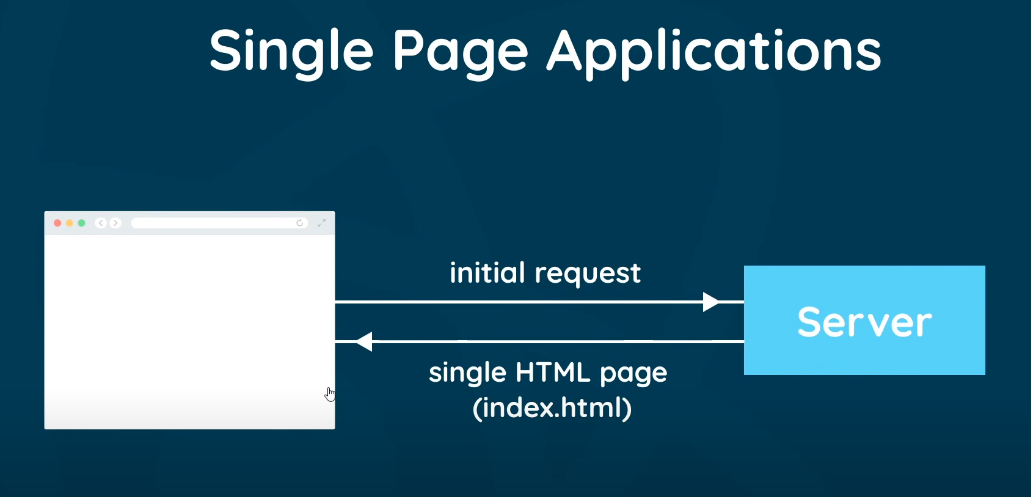
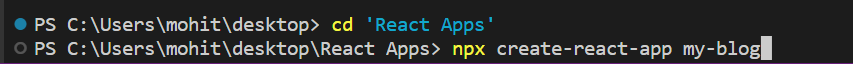
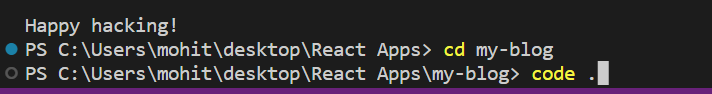
React



Command to create react app:



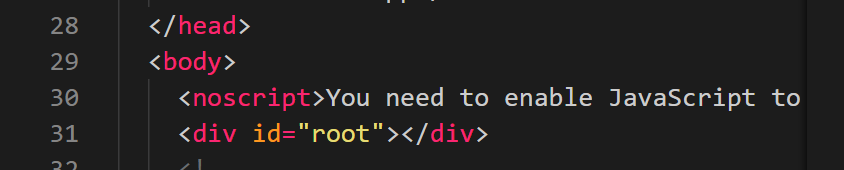
How to open newly created app in code:



Node Modules: All the libraries installed.

Public Folder: Files public to the browser.

Index.html: File that is served to the browser. All of our code is injected to this html file.

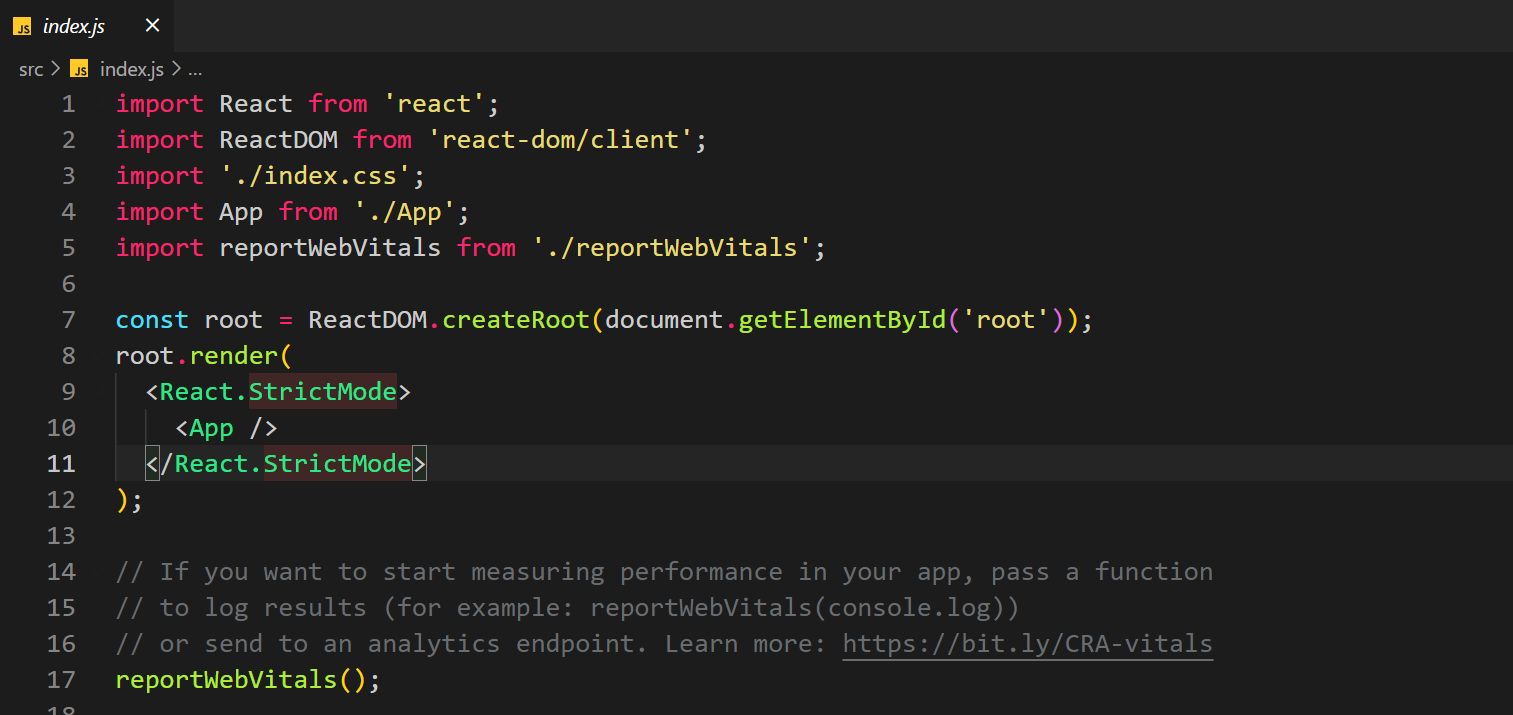


This **div id = “root”** is where all of our code is injected.

Src Folder: All the code we write. Components etc.

App.js: This is a component.

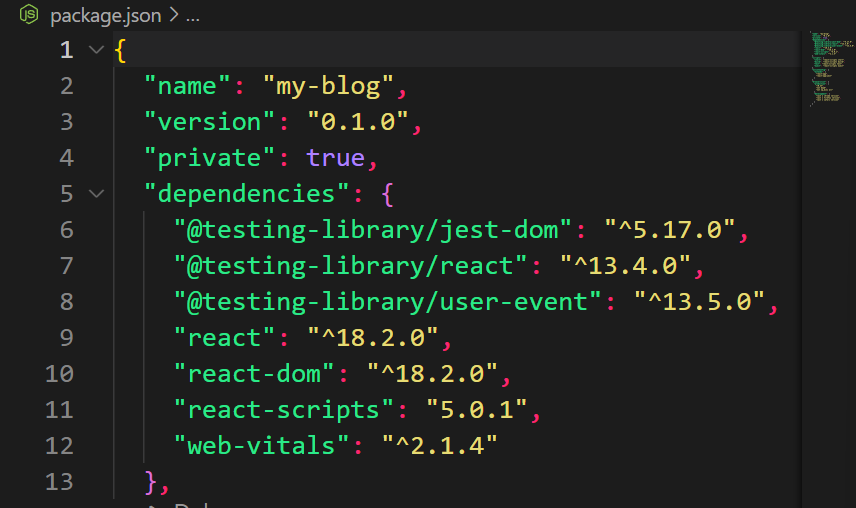
Index.js: Kickstarts our app. This takes all of our components and mounts it to index.html

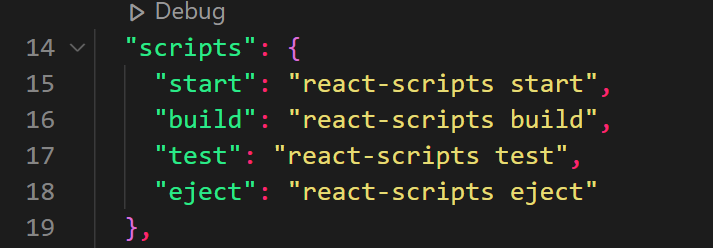


Here the app component is rendered on the element which has the id = ‘root’ in the index.html

<React.StrictMode> means react will do additional checks during dev and show warnings in the console.

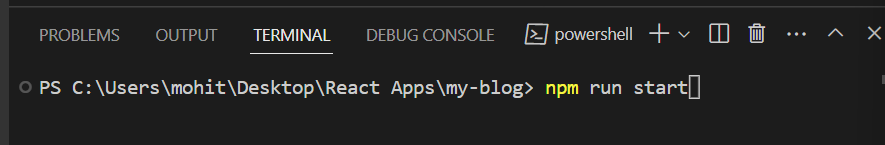
Package.json: Lists all the dependencies of the project



Scripts: 

This are scripts you can use to start, build (for prod), test etc the react app.

Running react app:



Remember start is a script which is used to start the react app.

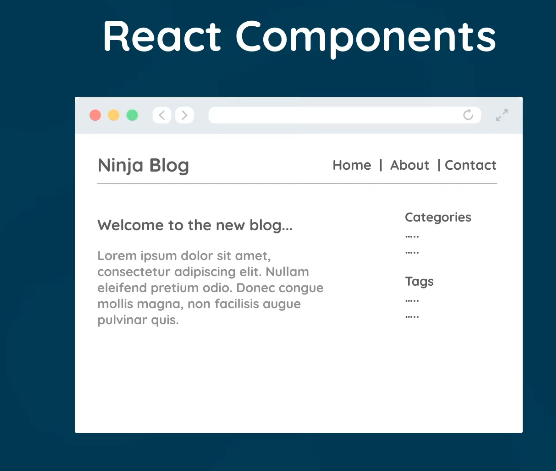
React app starts at: <http://localhost:3000/>

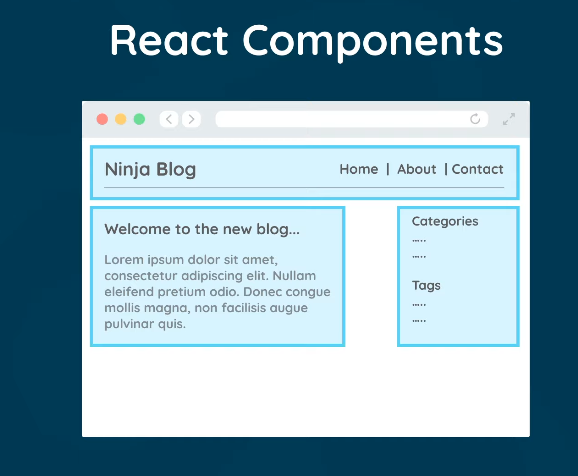
To install node modules, run this command:

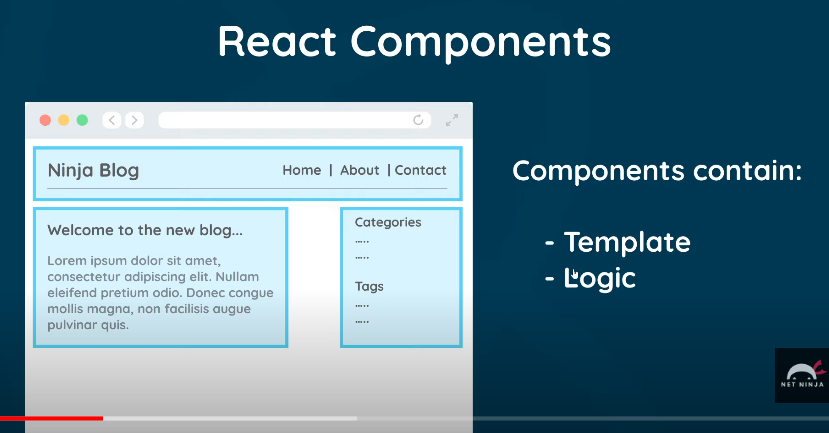
**Npm install or npm i**

This will look into package.json folder and install all of the dependencies mentioned there.

React Components:







App component is the root component.

The template of the react component is not HTML but JSX.

It is similar to html but not exactly HTML.

A transpiler called babel converts all of these JSX into HTML when we save the file.

And then HTML is rendered to the browser.

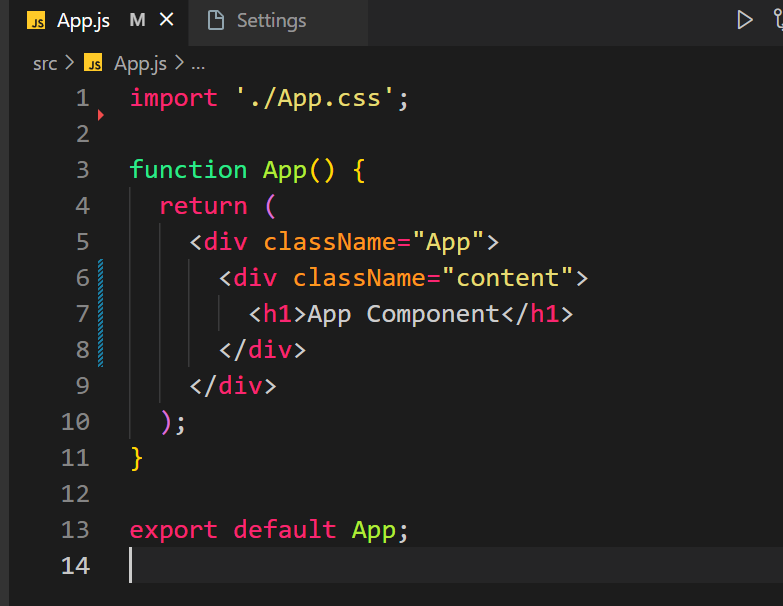
**Adding css class in html:**

**<h2 class = “abc”></h2>**

**Adding css class in JSX:**

**<h2 className = “abc”></h2>**

App Component:

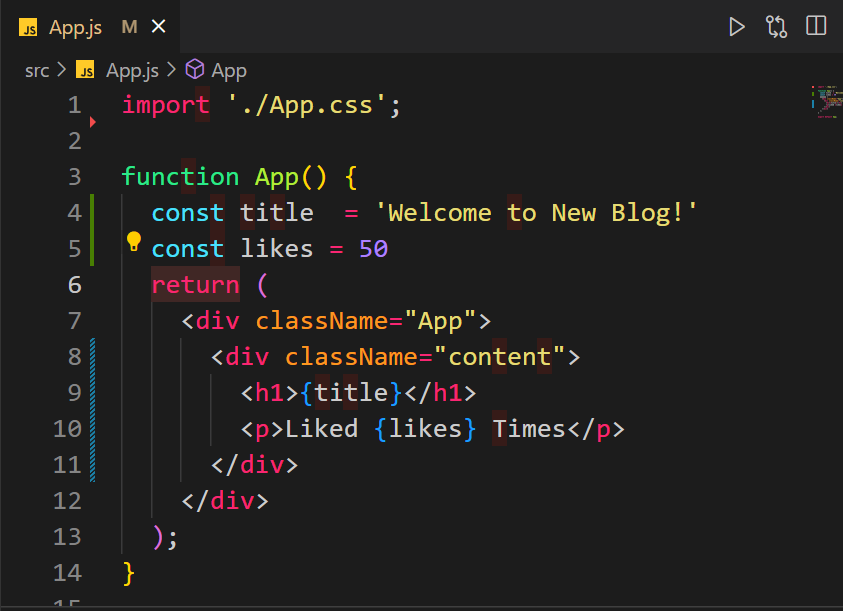


See that we are exporting it at the end.

Also, the component created using a function must return some amount of JSX.

See that the return has some JSX in it which is the template JSX of this component.

You can show JS vars in the html like this:





Use {}.

We can’t do this with Boolean and JS objects. Arrays you can do with this method.

You can also write JS directly inside {}:



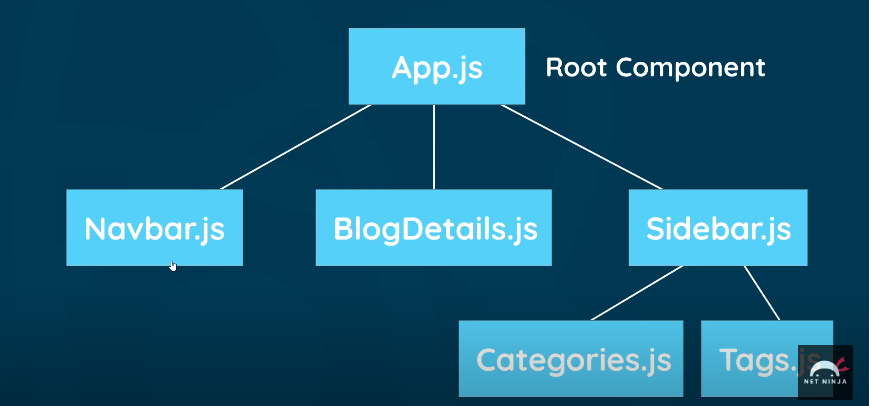
Gives same output.

Everything will be printed as a string in HTML.

You can also use this method to put dynamic values in the attributes of the tags:



Component Tree:

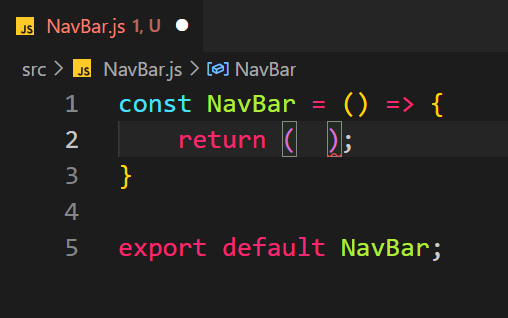


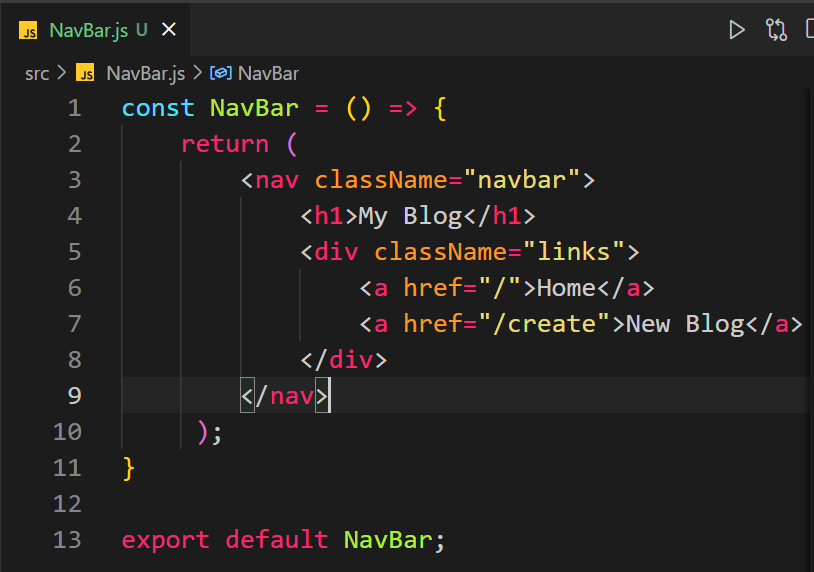
Creating a component:

Create a new file and call it Navbar.js

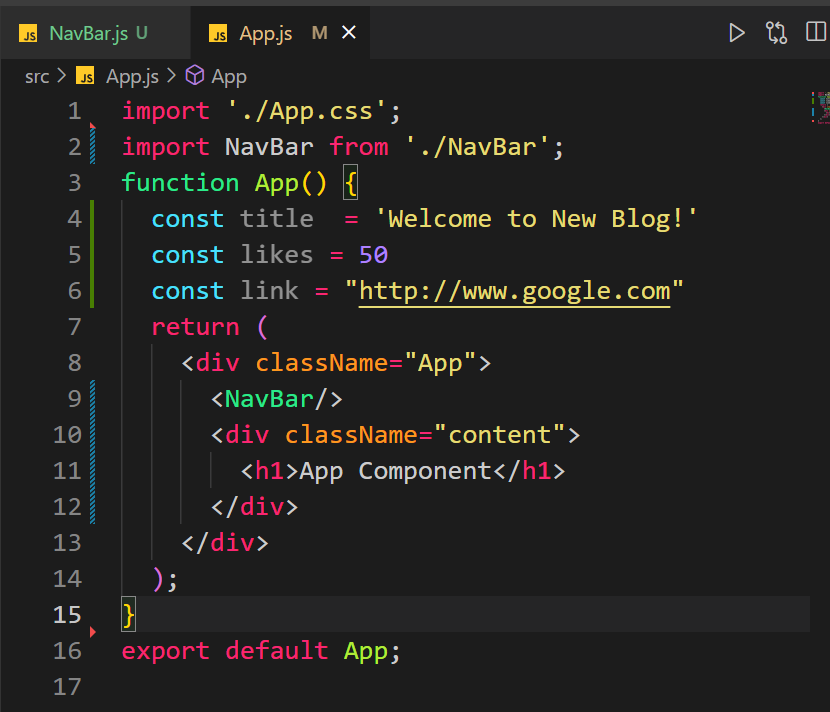
Once created, go to this file and **type sfc then press tab**, this will create a new component.

Also, you need react snippet extension installed on VS code for this.





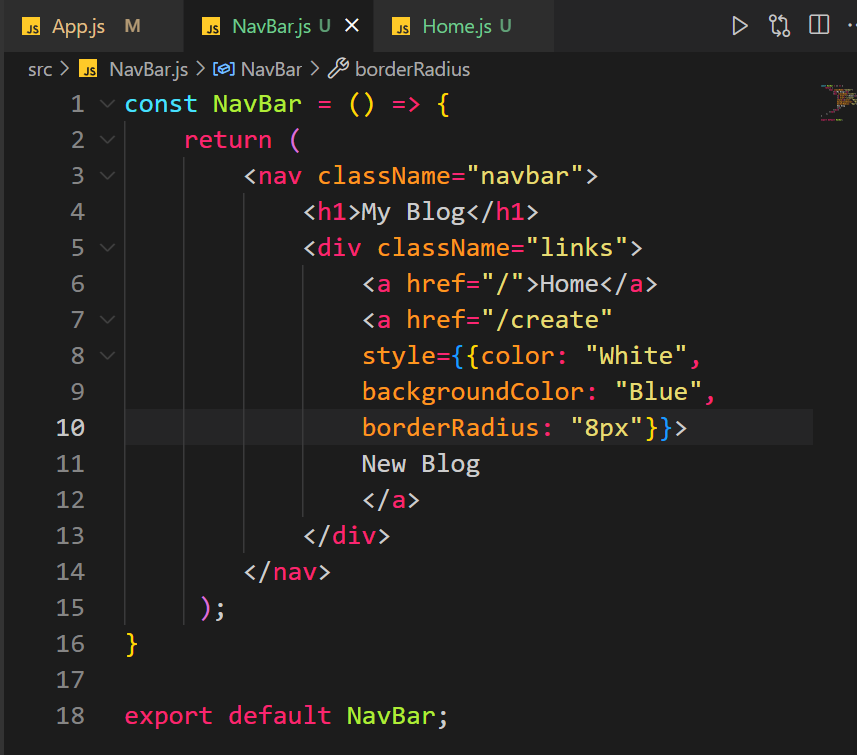
Now import this component to the component (in this case it is app component) where you want to use it.



We can add our styles to the index.css because this is applied to the index.html file.

And this the html file which holds all of the components.

How to apply inline styles:

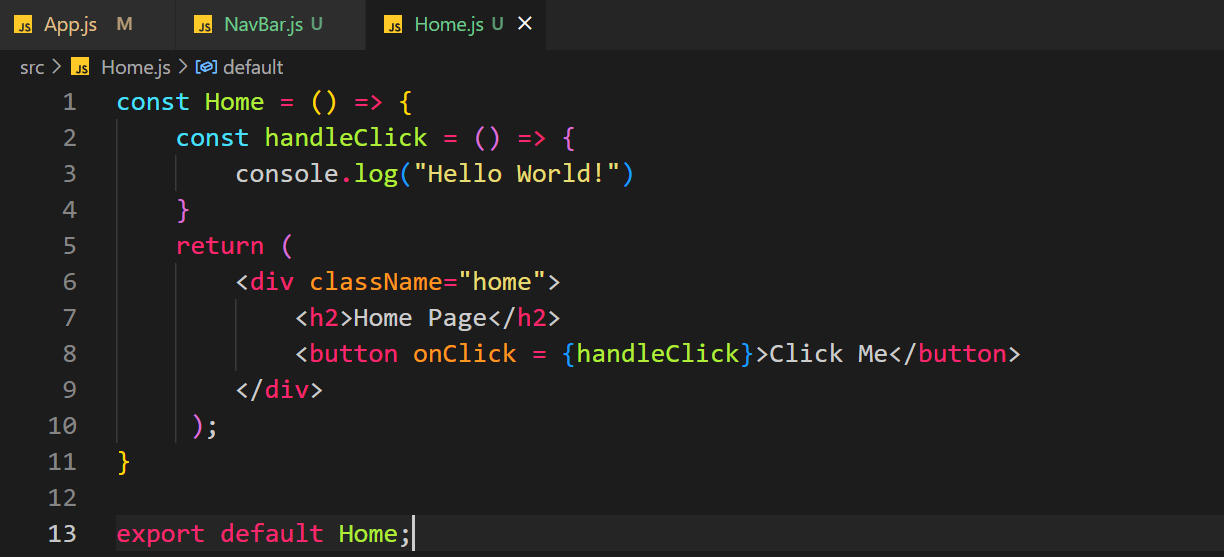


Style = {{}}, two brackets because the style will be given as a js object. Also, notice how the name of css properties is different from the pure css.

Background-color becomes backgroundColor and so on.

All the properties will lose the – and the next word’s first char will be capitalized.

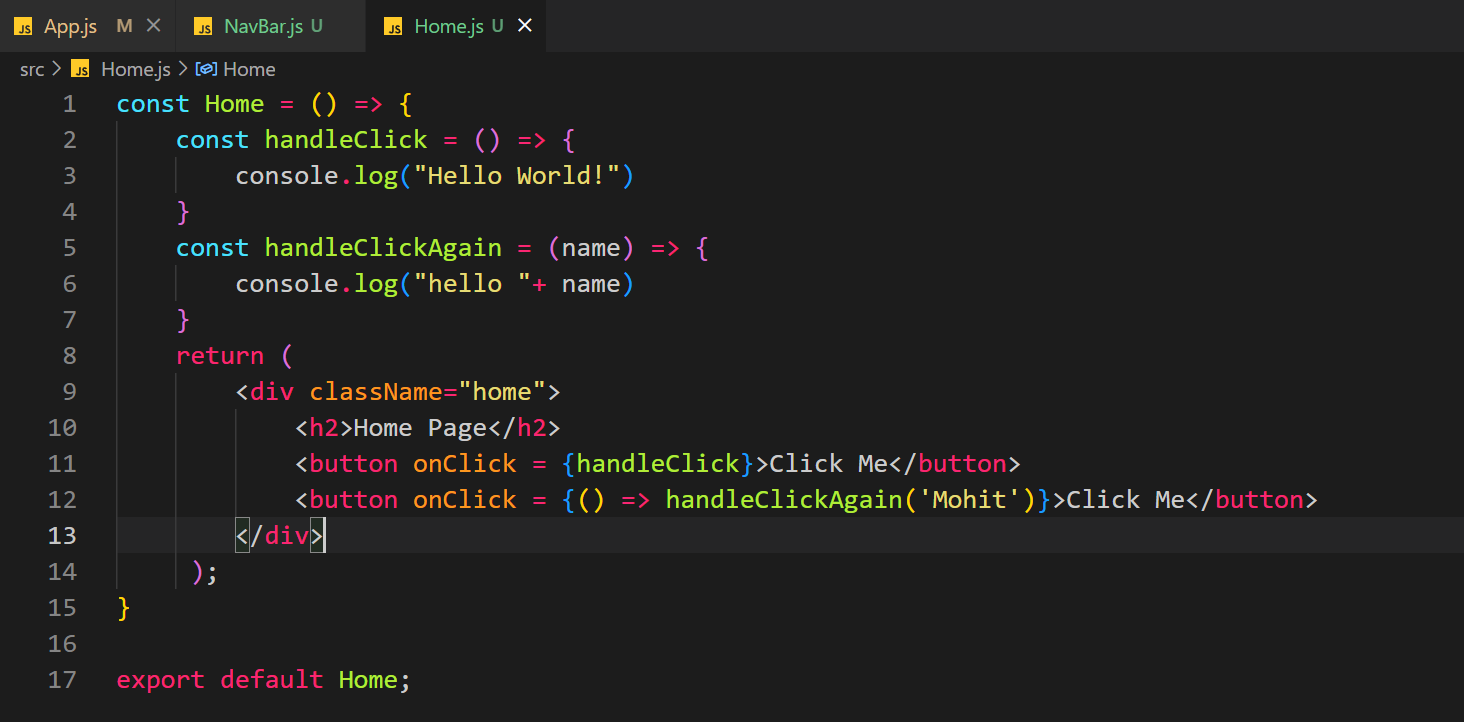
Eg border-radius will become borderRadius.

Reacting to a button click or any event: 

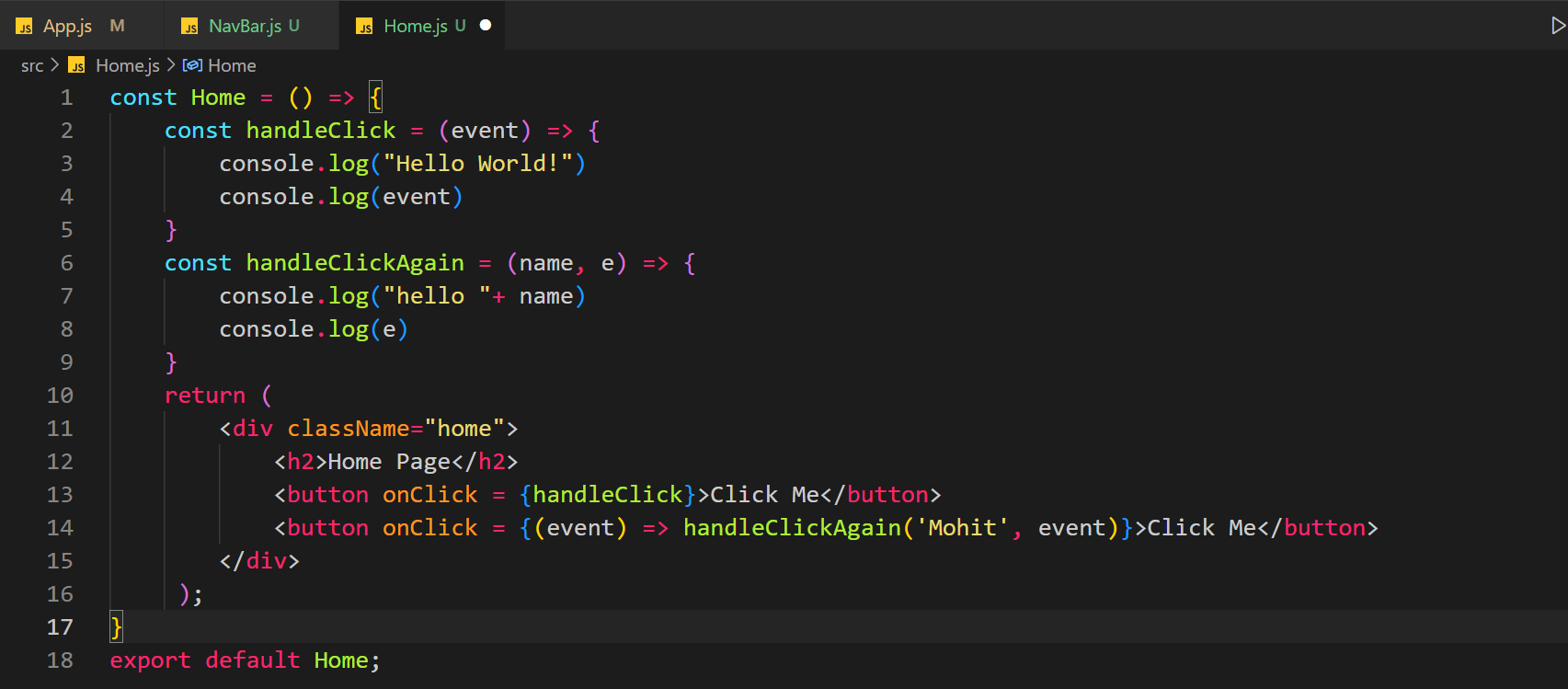
Just pass the reference of the function. Don’t pass () with the function otherwise it will be called even when the button is not clicked.

Passing argument:

If we want to pass arguments we have to pass an anonyms function like this:



The event parameter: When we call a function for any event the function will automatically get an event object.



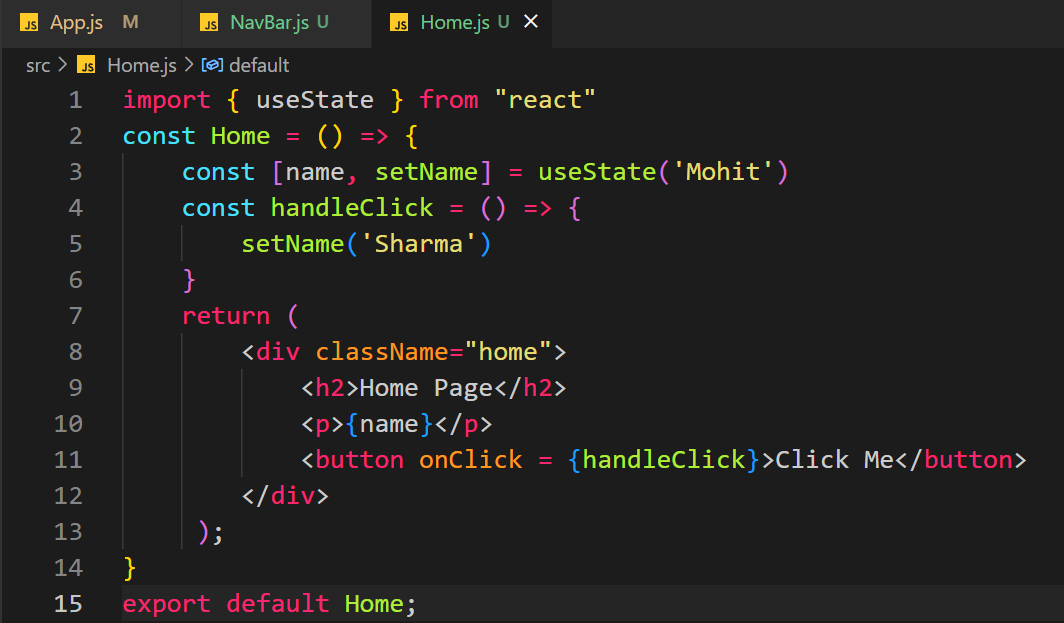
For the first function the event is automatically passed and we can use it inside the fuction. For the second function we pass it from the anonyms function.

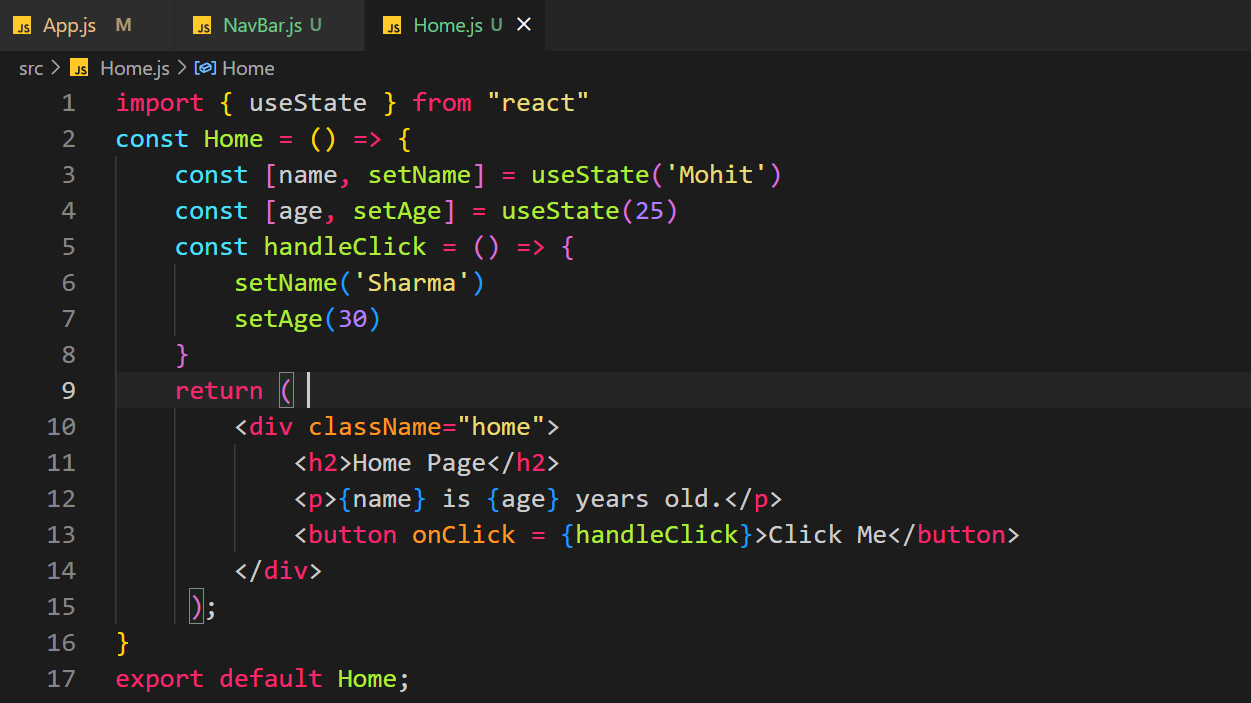
State: State are the variables/data which we want react to capture when changed.

When something is changed in the DOM, we want react to know about this change. This is called a state.

For state changes, we use a hook: useState()

Hooks start with **use.**

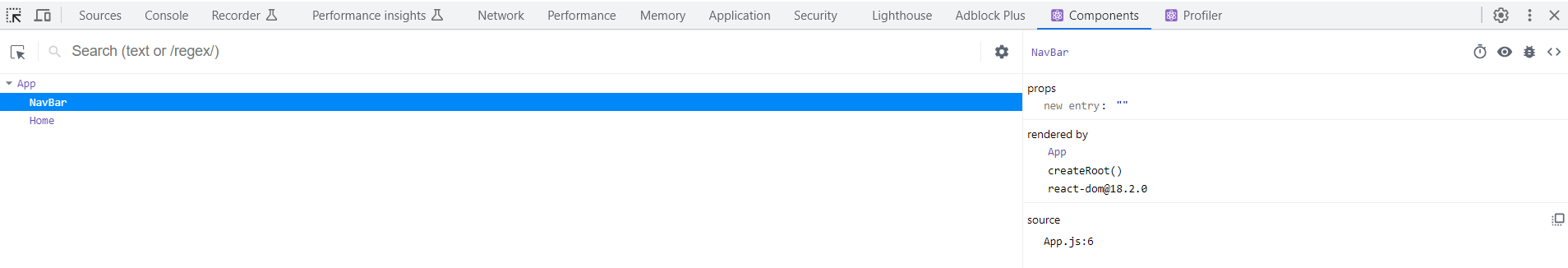
First import the hook. 



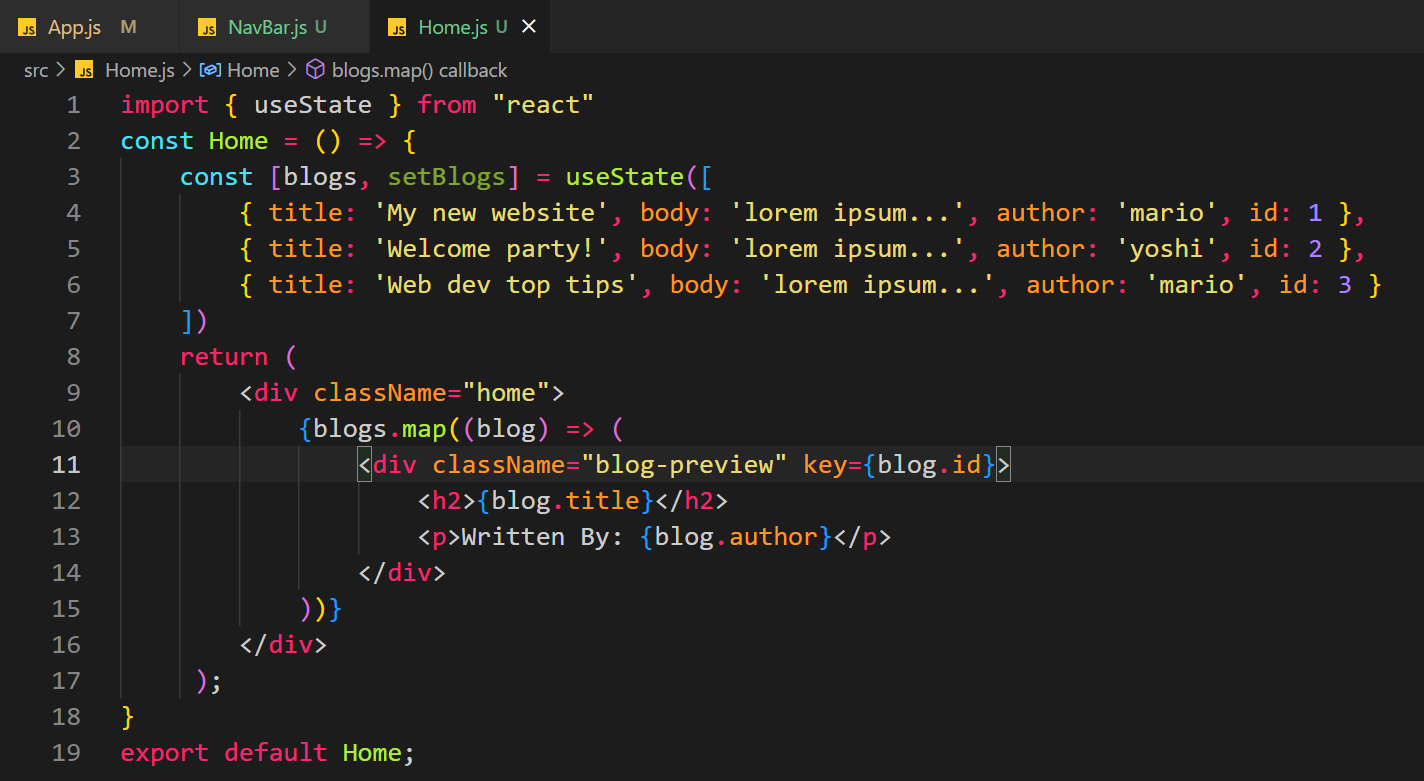
Add React Developer Tools extension for chrome.

Use this link:

[React Developer Tools extension](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi/related)



How would be output a list:



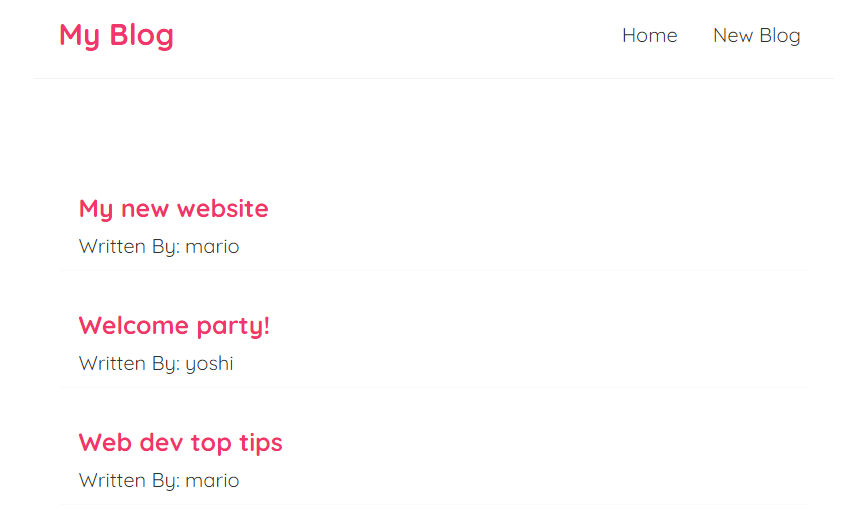
We want to output a list of blog titles and their authors.

Here we have a state blog which is a list. This list can be populated through an API.

Now when we want to output this list. We will use a map function of JS.

The map function will take every item from the list; blogs in this case, and then apply the html for every item. Finally, this html for every list item will be shown on browser.

If we use this map method, we must have some sort of primary key in our list, in this case it is ID, this we will pass to the parent element of html inside the map, in this case it is div.



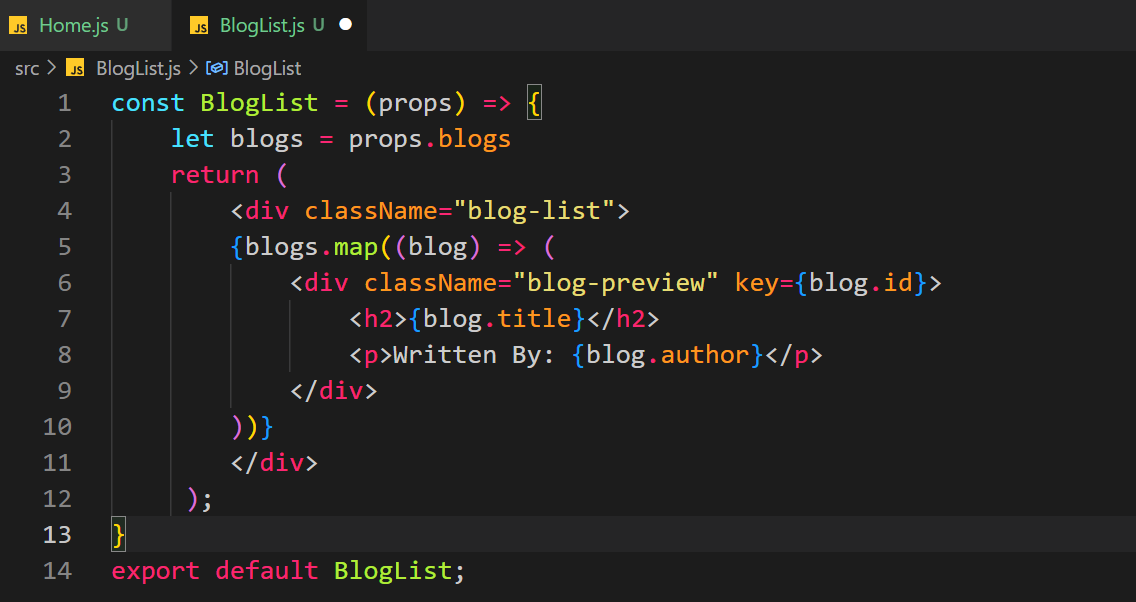
If your code is getting use again and again and will be used in different components,

Just create a new component for this code.

Props: Data is passed from a parent component to child component through props.

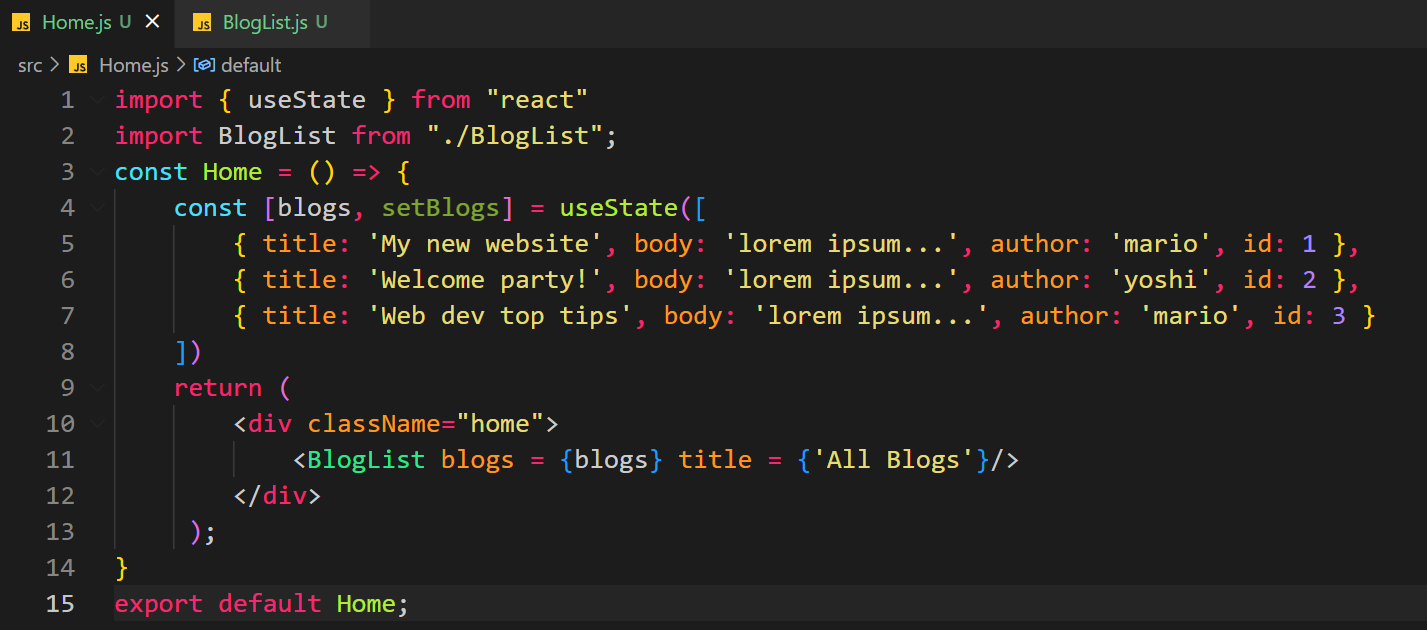
Let’s create a new component: BlogList which will show all the blogs in our list.

Let’s pass blogs list from Home(parent) to BlogList(child) component.

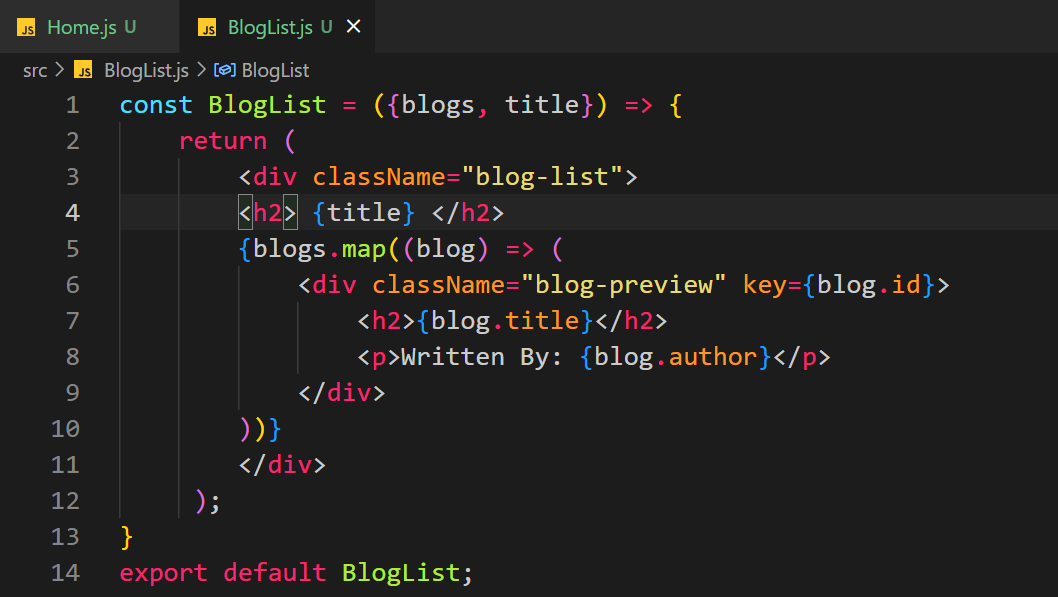
 

You can pass any many props you want from the parent component.

Let’s pass multiple props:



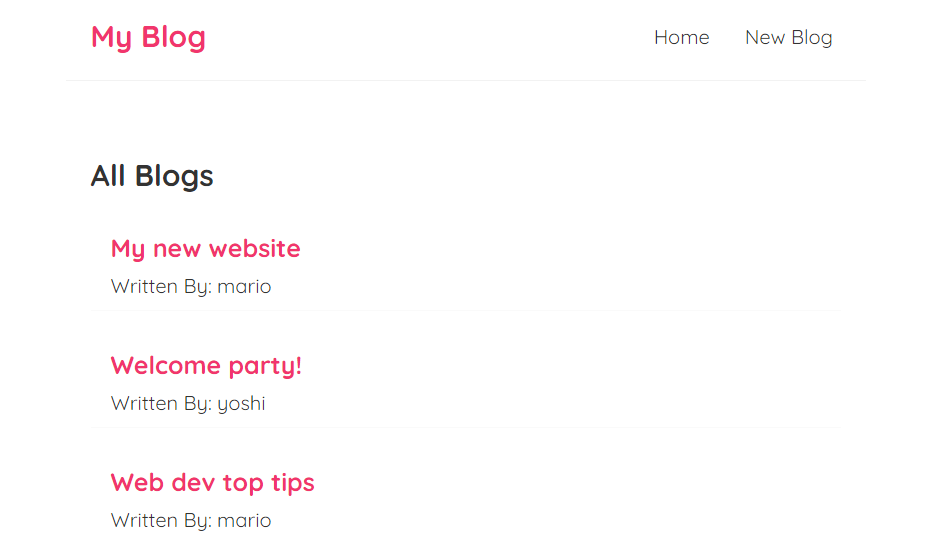
We are passing blogs list and title string as props to BlogList component.



Here we are accepting blogs, title directly in the (), this is called destructuring.

Once done we are showing the tile at h2 and then using map we show the list of blogs.

Output:



Lets say we now want to only show blogs from the author ‘mario’.

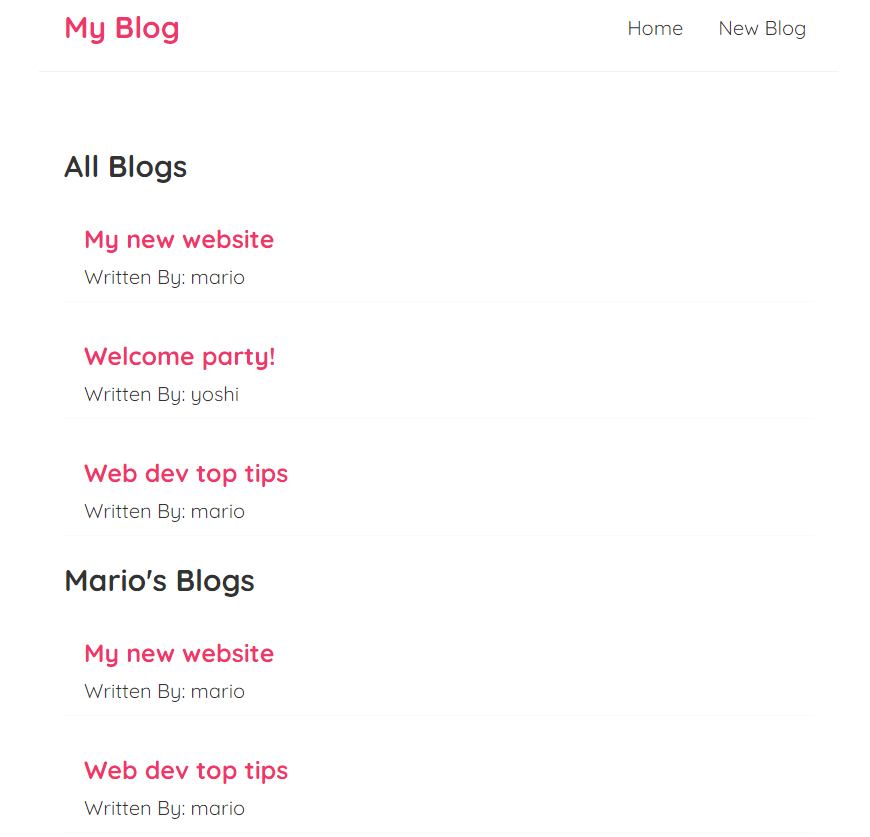
Lets reuse the BlogList component to do just that:



Instead of passing the whole list we will pass a filered list containing the blogs where the author is ‘mario’.

We don’t need to change anything in the BlogList component.

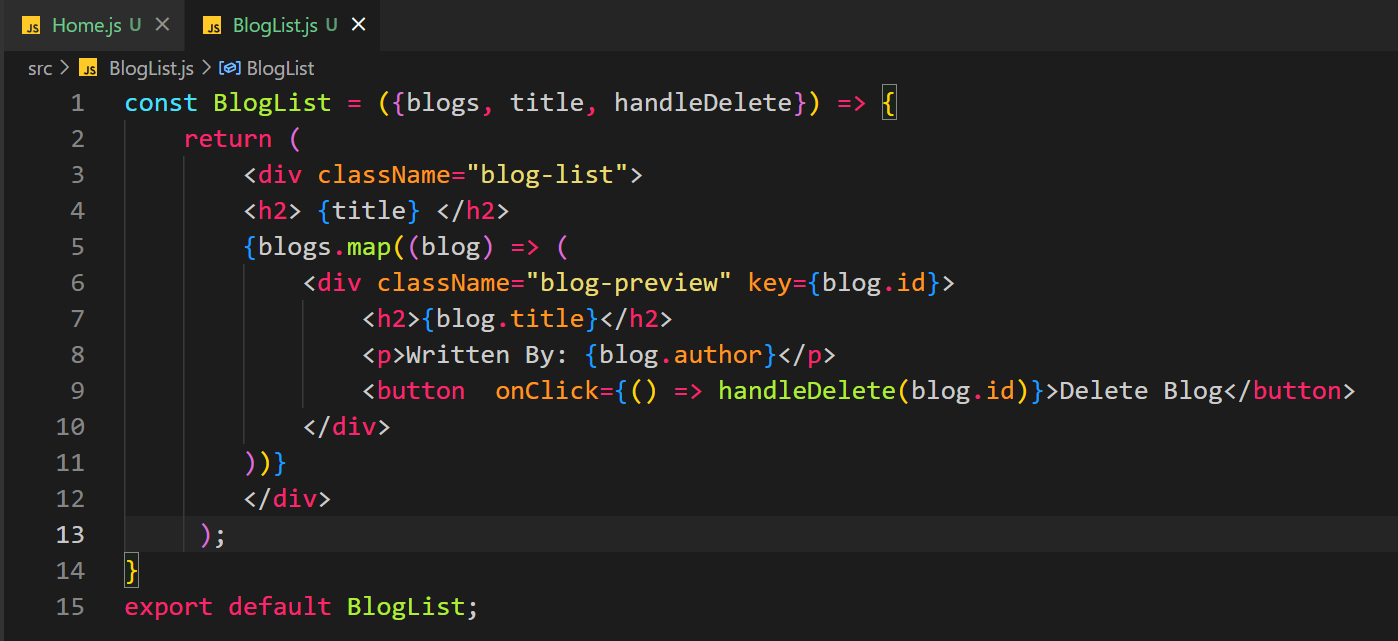
See the output:



Passing Function as a prop:

We now want to delete a blog:

We will create this handle delete function in the home component and from there we will pass it to the blogList component.

useEffect hook:

When state is updated, DOM is refreshed.

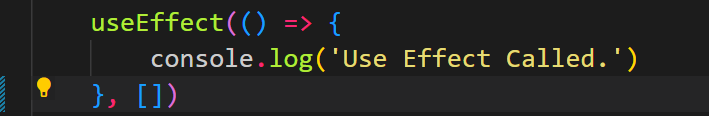
First import: 

useEffect function will be executed when the component first loads and after that every time the state is changed.

When the state is changed, useEffect is called. If you change the state inside the useEffect function. Then when useEffect is called then the state will change which will cause a rerender, which will call the useEffect. It will cause a infinite loop.

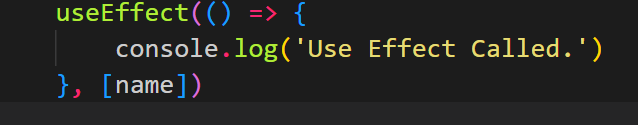
If we want to rerender only on certain changes then we will use dependencies array with useEffcet:

1. If you pass empty array as a second argument then it will only run on initial render. It will not run after that.



1. If you put some states in the dependecy list then useEffect will be called when you first load and also when any of the states provided in the dependecy list changes.





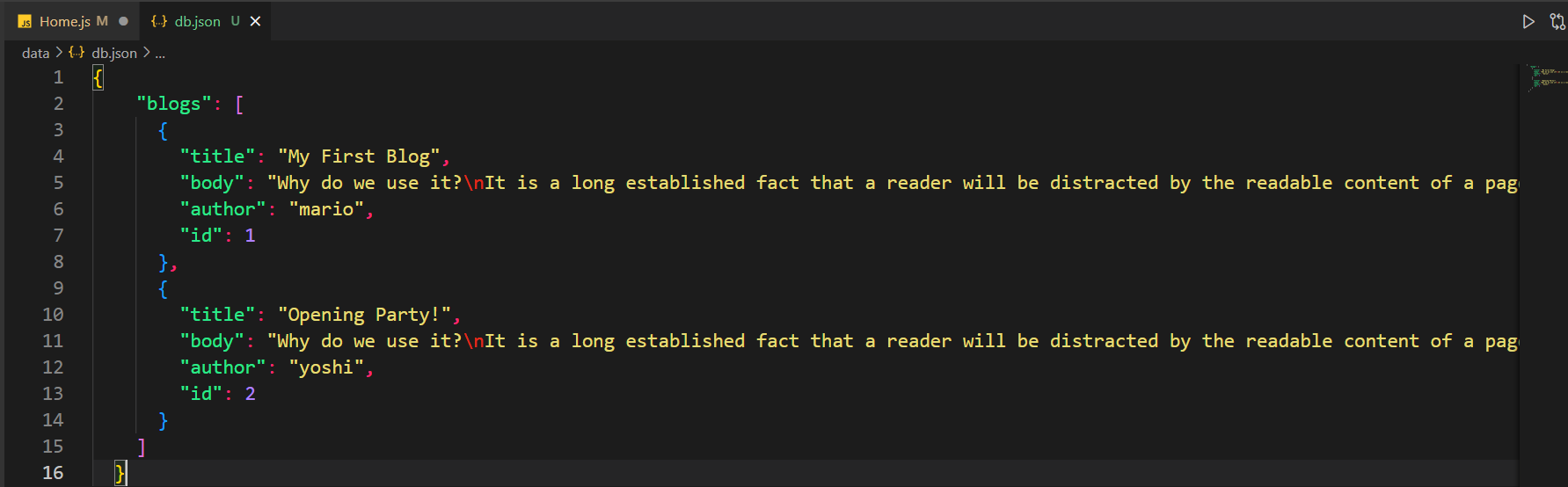
useEffect will also be called when state name changes.

useEffect is a good place to fetch data.

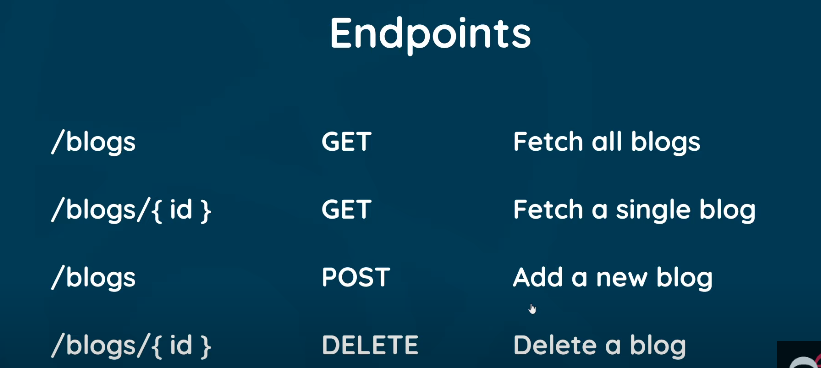
Lets use a JSON server to get fake data to test our application.

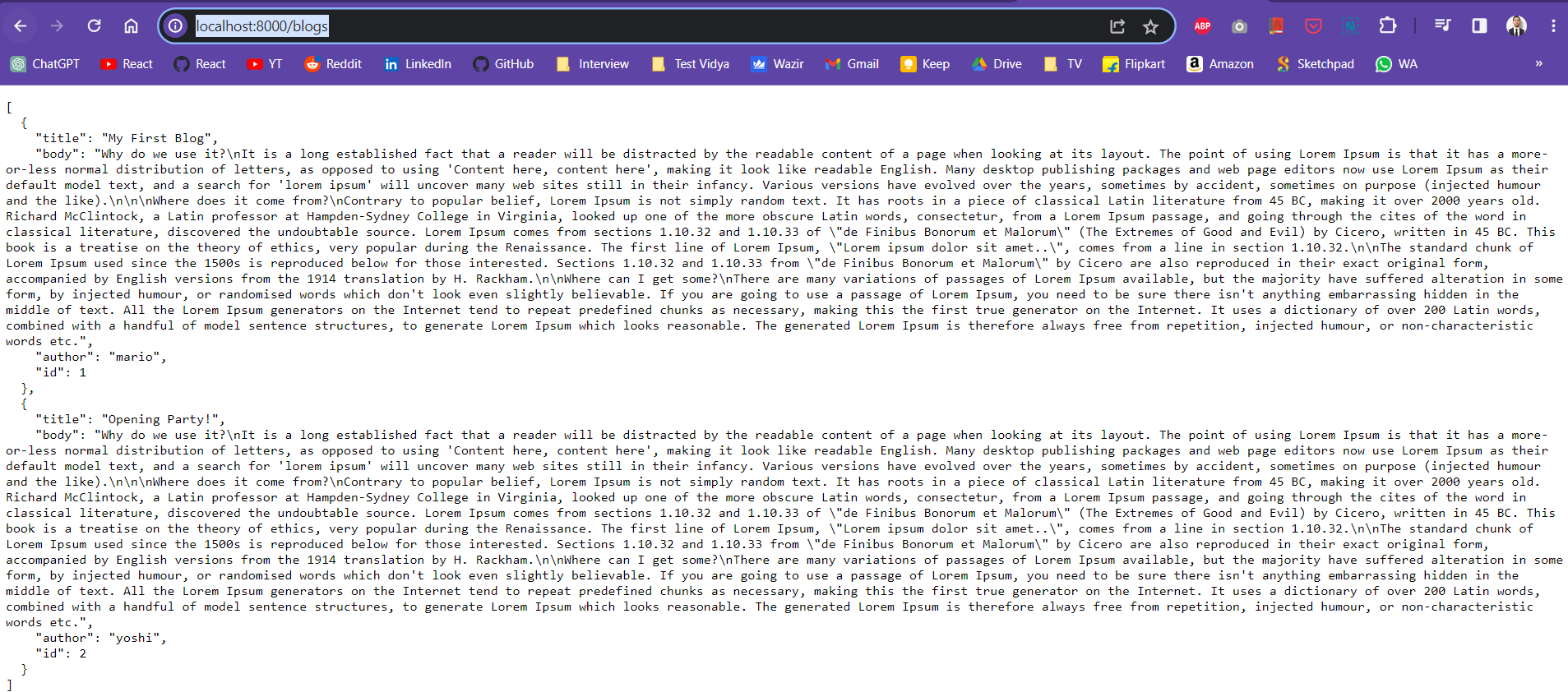
Using we have careted a json file and we will create some APIs for it using json server package.

JSON File:



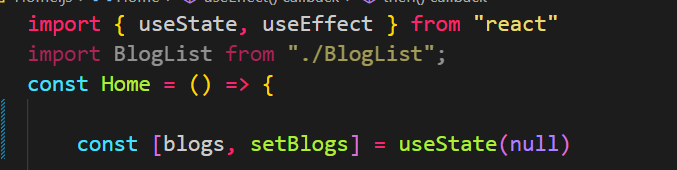
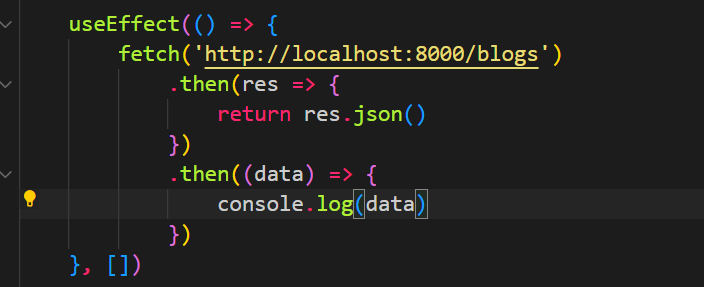
APIs Created:



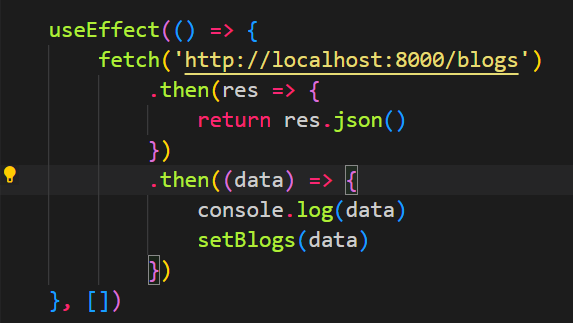


We will use these APIs in our react application.

Using fetch function to call an API in useEffect to get data:

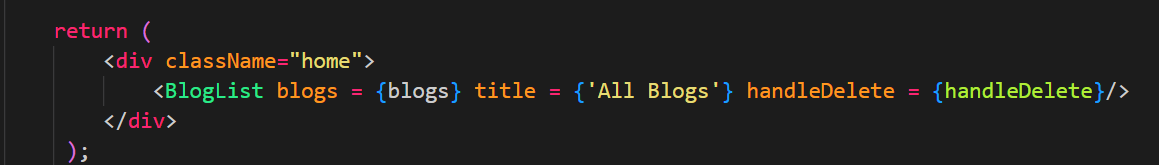


Blogs is a state initially empty. In useEffect we are using empty dependency array as a second parameter which means it will be called only once when component is first rendered. In the fetch method we call an API (we pass the API endpoint), then we chain some then methods because API takes time and returns a promise. That promise we take in the var res and this then return a JSON of the res. This json method is also Async and returns a promise which we capture in the next then method. This is captured in data var and we print data to the console.

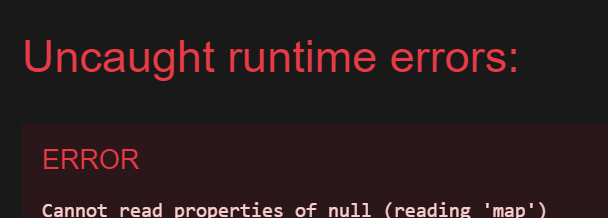


In the end of the second then we call the setBlogs method and set data to the state blogs.

The JSX of this component:

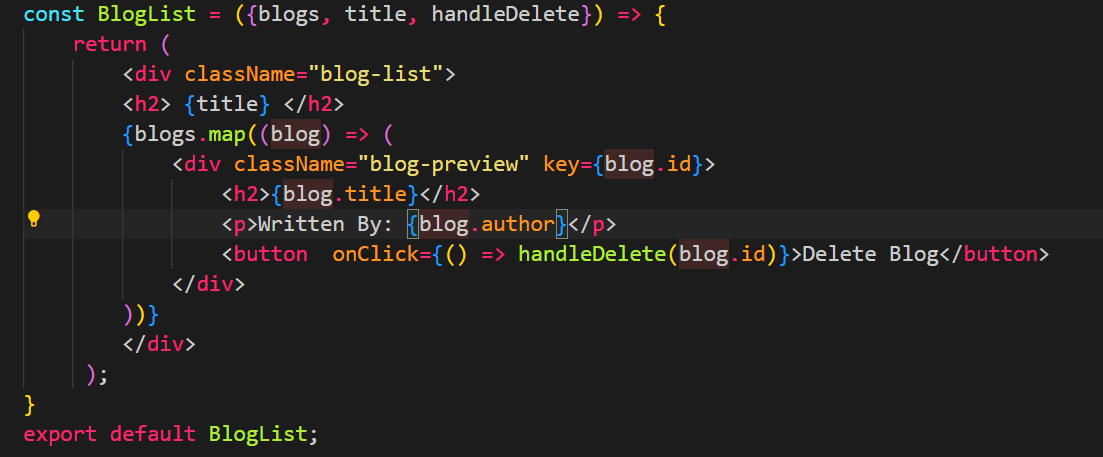


We get error:



This is because we are passing blogs state to blogList component as a prop. But it takes some time to set the data to the blogs state. We are passing it before the data is set.

Since we pass it before that the value of blog is null.

In the blogList component we have: 

We are using map on null which causes this error.

To remove we have to write this in home component:

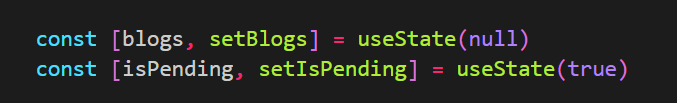


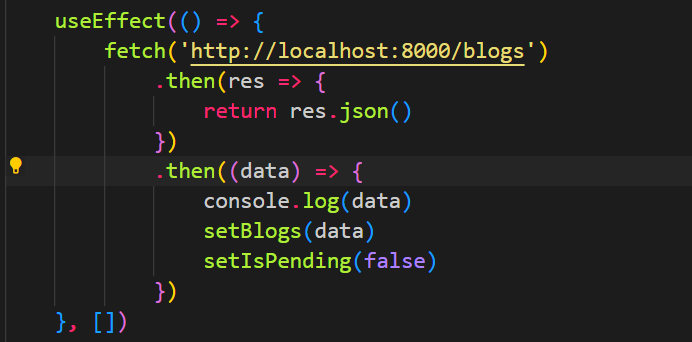
When blogs var is not null only then pass blogs to the blogList component.

When blogs var is null, the next condition is not executed because we have && and the first condition is not true.

This is conditional rendering.

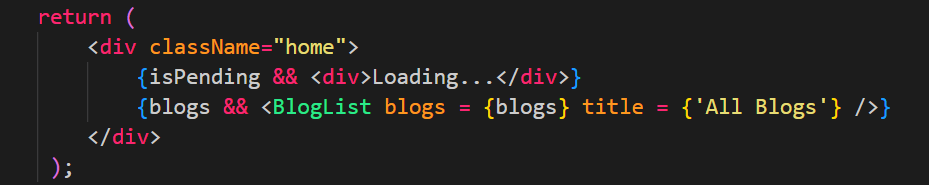
Lets show Loading.. when we are fetching the data:





Let’s set the isPending var to false once the data is loaded through fetch.

Let’s do the conditional rendering:



When isPending is true we will return loading, when blogs is true will call blogList component.

Handling errors which may come from fetch: 

Write a catch block after the then method of fetch.

We can also throw error by ourself:

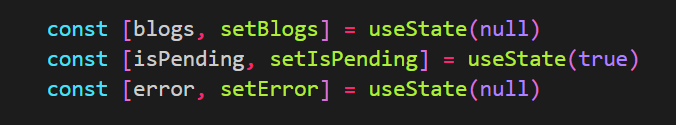


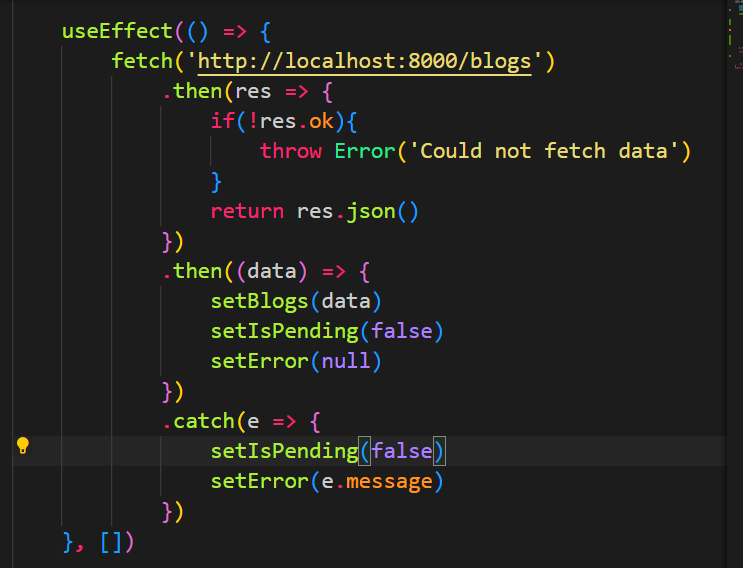
When we get a response from fetch the response has a property called ok. If ok is true resonse was fetched correctly. If not, response was not fetched due to some error.

So, we check that ok property in the then method and if it is false, we throw an error.

We also send a custom message in the error.

If an error occurs, we can put that on browser as well:

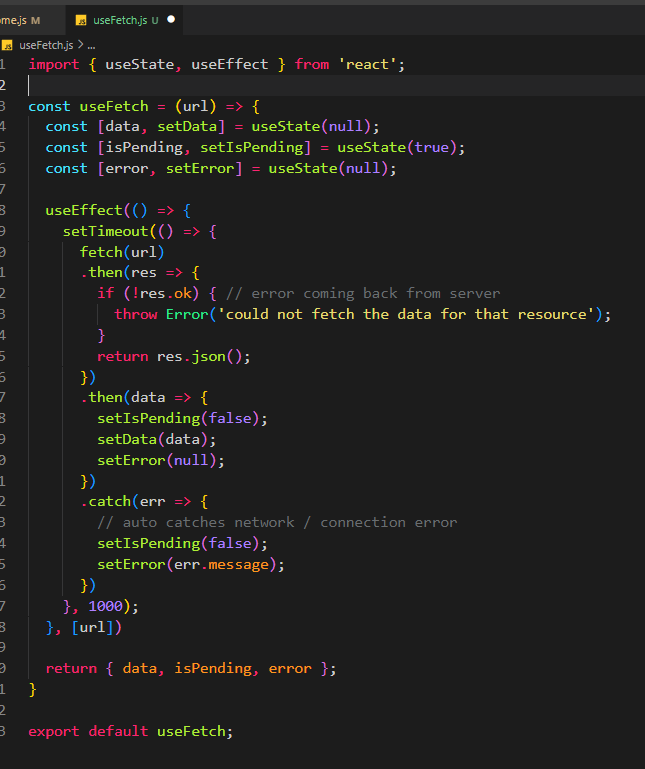




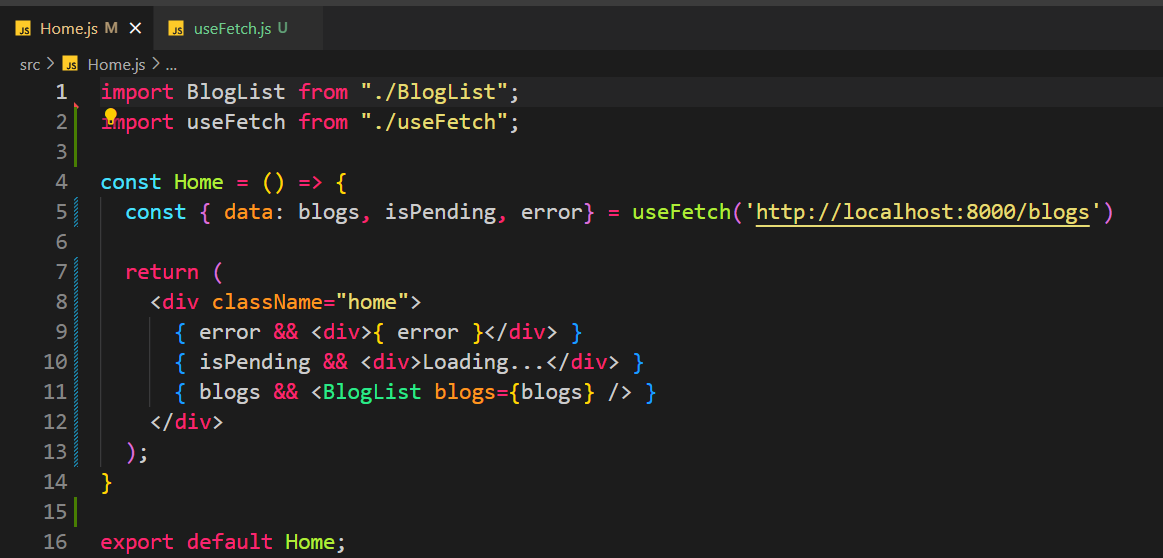
If we want to use this useEffect code in different component we can write custom hook.

Take all the code from useEffect and put it in a separate JS file. Import that JS file and use it in different components.

Custom hooks in react must start with the word **use**, otherwise it does not work.



How to use:



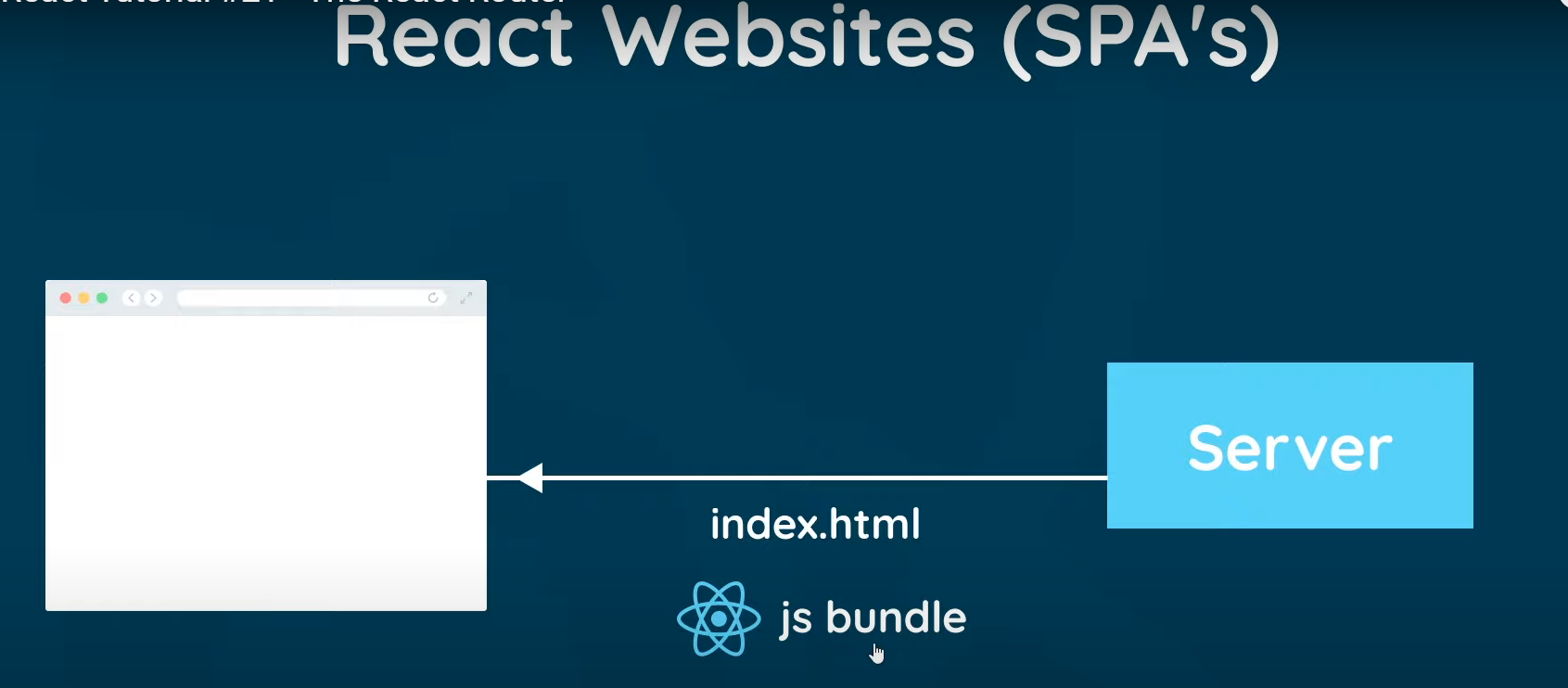
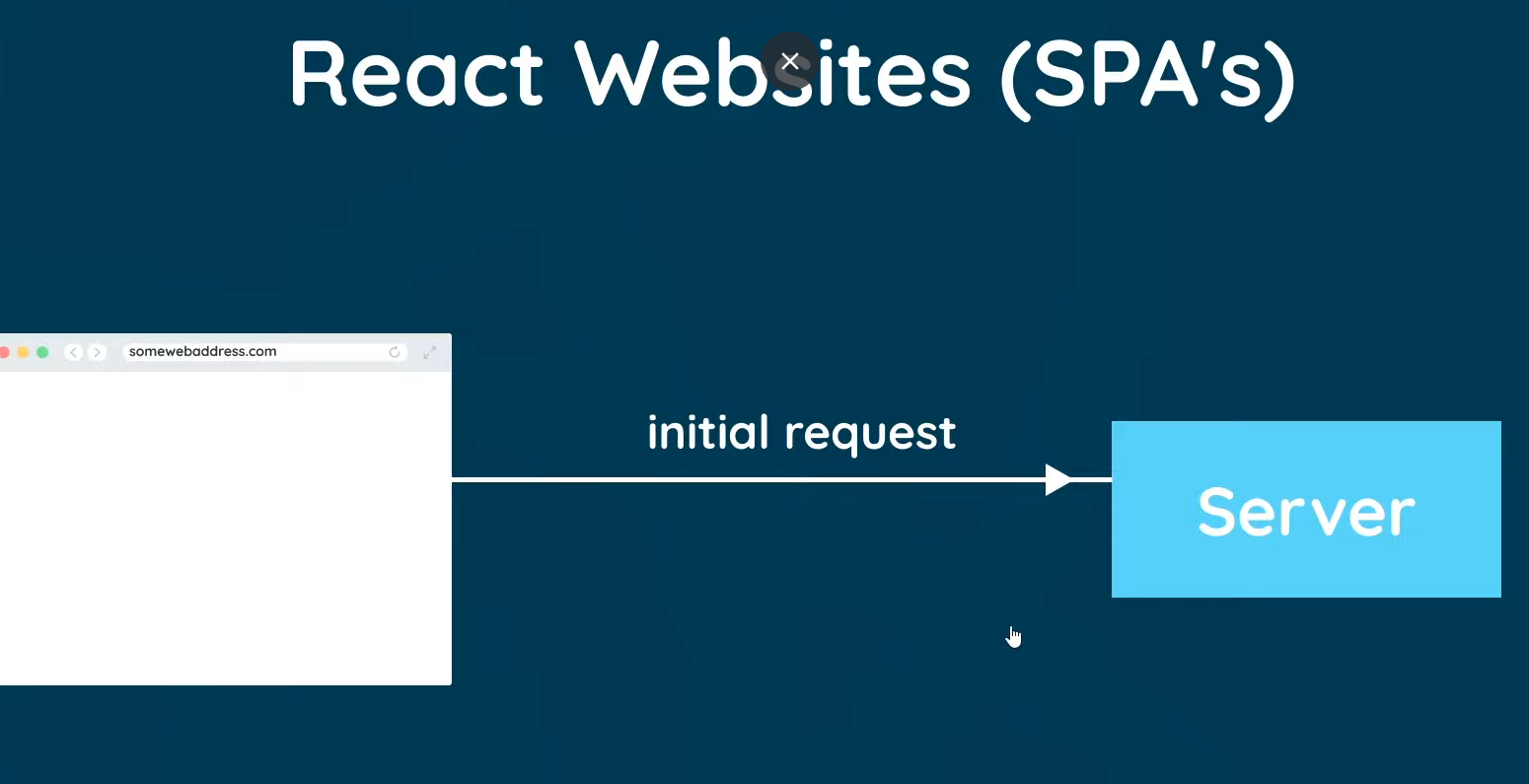
The custom hook will accept an argument that is url. Then fetch will be called for that url. There could be data or error. Also, while we fetch isPending will be true.

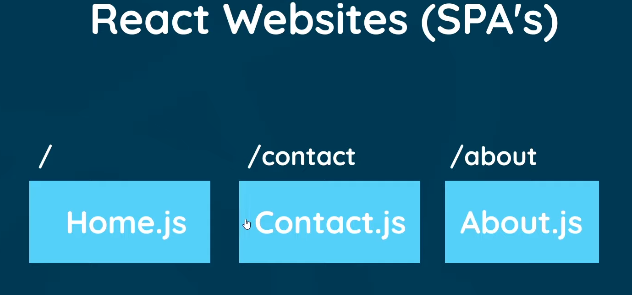
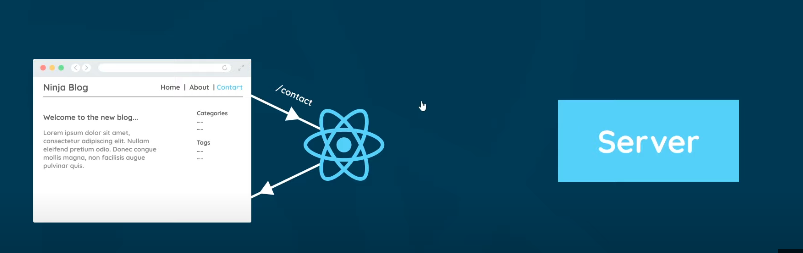
We change some var names for better understanding.

Also, we put url in the dependency hook which means that if the url is changed the hook is triggered again.

This hook can be called from any component.

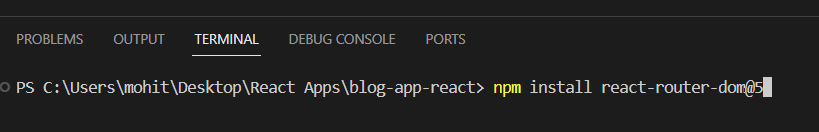
React Router:





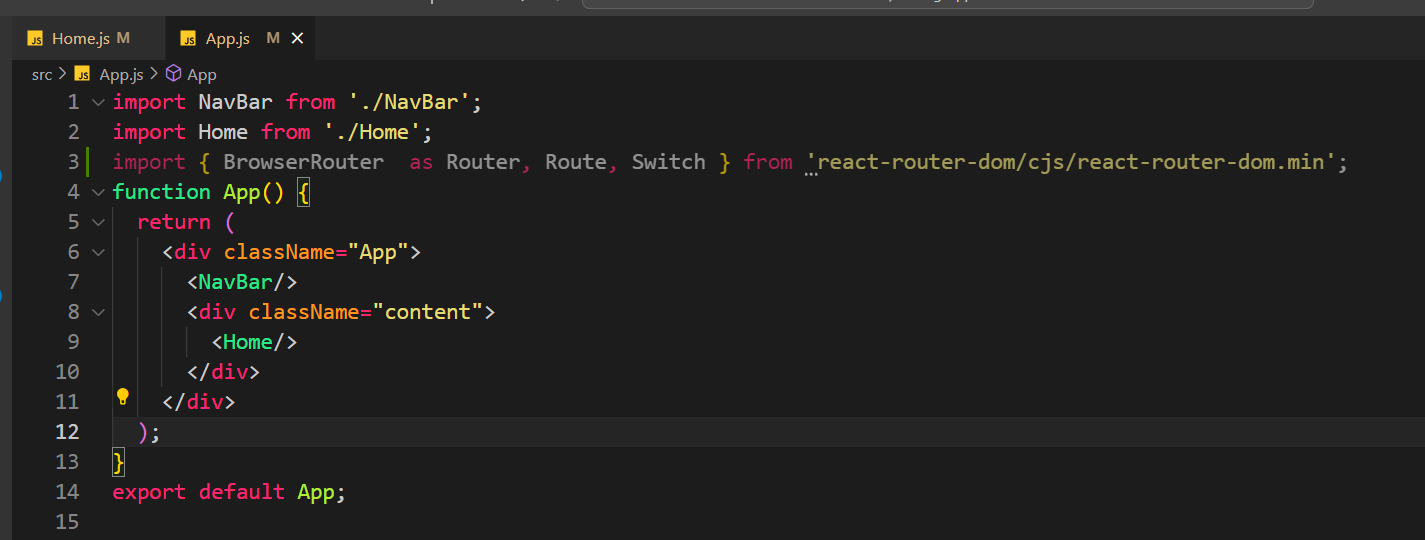
Step 1:

Intall the packge:



Step 2:

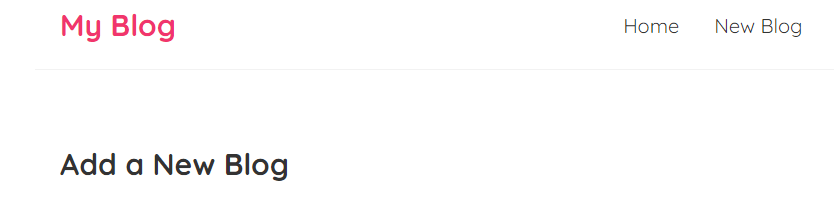
Import it in App.js file:



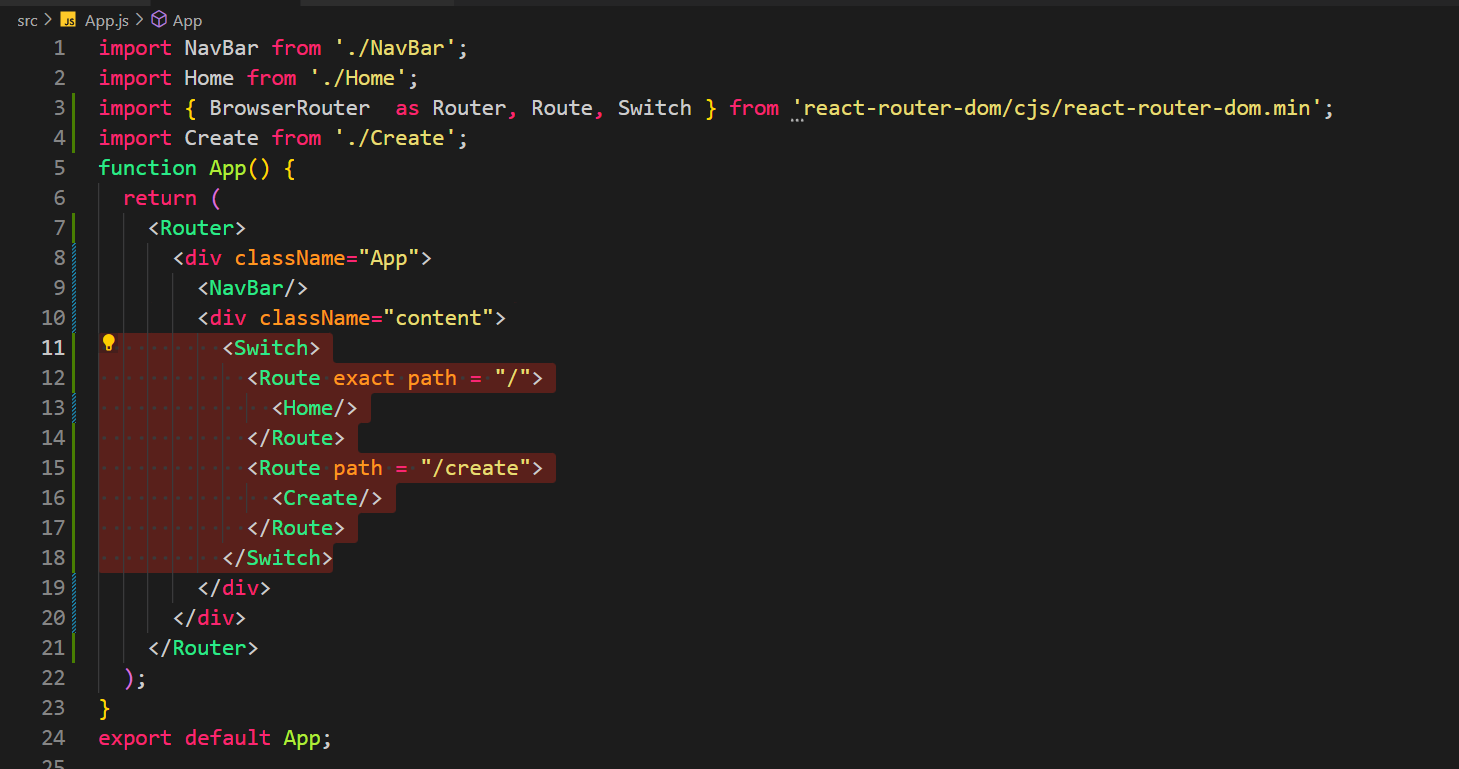
Once it is imported here you can use it in any component.

Let’s first create a component called Create which will be used to create a blog.





Now we have a home component and a create component. Let’s add routing:

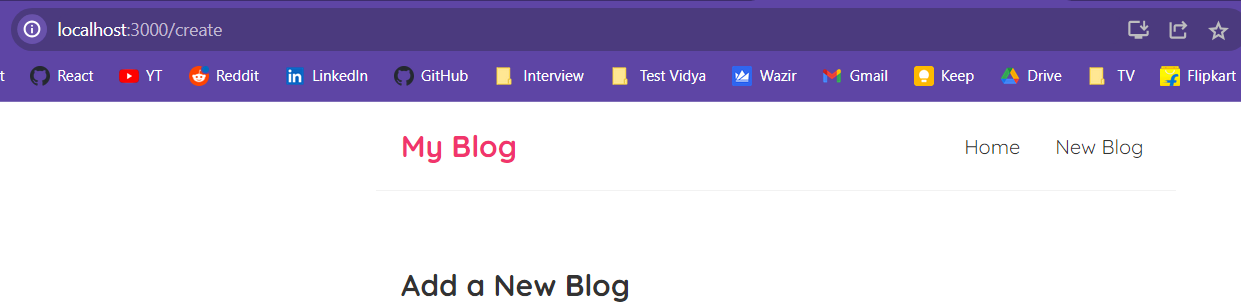
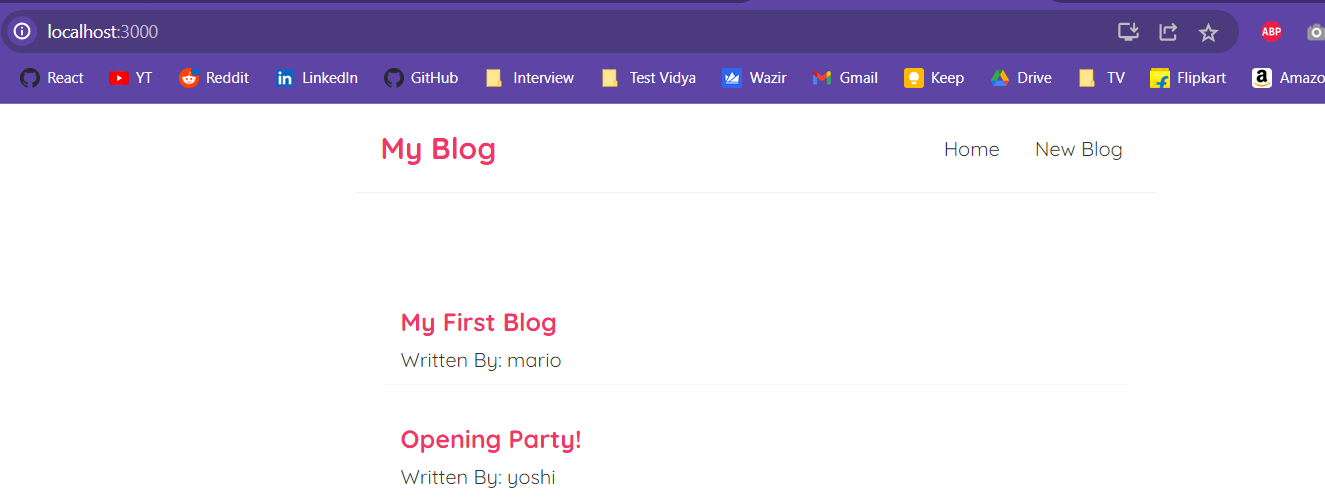


Include everything in Router tags. Now we have Navbar component outside the switch. So, the navbar component will be shown at all the pages, irrespective of routing.

We have our routes inside Switch tags. When we enter a url, react will go to the switch tags and look at each Route tag and its path. Whenever the url matches with the path of the route that component is rendered. Switch works just like switch statements in c or java. So, if we find a match then next paths are not seen.

Also, we have put exact path for the path /. This is because path will match with the url in a way such that if path is included in the url that switch is taken. So, add exact path in the paths so that exact path is matched with the url.

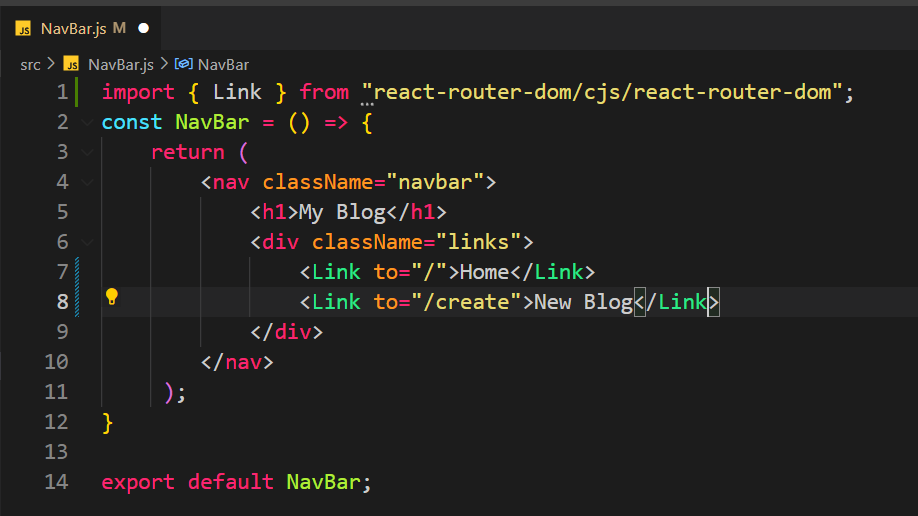
/ means the path of the website:



See the Navbar component:



We need to use react routing instead of using anchor tags:

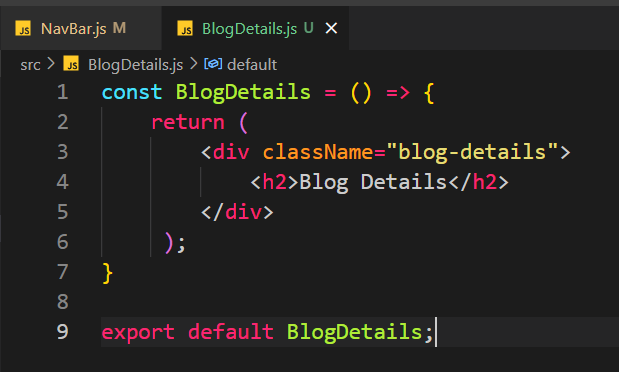


We will use Link tags and instead of **href** we will use **to** property.

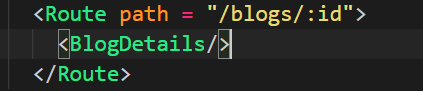
Sometimes we want to pass some parameters to the route. Some path of the route is dynamic and can change. Eg /blogs/123. 123 is the id of the blog. This part may change with the id of the blog. Eg if we now want to show a blog of id 134 then route becomes:

/blogs/134.

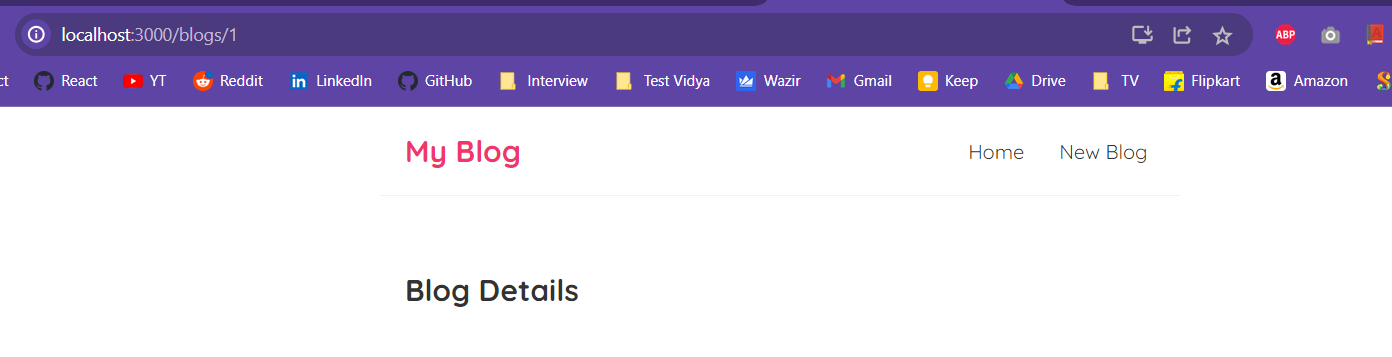
Let’s create a BlogDetail component first:



In the App.js:

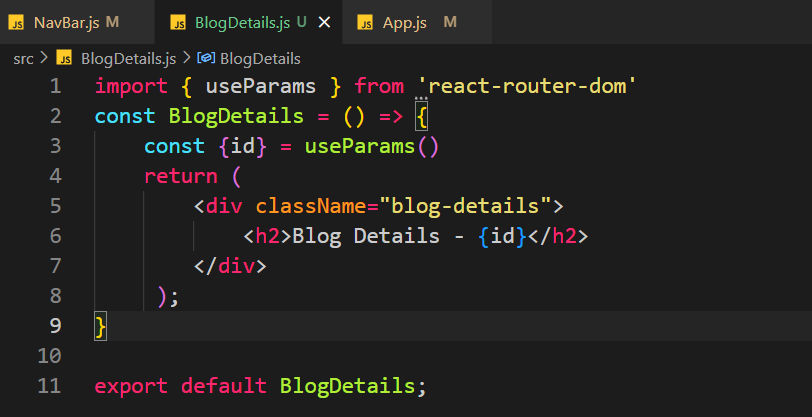


:id means id is a dynamic part and can change but same component will be called.



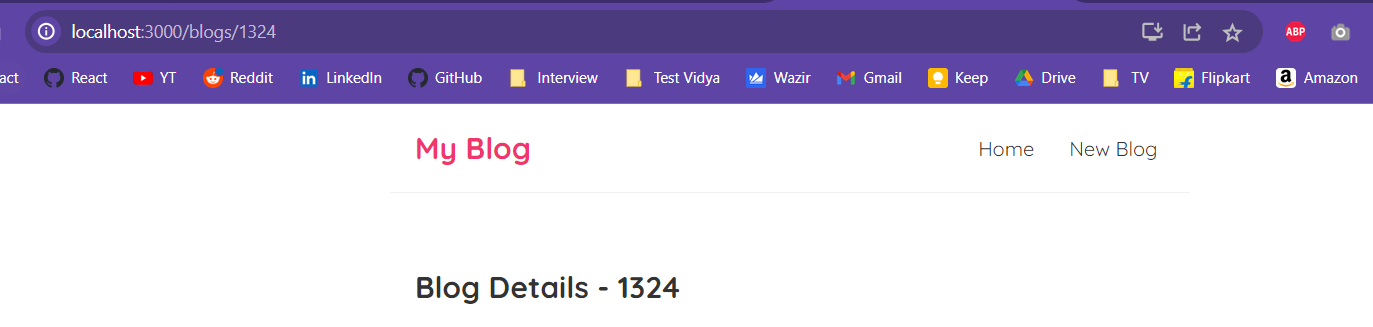
Any number you put after blogs/ will send you the same page.

But we want to read the id passed in the url.

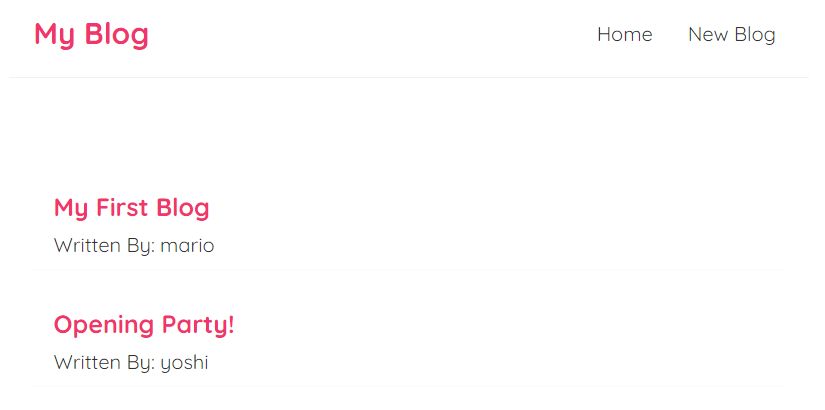
First import the useParams in the component: 

Second destructure the data in the url. Here we only have one item id. So, we call the useParams() and it returns id which we capture in the const var id.

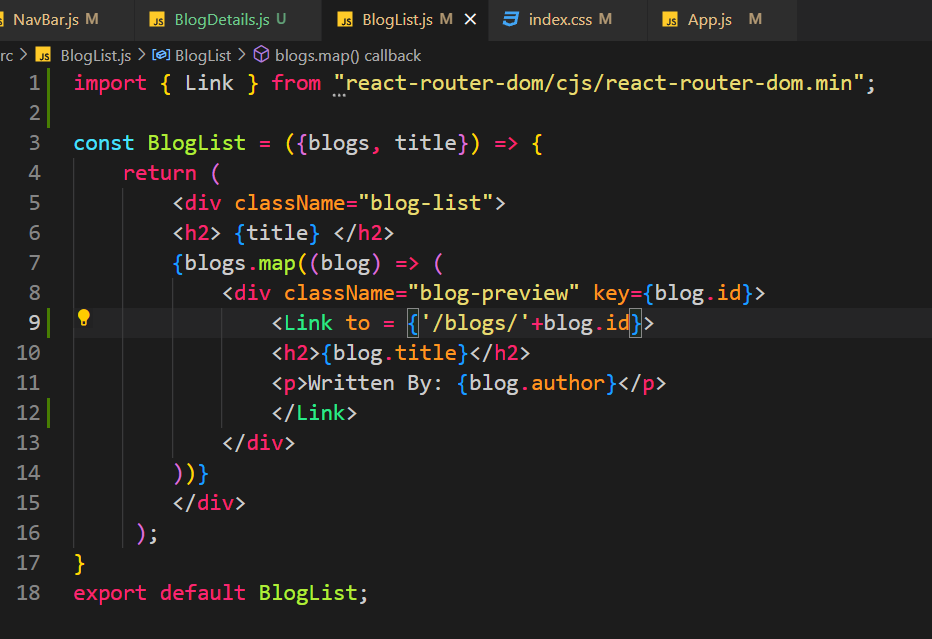
This id then we can use in the page:



We have this blog list component:



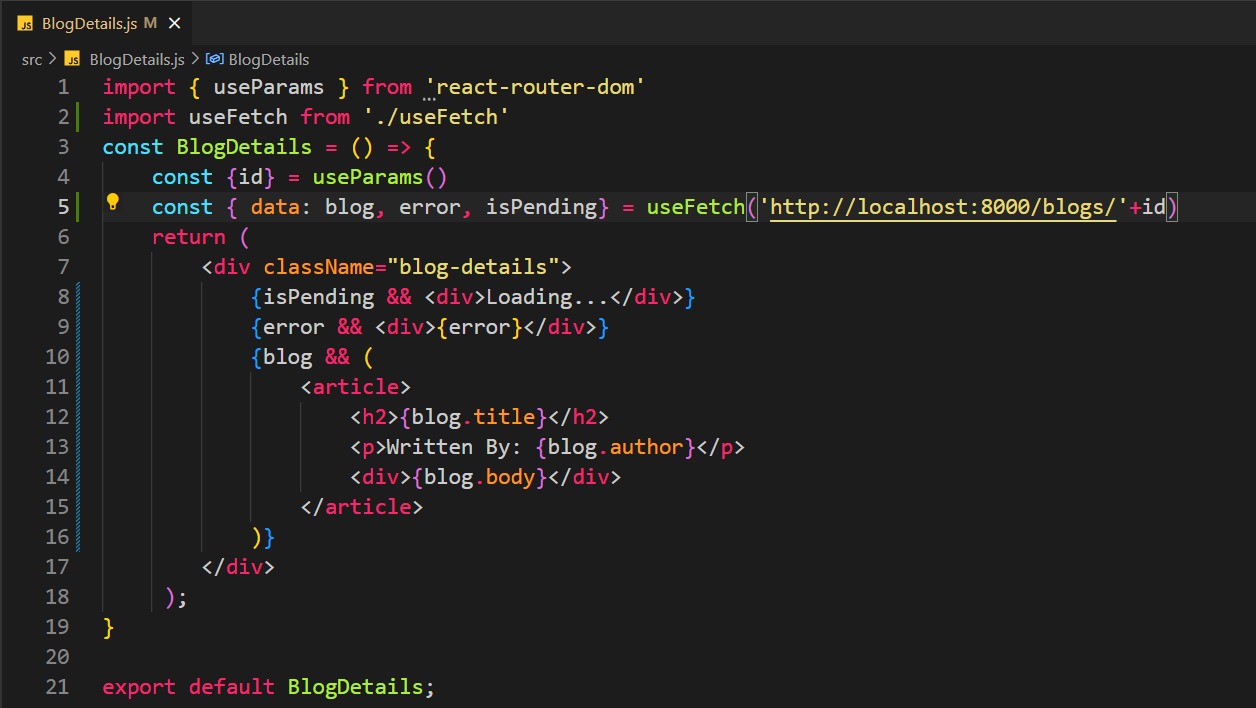
We want to show the blog id when we click on these blogs. Let’s do that:



Here we are enclosing the blog preview inside the Link tag making it basically a link. And in the to attribute we are using JS variable. We are passing the blog id as a param. Which we will receive in the blog details component and show the id there.

So, whichever blog is clicked, its id will be sent in the url parameter. This url will then trigger /blogs/id route and in that we have blog details component. This component will then show the id.

Now we want to show blog details when we click on the blog. We will reuse our custom hook useFetch to fetch the blog we clicked and then show it on our blog details component.

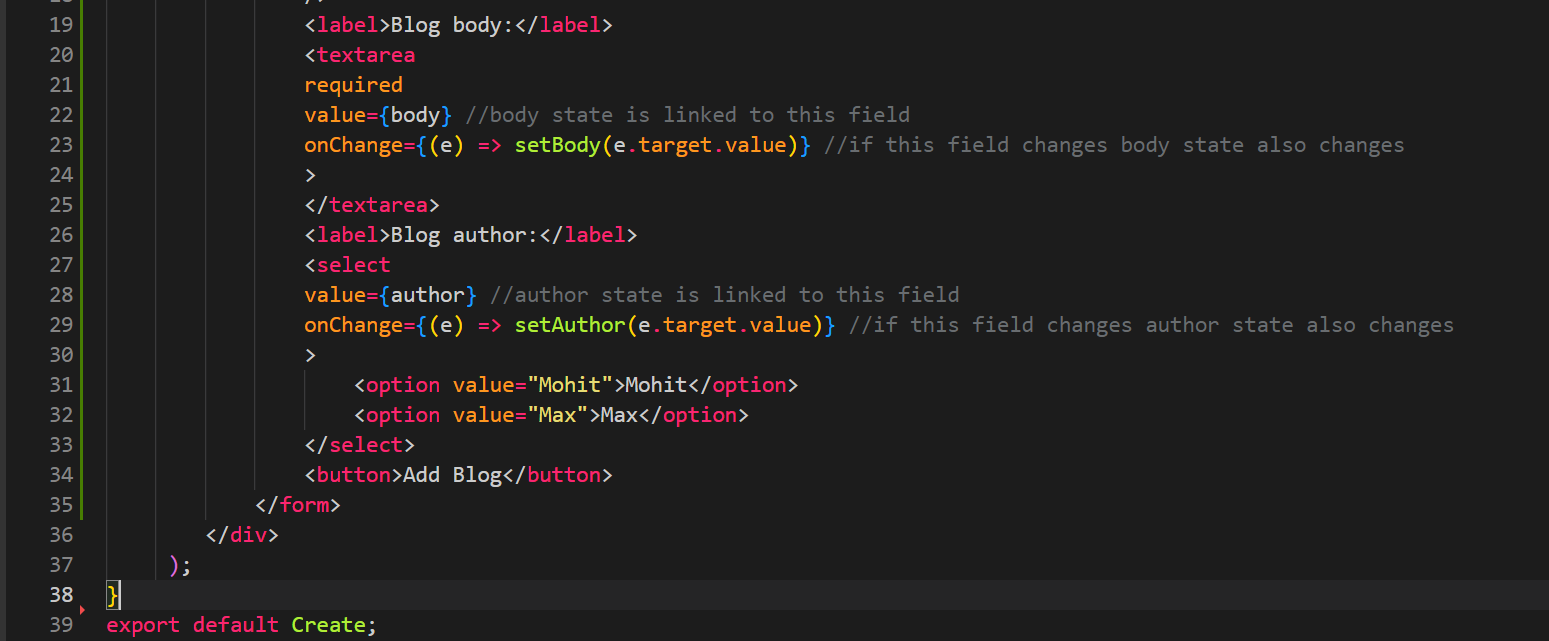


When we get to this component we will have an id in the url. This is due to the fact that we have set the routing accordingly.

We will fetch that id using useParams method and then use the useFetch hook to get the data according to the blog id. Once we get the data, we will use some JSX to show this data on the screen.

Let’s create the create blog component. This will be a form with multiple fields.

We will see how to create fields and how to link them with the state of the components:



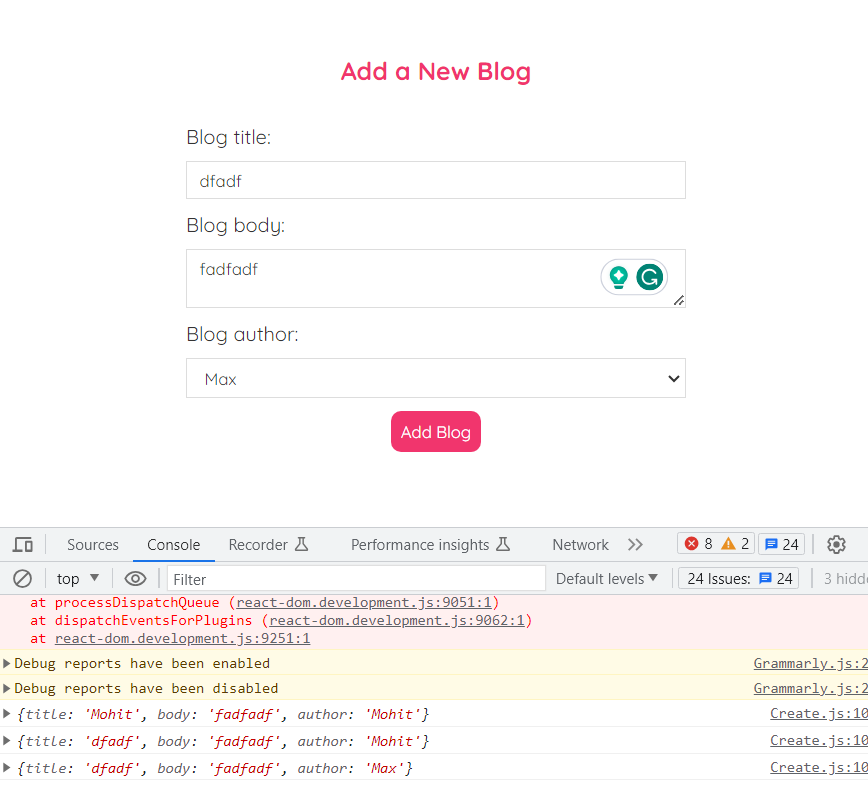
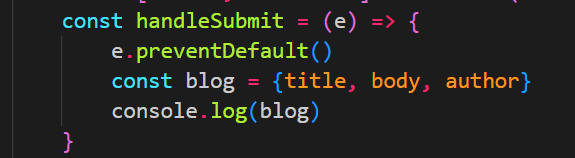
Three states title, body and author and three fields in the form.

Let’s submit this form:

Let’s try to submit this form:

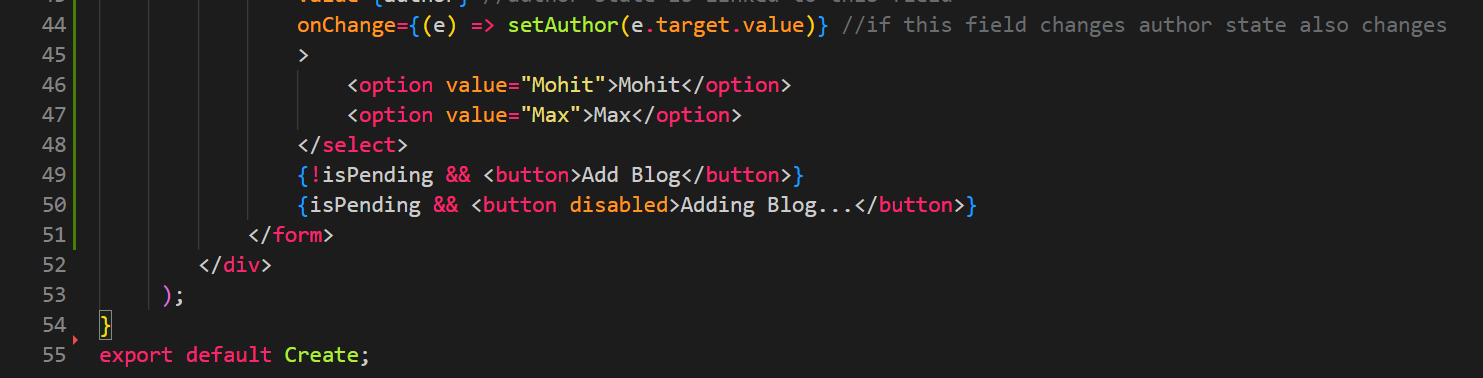
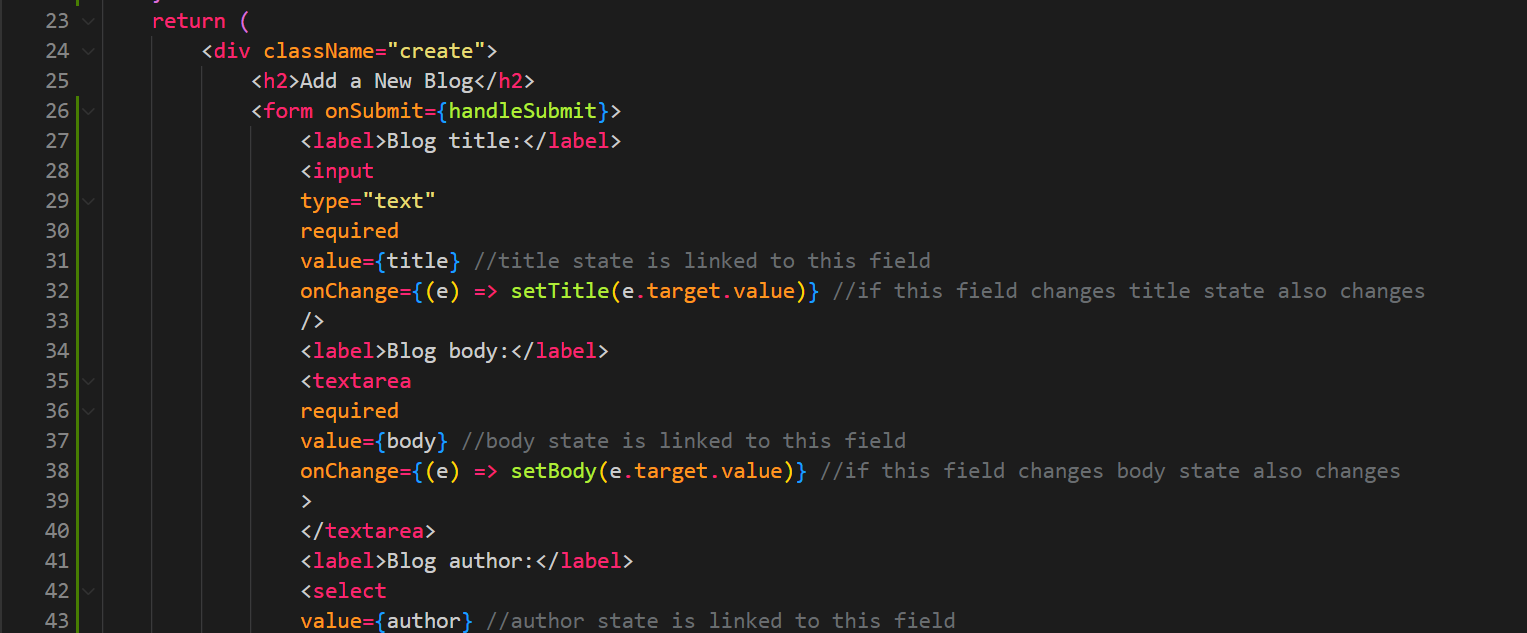
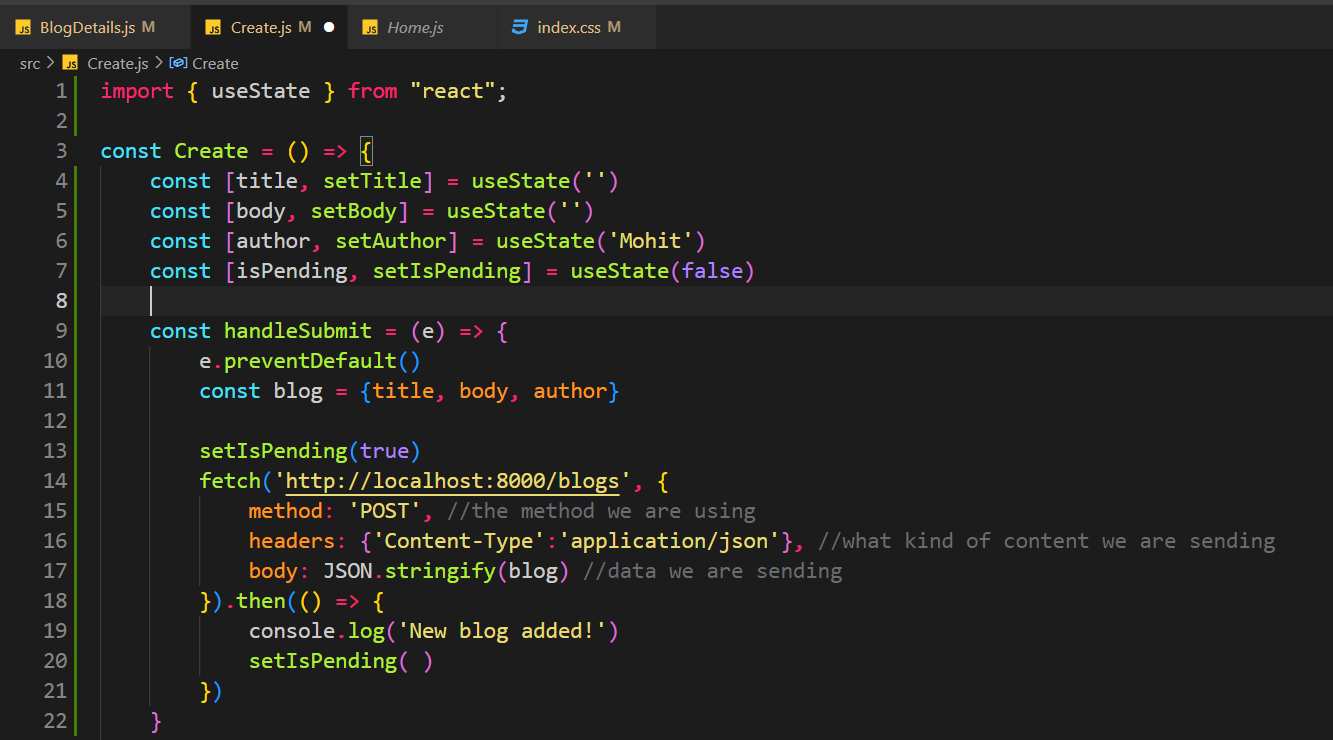
To do so, we will add onSubmit on the form and we will call a function from there.

The function will create an object with title body and author, which we will then log to the console.



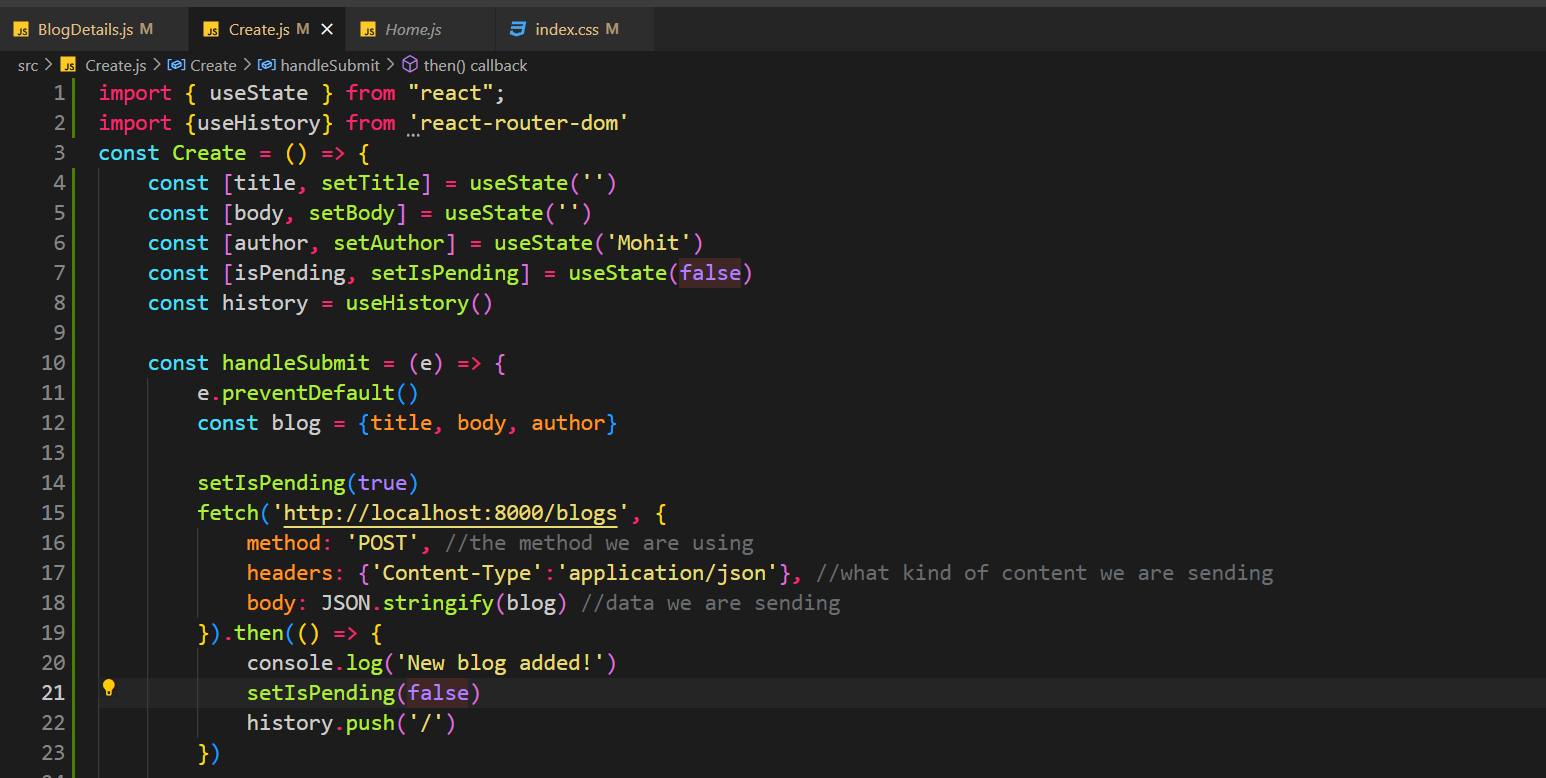
Once the form is filled, we need to save this data in the json file (in json server we are using). For that we need to do a post request.

We will also create a isPending state which we be true when our post req is being sent and we have not received any response. We receive response then it turns to false.



Now we want to go back to the home page once we have added our blog. To do this we will use useHistory from router. useHistory is used to go one step back or two steps back etc by using -1, -2, -3 in the useHistory function. But, we can also use it to go to any route.

See:



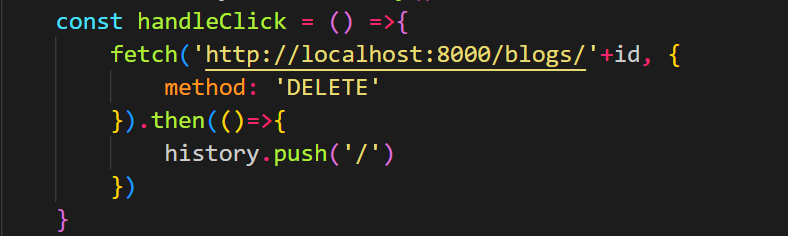
Import history.

History.push will send you to the route passed as argument.

Let’s create a delete button in the blog details component and let’s delete the particular blog using fetch.



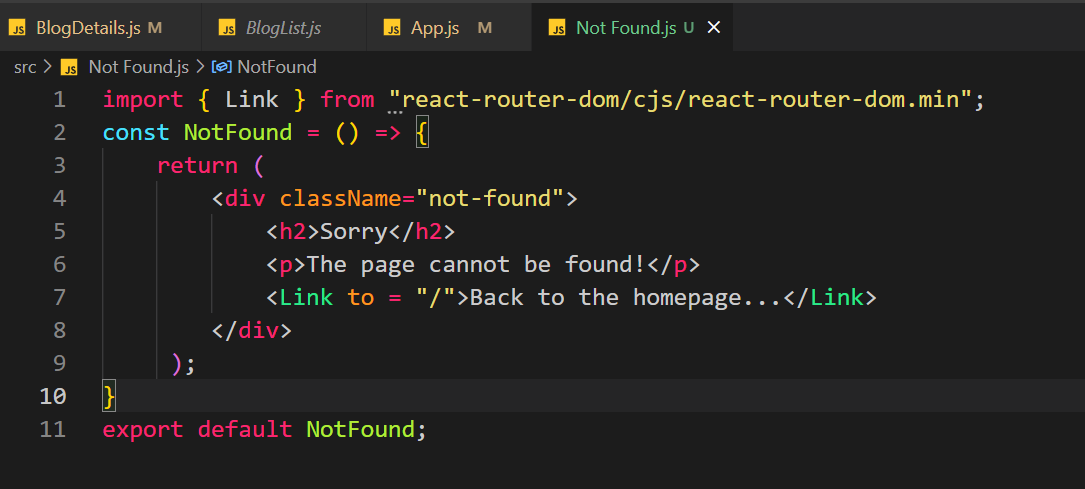
Delete button in blog details component.



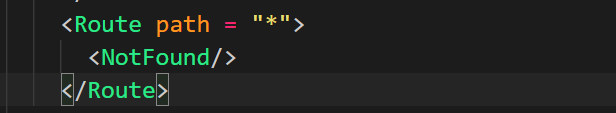
Handle click method deletes the blog. We pass the blog id we get from the url params. We pass that to fetch and pass method: delete to delete the blog.

Now one last thing we need to do is, if the user types any url which we have not mentioned in the routes, then the user must be sent to a not found page.

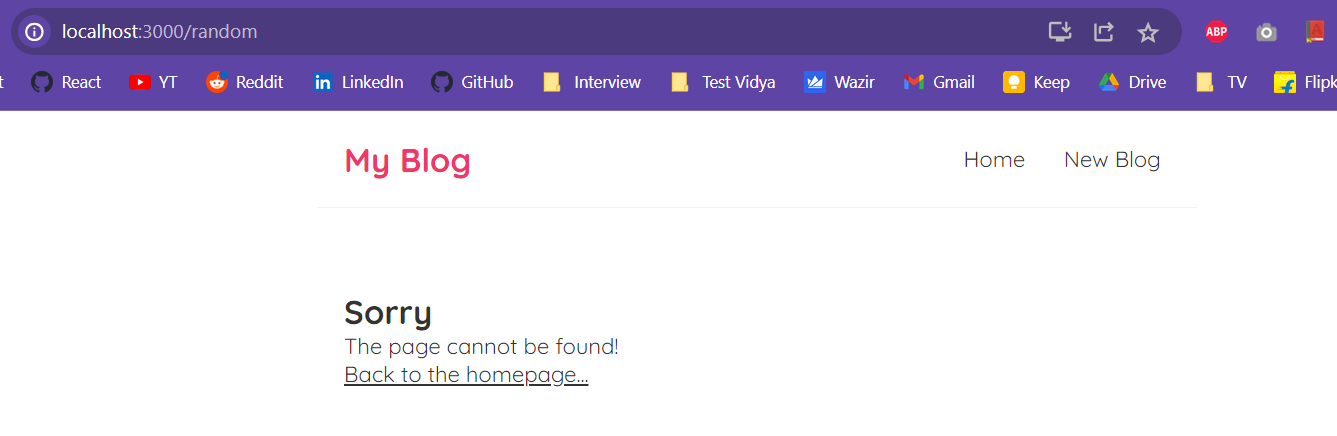
Let’s create a not found component, this component has a link to home page as well.



Let’s see the routing:



This path means if the user enters any path which is not found in any of the above paths then go to this path. Remember this route path = ‘\*’ must always be the last route mentioned in the switch statement.



User enters any path which does not match then we are moved to this page.