

React Basics

Bro Code - React Full Course for free (2024)

Installation and Set-up

Extensions

Eslint

Airbnb (or do your custom config)

Prettier

Links

Testing

Vitest

React Testing Library

How To Actually test

Option 1

Option 2

Other test function

Run your tests:

Links

React routing

React Router

React Router Install

Use React Router

Links to pages in your app (type of shit)

Test react router

Links

Basics

Components

How to create components

Styling

List

Links

Folder structure

assets

components

ui components subfolder

form components subfolder

context

data

hooks

layouts

pages

features

utils

__tests__

Links

Props:

Context

useState

ref and forwardRef

children, reactNode and how passing props actually works

exports

Installation and Set-up

(with Vite, eslint, prettier, vitest, testing-library and react-router)

- install node
- In Terminal:
npm install -g app-name → cd app-name → npm start

or w npx
- ACTUALLY do it with vite:
npm create vite@latest
and then it's pretty simple from then on;
npm install - for all dependencies
- npm run build + npm run dev

Extensions

- use react dev tools - extension (chrome)
- es7 react/redux extension for vs code

Eslint

- npm i -D eslint (install eslint as a dev dependency)
- npx eslint - --init generate eslint config

Airbnb (or do your custom config)

- MAYBE install: npx install-peerdeps --dev eslint-config-airbnb (airbnb's eslint config)
add "airbnb" and "airbnb/hooks" .eslintrc.cjs "extends" instead of "eslint:recommended"
then install eslint-config-airbnb-typescript (you can find all on their github docs)

Prettier

- npm i -D prettier eslint-config-prettier eslint-plugin-prettier
- create and config your .prettierrc file
- add to .eslintrc: to plugins 'prettier' and 'plugin:prettier/recommended' - extends

Links

<https://www.youtube.com/watch?v=cchqeWY0Nak&pp=ygUlc2V0dXAgcmlVhY3QgdYB2aXRlIGVzbGludCB0cyBwcmV0dGllcg%3D%3D>

Testing

Vitest

```
npm install -D vitest
add <reference type="vitest"/> type="vite/client"
and test:{
  globals: true,
  environment: 'jsdom',
  setupFiles:['./src/setup.ts']}
```

React Testing Library

```
npm i -D @testing-library/react @testing-library/jest-dom
```

Setup with Vitest:

```
add import matchers from '@testing-library/jest-dom/matchers'
import {expect} from 'vitest';
expect.extend(matchers); - to your setupTests.ts file
```

create your tests folder and create files:

App.test.tsx

Navbar.test.tsx etc. - for each thing you want tested

App.test.tsx:

```
import '@testing-library/jest-dom'
import {fireEvent, render, screen} from '@testing-library/react'
import {expect, test, vi, describe, it} from 'vitest'
// (there are more ways to actually test)
describe('App', () => {
  it('test smth', () => {
    //actual test:
    //ARRANGE
    //RENDER
    //ACT
  });
});
```

OR

```
test('your-test', () => {
  //actual test
});
```

create a script "test": "vitest" to run all tests

How To Actually test

you need to give your components and data-testid so you can access them and see if they render and show on the screen and so on

Option 1

```
test('test button render without extra class', () => {
  render(<Button type='button' buttonText='testing button'/></Button>);
```

```
const renderedButton = screen.getByTestId('button-test-id');
expect(renderedButton).toBeInTheDocument;
```

```
import {describe, expect, it, test} from 'vitest';

import '@testing-library/jest-dom';
import {render, screen} from '@testing-library/react';
import Button from '../Button/Button';

// 1st type of testing
test('test button rendering without extra class', () => {
  render(<Button type='button' buttonText='Remove player'></Button>);

  const renderedButton = screen.getByTestId('button-test-id');
  expect(renderedButton).toBeInTheDocument();
});

test('test button rendering with extra class', () => {
  render(
    <Button
      type='button'
      buttonText='Remove player'
      className='button-light remove-button'
    ></Button>,
  );

  const renderedButton = screen.getByTestId('button-test-id');
  expect(renderedButton).toBeInTheDocument();
});
```

Option 2

```
describe('Button component', () => {
  it('Render without extra class', () => {
    render(<Button type='button' buttonText='Remove player'></Button>);

    const renderedButton = screen.getByTestId('button-test-id');
    expect(renderedButton).toBeInTheDocument();
  });
  it('Render with extra class', () => {
    render(
      <Button
        type='button'
        buttonText='Remove player'
        className='button-light remove-button'
      ></Button>,
    );
  });
});
```

```

    const renderedButton = screen.getByTestId('button-test-id');
    expect(renderedButton).toBeInTheDocument();
  });
});

```

basically the same, but a different syntax;

he imports are quite the same too

Other test function

`expect(screen.getByRole('heading',{level:1}))` - to check if there is an h1 there

- `.toHaveTextContent('Caca maca eeee');`
- `.toHaveAttribute('attribute', 'attributeValue')`
- `.toHaveClass('className');`
- IF YOU WANT TO TEST THINGS THAT HAVE ROUTING(`<Link to....>`)
you need to import `BrowserRouter` or whatever you used and render them with the router:

```
<BrowserRouter> <Navbar></Navbar></BrowserRouter>
```

- if you want to test smth that has a `useNavigate`:

```
const {mockedUseNavigate} = vi.hoisted (() => {
  return {
    mockedUseNavigate: vi.fn(),
  };
});
```

```
vi.mock('react-router-dom', async() => {
  const router = await vi.importActual<typeof import('react-router-dom')>('react-router-dom');
  return {
    ...router,
    useNavigate: () => mockedUseNavigate,
  };
});
```

and then in your test:

```
const mockRemoveMethod = vi.fn(); - for example
```

- To test if a button has been pressed, you check if the function `onClick` to it has been called and the `renderedRemoveButton` has that `mockRemovePlayer` function as on click

```

renderedRemoveButton.click();

expect(mockRemovePlayer).toHaveBeenCalled();

```

- `expect(renderedCharacterList.childNodes.length).toBe(1);`
you check how many `childNodes` a rendered element has

Run your tests:

create a script in your package.json called "test" that does "vitest" and then use the command:
npm run test

Links

<https://testing-library.com/docs/react-testing-library/intro/#>

React routing

You need to use React Router to do it

React Router

React Router Install

from their website

Use React Router

create a folder with your pages (top lvl components)

import a {YourRouter} from 'react-router-dom'
and wrap everything in your

```
<YourRouter>
  <Routes>
    <Route path="/" /> // average path
    <Route />...
  </Routes>
</YourRouter>
```

basically each route is one of your pages;

you will import whatever you need and in your App.tsx you need to set the 'paths'

example:

```
<BrowserRouter>
<Routes>
<Route path="/" element={<SomePage/>}> </Route> (idk the closing tags by heart)
<Route path='/chewbacca' element={<ChewbaccaPage/>}> </Route>
...
</Routes>
</BrowserRouter>
```

this basically means that when you access

<yourwebsite.com/chewbacca> it will open the Chewbacca Page;

to navigate between pages you use the: `useNavigate()`; function from 'react-router-dom':

example: `const navigate = useNavigate();`

`const handleClick = () =>{`

`navigate('/chewbacca');`} this will basically go to your chewbacca page

if the handleClick event is triggered

you give it an element = `{<OneOfYourPages/>}`

Your app should forward you to your HomePage with a Router

path="*" → if none of the paths are found, you should have a default not found page

Links to pages in your app (type of shit)

to have links from other page:

```
<Link to="/">Home</Link>
```

```
<Link to="">Page1</Link>
```

Test react router

- you CAN test it with <MemoryRouter> - if you give it a bad path, you have to check if it goes to not found

example

```
<MemoryRouter initialEntries={['/ -your pth or "/" for home]}'>
```

```
<App />
```

```
</MemoryRouter>
```

```
expect(
```

```
screen.getByRole('heading', { //for h1
```

```
level:1,
```

```
})
```

```
).toHaveTextContent('Not Found');
```

```
});
```

```
});
```

Links

<https://reactrouter.com/en/main>

Basics

- UI - made of separate, reusable components
- components can be created w function w Hooks or w classes
- you write JSX - Js Syntax Extension - similar to html
- components have a "state" - determines how a component behaves
- hooks
- jsx/tsx must have only one parent elemt
- in your function App() (App.tsx you will return one div that contains others)
- you can add expressions and variables:

```
<h2> Hello {var}, 1+1 is {1+1}
```

```
let x = true
```

```
<h2>{x ? 'Yes' : 'No'}</h2>
```
- assets folder - for images and videos
- main.tsx - your main.js file basically, which usually only contains App.jsx
- App.css - css for your app component
- index.css - css for your whole html

- package.json -

Components

How to create components

- create a new folder under src for components
then create a subfolder for each component you want to create
create a CompName.tsx/jsx and CompName.css - inside the folder
(tsx - types, jsx-js)
- create a new const with the name of your components and work with it as you do in the App.tsx; import it in App.tsx and then use it: <Component/>
- you can pass elements in your component and handle them:
EXAMPLE:
App.tsx:

```
return( <Header title="Astral Odyssey"/>
```


Header.tsx

```
const Header = (props) => { || const Header = ({title}:{title:string}) - if you do this, you use title directly  
return(  
  <header>  
    <h1>props.title</h1>  
  </header>  
)  
}  
Header.defaultProps = {  
  title: 'defaultTitle',  
}  
export default Header
```
- you can specify the type of what you want to transmit:

```
import PropTypes from 'prop-types'  
Header.propTypes = {  
  title: PropTypes.string  
} => you can't transmit anything else but strings  
// didn't work for me ( import PropTypes from prop-types  
what worked:  
const Header ({title}:{title:string}){...}  
working with props as a whole container of parameters also didnt work  
defaultProps works tho
```

Styling

- you can use a stylesheet - header.css or whatever for your Header component and so on
- you can use style inside of your react (inline):

```
<h1 style={{color:'red', backgroundColor:'blue'}}>My Title</h1>
```


OR


```
not inline:
<h1 style={headingStyle}>...</h1>
const headingStyle = {
  color:'red',
  backgroundColor:'blue'
}
```

List

```
{yourArray.map( (item) => (
<html-here>{item.name}</html-here>
))}
```

Links

<https://www.youtube.com/watch?v=CgkZ7MvWUAA&pp=ygUOYnJvIGNvZGUgcmVhY3Q%3D>

<https://www.youtube.com/watch?v=SqcY0GIETPk&pp=ygUOcmVhY3QgdHV0b3JpYWw%3D>

<https://www.youtube.com/watch?v=w7ejDZ8SWv8&t=2436s&pp=ygUdcMvhY3QgdHV0b3JpYWwgdHJhdmVyc3kgbWVkaWE%3D>

Folder structure

You should have the following folders in your src folder

assets

here's where you store you assets

components

ui components subfolder

Have a folder for your ui components

form components subfolder

Have a folder for form components

context

separate folder for your contexts

data

here you store your data - like configuration.json, constant.ts/js and so on

hooks

Have a folder for your hooks

layouts

for header, footer and so on - layout elements that you use frequently
have a mainLayout for example that has a div w header, footer, navbar and etc

pages

have a folder for each page in your screen

features

have a folder for your 'features' - like crud operations and so on

utils

have a folder for utilities

__tests__

you have a __tests__ folder in each folder that you're testing
so they are close to one another

Links

<https://www.youtube.com/watch?v=UUga4-z7b6s&pp=ygUXcmVhY3Qgc3RydWN0dXJlIHByb2plY3Q%3D>

<https://www.youtube.com/watch?v=ANrYhHN8DI4&pp=ygUXcmVhY3Qgc3RydWN0dXJlIHByb2plY3Q%3D>

Props:

props are basically a set of parameters that you can pass from object to object
or from component to component

you define them in the following way:

```
export type MyProps = {  
  name: string,  
  thisIsOptional? : number,  
  thisCanHaveAnyType: any,  
} and so on. you then import them in your components
```

once you are in components, you use them like this:

```
export function myComponent = ({props}:{props:MyProps}) ⇒ {  
  <div name={props.name}>...  
  and something along the way
```

Context

<context.Provider value=...> - provider is for passing down props

context is basically a global variable props type of shit

if you have
component 1 >> component 2 >> ... >> component n
and you want to pass some props all the way to comp n, but you access comp 1 then 2 and so on
then you could do it with a context:
this basically lets you do it directly
or that's how i understand it for now

useState

- everytime you render a component it declares all the variables
 - with useState, you basically tell it that if it's already declared, it doesn't redeclare it but it takes the last value
 - it's a persistence before refreshing the page
-

ref and forwardRef

todo

children, reactNode and how passing props actually works

todo

exports

export - a function / comp can be exported more than once
but only once as a default

Material UI

this is a library for react components; it's better to use than normal html tags + css
