**Namaste React**

**Ep04: Building a Food Ordering App**

\*JS JSX mandatory? => NO

\*JS Typescript mandatory? => NO

\*JS ES6 mandatory? => No

3 ways of component composition

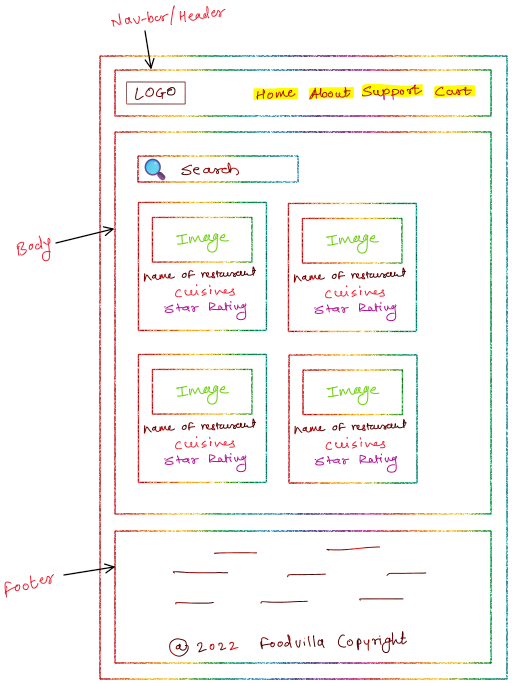
1. {Title()}
2. <Title/> -> used generally
3. <Title></Title>

**Building our App**

Name: FoodVilla

Whenever we are writing code, do planning.

after planning our App will something look like this,



So the app layout should have following components.

* Header
* Body
* Footer

const AppLayout = () =>{

return(

**-Header**

-Logo

-NavItems(on right side)

-Cart

**-Body**

-SearchBar

-ReastaurantList

-RestaurantCard

-Image

-Named

-Rating

-Cusines

**-Footer**

-Links

-Copyrights

)

}

**Header Component**

For building the header component we will write the component as below,

const **Title** = () => {

return (

<a href="/">

<img

className="title"

alt="logo"

src="https://...">

</img>

</a>

);

};

const **HeaderComponent** = () => {

return (

<div className="header">

<Title />

<div className="nav-items">

<ul>

<li>Home</li>

<li>About</li>

<li>Contact</li>

<li>Cart</li>

</ul>

</div>

</div>

);

};

Any piece of JSX expressions inside the component should have only one parent element. That’s why we need to wrap the entire inside of <div> tag or something known as <React.Fragment>.

What is React.Fragment

-it a component which is exported by React, it is like and empty tag.

-whenever we wrap the JSX inside of <div> tag it will add an extra node to the DOM. But by using the React.Fragment we can avoid this extra node.

e.g.

const AppLayout=()=>{

return(

**<React.Fragment>**

<Header/>

<Body/>

<Footer/>

**</React.Fragment>**

)

}

Shorthand syntax is like => <></>, so we can use this syntax instead of <React.Fragment><React.Fragment/>.

**Styling in React**

There are several ways of styling in React, some of them are as follow.

1. Using JS objects for styling the components.

const styleObj = {

color: "red",

};

**styleObj** is normal JS object

const someComponent = () => {

<div style={styleObj}>

<h1>Helllo World</h1>

</div>;

};

1. Using external CSS file

const someComponent = () => {

<div className=”hello”>

<h1>Helllo World</h1>

</div>;

};

**CSS file**

.hello{

color: "red"

}

1. Using the external library like **Tailwind CSS, Bootstrap, MaterialUI** etc.

Can I use a **React.Fragment**  inside another **React.Fragment**?

=>***Yes****, we can nest* ***React.Fragment*** *component inside another* ***React.Fragment*** *component.*

**Body Component**

While building Restaurant Card we need some data for this card and there are two ways to get,

1. Using hard coded data
2. Integrate with API

**Config Driven UI**

All the UI is driven by configuration sent from backend, for example- depending on the location we gets the recommendation and coupon codes on food delivering apps, which are only specific to that particular area.

**Optional chaining (?.)**

* The optional chaining (?.) operator accesses an object's property or calls a function. If the object accessed or function called is undefined or null, it returns undefined instead of throwing an error.
* The ?. operator is like the . chaining operator, except that instead of causing an error if a reference is nullish (null or undefined), the expression short-circuits with a return value of undefined. When used with function calls, it returns undefined if the given function does not exist.
* This results in shorter and simpler expressions when accessing chained properties when the possibility exists that a reference may be missing. It can also be helpful while exploring the content of an object when there's no known guarantee as to which properties are required.

**Optional chaining with function calls**

You can use optional chaining when attempting to call a method which may not exist. This can be helpful, for example, when using an API in which a method might be unavailable, either due to the age of the implementation or because of a feature which isn't available on the user's device.

Using optional chaining with function calls causes the expression to automatically return undefined instead of throwing an exception if the method isn't found:

const result = someInterface.customMethod?.();

Copy to Clipboard

However, if there is a property with such a name which is not a function, using ?. will still raise a TypeError exception "someInterface.customMethod is not a function".

Note: If someInterface itself is null or undefined, a TypeError exception will still be raised ("someInterface is null"). If you expect that someInterface itself may be null or undefined, you have to use ?. at this position as well: someInterface?.customMethod?.().

eval?.() is the shortest way to enter indirect eval mode.

**Optional chaining with expressions**

You can also use the optional chaining operator with bracket notation, which allows passing an expression as the property name:

const nestedProp = obj?.["prop" + "Name"];

This is particularly useful for arrays, since array indices must be accessed with brackets.

function printMagicIndex(arr) {

console.log(arr?.[42]);

}

printMagicIndex([0, 1, 2, 3, 4, 5]); // undefined

printMagicIndex(); // undefined; if not using ?., this would throw

**Optional chaining not valid on the left-hand side of an assignment**

It is invalid to try to assign to the result of an optional chaining expression:

const object = {};

object?.property = 1; // SyntaxError: Invalid left-hand side in assignment

**Short-circuiting**

When using optional chaining with expressions, if the left operand is null or undefined, the expression will not be evaluated. For instance:

const potentiallyNullObj = null;

let x = 0;

const prop = potentiallyNullObj?.[x++];

console.log(x); // 0 as x was not incremented

Subsequent property accesses will not be evaluated either.

const potentiallyNullObj = null;

const prop = potentiallyNullObj?.a.b;

// This does not throw, because evaluation has already stopped at

// the first optional chain

This is equivalent to:

const potentiallyNullObj = null;

const prop =

potentiallyNullObj === null || potentiallyNullObj === undefined

? undefined

: potentiallyNullObj.a.b;

However, this short-circuiting behavior only happens along one continuous "chain" of property accesses. If you group one part of the chain, then subsequent property accesses will still be evaluated.

const potentiallyNullObj = null;

const prop = (potentiallyNullObj?.a).b;

// TypeError: Cannot read properties of undefined (reading 'b')

This is equivalent to:

const potentiallyNullObj = null;

const temp = potentiallyNullObj?.a;

const prop = temp.b;

Except the temp variable isn't created.

**Props**

-Shorthand for Properties

-passed props means I’m passing some data as properties into my Functional/Class component