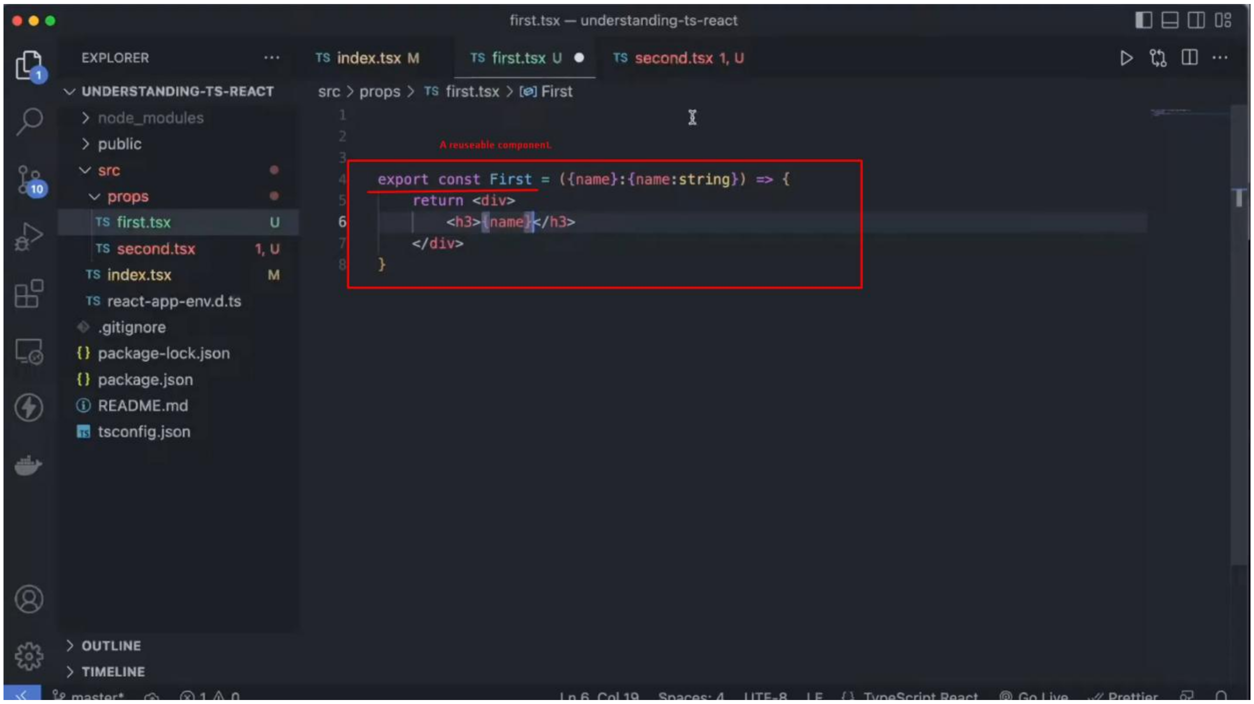
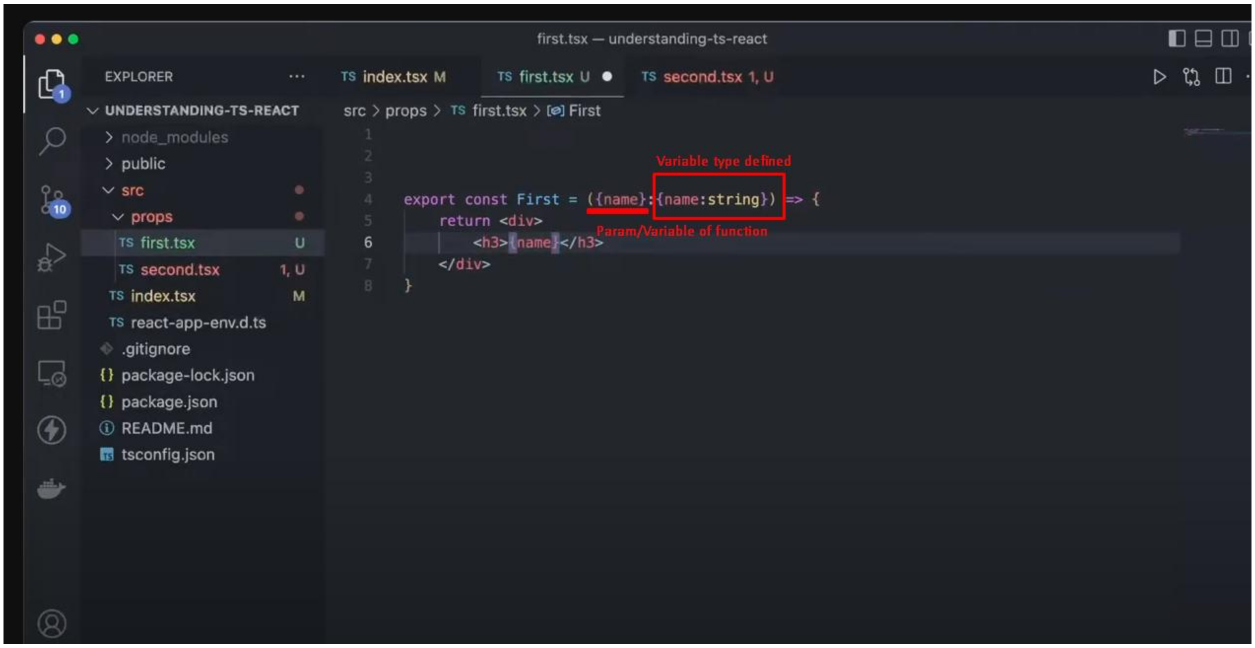


A reuseable component.

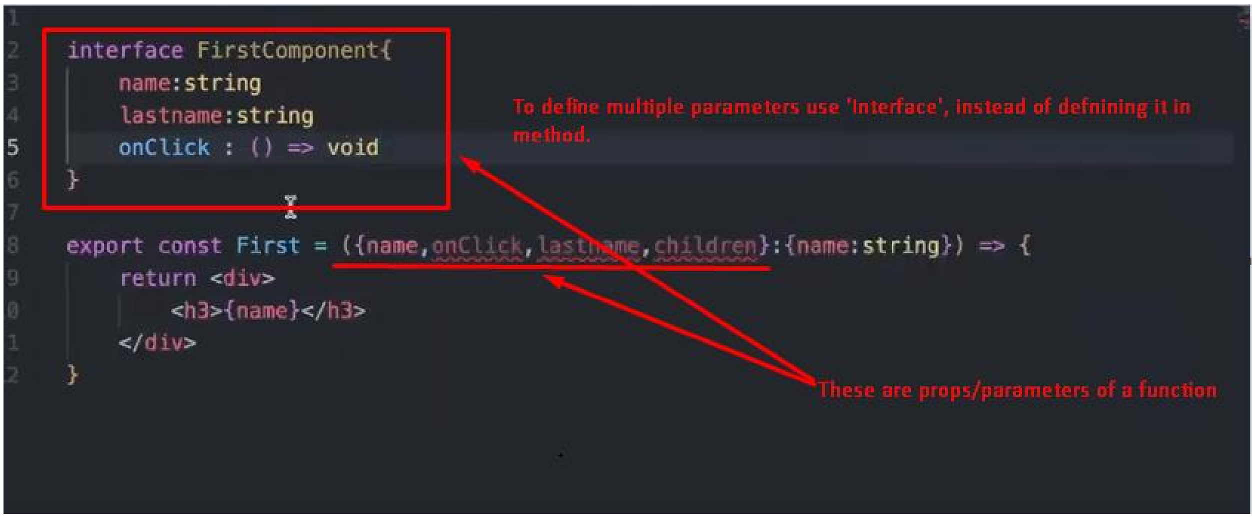
```tsx
export const First = ({name}:{name:string}) => {
    return <div>
        <h3>{name}</h3>
    </div>
}
```



Variable type defined

Param/Variable of function

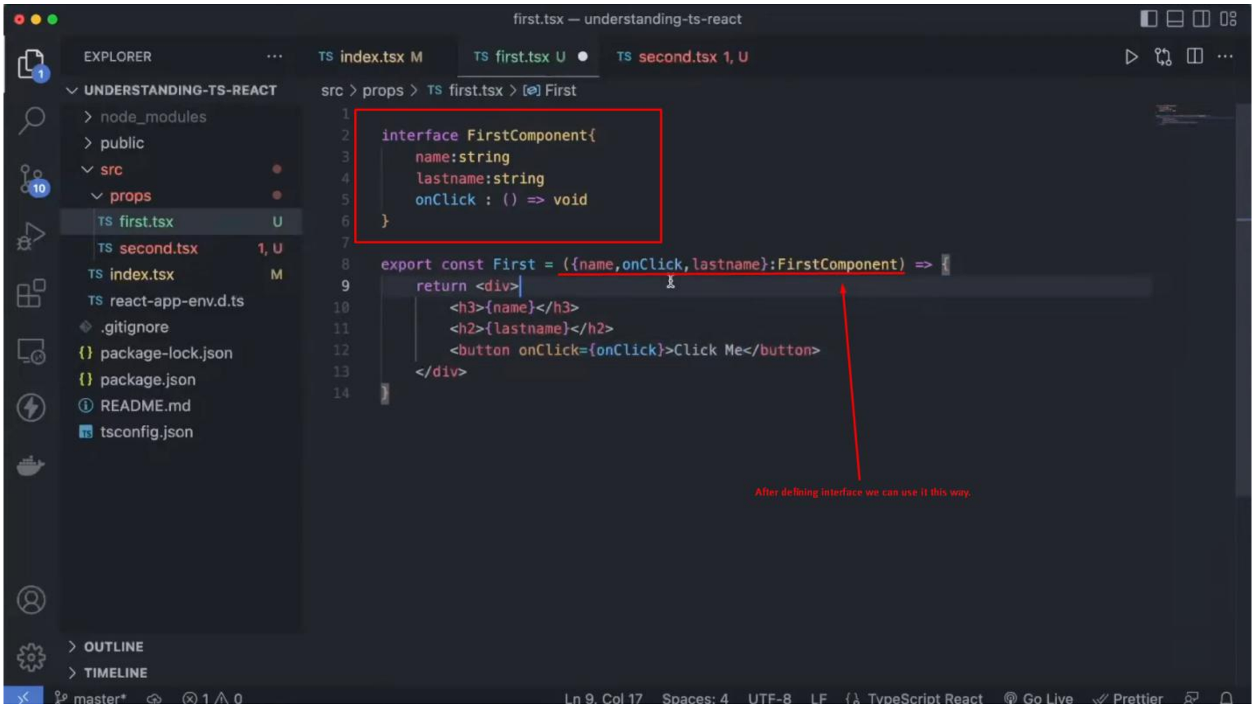
```tsx
export const First = ({name}:{name:string}) => {
    return <div>
        <h3>{name}</h3>
    </div>
}
```



```tsx

interface FirstComponent{
    name:string
    lastname:string
    onClick : () => void
}

export const First = ({name,onClick,lastname,children}:{name:string}) => {
    return <div>
        <h3>{name}</h3>
    </div>
}
```

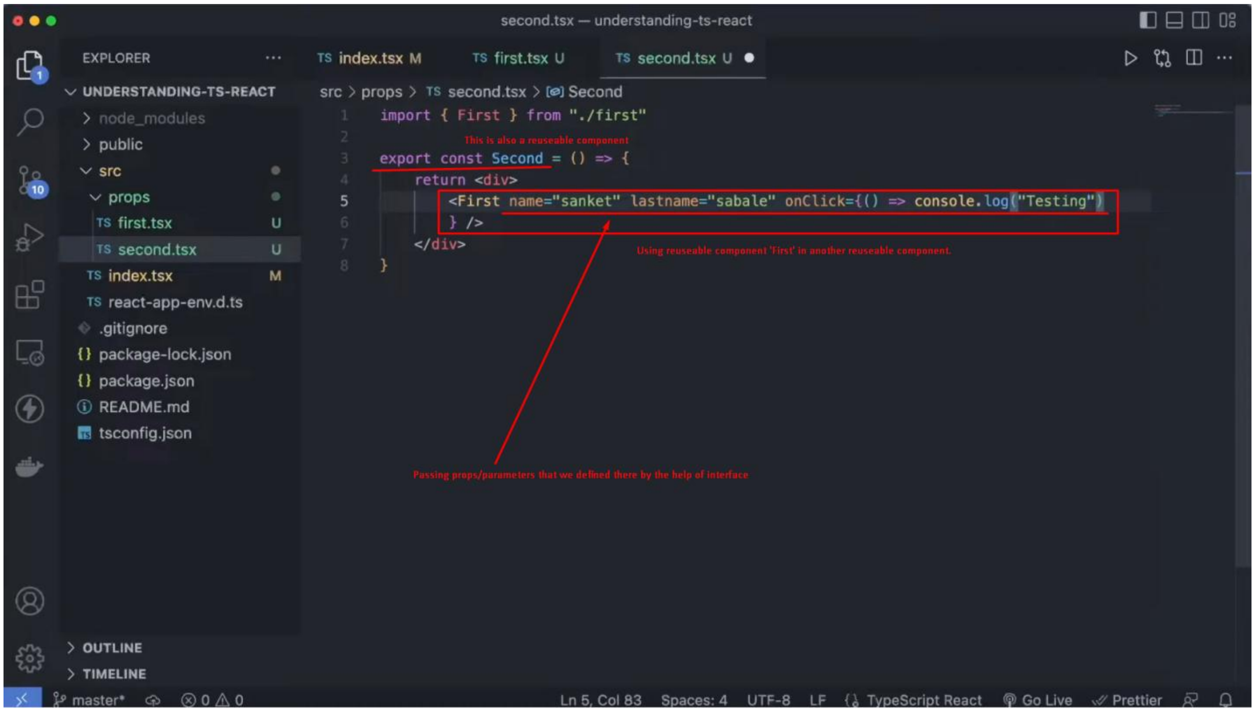
To define multiple parameters use 'Interface', instead of defnining it in method.

These are props/parameters of a function



```tsx

interface FirstComponent{
    name:string
    lastname:string
    onClick : () => void
}

export const First = ({name,onClick,lastname}:FirstComponent) => {
    return <div>
        <h3>{name}</h3>
        <h2>{lastname}</h2>
        <button onClick={onClick}>Click Me</button>
    </div>
}
```

After defining interface we can use it this way.

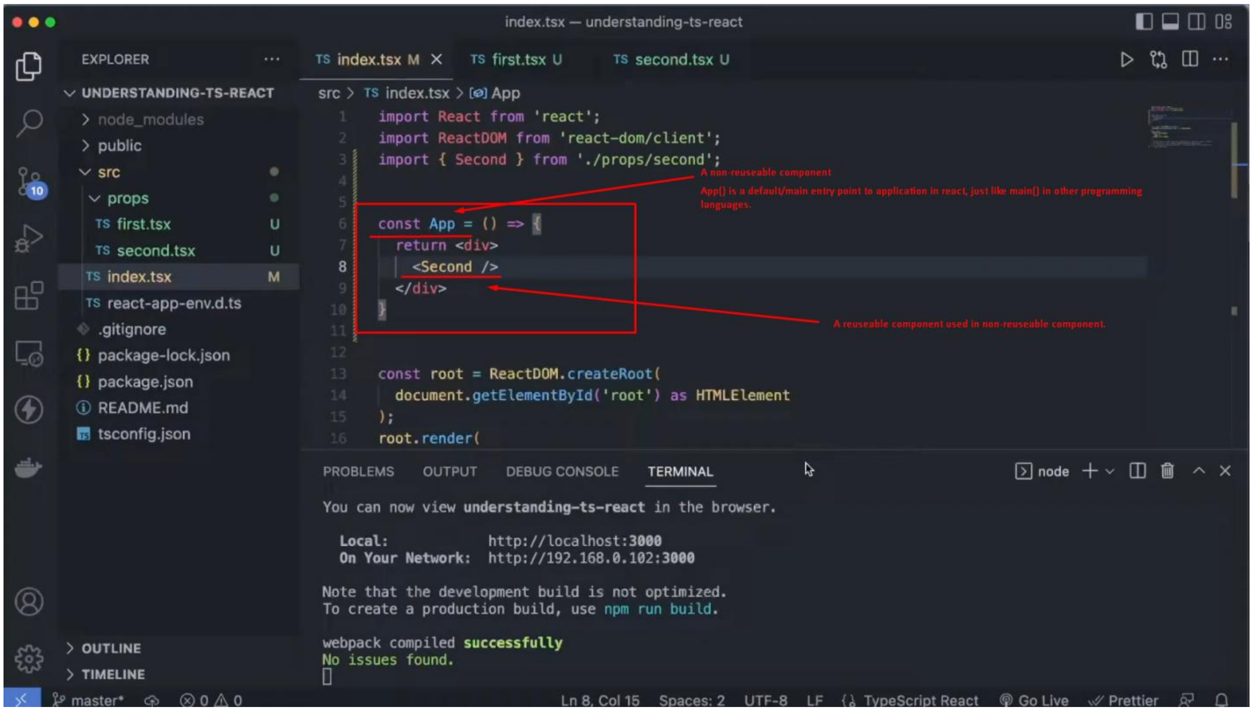


```tsx
import { First } from "./first"

export const Second = () => {
    return <div>
        <First name="sanket" lastname="sabale" onClick={() => console.log("Testing")
        } />
    </div>
}
```

This is also a reuseable component

Using reuseable component 'First' in another reuseable component.

Passing props/parameters that we defined there by the help of interface



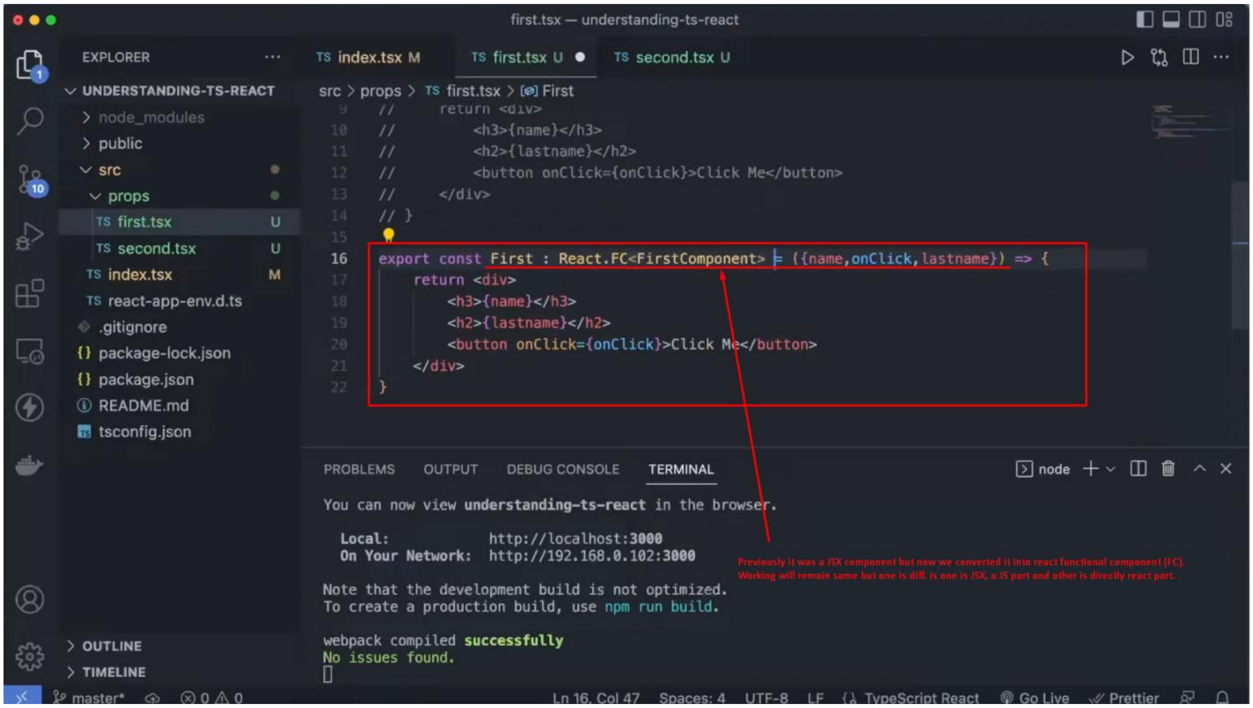
```tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Second } from './props/second';


const App = () => {
  return <div>
    <Second />
  </div>
}


const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
```

A non-reuseable component

App() is a default/main entry point to application in react, just like main() in other programming languages.

A reuseable component used in non-reuseable component.

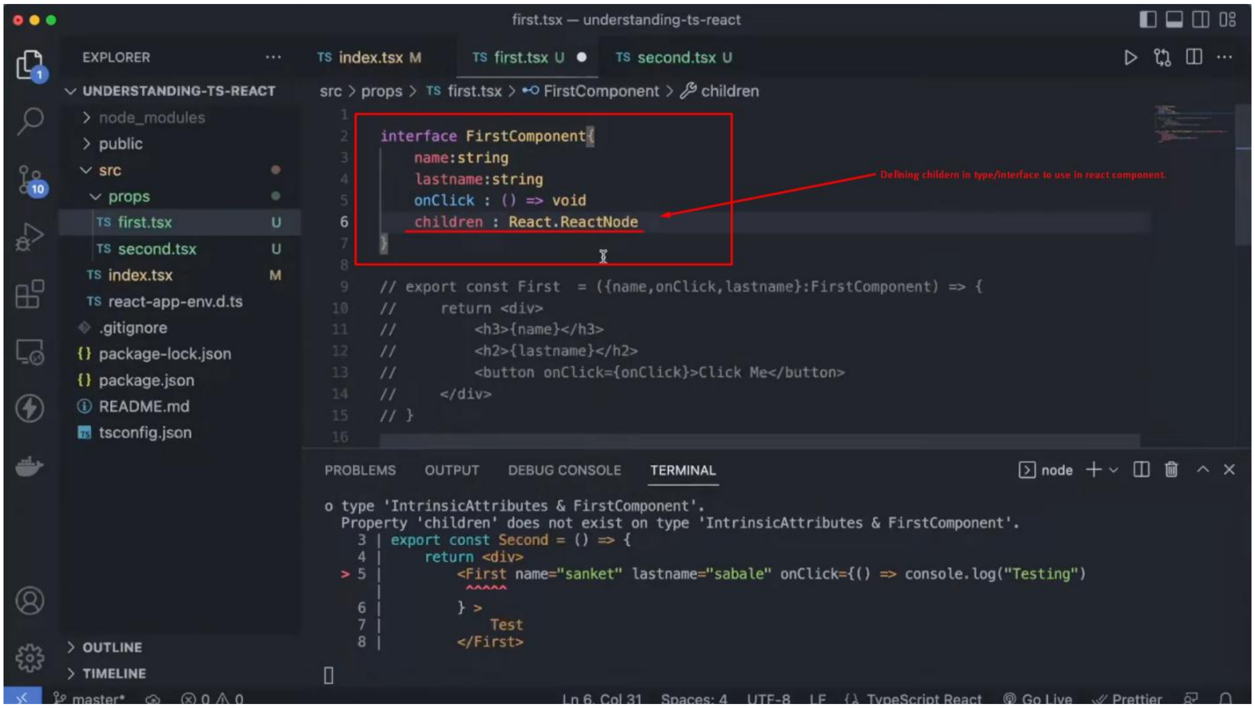
```
You can now view understanding-ts-react in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.0.102:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
No issues found.
```



```
//    return <div>
//        <h3>{name}</h3>
//        <h2>{lastname}</h2>
//        <button onClick={onClick}>Click Me</button>
//    </div>
// }

export const First : React.FC<FirstComponent> = ({name,onClick,lastname}) => {
    return <div>
        <h3>{name}</h3>
        <h2>{lastname}</h2>
        <button onClick={onClick}>Click Me</button>
    </div>
}
```

```
You can now view understanding-ts-react in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.0.102:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
No issues found.
```

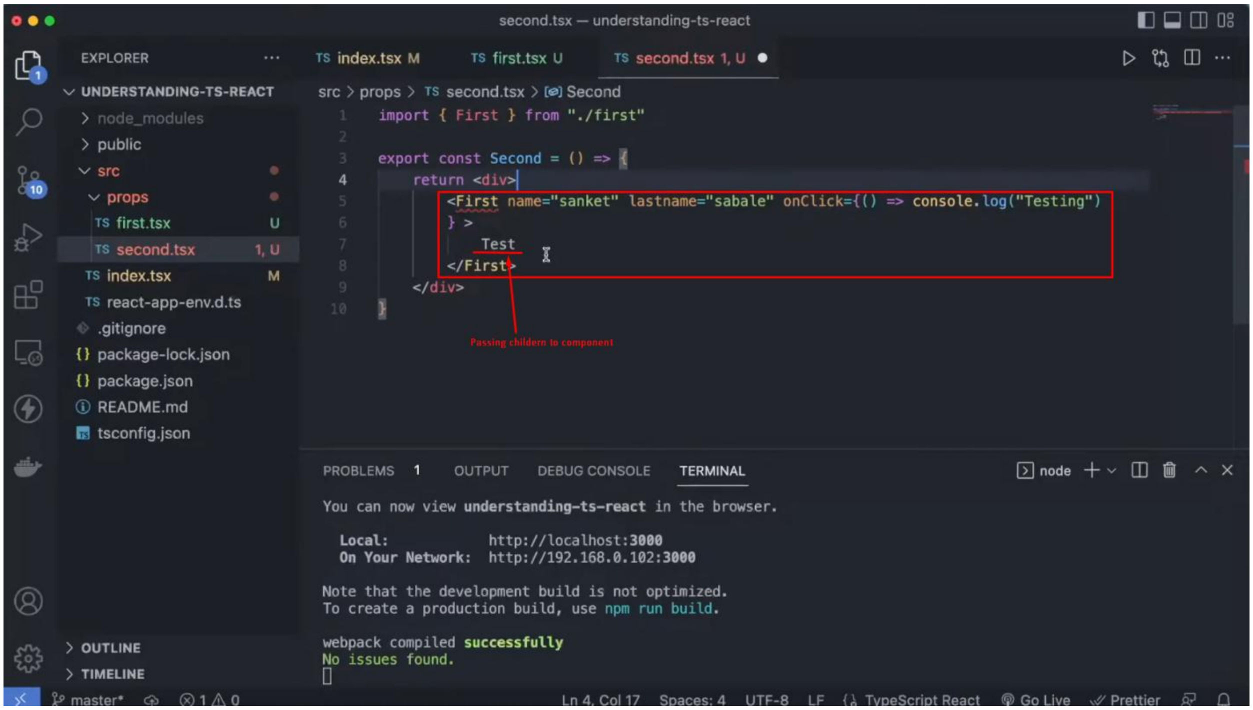
Previously it was a JSX component but now we converted it into react functional component (FC).
Working will remain same but one is diff. is one is JSX, a JS part and other is directly react part.



```

interface FirstComponent{
    name:string
    lastname:string
    onClick : () => void
    children : React.ReactNode
}

// export const First  = ({name,onClick,lastname}:FirstComponent) => {
//    return <div>
//        <h3>{name}</h3>
//        <h2>{lastname}</h2>
//        <button onClick={onClick}>Click Me</button>
//    </div>
// }

```

Defining childern in type/interface to use in react component.

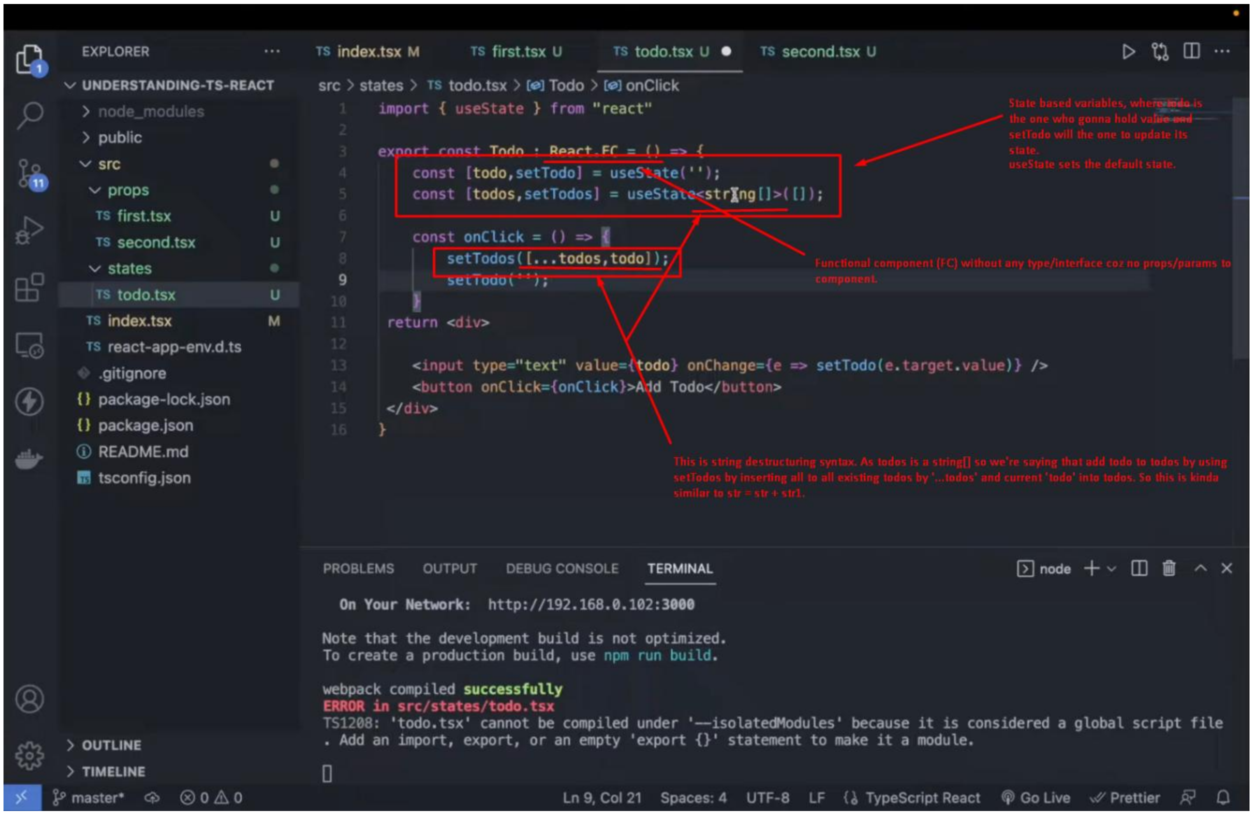
```
o type 'IntrinsicAttributes & FirstComponent'.
  Property 'children' does not exist on type 'IntrinsicAttributes & FirstComponent'.
    3 | export const Second = () => {
    4 |     return <div>
  > 5 |         <First name="sanket" lastname="sabale" onClick={() => console.log("Testing")
      |          ^^^^^
    6 |         } >
    7 |             Test
    8 |         </First>
```



```tsx
import { First } from "./first"

export const Second = () => {
    return <div>
        <First name="sanket" lastname="sabale" onClick={() => console.log("Testing")
        } >
            Test
        </First>
    </div>
}
```

Passing childern to component

```
You can now view understanding-ts-react in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.0.102:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
No issues found.
```



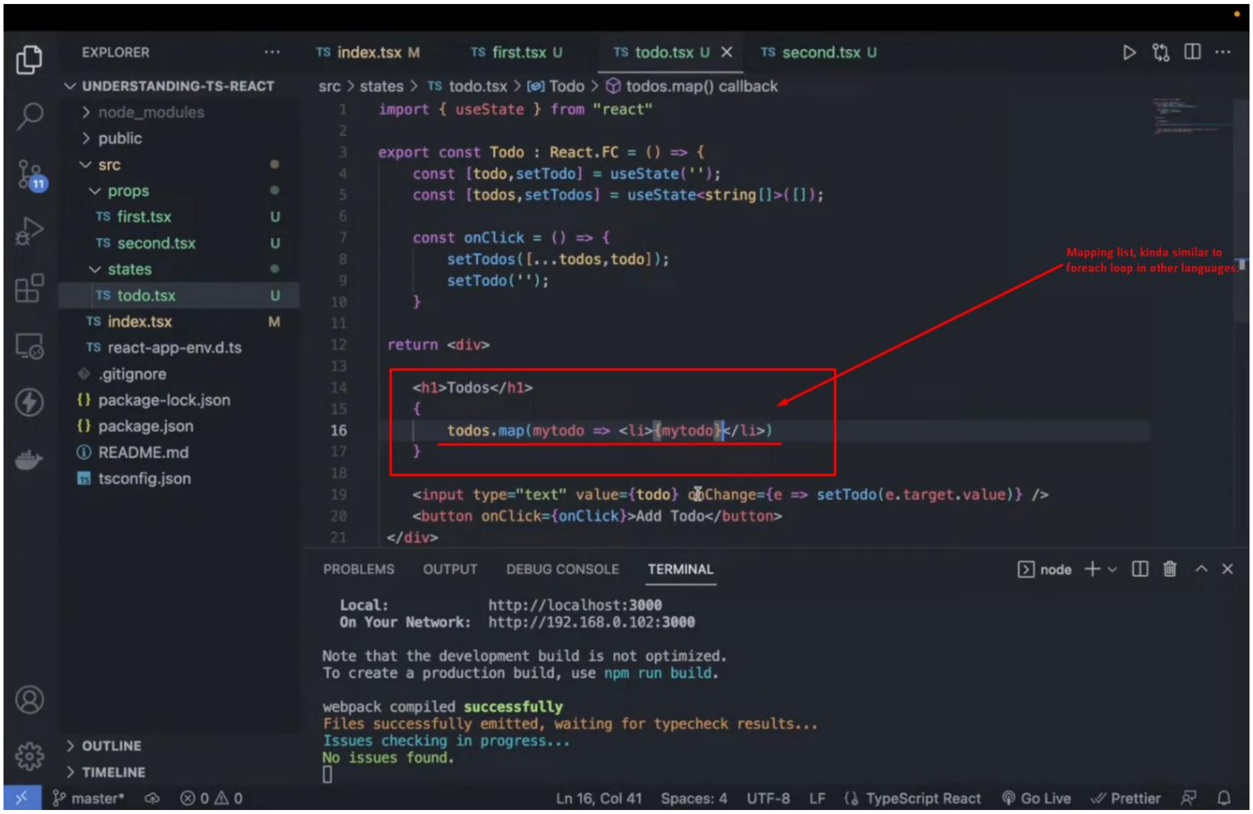
```tsx
import { useState } from "react"

export const Todo : React.FC = () => {
    const [todo,setTodo] = useState('');
    const [todos,setTodos] = useState<string[]>([]);

    const onClick = () => {
        setTodos([...todos,todo]);
        setTodo('');
    }
  return <div>

    <input type="text" value={todo} onChange={e => setTodo(e.target.value)} />
    <button onClick={onClick}>Add Todo</button>
  </div>
}
```

State based variables, where todo is the one who gonna hold value and setTodo will the one to update its state.
useState sets the default state.

Functional component (FC) without any type/interface coz no props/params to component.

This is string destructuring syntax. As todos is a string[] so we're saying that add todo to todos by using setTodos by inserting all to all existing todos by '...todos' and current 'todo' into todos. So this is kinda similar to str = str + str1.

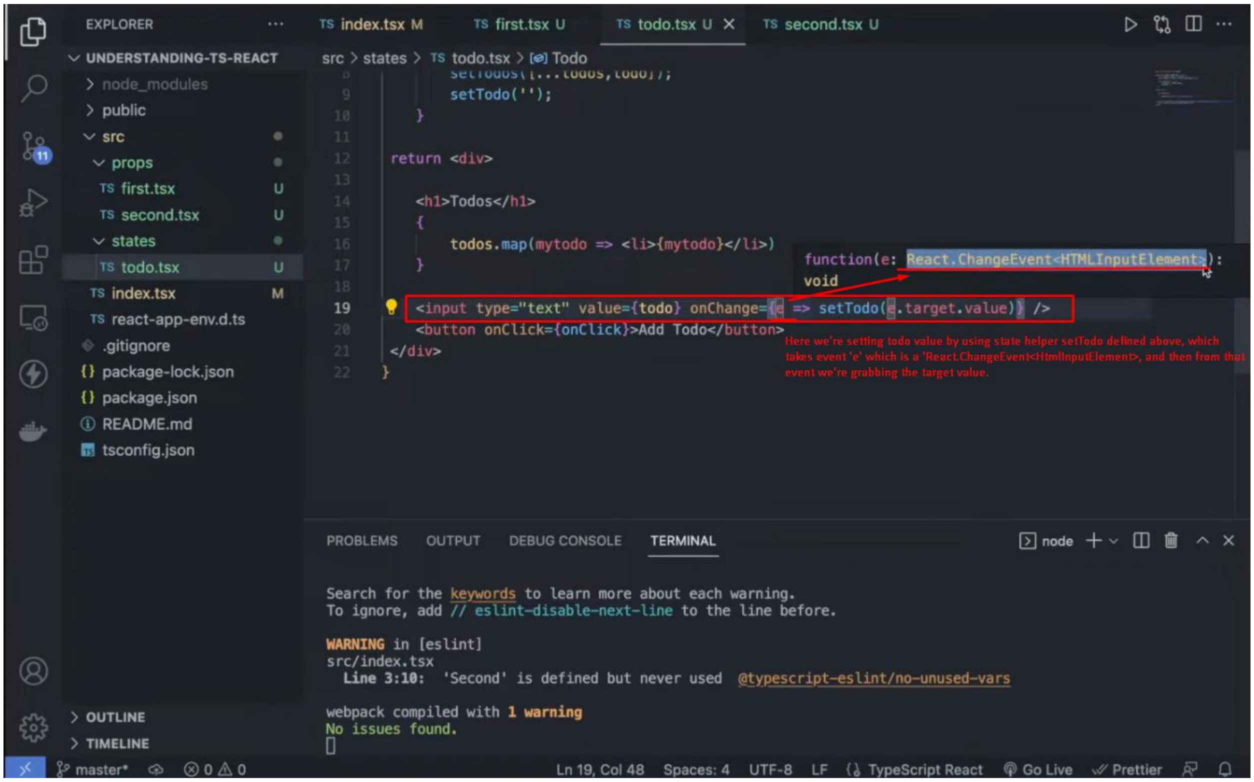
```
  On Your Network:  http://192.168.0.102:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
ERROR in src/states/todo.tsx
TS1208: 'todo.tsx' cannot be compiled under '--isolatedModules' because it is considered a global script file
. Add an import, export, or an empty 'export {}' statement to make it a module.
```



```
import { useState } from "react"

export const Todo : React.FC = () => {
    const [todo,setTodo] = useState('');
    const [todos,setTodos] = useState<string[]>([]);

    const onClick = () => {
        setTodos([...todos,todo]);
        setTodo('');
    }

  return <div>

    <h1>Todos</h1>
    {
        todos.map(mytodo => <li>{mytodo}</li>)
    }

    <input type="text" value={todo} onChange={e => setTodo(e.target.value)} />
    <button onClick={onClick}>Add Todo</button>
  </div>
```

Mapping list, kinda similar to foreach loop in other languages.

```
  Local:            http://localhost:3000
  On Your Network:  http://192.168.0.102:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
Files successfully emitted, waiting for typecheck results...
Issues checking in progress...
No issues found.
```

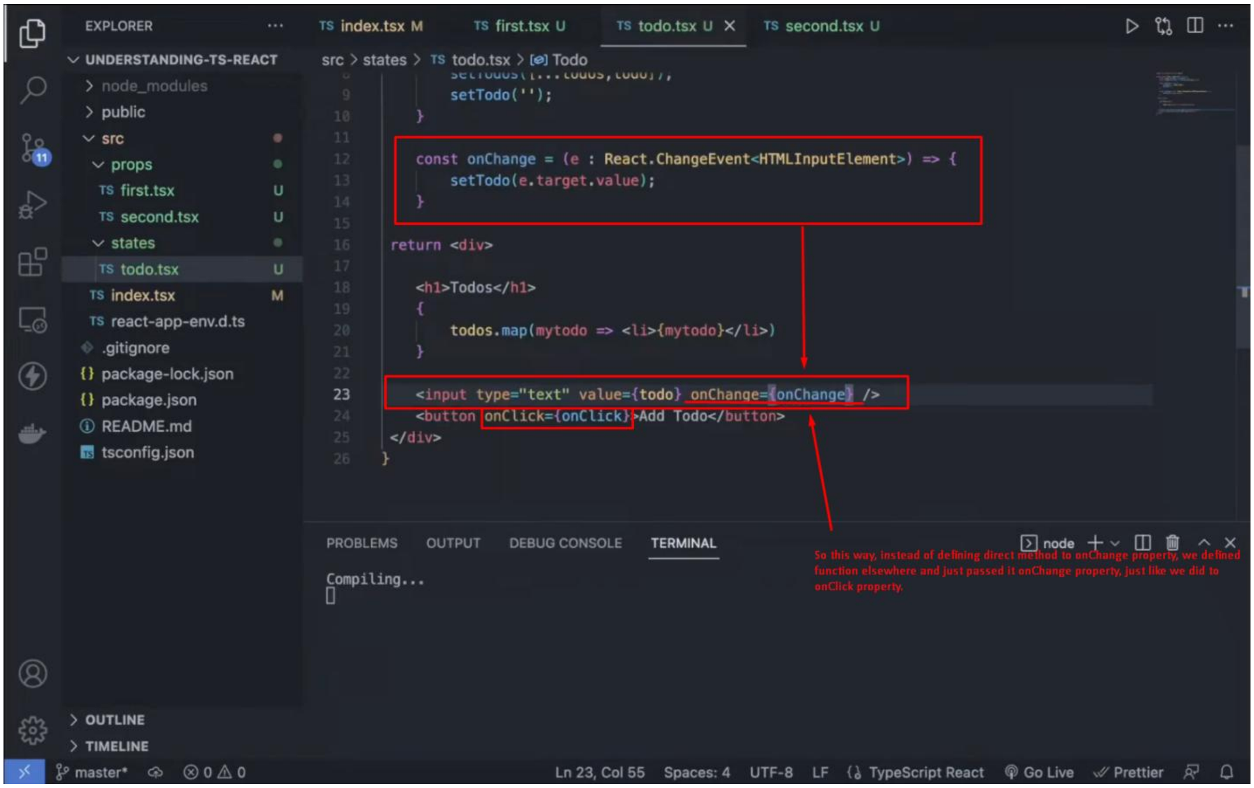


```
        setTodos([...todos,todo]);
        setTodo('');
    }

  return <div>

    <h1>Todos</h1>
    {
        todos.map(mytodo => <li>{mytodo}</li>)
    }

    <input type="text" value={todo} onChange={e => setTodo(e.target.value)} />
    <button onClick={onClick}>Add Todo</button>
  </div>
}
```

```
function(e: React.ChangeEvent<HTMLInputElement>):
void
```

Here we're setting todo value by using state helper setTodo defined above, which takes event 'e' which is a 'React.ChangeEvent<HtmlInputElement>, and then from that event we're grabbing the target value.

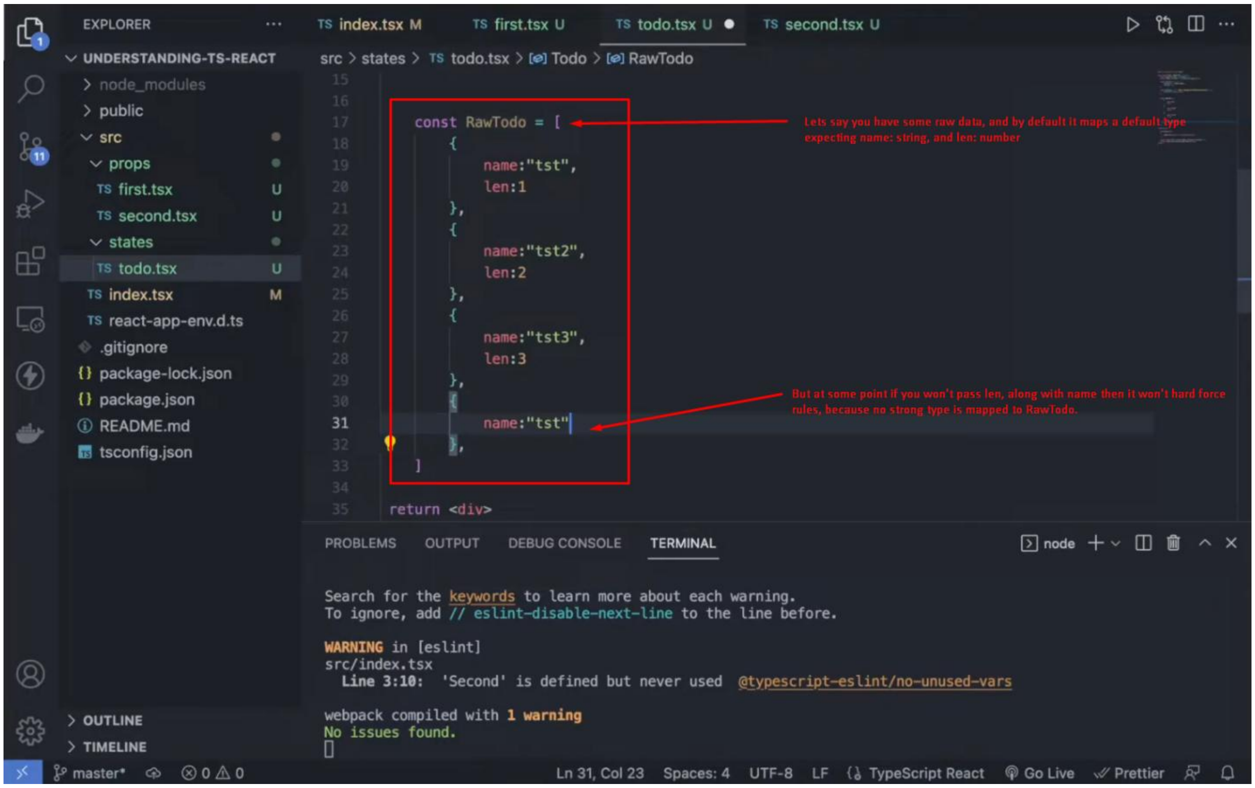
```
Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.

WARNING in [eslint]
src/index.tsx
  Line 3:10:  'Second' is defined but never used  @typescript-eslint/no-unused-vars

webpack compiled with 1 warning
No issues found.
```



```
        setTodos([...todos,todo]);
        setTodo('');
    }

    const onChange = (e : React.ChangeEvent<HTMLInputElement>) => {
        setTodo(e.target.value);
    }

  return <div>

    <h1>Todos</h1>
    {
        todos.map(mytodo => <li>{mytodo}</li>)
    }

    <input type="text" value={todo} onChange={onChange} />
    <button onClick={onClick}>Add Todo</button>
  </div>
}
```

So this way, instead of defining direct method to onChange property, we defined function elsewhere and just passed it onChange property, just like we did to onClick property.

```
Compiling...
```

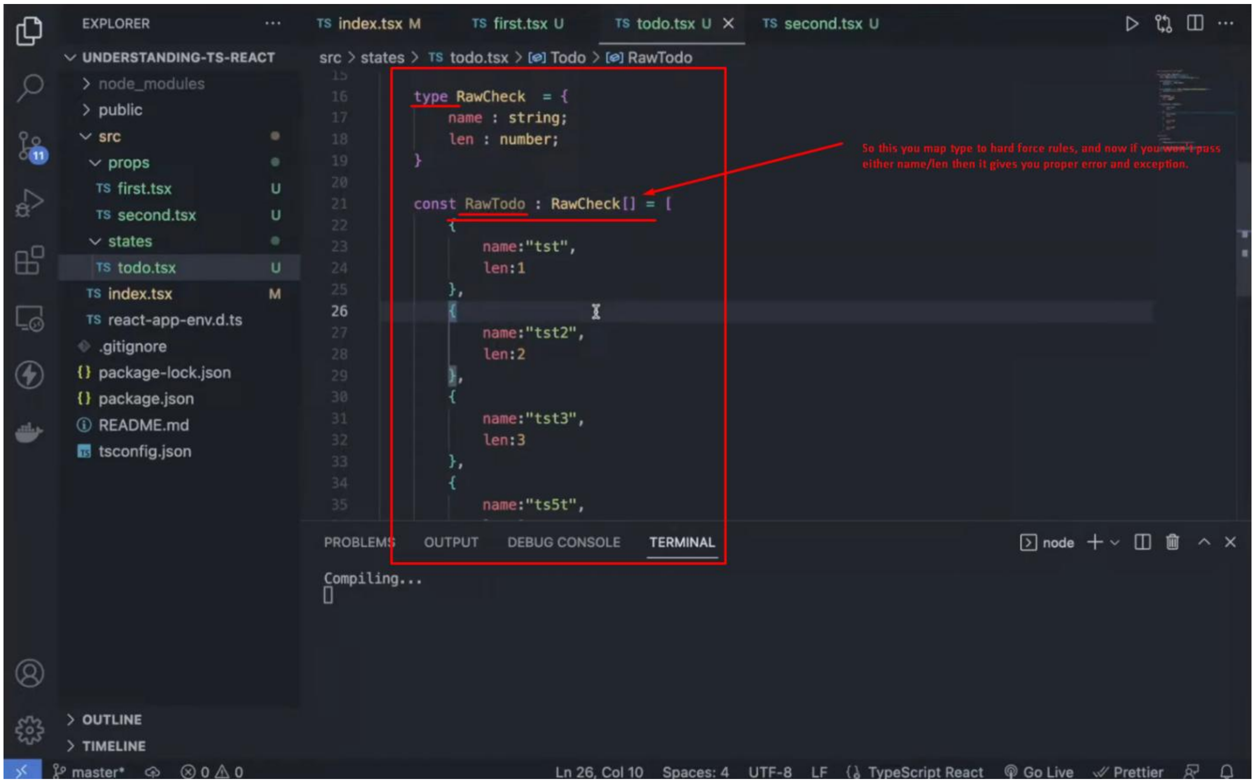


```
    const RawTodo = [
        {
            name:"tst",
            len:1
        },
        {
            name:"tst2",
            len:2
        },
        {
            name:"tst3",
            len:3
        },
        {
            name:"tst"
        },
    ]

  return <div>
```

Lets say you have some raw data, and by default it maps a default type expecting name: string, and len: number

But at some point if you won't pass len, along with name then it won't hard force rules, because no strong type is mapped to RawTodo.

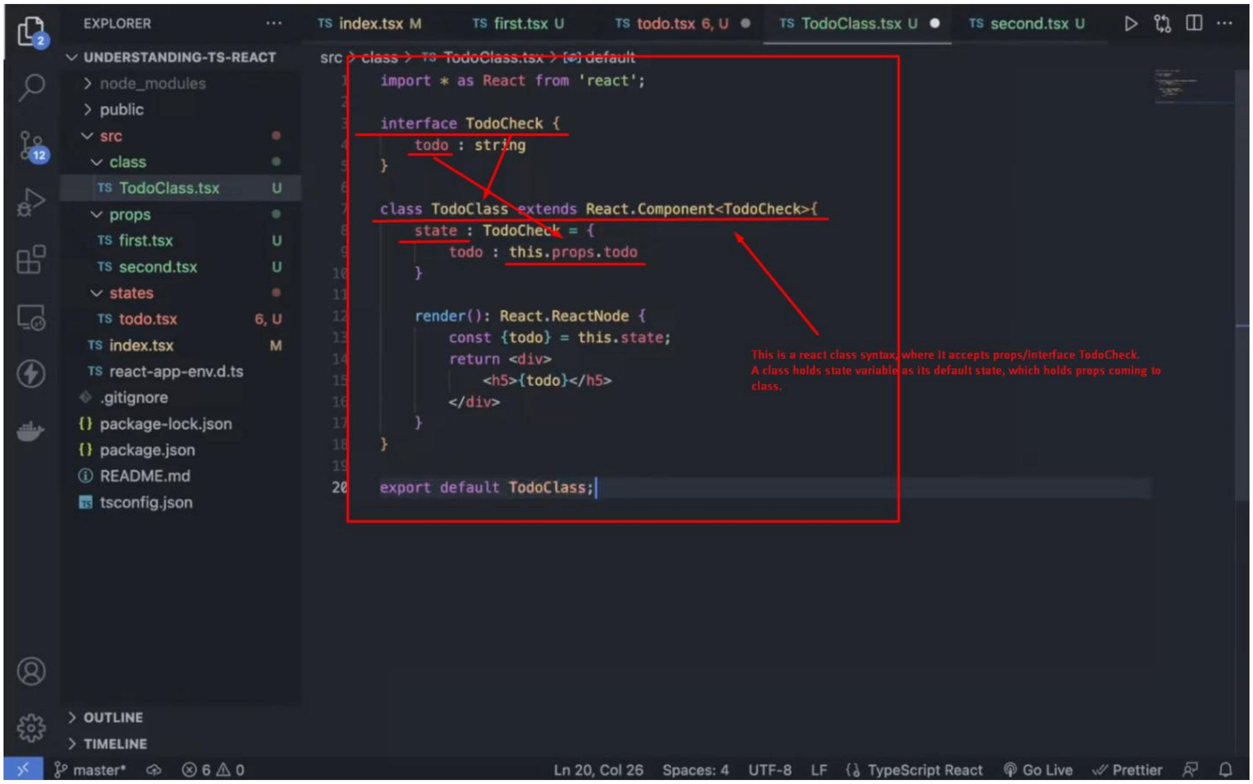
```
Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.

WARNING in [eslint]
src/index.tsx
  Line 3:10:  'Second' is defined but never used  @typescript-eslint/no-unused-vars

webpack compiled with 1 warning
No issues found.
```



```
type RawCheck  = {
    name : string;
    len : number;
}

const RawTodo : RawCheck[] = [
    {
        name:"tst",
        len:1
    },
    {
        name:"tst2",
        len:2
    },
    {
        name:"tst3",
        len:3
    },
    {
        name:"ts5t",
```

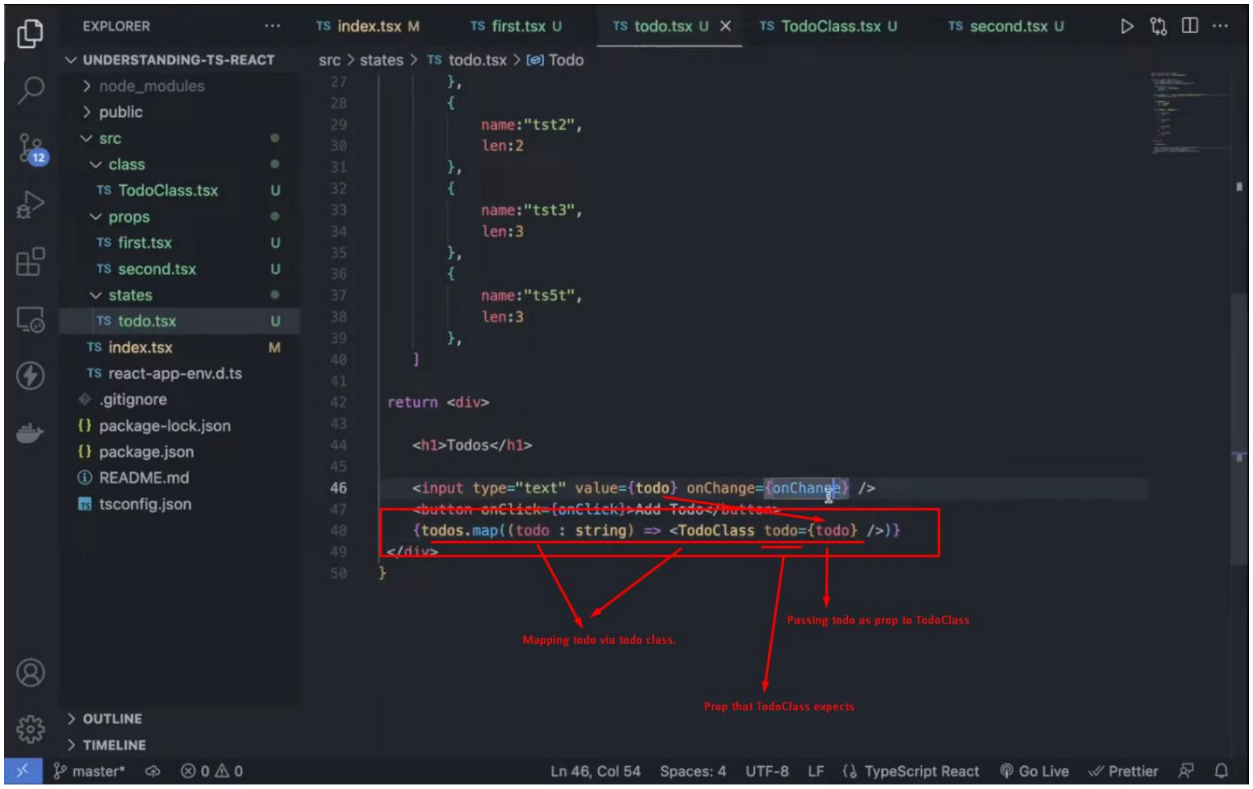
```
Compiling...
```

So this you map type to hard force rules, and now if you won't pass either name/len then it gives you proper error and exception.



```
import * as React from 'react';

interface TodoCheck {
    todo : string
}

class TodoClass extends React.Component<TodoCheck>{
    state : TodoCheck = {
        todo : this.props.todo
    }

    render(): React.ReactNode {
        const {todo} = this.state;
        return <div>
            <h5>{todo}</h5>
        </div>
    }
}

export default TodoClass;
```

This is a react class syntax, where it accepts props/interface TodoCheck.
A class holds state variable as its default state, which holds props coming to class.

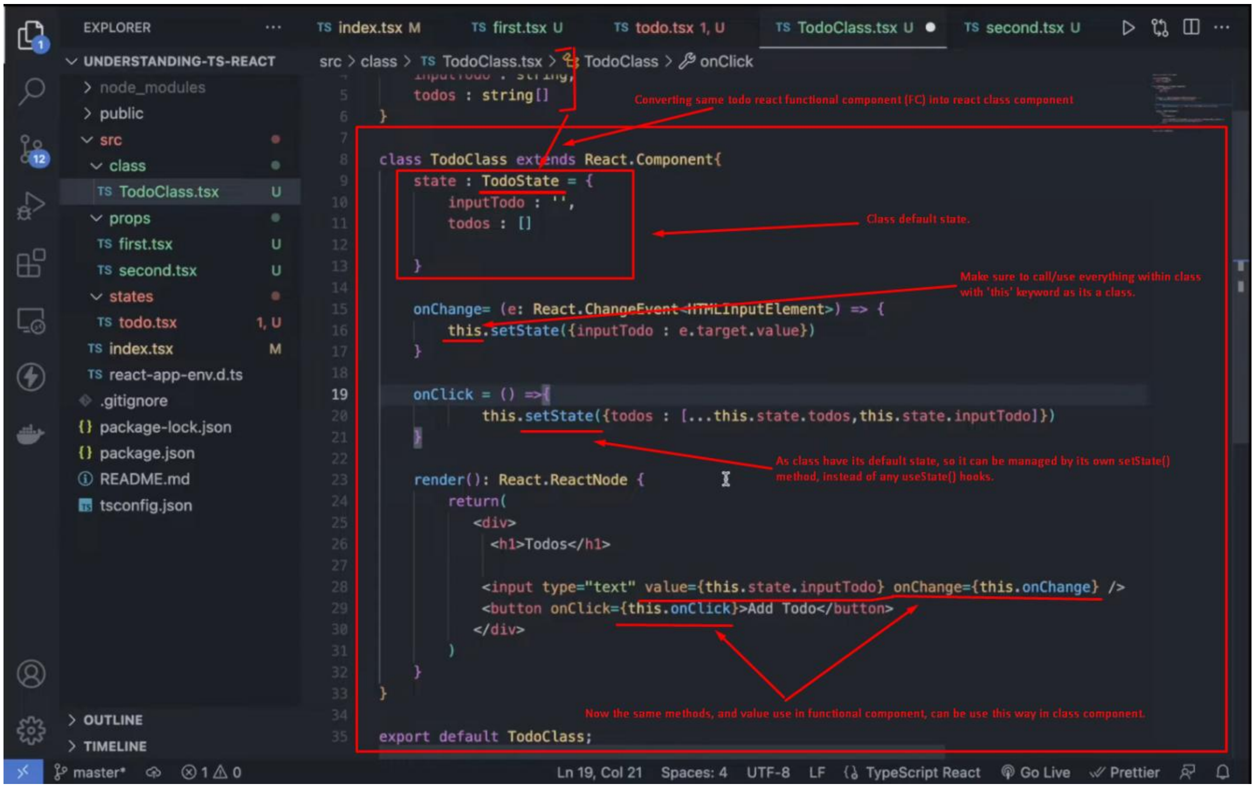


```
            },
            {
                name:"tst2",
                len:2
            },
            {
                name:"tst3",
                len:3
            },
            {
                name:"ts5t",
                len:3
            },
        ]

    return <div>

        <h1>Todos</h1>

        <input type="text" value={todo} onChange={onChange} />
        ~~<button onClick={onClick}>Add Todo</button>~~
        {todos.map((todo : string) => <TodoClass todo={todo} />)}
    </div>
}
```

Mapping todo via todo class.

Passing todo as prop to TodoClass

Prop that TodoClass expects



```
    inputTodo : string,
    todos : string[]
}

class TodoClass extends React.Component{
    state : TodoState = {
        inputTodo : '',
        todos : []

    }

    onChange= (e: React.ChangeEvent<HTMLInputElement>) => {
        this.setState({inputTodo : e.target.value})
    }

    onClick = () =>{
            this.setState({todos : [...this.state.todos,this.state.inputTodo]})
    }

    render(): React.ReactNode {
        return(
            <div>
              <h1>Todos</h1>

             <input type="text" value={this.state.inputTodo} onChange={this.onChange} />
             <button onClick={this.onClick}>Add Todo</button>
            </div>
        )
    }
}

export default TodoClass;
```

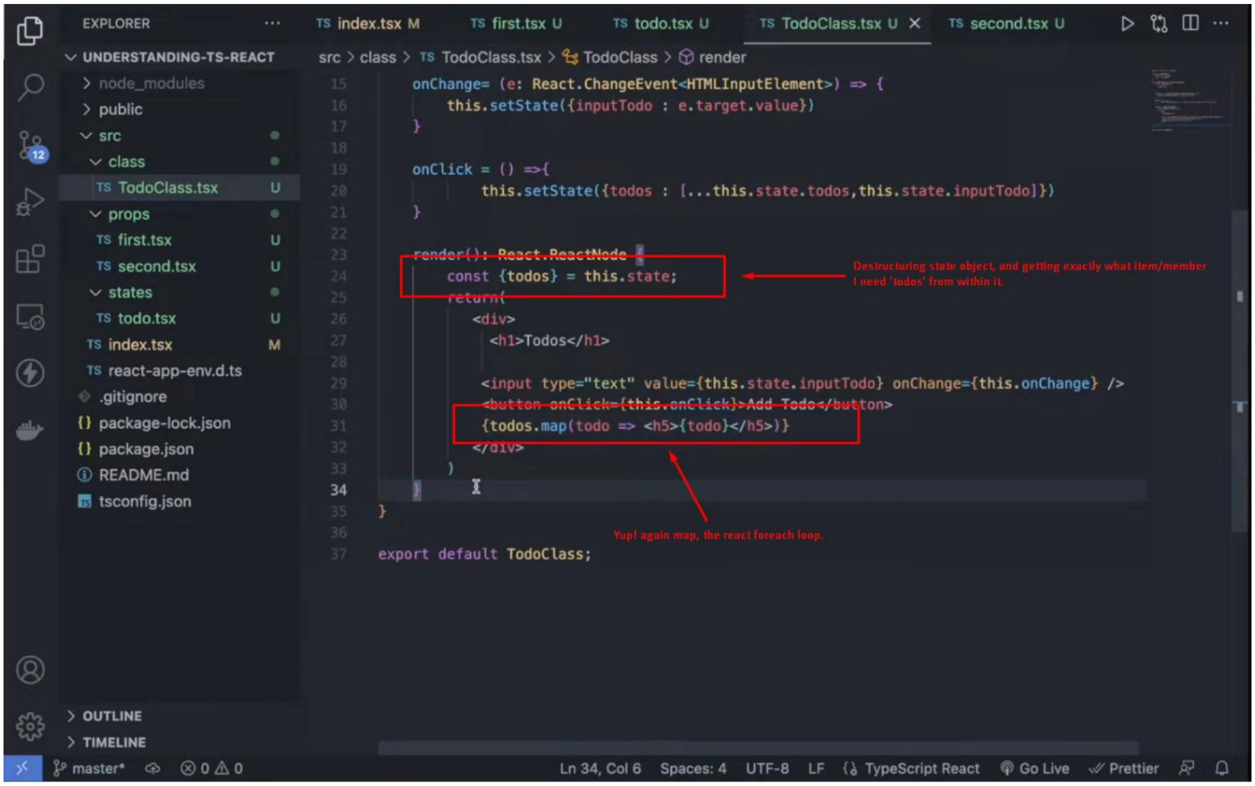
Converting same todo react functional component (FC) into react class component

Class default state.

Make sure to call/use everything within class with 'this' keyword as its a class.

As class have its default state, so it can be managed by its own setState() method, instead of any useState() hooks.

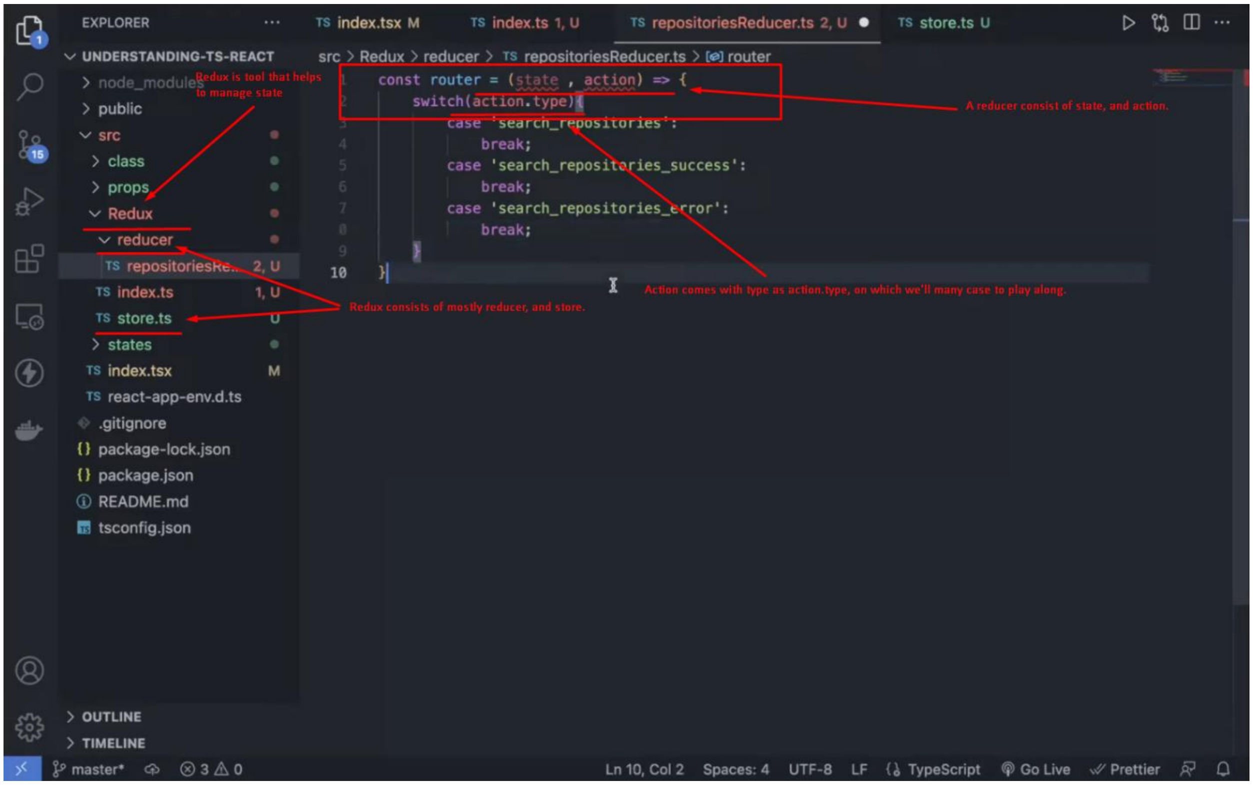
Now the same methods, and value use in functional component, can be use this way in class component.



```tsx
    onChange= (e: React.ChangeEvent<HTMLInputElement>) => {
        this.setState({inputTodo : e.target.value})
    }

    onClick = () =>{
          this.setState({todos : [...this.state.todos,this.state.inputTodo]})
    }

    render(): React.ReactNode {
        const {todos} = this.state;
        return(
            <div>
              <h1>Todos</h1>

              <input type="text" value={this.state.inputTodo} onChange={this.onChange} />
              <button onClick={this.onClick}>Add Todo</button>
              {todos.map(todo => <h5>{todo}</h5>)}
            </div>
        )
    }
}

export default TodoClass;
```

Destructuring state object, and getting exactly what item/member I need 'todos' from within it.

Yup! again map, the react foreach loop.



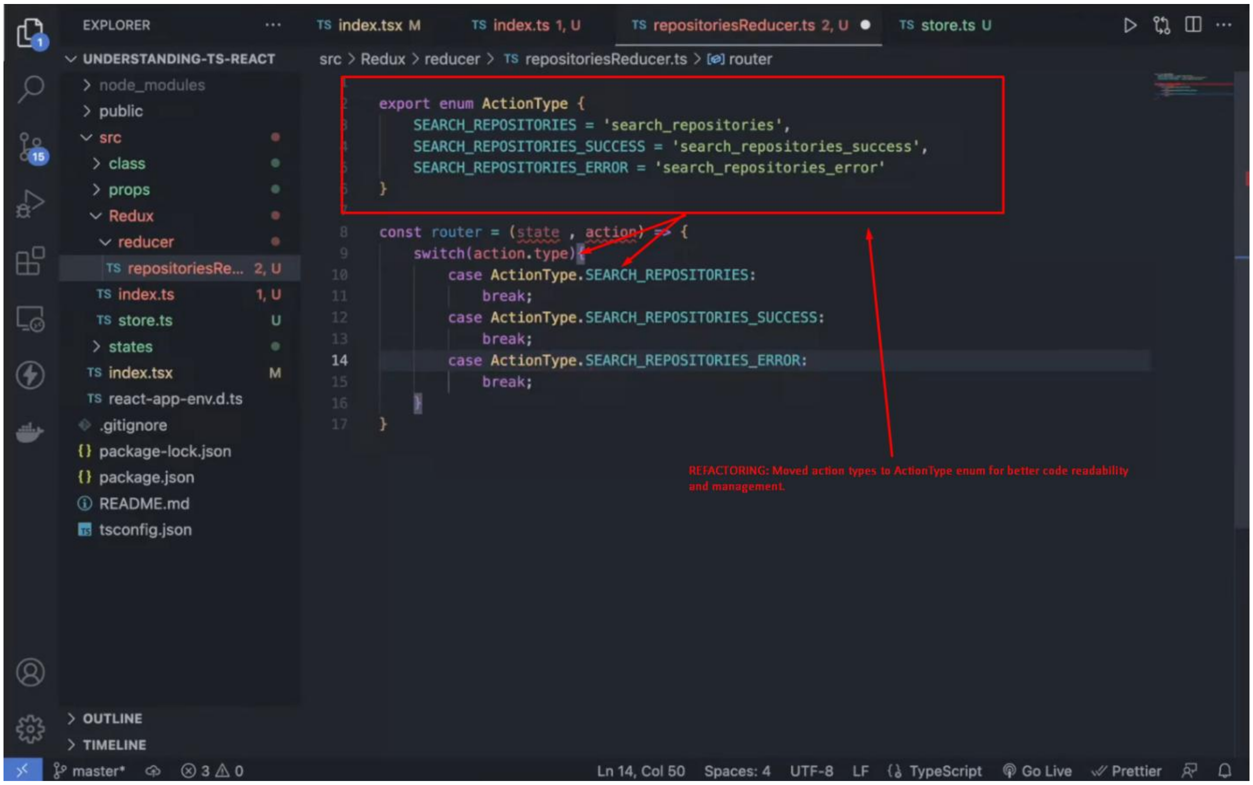
```ts
const router = (state , action) => {
    switch(action.type){
        case 'search_repositories':
            break;
        case 'search_repositories_success':
            break;
        case 'search_repositories_error':
            break;
    }
}
```

Redux is tool that helps to manage state

A reducer consist of state, and action.

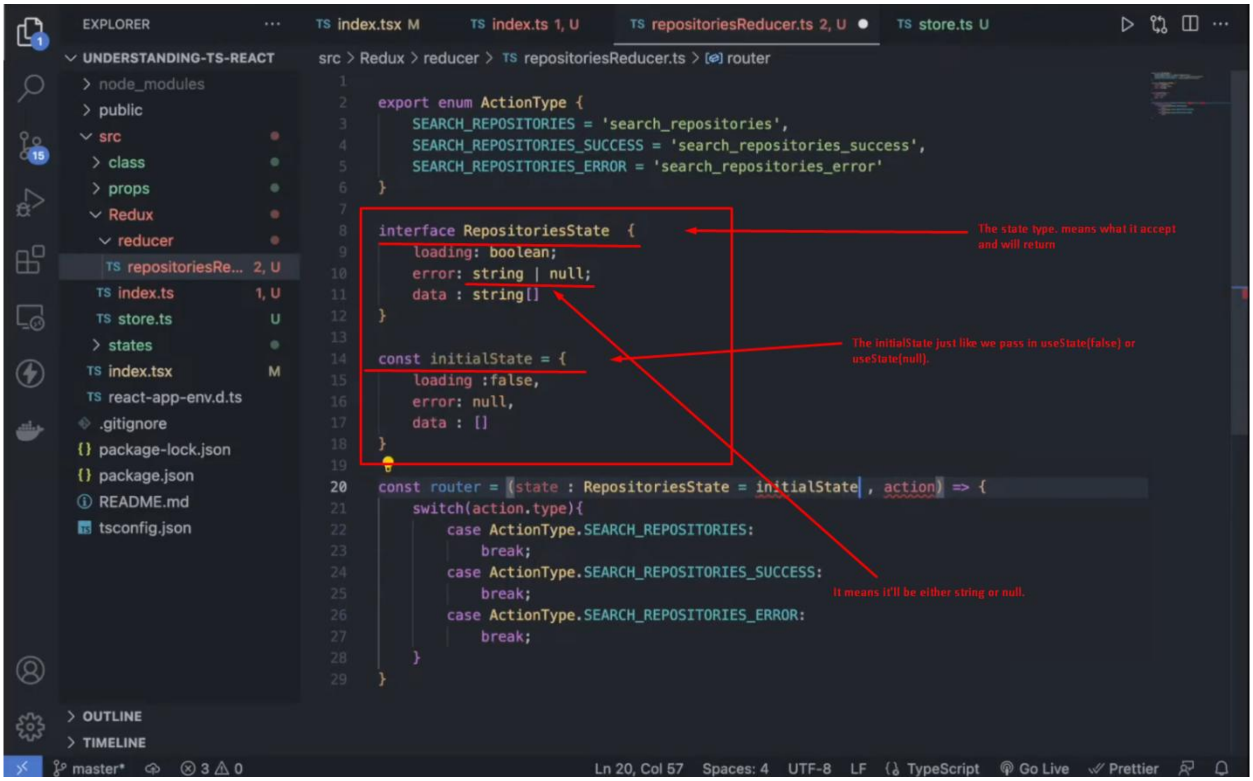
Redux consists of mostly reducer, and store.

Action comes with type as action.type, on which we'll many case to play along.



```
export enum ActionType {
    SEARCH_REPOSITORIES = 'search_repositories',
    SEARCH_REPOSITORIES_SUCCESS = 'search_repositories_success',
    SEARCH_REPOSITORIES_ERROR = 'search_repositories_error'
}

const router = (state , action) => {
    switch(action.type){
        case ActionType.SEARCH_REPOSITORIES:
            break;
        case ActionType.SEARCH_REPOSITORIES_SUCCESS:
            break;
        case ActionType.SEARCH_REPOSITORIES_ERROR:
            break;
    }
}
```

REFACTORING: Moved action types to ActionType enum for better code readability and management.

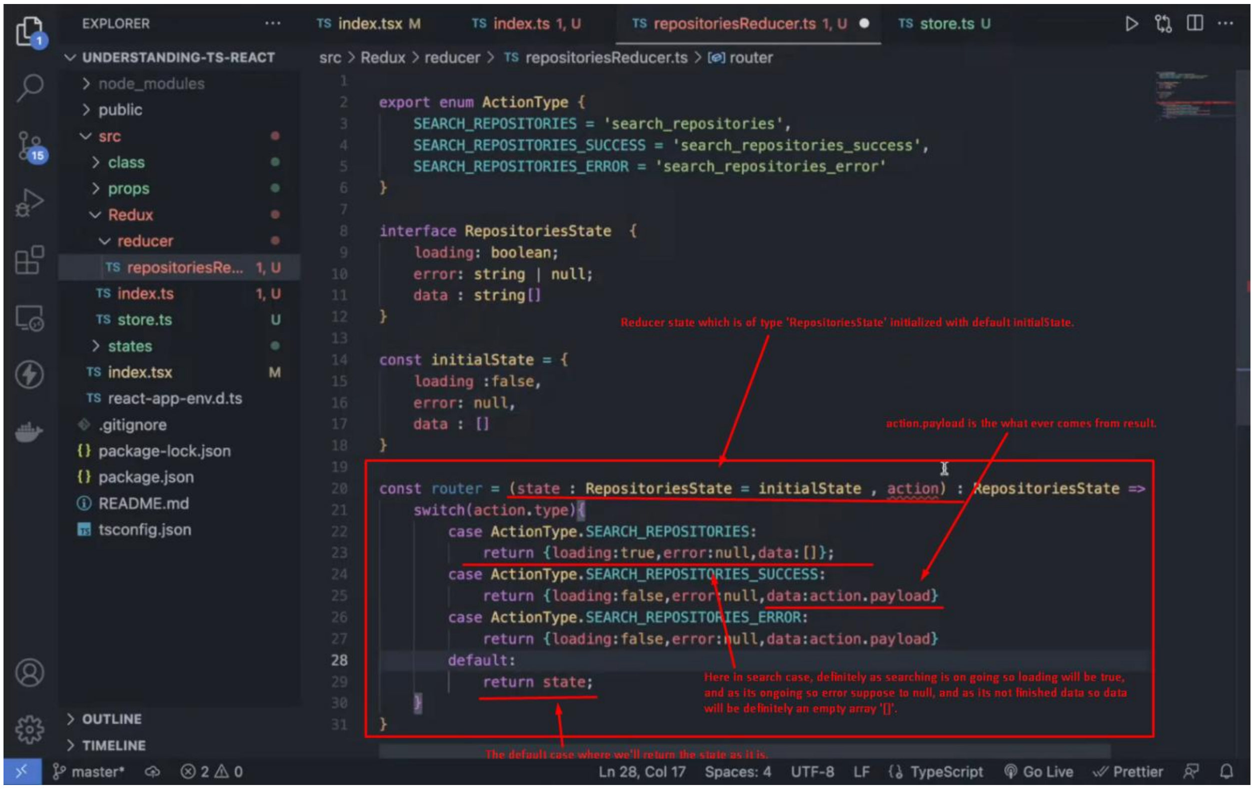


```

export enum ActionType {
    SEARCH_REPOSITORIES = 'search_repositories',
    SEARCH_REPOSITORIES_SUCCESS = 'search_repositories_success',
    SEARCH_REPOSITORIES_ERROR = 'search_repositories_error'
}

interface RepositoriesState  {
    loading: boolean;
    error: string | null;
    data : string[]
}

const initialState = {
    loading :false,
    error: null,
    data : []
}

const router = (state : RepositoriesState = initialState , action) => {
    switch(action.type){
        case ActionType.SEARCH_REPOSITORIES:
            break;
        case ActionType.SEARCH_REPOSITORIES_SUCCESS:
            break;
        case ActionType.SEARCH_REPOSITORIES_ERROR:
            break;
    }
}
```

The state type. means what it accept and will return

The initialState just like we pass in useState(false) or useState(null).

It means it'll be either string or null.



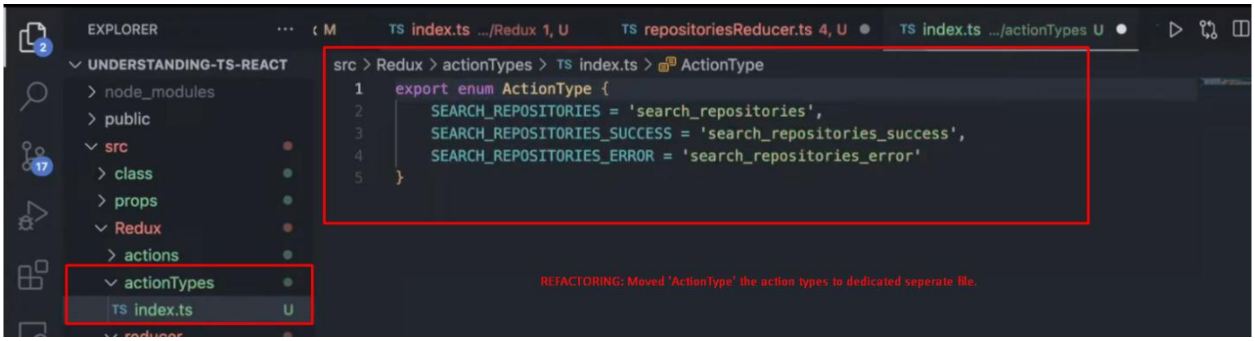
```
export enum ActionType {
    SEARCH_REPOSITORIES = 'search_repositories',
    SEARCH_REPOSITORIES_SUCCESS = 'search_repositories_success',
    SEARCH_REPOSITORIES_ERROR = 'search_repositories_error'
}

interface RepositoriesState  {
    loading: boolean;
    error: string | null;
    data : string[]
}

const initialState = {
    loading :false,
    error: null,
    data : []
}

const router = (state : RepositoriesState = initialState , action) : RepositoriesState =>
    switch(action.type){
        case ActionType.SEARCH_REPOSITORIES:
            return {loading:true,error:null,data:[]};
        case ActionType.SEARCH_REPOSITORIES_SUCCESS:
            return {loading:false,error:null,data:action.payload}
        case ActionType.SEARCH_REPOSITORIES_ERROR:
            return {loading:false,error:null,data:action.payload}
        default:
            return state;
    }
}
```

Reducer state which is of type 'RepositoriesState' initialized with default initialState.

action.payload is the what ever comes from result.

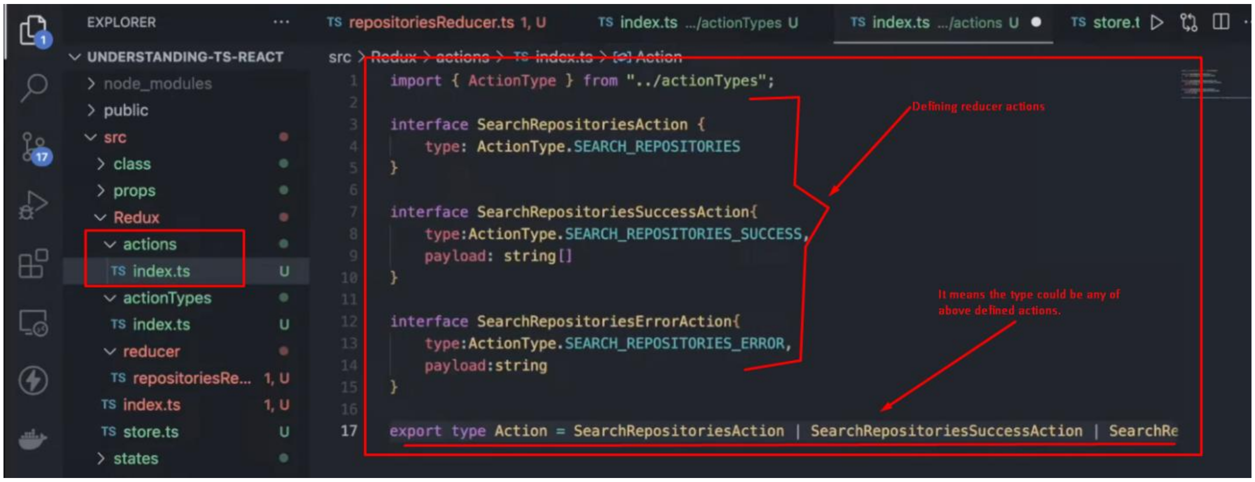
Here in search case, definitely as searching is on going so loading will be true, and as its ongoing so error suppose to null, and as its not finished data so data will be definitely an empty array '[]'.

The default case where we'll return the state as it is.



```
export enum ActionType {
    SEARCH_REPOSITORIES = 'search_repositories',
    SEARCH_REPOSITORIES_SUCCESS = 'search_repositories_success',
    SEARCH_REPOSITORIES_ERROR = 'search_repositories_error'
}
```

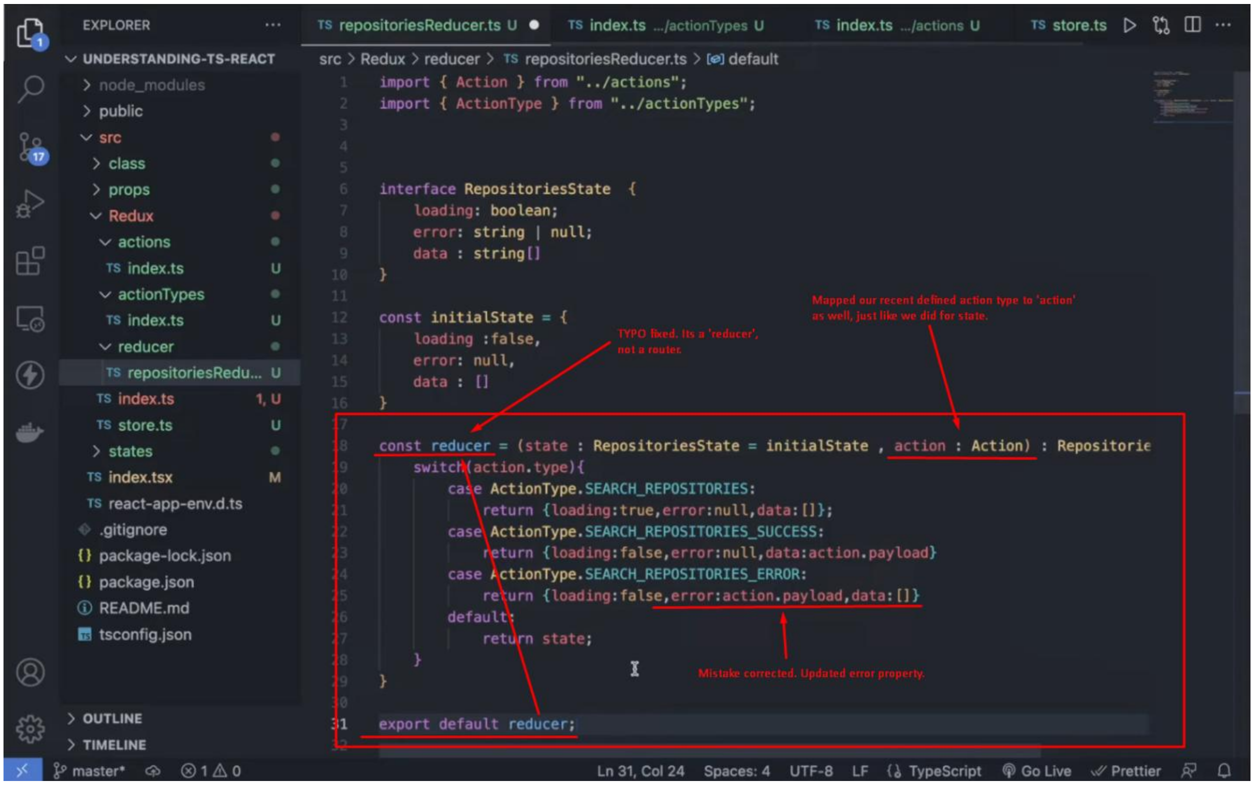
REFACTORING: Moved 'ActionType' the action types to dedicated seperate file.



```
import { ActionType } from "../actionTypes";

interface SearchRepositoriesAction {
    type: ActionType.SEARCH_REPOSITORIES
}

interface SearchRepositoriesSuccessAction{
    type:ActionType.SEARCH_REPOSITORIES_SUCCESS,
    payload: string[]
}

interface SearchRepositoriesErrorAction{
    type:ActionType.SEARCH_REPOSITORIES_ERROR,
    payload:string
}

export type Action = SearchRepositoriesAction | SearchRepositoriesSuccessAction | SearchRe
```

Defining reducer actions

It means the type could be any of above defined actions.

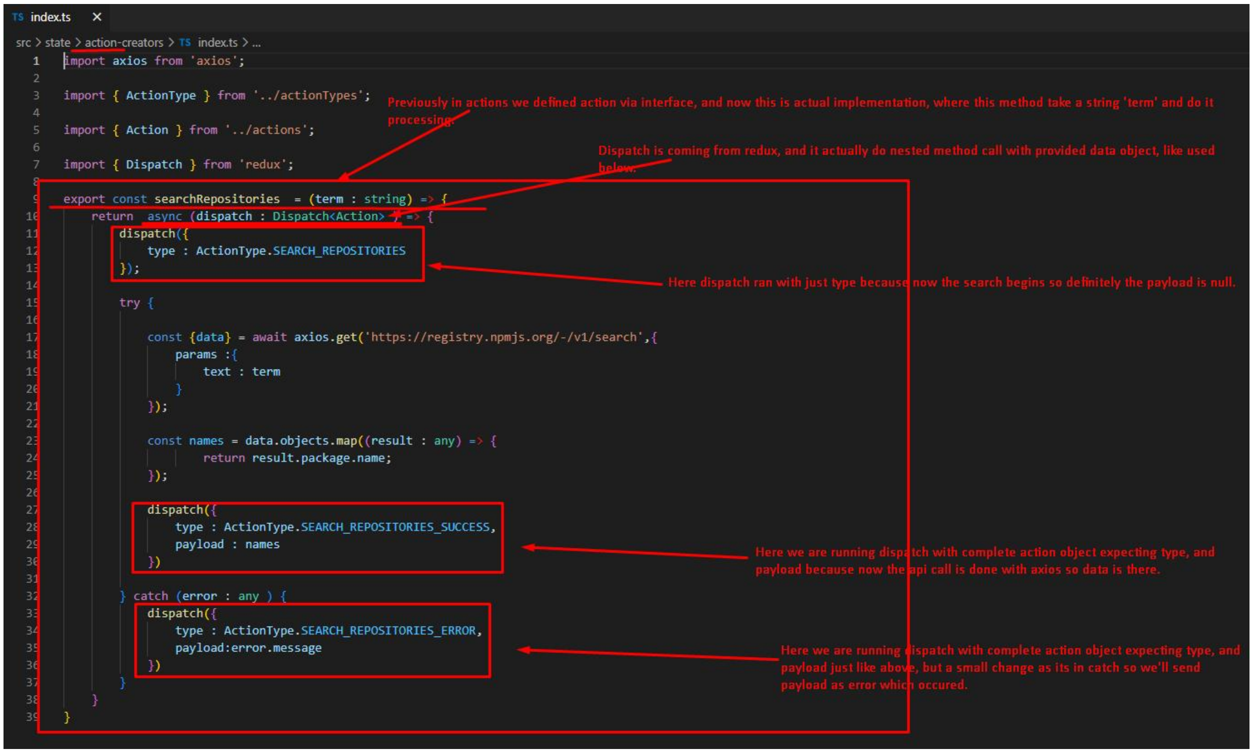


```
import { Action } from "../actions";
import { ActionType } from "../actionTypes";


interface RepositoriesState  {
    loading: boolean;
    error: string | null;
    data : string[]
}

const initialState = {
    loading :false,
    error: null,
    data : []
}

const reducer = (state : RepositoriesState = initialState , action : Action) : Repositorie
    switch(action.type){
        case ActionType.SEARCH_REPOSITORIES:
            return {loading:true,error:null,data:[]};
        case ActionType.SEARCH_REPOSITORIES_SUCCESS:
            return {loading:false,error:null,data:action.payload}
        case ActionType.SEARCH_REPOSITORIES_ERROR:
            return {loading:false,error:action.payload,data:[]}
        default:
            return state;
    }
}

export default reducer;
```

Mapped our recent defined action type to 'action' as well, just like we did for state.

TYPO fixed. Its a 'reducer', not a router.

Mistake corrected. Updated error property.



```
import axios from 'axios';

import { ActionType } from '../actionTypes';

import { Action } from '../actions';

import { Dispatch } from 'redux';

export const searchRepositories  = (term : string) => {
    return  async (dispatch : Dispatch<Action> ) => {
        dispatch({
            type : ActionType.SEARCH_REPOSITORIES
        });

        try {

            const {data} = await axios.get('https://registry.npmjs.org/-/v1/search',{
                params :{
                    text : term
                }
            });

            const names = data.objects.map((result : any) => {
                    return result.package.name;
            });

            dispatch({
                type : ActionType.SEARCH_REPOSITORIES_SUCCESS,
                payload : names
            })

        } catch (error : any ) {
            dispatch({
                type : ActionType.SEARCH_REPOSITORIES_ERROR,
                payload:error.message
            })
        }
    }
}
```

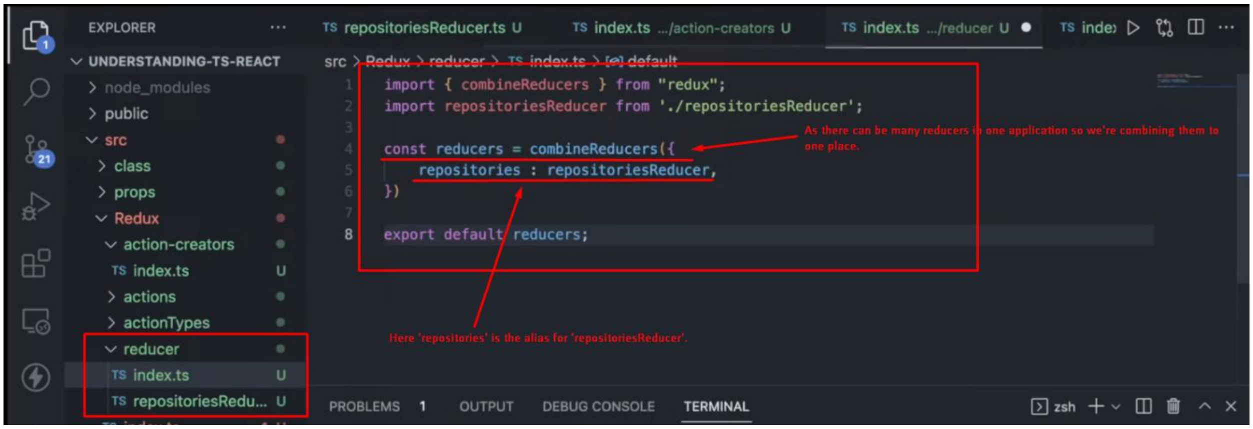
Previously in actions we defined action via interface, and now this is actual implementation, where this method take a string 'term' and do it processing.

Dispatch is coming from redux, and it actually do nested method call with provided data object, like used below.

Here dispatch ran with just type because now the search begins so definitely the payload is null.

Here we are running dispatch with complete action object expecting type, and payload because now the api call is done with axios so data is there.

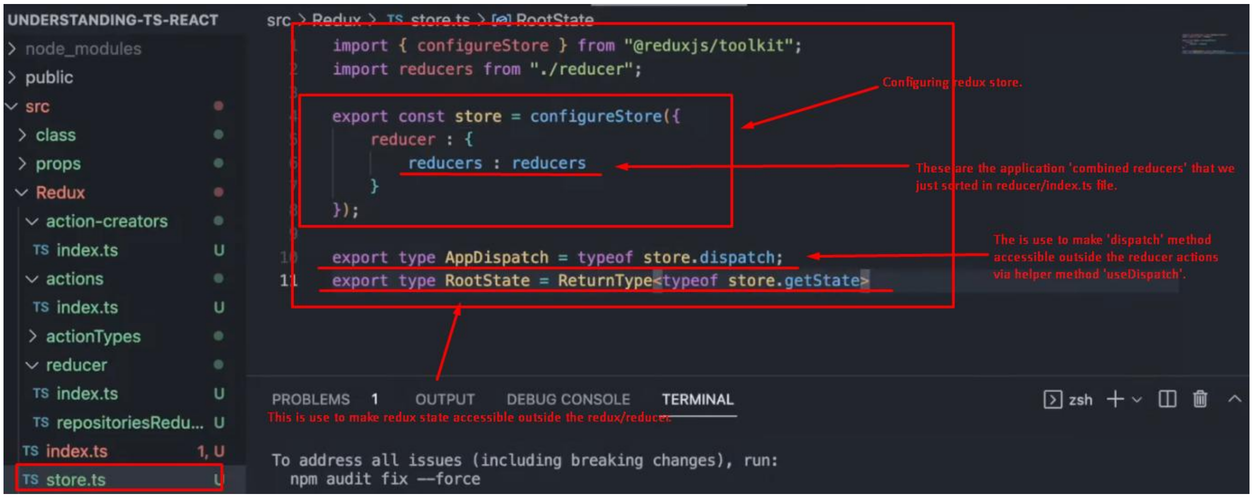
Here we are running dispatch with complete action object expecting type, and payload just like above, but a small change as its in catch so we'll send payload as error which occured.



```
import { combineReducers } from "redux";
import repositoriesReducer from './repositoriesReducer';

const reducers = combineReducers({
    repositories : repositoriesReducer,
})

export default reducers;
```

As there can be many reducers in one application so we're combining them to one place.

Here 'repositories' is the alias for 'repositoriesReducer'.



```
import { configureStore } from "@reduxjs/toolkit";
import reducers from "./reducer";

export const store = configureStore({
    reducer : {
        reducers : reducers
    }
});

export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof store.getState>
```

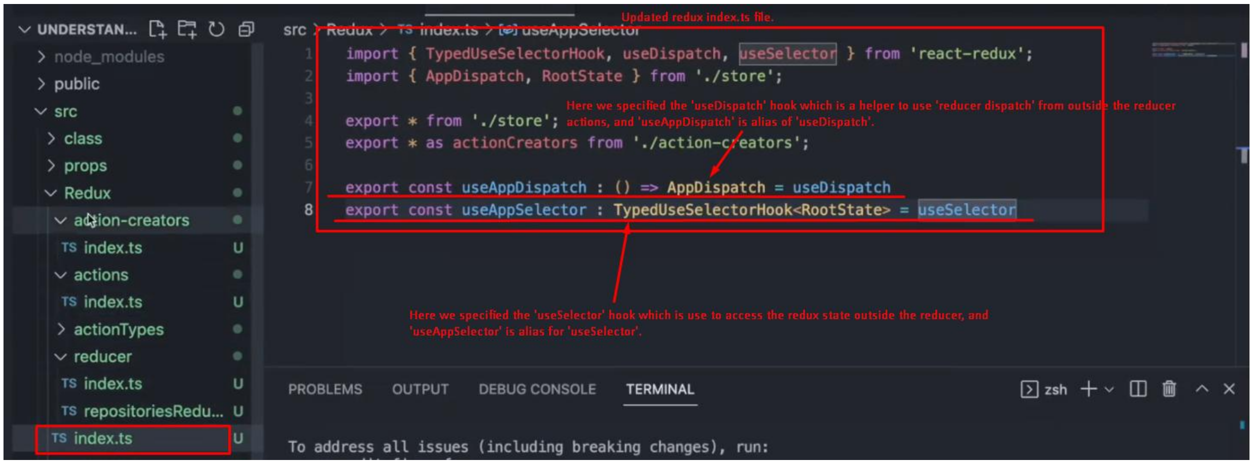
Configuring redux store.

These are the application 'combined reducers' that we just sorted in reducer/index.ts file.

The is use to make 'dispatch' method accessible outside the reducer actions via helper method 'useDispatch'.

This is use to make redux state accessible outside the redux/reducer.

```
To address all issues (including breaking changes), run:
  npm audit fix --force
```



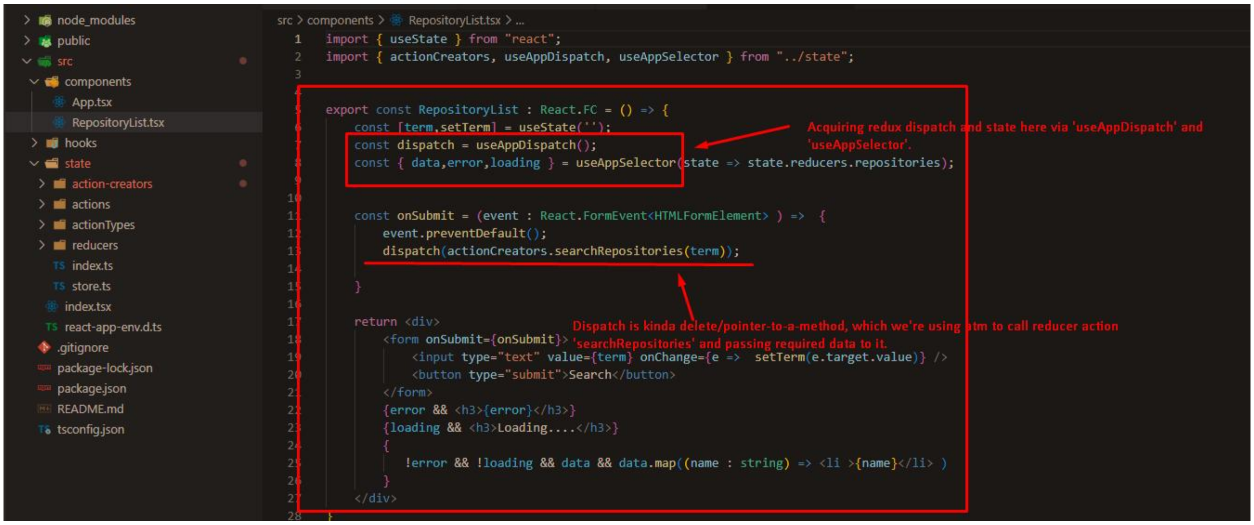
Updated redux index.ts file.

```
import { TypedUseSelectorHook, useDispatch, useSelector } from 'react-redux';
import { AppDispatch, RootState } from './store';

export * from './store';
export * as actionCreators from './action-creators';

export const useAppDispatch : () => AppDispatch = useDispatch
export const useAppSelector : TypedUseSelectorHook<RootState> = useSelector
```

Here we specified the 'useDispatch' hook which is a helper to use 'reducer dispatch' from outside the reducer actions, and 'useAppDispatch' is alias of 'useDispatch'.

Here we specified the 'useSelector' hook which is use to access the redux state outside the reducer, and 'useAppSelector' is alias for 'useSelector'.

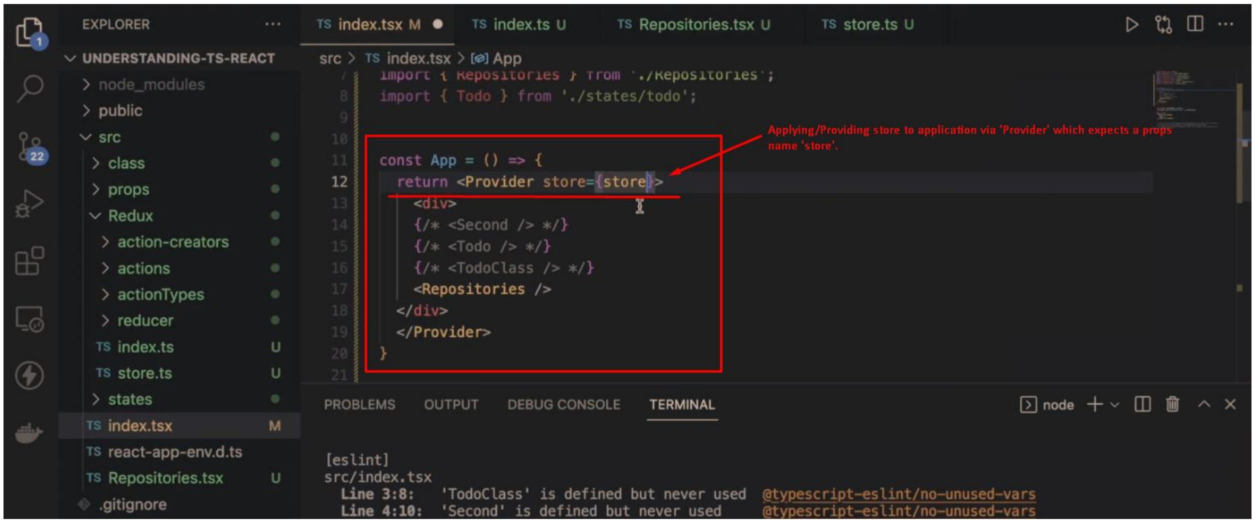
```
To address all issues (including breaking changes), run:
```



```tsx
import { useState } from "react";
import { actionCreators, useAppDispatch, useAppSelector } from "../state";

export const RepositoryList : React.FC = () => {
    const [term,setTerm] = useState('');
    const dispatch = useAppDispatch();
    const { data,error,loading } = useAppSelector(state => state.reducers.repositories);

    const onSubmit = (event : React.FormEvent<HTMLFormElement> ) =>  {
        event.preventDefault();
        dispatch(actionCreators.searchRepositories(term));

    }

    return <div>
        <form onSubmit={onSubmit}>
            <input type="text" value={term} onChange={e =>  setTerm(e.target.value)} />
            <button type="submit">Search</button>
        </form>
        {error && <h3>{error}</h3>}
        {loading && <h3>Loading....</h3>}
        {
          !error && !loading && data && data.map((name : string) => <li >{name}</li> )
        }
    </div>
}
```

Acquiring redux dispatch and state here via 'useAppDispatch' and 'useAppSelector'.

Dispatch is kinda delete/pointer-to-a-method, which we're using atm to call reducer action 'searchRepositories' and passing required data to it.



```tsx
import { Repositories } from './Repositories';
import { Todo } from './states/todo';

const App = () => {
  return <Provider store={store}>
    <div>
    {/* <Second /> */}
    {/* <Todo /> */}
    {/* <TodoClass /> */}
    <Repositories />
  </div>
  </Provider>
}
```

Applying/Providing store to application via 'Provider' which expects a props name 'store'.

```
[eslint]
src/index.tsx
  Line 3:8:   'TodoClass' is defined but never used  @typescript-eslint/no-unused-vars
  Line 4:10:  'Second' is defined but never used     @typescript-eslint/no-unused-vars
```