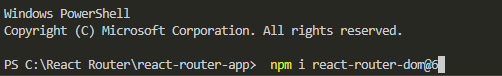
**REACT ROUTER**

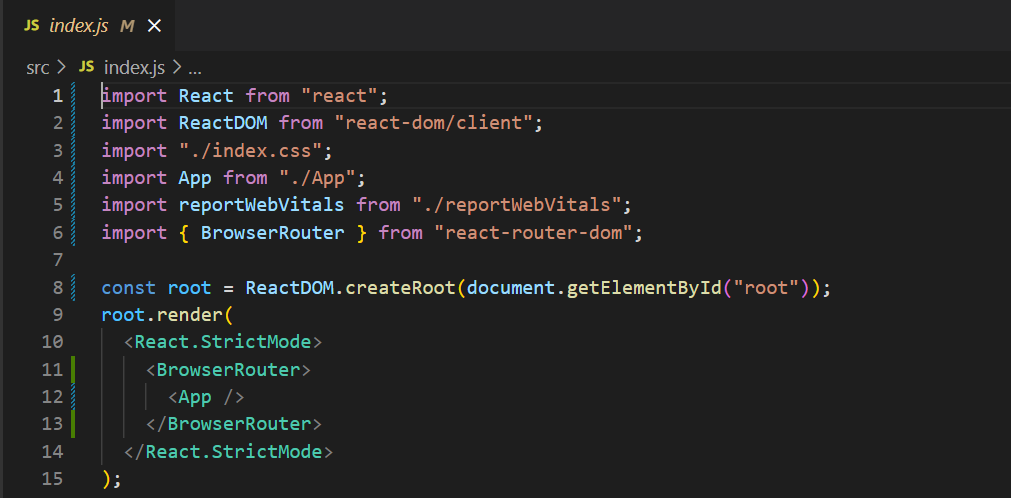
1. **Installation & Setup:**

After creating a react app, install *react router package* via the terminal.



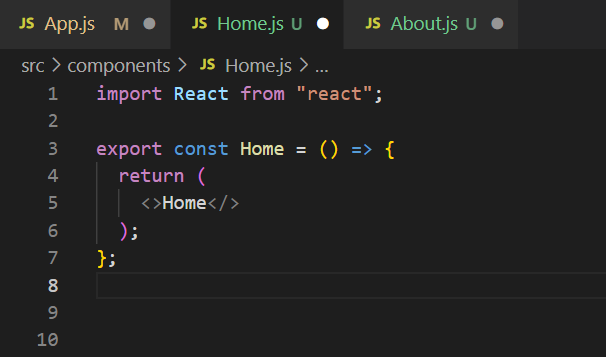
1. **Configuring Routes:**

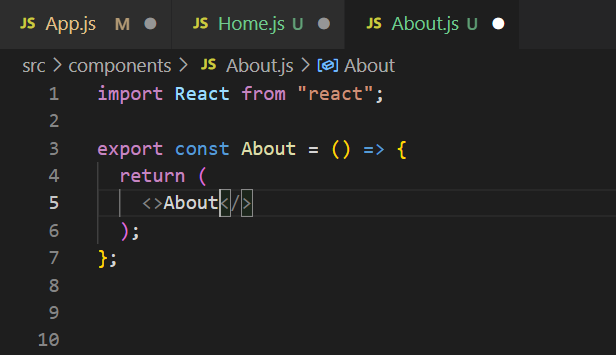
In *index.js* wrap the entire app with *<Browser.Router>.*



Create *component* folder in *src*.

Create files *Home.js* & *About.js* in *components.*



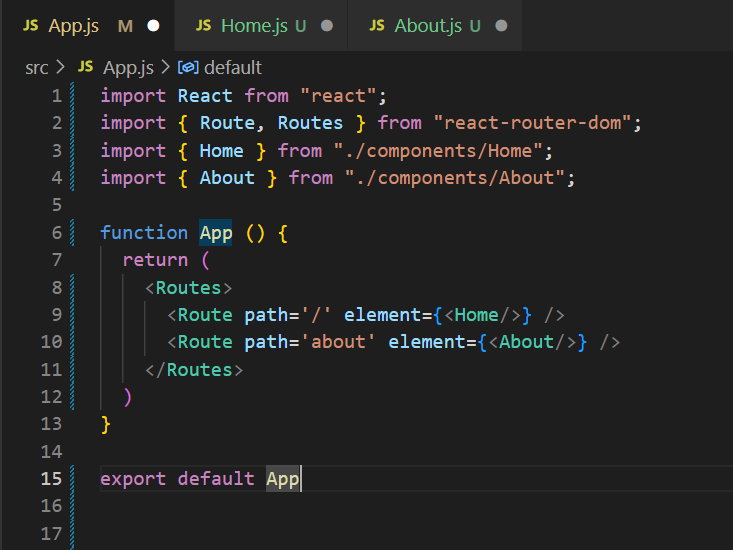


We need to import two components in App.js => *Routes* & *Route.*

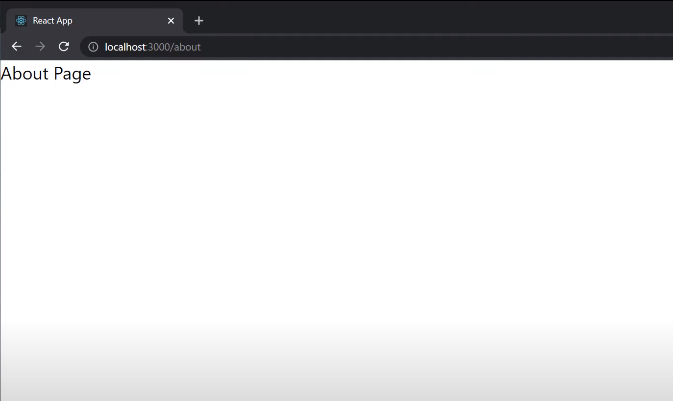
Within the *Routes* component we define the individual route using the *Route* component.

We pass the props, *path* => gives the path in the URL and

*element* => the component to be rendered.



Output:



We can also access *Home.js* by entering http://*localhost:3000* and *About.js*  by entering http://*localhost:3000/about* in the browser address bar.

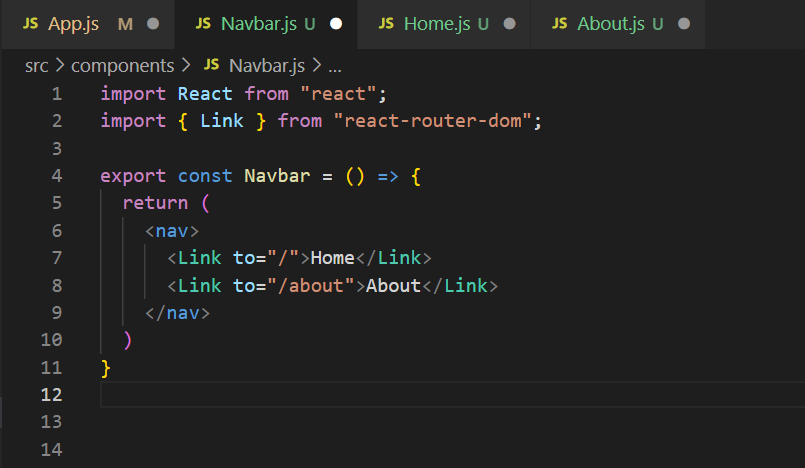
**3) Links:**

We now learn how to navigate to different routes using an element.

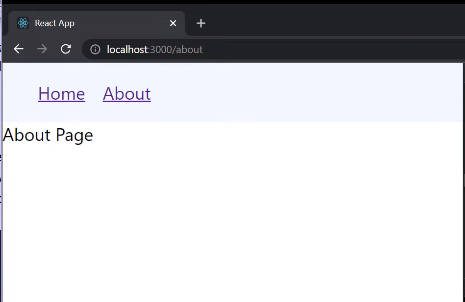
Create a *Navbar.js* file as shown below and add it as a component in *App.js*.

*React Router* provides us with a *Link* component for this purpose.

It contains a *to=””* prop that takes the relative path.



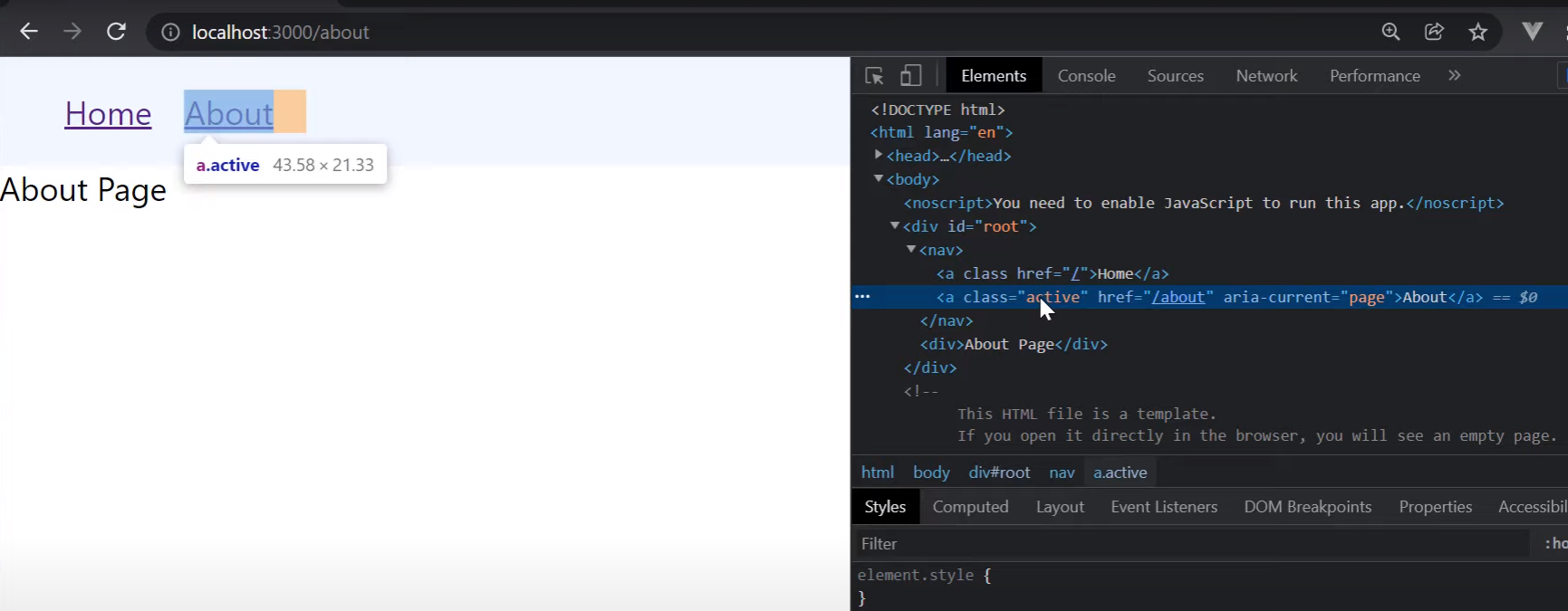
Output:



**4) Active Links:**

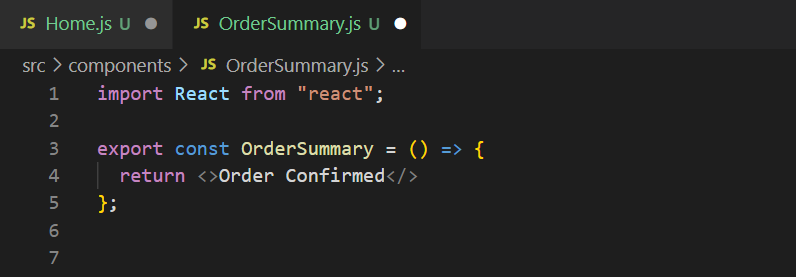
Replace *<Link>* component with *<NavLink>* component in *Navbar.js*

In *<NavLink>* component, whenever a given link is active, it is assigned by default ‘active’ class.



**5) Navigating Programmatically:**

Create a file *OrderSummary.js* in *components* folder.



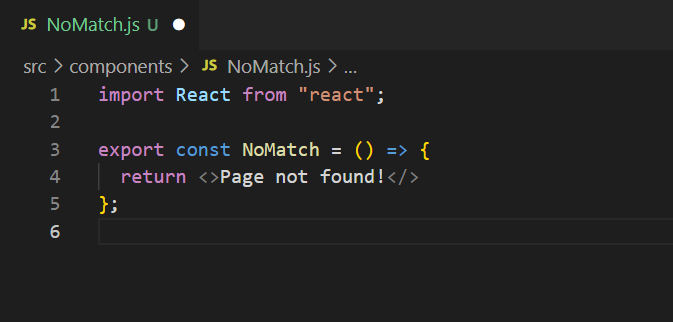
To navigate programmatically, we will use the *useNavigate()* hook which basically accepts a path to navigate to when an event occurs (here *onClick*).



**6) No Match Route**

In case we navigate to a route which isn’t configured, the user must be informed about it as React remains on the same page (although we do get a warning in the *console*).

Create a file *NoMatch.js* in *components* folder.

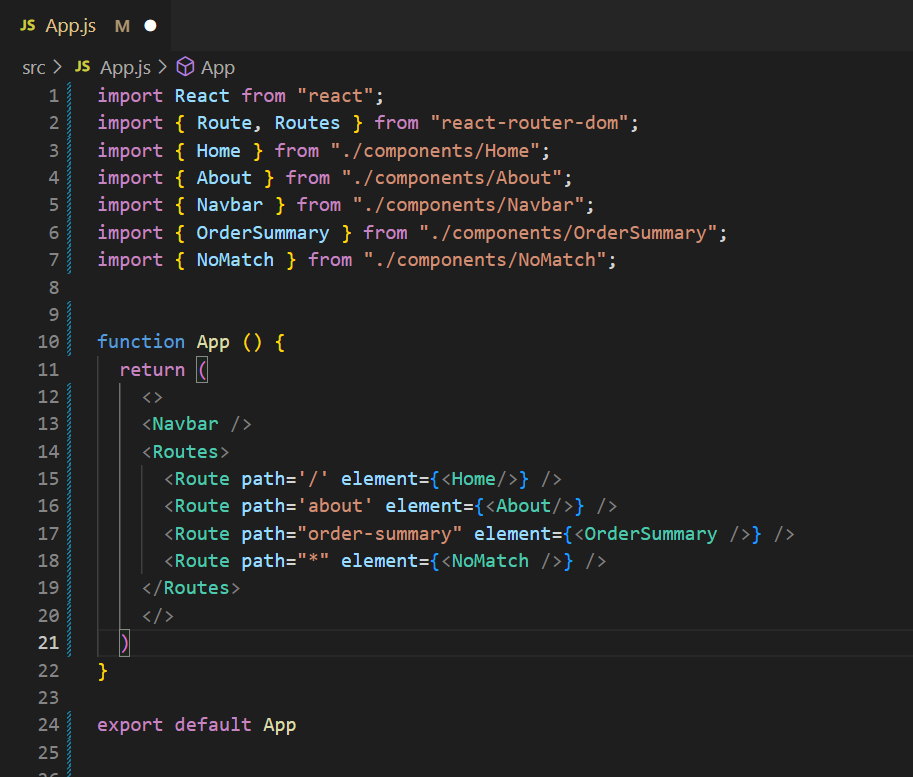


Pass in a new *Route* that will navigate to *NoMatch.js* whenever an non-configured URL is passed in the address bar.

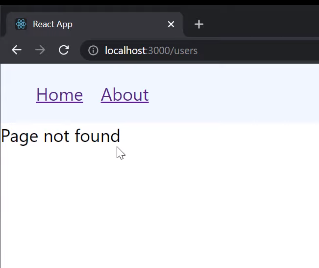
This is achieved by adding a new *Route* with *path=”\*”.*

This path will match only when none of the other routes do.

*App.js:*



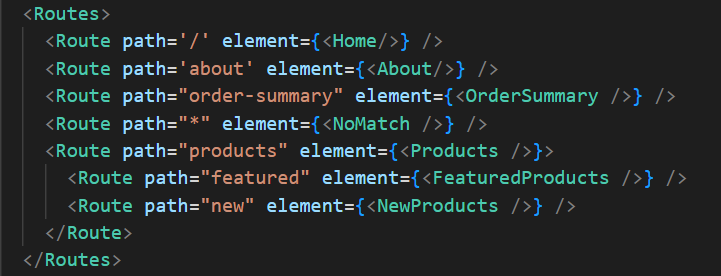
Output:



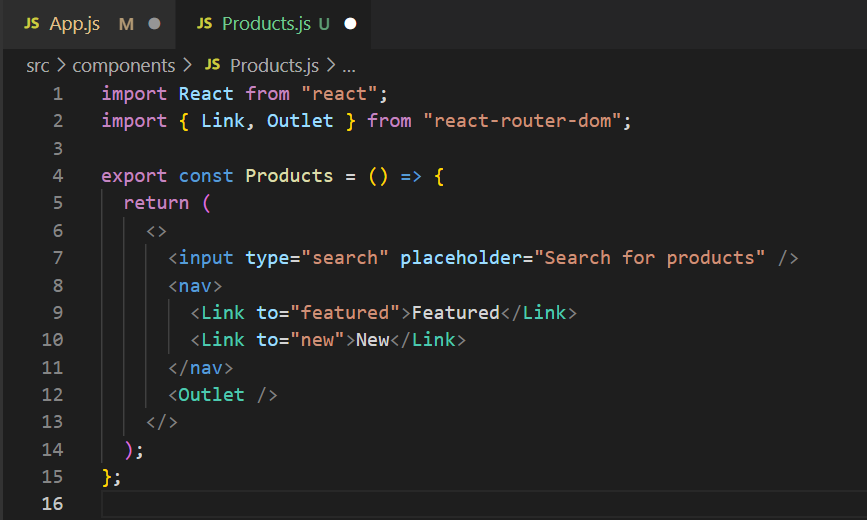
When *localhost:3000/users* (a non-configured URL) was entered, *<NoMatch/>* was rendered.

**7) Nested Routes**

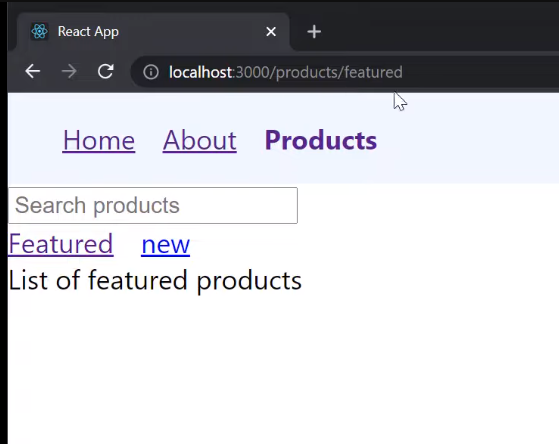
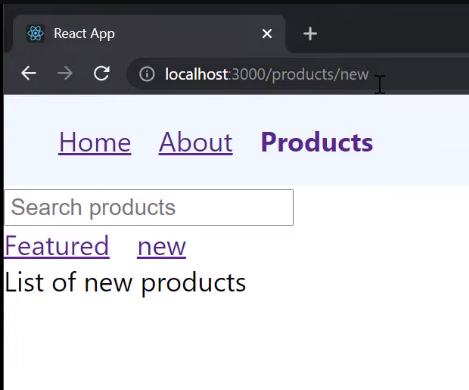
React lets us render a *Route* inside another *Route*.

*App.js:* 

Create files *Product.js, FeaturedProduct.js* & *NewProduct.js* in *components* folder.



*<Outlet/>* component tells *Products.js* where to render the child components (*FeaturedProduct.js* & *NewProduct.js*).

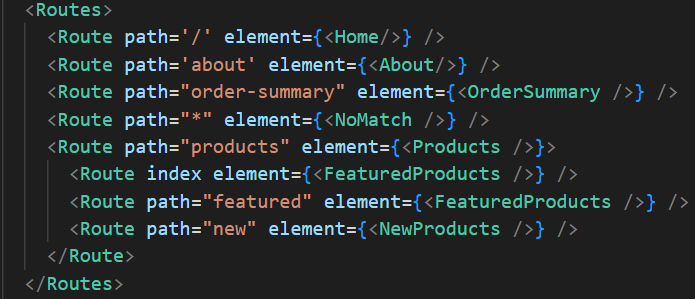
 

React Routers automatically form the full path for the child Route by joining the relative path of the parent to the relative path of the child, which in turn forms an entire URL.

**8) Index Routes**

The *index* prop when passed into the *Route* component renders the component which the *element* prop receives as by default when the parent component is rendered.

*App.js:*

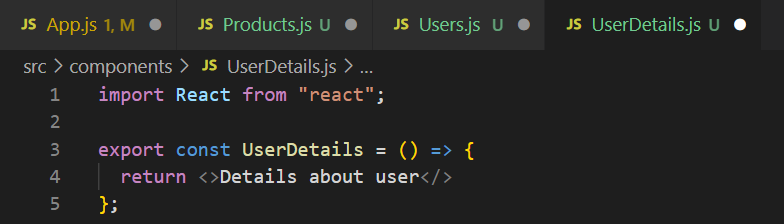


**9) Dynamic Routes**

Create *User.js*

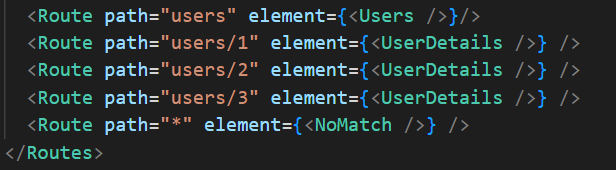


Create *UserDetails.js*



Using this method to render *UserDetails.js* for every user isn’t feasible as for a large number of user we will have to write a large number of lines of code

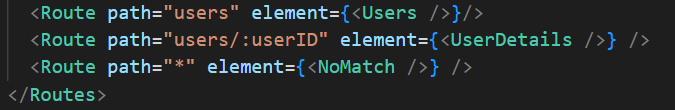
*App.js:*



To overcome this problem, we use this syntax *path=“users/:userID”*

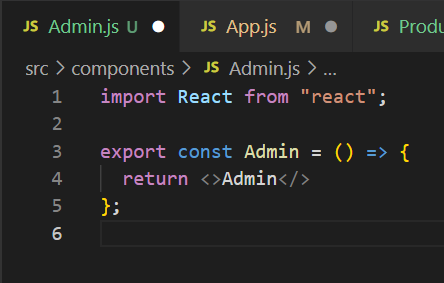
The *users/:userID* parameter will match any value as long as the pattern is same.

*App.js:*



Note: *userID* can also be a string not just a number.

Create *admin.js* file in *components* folder

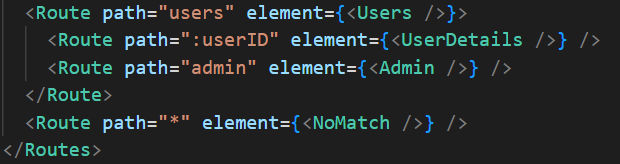


Dynamic routes can also be nested routes.

As both routes (*UserDetails.js* & *Admin.js*) have the same prefix URL, we can nest them.

They’ll be rendered within the parent component (*Users.js*).

*App.js:*



Also, add *Outlet.js* component in *Users.js* file.

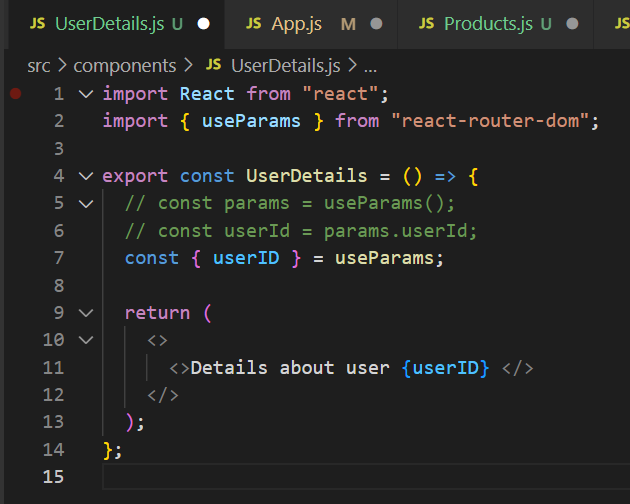
**10) URL Params**

*useParams()* hook can be used to render unique content for any given user in *UserDetails.js*

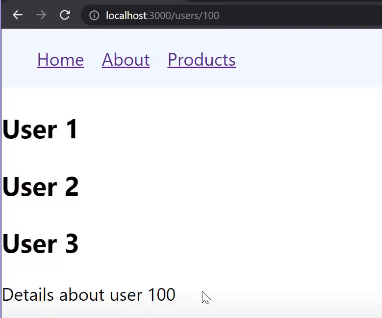
*useParams()* returns an object with key-value pairs which we store in *const params*.

We want to access *userId* parameter that we store in *userId*.

We destructor the same.



Output:

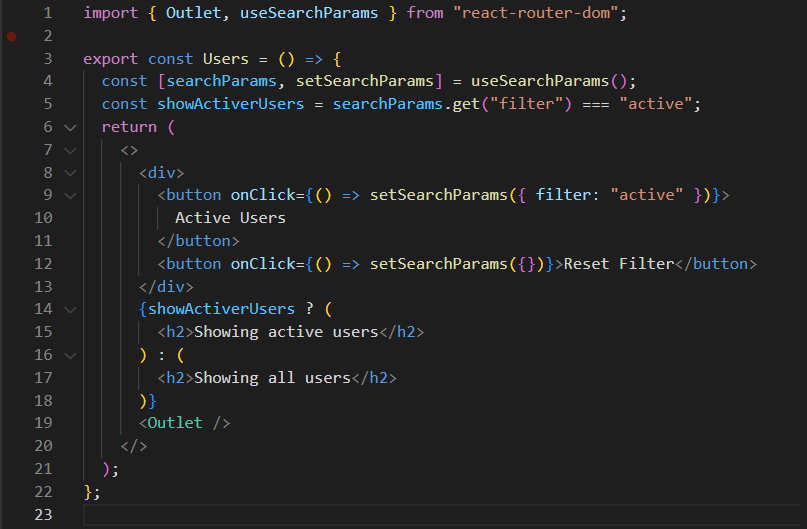


**11) Search Params**

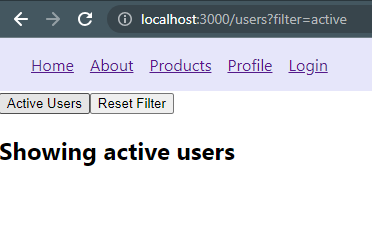
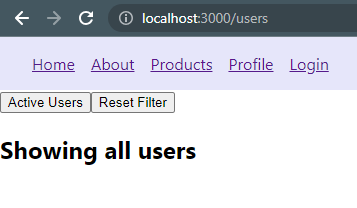
The *useSearchParams()* hook is used to read and modify the query string in the URL for the current location.

*useSearchParams()* returns an array of two values: the current location's search params and a function that may be used to update them (similar to *useState()* hook).

Modify *User.js* accordingly.



Output:

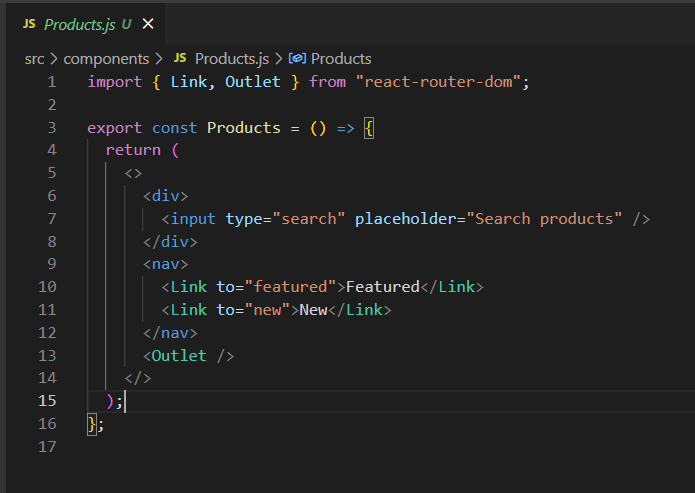


To make a check on the *filter* parameter, we declare a *const showActiveUsers* which is a Boolean value.

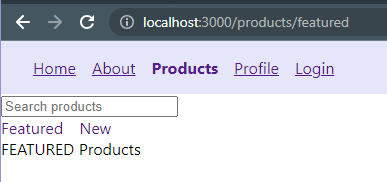
When active, we need to pass a custom message *“Showing active users”* else *“Showing all users”*. To do that we use the *get() function* on *searchparams*.

**12) Relative Links**

A relative link is a link that does not start with a forward slash and will inherit the closest route in which they are rendered.



Output:



**13) Lazy Loading**

Lazy loading is a technique where components not required on the homepage can be split into separate code bundles and downloaded only when the user navigates to that page.

We will implement Lazy Loading on *About.js* by making is bulky.

We need the dynamic import syntax for Lazy Loading and now we need a default export of the component () (in this case *About.js*)

*App.js:*



The *<React.Suspense>* component is required to provide a fallback UI while a given page loads. It needs to be defined in the following manner in *App.js.*

