Component {
    constructor(props) {
        super(props);
        console.log("Parent Constructor");
    }

    componentDidMount() {
        // Do API calls here
        console.log("Parent componentDidMount")
    }

    render() {
        console.log("Parent render");
        return (
            <div>
                <h1>About Us Page</h1>
                <p>This is a part of the Namaste React Day 07 Code</p>
                <ProfileClass name="ArpanClassChild 1" />
            </div>
        )
    }
}

export default About;
```

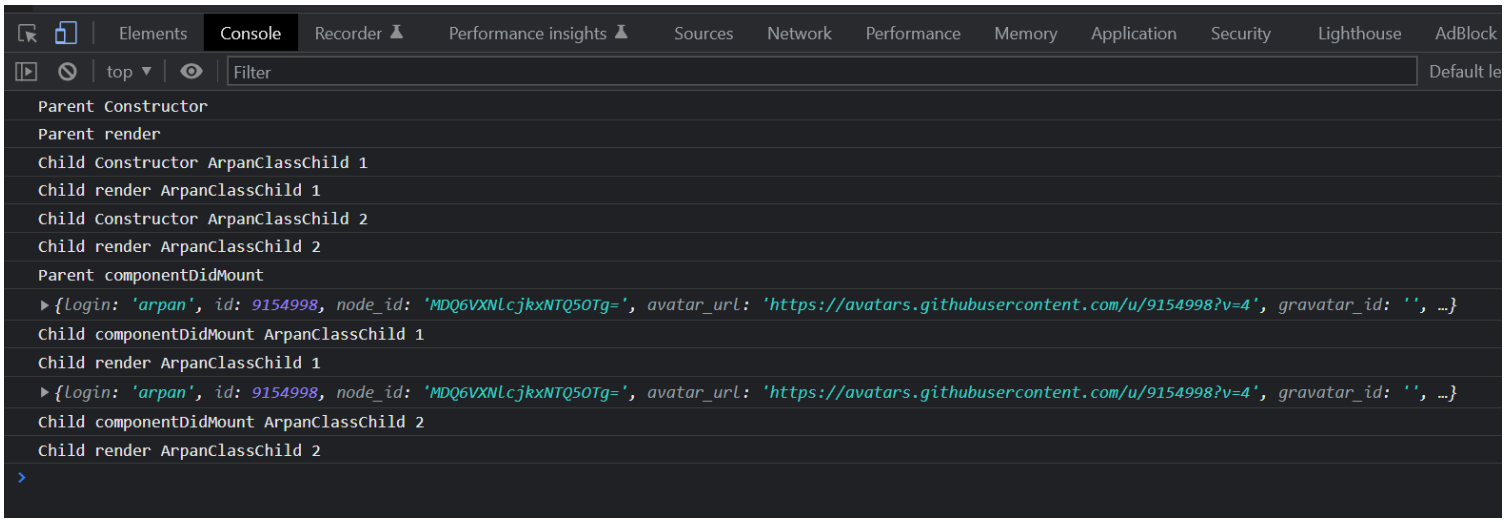
We will get output in the console like :-

Console | Recorder | Performance insights | 1 Issue: 1 | 2 hidden

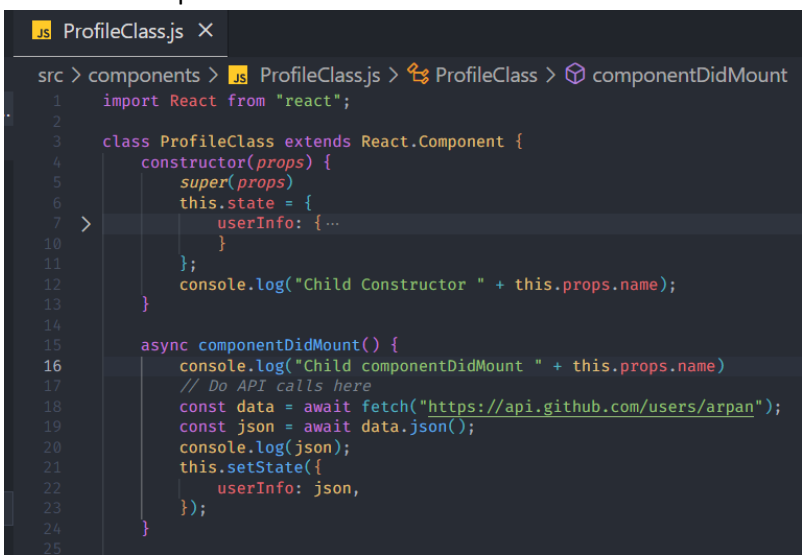
Filter | Default levels

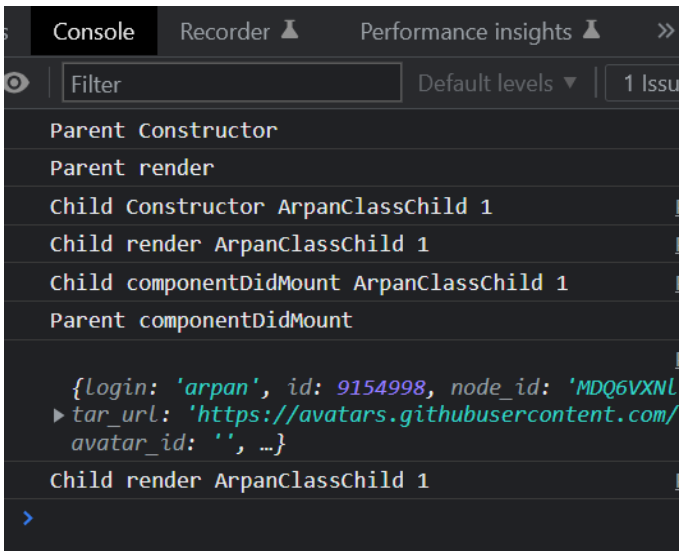
Parent Constructor	About.js:7
Parent render	About.js:16
Child Constructor ArpanClassChild 1	ProfileClass.js:12
Child render ArpanClassChild 1	ProfileClass.js:27
Parent componentDidMount	About.js:12
{login: 'arpan', id: 9154998, node_id: 'MDQ6VXNlcjkxNTQ5OTg=', ava ▶ tar_url: 'https://avatars.githubusercontent.com/u/9154998?v=4',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

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 :-

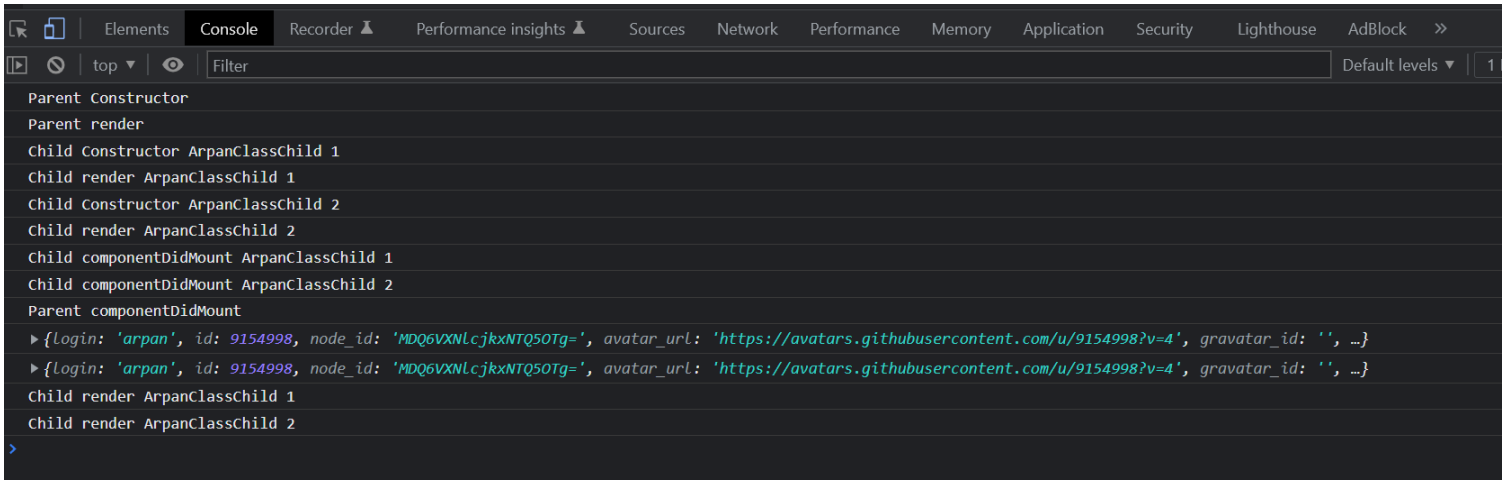


Now, again we just use the “ArpanClassChild 1” and remove the 2nd one in the About.js, which is the parent CBC. However, if we write the code of console logging the “Child componentDidMount” before making the API call like below, then we will get a different output :-

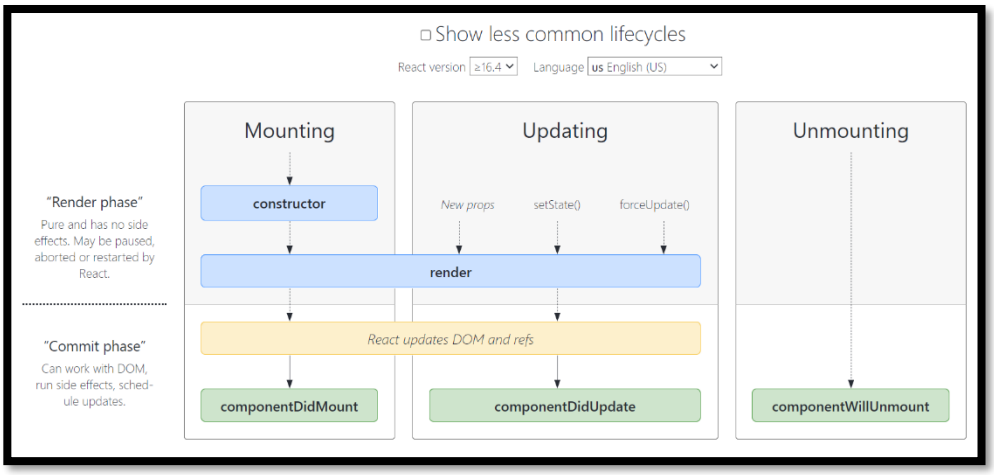




Now, if there are again 2 child CBC in the About.js, which is the parent CBC, with names “ArpanClassChild 1” and “ArpanClassChild 2”, then below will be the output :-



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.

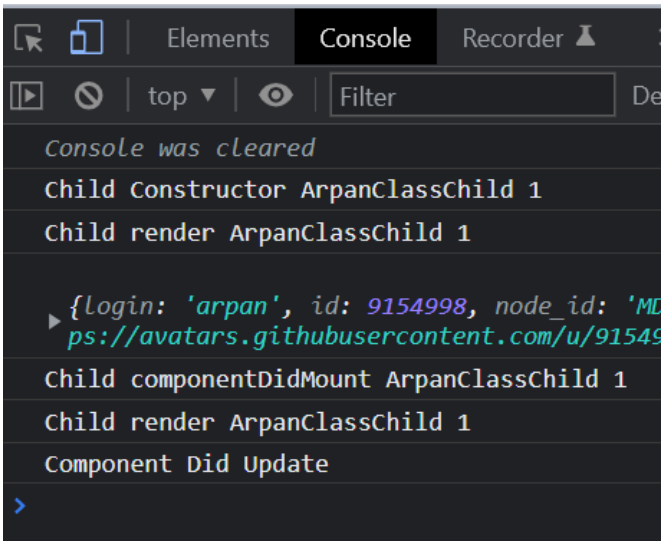
Now, we actually use that data to update an existing component i.e. rendering the component with the help of setState(). So, we enter the updating phase of the diagram.

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.

```
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {···
        };
        console.log("Child Constructor " + this.props.name);
    }

    async componentDidMount() {···
    }

    componentDidUpdate() {
        console.log("Component Did Update");
    }

    render() {
        console.log("Child render " + this.props.name);
        return (···
        )
    }
}

export default ProfileClass;
```

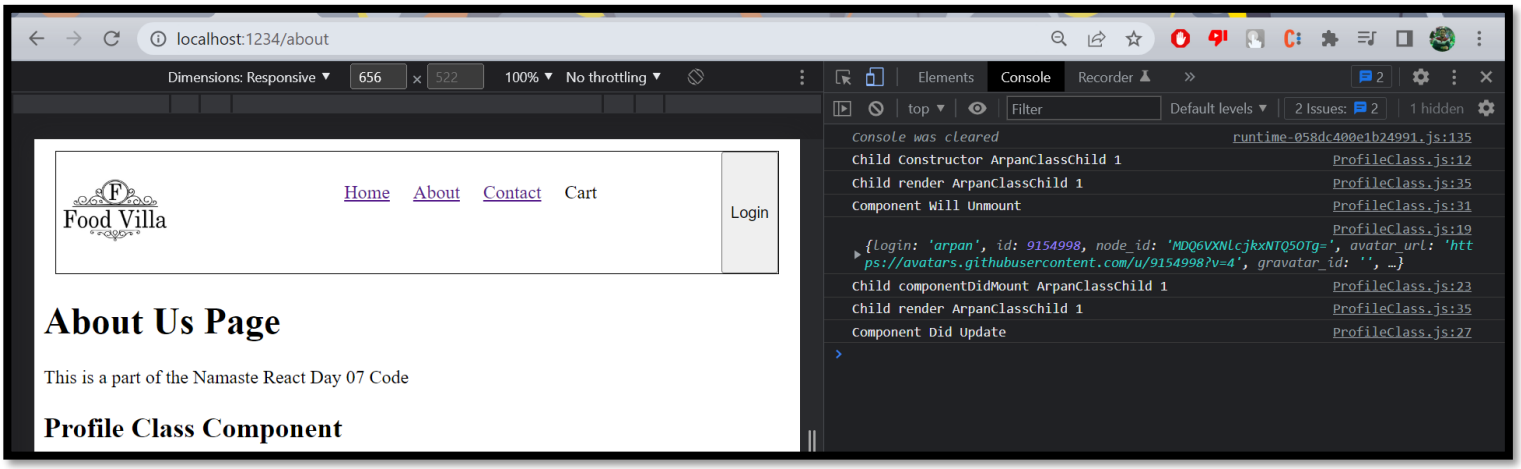


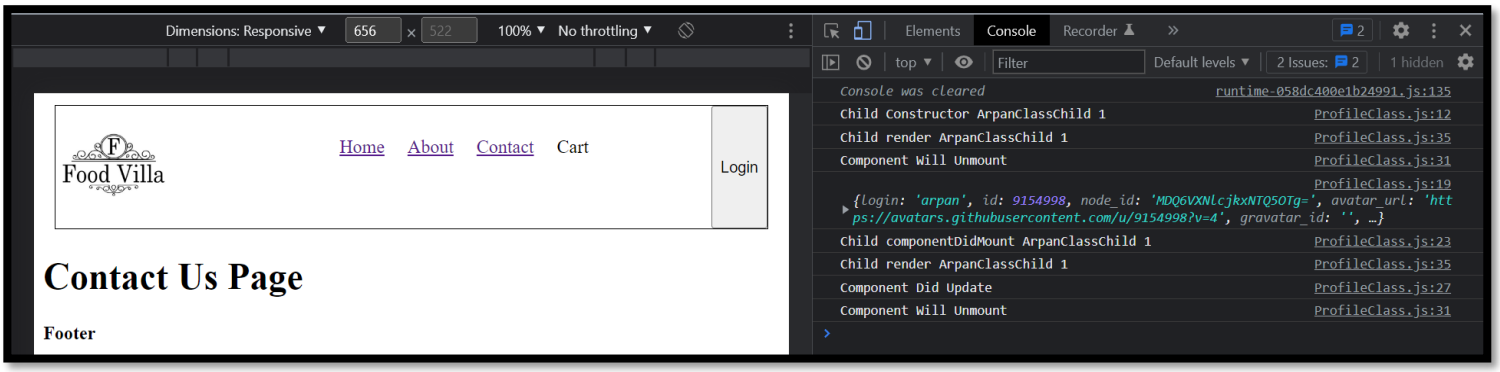
ComponentWillUnmount() Life-Cycle Method :-

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.

```
ProfileClass.js  About.js
src > components > 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 = {···
11          };
12          console.log("Child Constructor " + this.props.name);
13      }
14
15      async componentDidMount() {···
24      }
25
26      componentDidUpdate() {
27          console.log("Component Did Update");
28      }
29
30      componentWillUnmount() {
31          console.log("Component Will Unmount");
32      }
33
34      render() {
35          console.log("Child render " + this.props.name);
36          return (···
45          )
46      }
47  }
```





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 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 leftside 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.

ProfileClass.js

src > components > ProfileClass.js > ProfileClass > componentDidUpdate

```
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            count: 0,
            count2: 0,
            userInfo: {...
            }
        };
        console.log("Child Constructor " + this.props.name);
    }

    async componentDidMount() {...
    }

    componentDidUpdate(prevprops, prevState) {
        if(this.state.count !== prevState.count ||
           this.state.count2 !== prevState.count2
        ){
            // Do something
        }

        console.log("Component Did Update");
    }
```

Profile.js

src > components > Profile.js > Profile

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

const Profile = (props) => {
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(0);
    useEffect(() => {
        // API Call
        console.log("API functional");
    }, [count, count2]);
    console.log("render functional");
    return (
        <div>
            <h2>Profile Functional Component</h2>
            <h3>Name: {props.name}</h3>
            <h3>Count1: {count}</h3>
            <h3>Count2: {count2}</h3>
            <button onClick={
                () => {
                    setCount(1);
                    setCount2(2);
                }
            }>Button</button>
        </div>
    )
};

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 :-

ProfileClass.js

src > components > ProfileClass.js > ProfileClass

```
    async componentDidMount() {...
    }

    componentDidUpdate(prevprops, prevState) {
        if(this.state.count !== prevState.count){
            // Do something
        }

        if(this.state.count2 !== prevState.count2){
            // Do something else
        }

        console.log("Component Did Update");
    }
```

Profile.js

src > components > Profile.js > Profile

```
const Profile = (props) => {
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(0);
    useEffect(() => {
        // API Call
        console.log("API functional");
        // Do something
    }, [count]);

    useEffect(() => {
        // API Call
        console.log("API functional");
        // Do something else
    }, [count2]);
```

Use Case of componentWillUnmount() :-

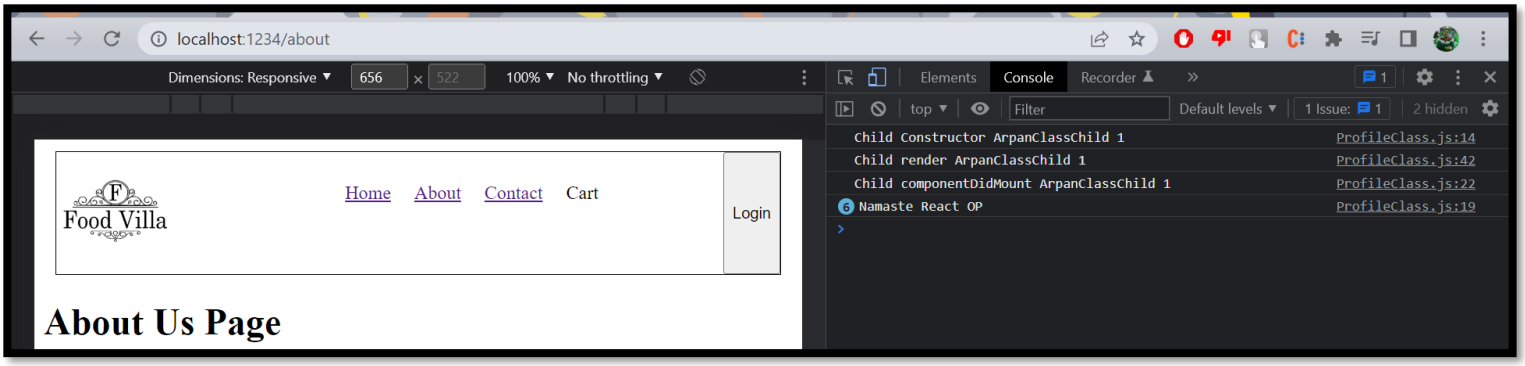
It is called when a component is unmounted. It means when we move from one component to another component, or rather one page to another page (not to be confused with React is a single page application. Here we are just using page to mean a component rather than a webpage). So, when we navigate to another 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

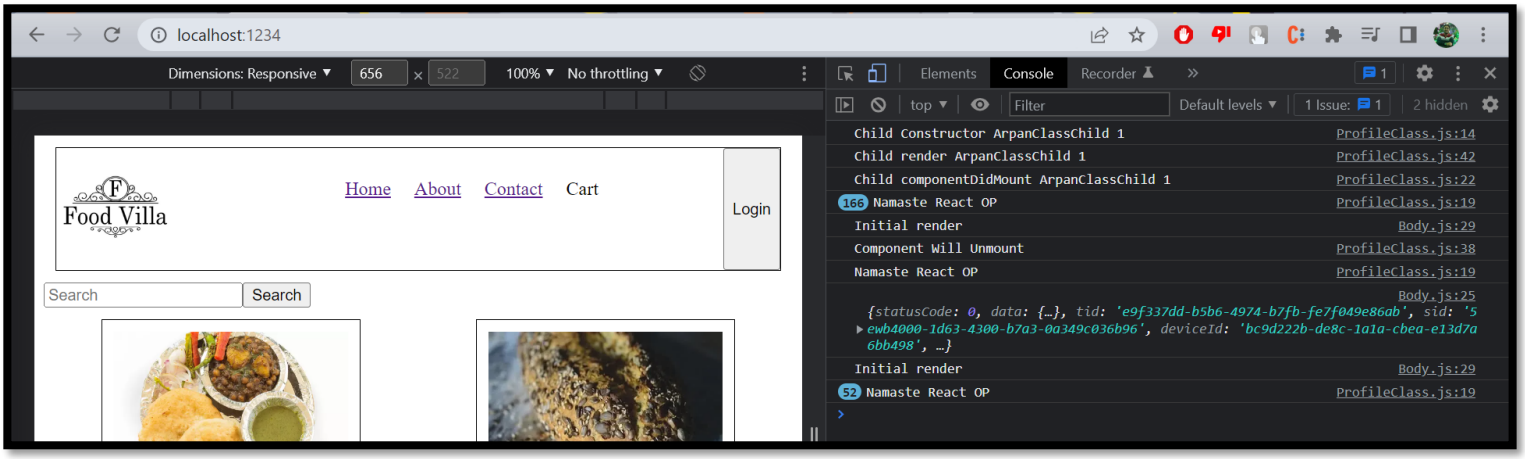
ProfileClass.js

src > components > ProfileClass.js > ProfileClass > componentDidMount

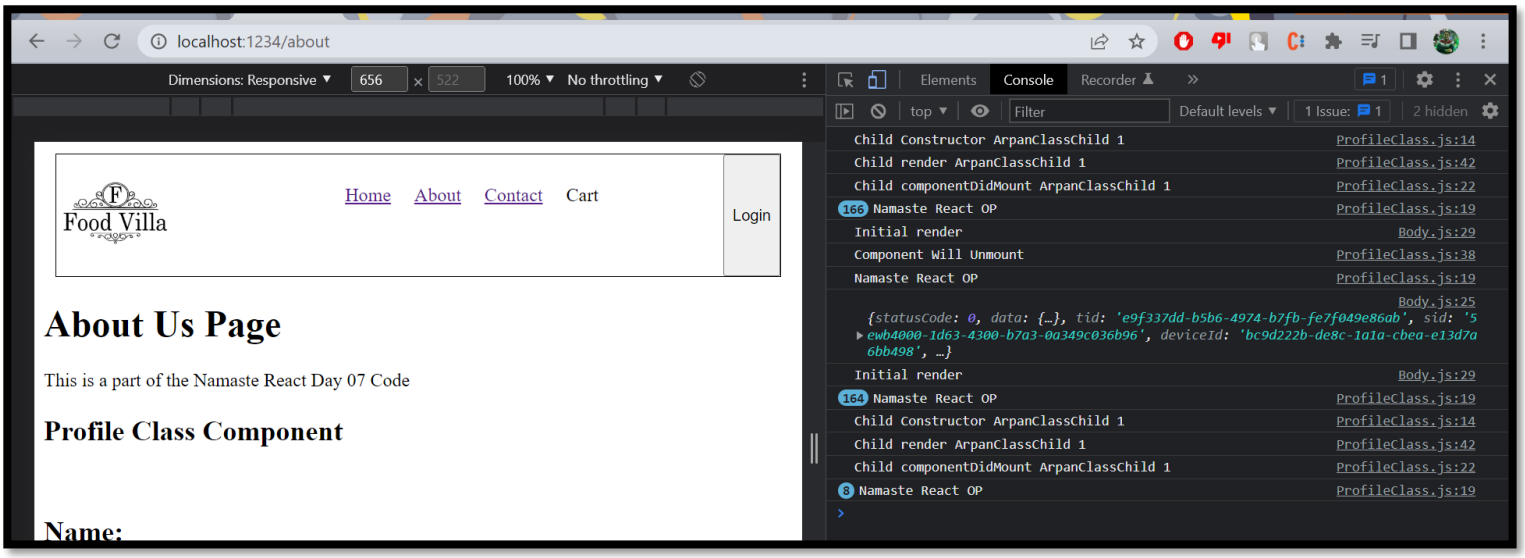
```
import React from "react";

class ProfileClass extends React.Component {
    constructor(props) {...
    }

    async componentDidMount() {
        setInterval(() => {
            console.log("Namaste React OP");
        }, 1000);

        console.log("Child componentDidMount " + this.props.name)
    }
```



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 it's reconciliation process. So, we will still see that statement getting printed in the console.



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 is major downside of single page application

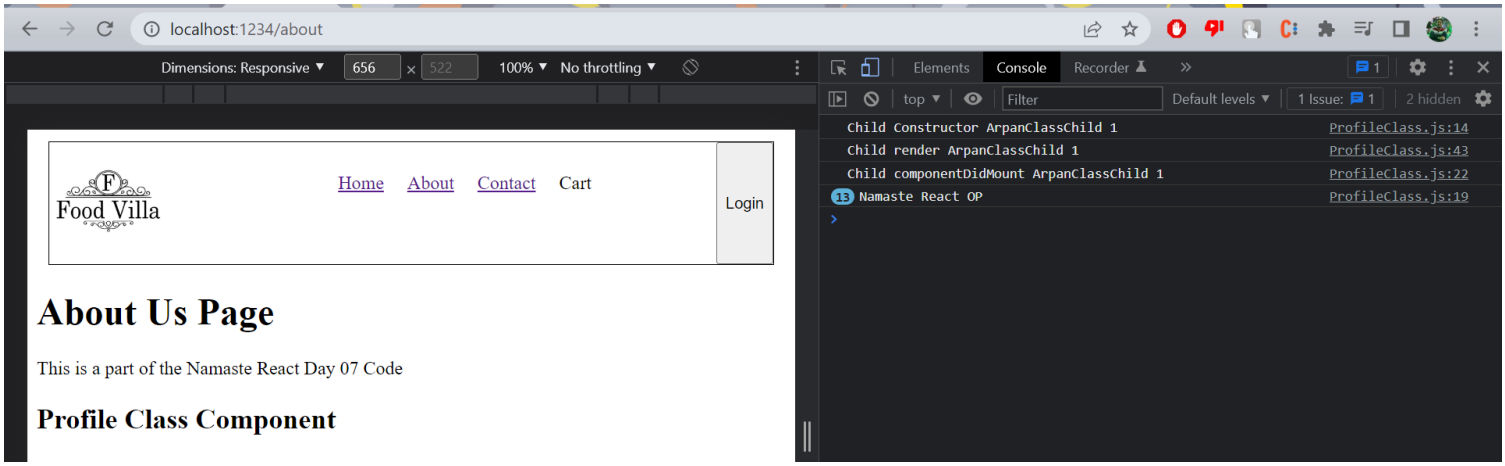
So, when we are changing our page, instead of reloading the page, React is changing the components. So, if we again move to the Contacts/Home page and again navigate back to the About page, the count of that statement getting printed will be 3 in every sec.

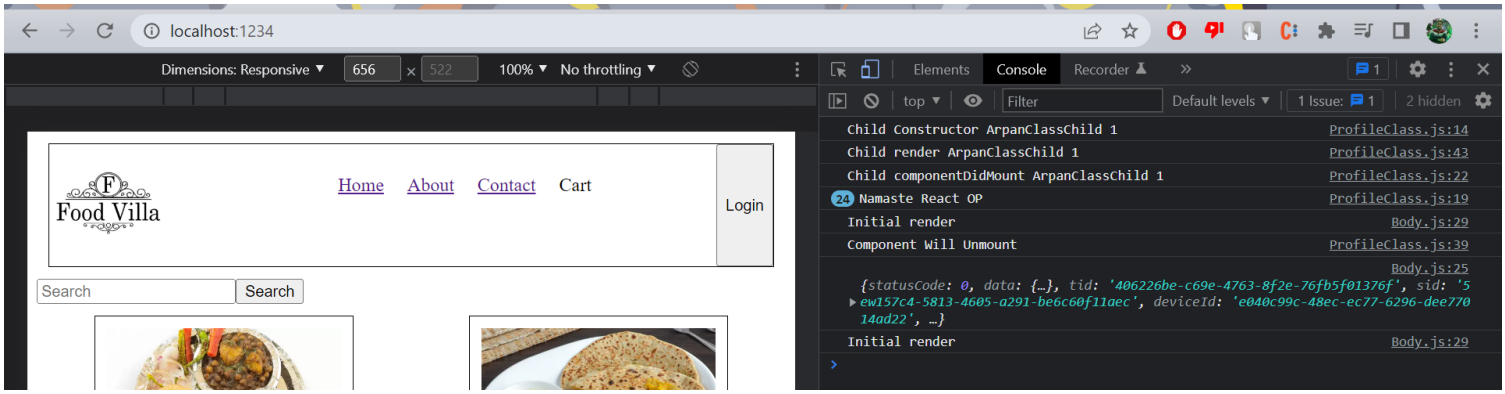
To avoid this, we need to cleanup this functionality. We know that when a component unmount, the componentWillUnmount() will be called and also that to clear a setInterval(), we use a clearInterval(). But, how can we reference the clearInterval() in the componentWillUnmount() to the setInterval() in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

```
ProfileClass.js
src > components > ProfileClass.js > ProfileClass > componentWillUnmount
1   import React from "react";
2
3   class ProfileClass extends React.Component {
4       constructor(props) {...
15      }
16
17      async componentDidMount() {
18          this.timer = setInterval(() => {
19              console.log("Namaste React OP");
20          }, 1000);
21
22          console.log("Child componentDidMount " + this.props.name)
23      }
24
25      componentDidUpdate(prevprops, prevState) {...
35      }
36
37      componentWillUnmount() {
38          clearInterval(this.timer);
39          console.log("Component Will Unmount");
40      }
```

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.



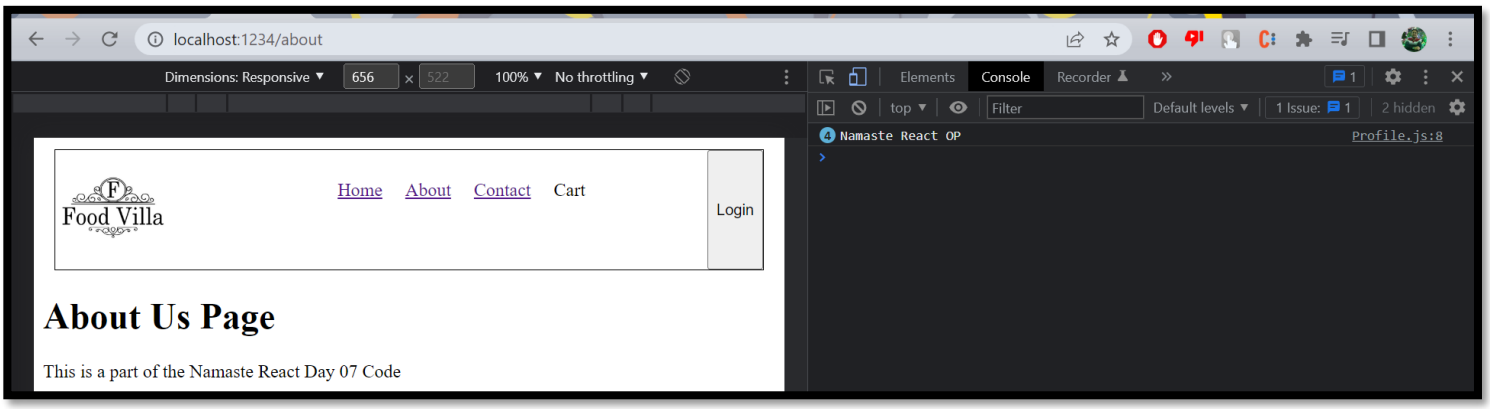


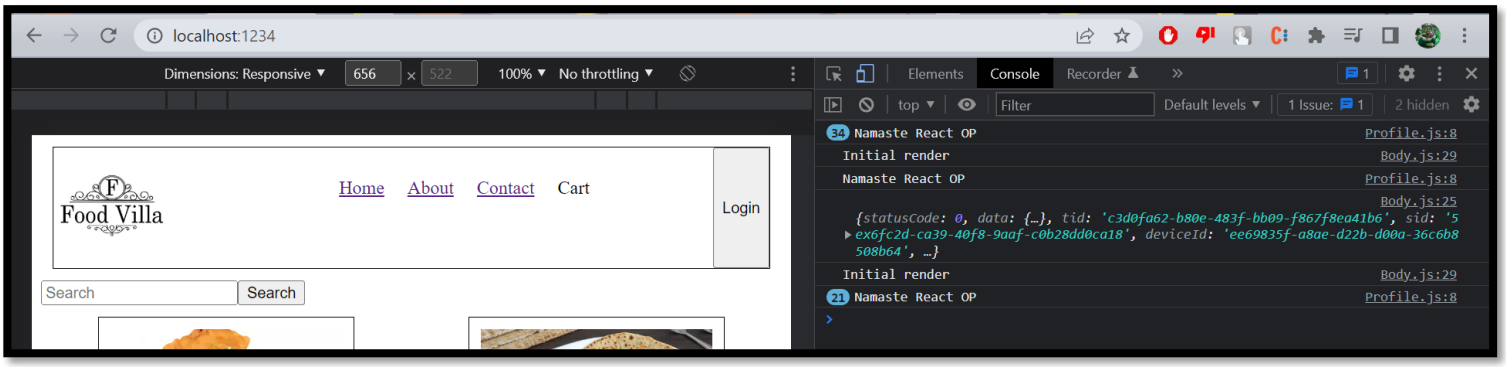
Performing cleanup in the Functional Component :-

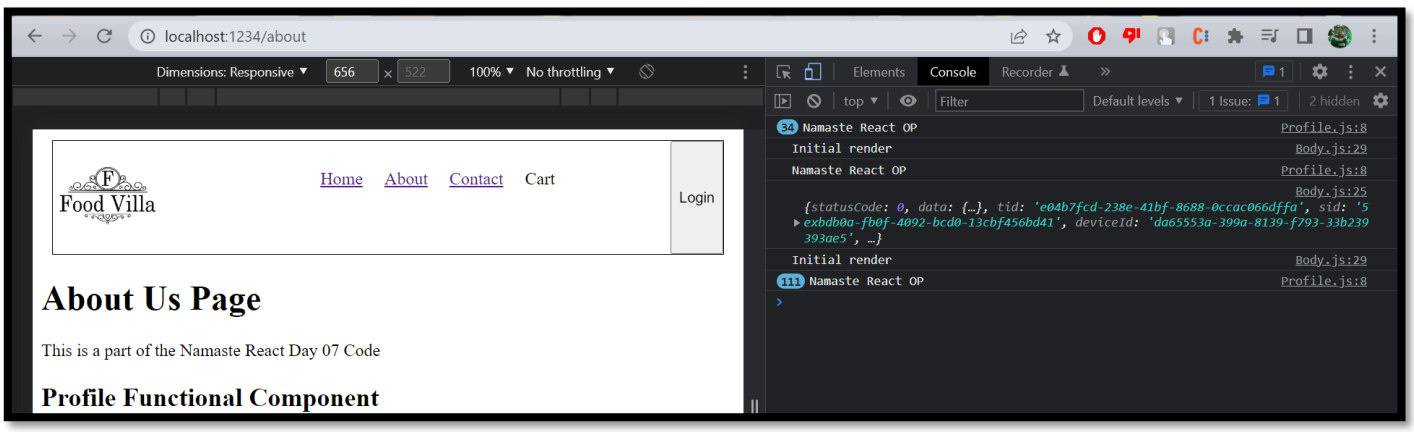
Just like in the CBC, if we write a setInterval() in useEffect() of a FC, then it will get executed and never stop. The 1st image is the code, the 2nd is the output in console when we reload the About page. The 3rd image is when we navigate to the Home Page and you can see that the statement is still 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.

```jsx
// About.js
import Profile from "./Profile";
import { Component } from "react";

class About extends Component {
    constructor(props) {
        super(props);
        // console.log("Parent Constructor");
    }

    componentDidMount() {
        // Do API calls here
        // console.log("Parent componentDidMount")
    }

    render() {
        // console.log("Parent render");
        return (
            <div>
                <h1>About Us Page</h1>
                <p>This is a part of the Namaste
                React Day 07 Code</p>
                <Profile name="ArpanClassChild 1" />
            </div>
        )
    }
}

export default About;
```

```jsx
// Profile.js
import { useEffect, useState } from "react";

const Profile = (props) => {
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(0);
    useEffect(() => {
        setInterval(() => {
            console.log("Namaste React OP");
        }, 1000);

    }, []);

    return (
        <div>
            <h2>Profile Functional Component</h2>
            <h3>Name: {props.name}</h3>
            <h3>Count1: {count}</h3>
            <h3>Count2: {count2}</h3>
            <button onClick={...
            }>Button</button>
        </div>
    )
};

export default Profile;
```





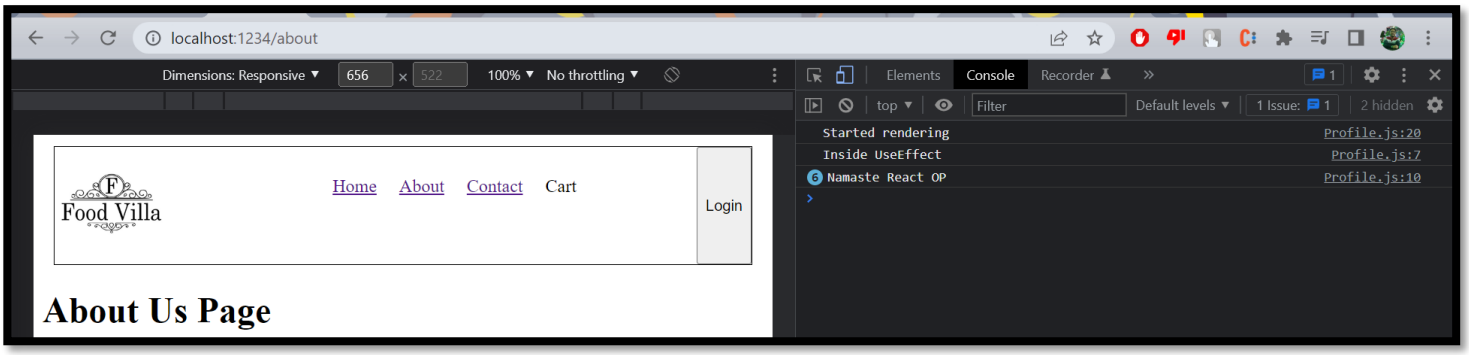


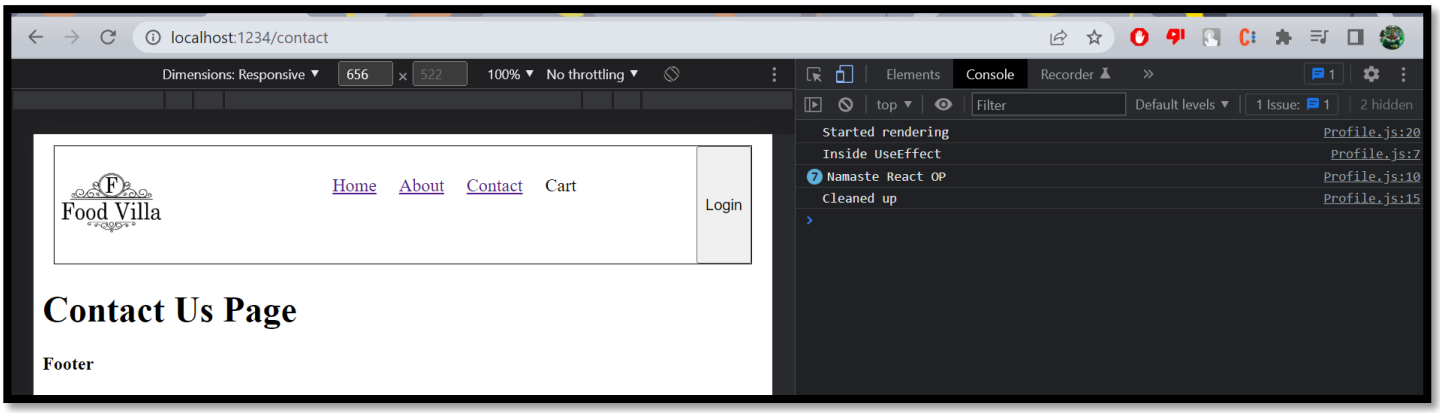
Till now we know that the callback of the useEffect function does not return any statement. However, it can return only 1 type of function that will get executed once the component is unmounted. That function is called a **cleanup function.**

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

```
const Profile = (props) => {
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(0);
    useEffect(() => {
        console.log("Inside UseEffect");

        const timer = setInterval(() => {
            console.log("Namaste React OP");
        }, 1000);

        return () => {
            clearInterval(timer);
            console.log("Cleaned up");
        }

    }, []);

    console.log("Started rendering");
    return (···
    )
};

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 ?

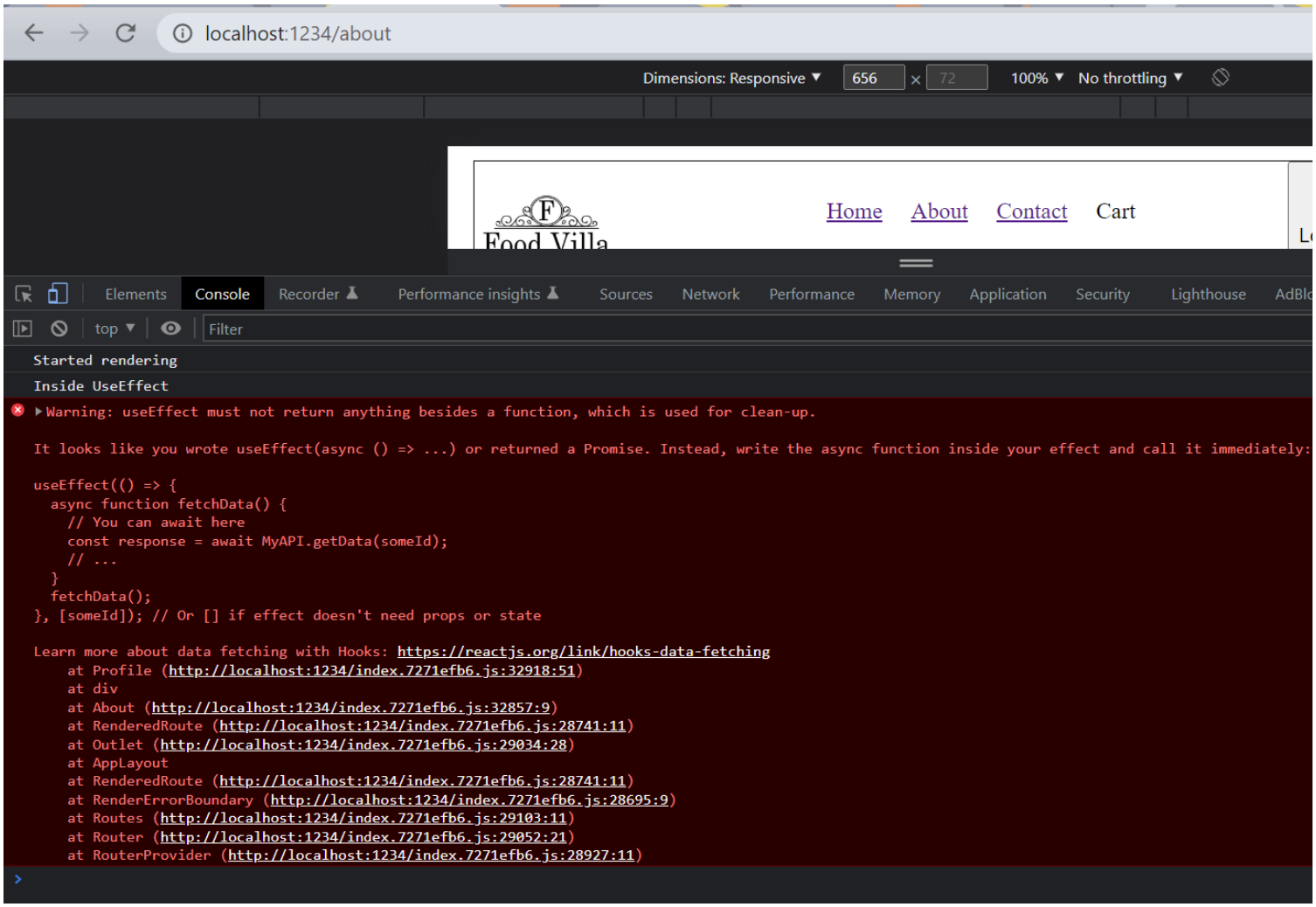
Before knowing the reason, lets make an useEffect()'s callback an async function and see what error or warning React throws us. For this example, I am just making the useEffect()'s callback in the Profie FC as an async function.

Profile.js

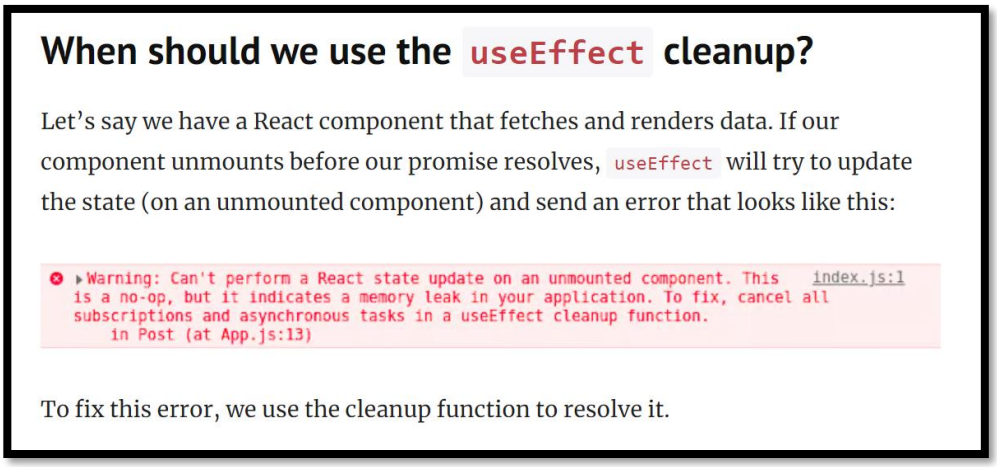
src > components > Profile.js > default

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

const Profile = (props) => {
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(0);
    useEffect(async () => {
        console.log("Inside UseEffect");
    }, []);

    console.log("Started rendering");
    return (···
    )
};

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 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 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=5gCtW7RCtQA&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)