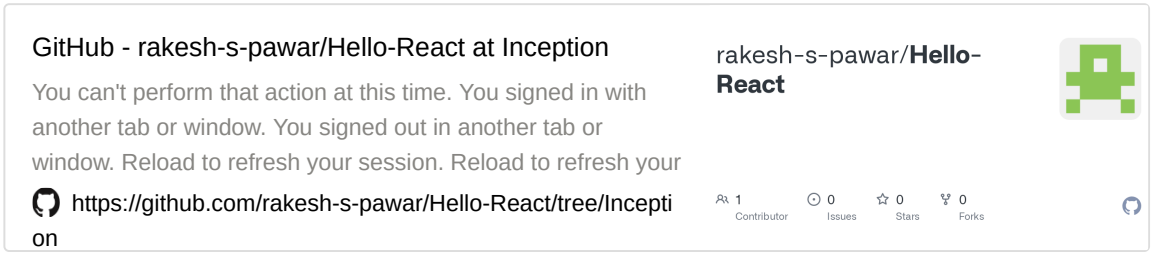
Hello React Course

▼ Inception



▼ Igniting our App

▼ Bundlers

There are lots of bundlers like Vite, Parcel, Webpack, etc. We are using Parcel in our project.

> 💡 In the original “Create-React-App” Webpack bundler was used.

Parcel is a bundler, it is a package. To use such package in our React app, we need package manager. We will be using NPM.

We need NPM, because we need to use lots of packages in our projects. To manage such packages we use NPM.

We need to minify our app, we need to bundle things, we need to remove console logs, we need to optimize our app, for these kind of tasks we need lots of helper packages. Those helper packages are managed using NPM.

```
// start with npm init in Terminal
npm init

// it will ask lots of questions. Click Enter if don't want to change
package name: (hello-react)
version: (1.0.0)
description: This is a Namaste React Tutorial course.
entry point: (App.js)
test command: jest
git repository: (https://github.com/rakesh-s-pawar/Hello-React.git)
keywords:
author: Rakesh Pawar
```

```
license: (ISC)
Is this OK? (yes)
```

💡 After this process, we will get package.json in our project. This file contains configuration which NPM needs to run.

To get “Parcel” in our App, we need to run some commands

```
// we can use
npm install parcel

// but we don't want Parcel on production, we want it on our local machine
// we use "-D", which means dev dependancies
npm install -D parcel

// Some people use "--save-dev", it is same as using "-D"
```

💡 Dependency means, all the packages that a project needs. Parcel is one of such dependency.

After running “npm install -D parcel”, “package-lock.json” file & “node_modules” folder are added in our project, and “Parcel” is added as devDependencies in package.json file as shown below.

```
"devDependencies": {
    "parcel": "^2.8.2"
  }
```

💡 **Difference Between Caret (^) & Tilde (~)**

NPM versions are written using three dots separated numbers the first number from the left shows the major release and the second number from the left shows the minor release and the third number shows the patch release of the package.

Syntax: The syntax of the npm version looks like the following.

```
Major.Minor.Patch
```

Tilde(~) notation	Caret(^) notation
Used for Approximately equivalent to version.	Used for Compatible with version.
It will update you to all future patch versions, without incrementing the minor version. ~1.2.3 will use releases from 1.2.3 to <1.3.	It will update you to all future minor/patch versions, without incrementing the major version. ^2.3.4 will use releases from 2.3.4 to <3.0.0
It gives you bug fix releases.	It gives you backwards-compatible new functionality as well.
It will update in decimals.	It will update to its latest version in numbers.
Not a default notation used by NPM.	Used by NPM as default notation.
Example: ~1.0.2	Example: ^1.0.2

💡 “package-lock.json” is very important file. It locks the version. We never have to add it in “gitignore”.

💡 “node_modules” folder is kind of database to npm. Don’t add “node_modules” folder on github. Add it in “gitignore” file. It is very heavy in size & our “package-lock.json” has sufficient information, hence we never add “node_modules” on git. “package-lock.json” helps us to generate “node_modules” folder on server.

💡 We are removing React CDN links from our project, because it is not the best way to use react. We will install React using npm, as shown below.

```
// we want React on local & server as well, hence we are not using "-D"
npm install react
```

After running “npm install react”, “React” is added as Dependency in package.json file as shown below.

```
// we havn't used "-D", hence it added as "dependencies" & not "devDependencies"
"dependencies": {
    "react": "^18.2.0"
  }
```

Now, we are installing “React-DOM” in our project

```
npm install react-dom

// we can also use "npm i react-dom" instead of "npm install react-dom"
// "npm i" is shortcut to "npm install"
```

Now, we will ignite our app, we will give entry point as index.html

```
// npx means execute using npm
npx parcel index.html


// After this import react & react-dom in App.js file
import React from "react";
import ReactDOM from "react-dom/client";
```

💡 when we run “npx parcel index.html”, “dist” creates faster development build of our project and serves it on the server.

Now, we need to edit <script> tag in index.html

```
// Change from
<script src="App.js"></script>
```

```
// Change to
<script type="module" src="App.js"></script>
```

💡 When we make any changes in any files, we don't need to refresh our page in browser, it will automatically show changes without any need to refresh page. This concept is called as "**Hot Module Replacement (HMR)**" and it is part of "Parcel". Parcel do this using "File Watcher Algorithm" which keeps track of changes in files.

```
// we are giving entry point as "indrx.html" in "npx parcel index.html", hence
// Remove below line from "package.json"

"main": "App.js",

// we can make production build by using below command
npx parcel build index.html
```

💡 "**Browserlist**" is used to make our app compatible with browsers. We can add it in "package.json" file, as shown below:

```
// Example
"browserslist": [
    "last 2 Chrome versions"
  ]
```

▼ Chapter 3: Laying the Foundation

💡 **Babel is a JavaScript compiler**

Babel comes along with Parcel. When we install Parcel, Babel comes as a dependency.

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Here are the main things Babel can do for you:

- Transform syntax
- Polyfill features that are missing in your target environment (through a third-party polyfill such as core-js)
- Source code transformations (codemods)
- And more!

Babel uses “Browserslist”, which keeps track of versions of browsers our code should work. It checks browser versions mentioned in the Browserslist, and convert newer version javascript code to older version of javascript, to make our code compatible with older versions. This concept is called “**Polyfill**”

```
// Till now we are using

npx parcel index.html

// to run our app, instead we can add code in "scripts" inside package.json

"start": "parcel index.html",

// now we can run our app using
npm run start

// There is shortcut to "npm run start", we can skip run as shown below
npm start

// instead of npx parcel index.html

// We can do similar thing for build command, instead of using

npx parcel build index.html

// we can add below code in "scripts" inside package.json
"build": "parcel build index.html",
```

```
// now we can create production build of our app using
npm run build
```

Parcel, Babel does not remove console.log from our projects, there is 1 plugin known as “**babel-plugin-transform-remove-console”**. We can install it in our app using below command:

`npm install babel-plugin-transform-remove-console --save-dev`

To use “**babel-plugin-transform-remove-console”** plugin, we need to configure it in our app, using below mentioned steps

- create `.babelrc` file in our Project
- Add below code inside .babelrc file

```
// with options
{
  "plugins": [ ["transform-remove-console", { "exclude": [ "error", "warn"] }] ]
}
```

```
// Now, console.log is showing below error message
Warning: Each child in a list should have a unique "key" prop.

// Because in our, App.js file, container div has 2 children (heading & heading2)
// we need to give key (highlighted green) to each children to solve this issue

// Whenever there are multiple siblings inside a parent, we give them keys.
// Keys are anything which uniquely identifies them. We should always attach keys
// to the siblings

const heading = React.createElement(
  "h1",
  {
    id: "title",
    key: "h1",
  },
  "Heading 1"
);

const heading2 = React.createElement(
  "h1",
  {
    id: "title",
    key: "h2",
  },
  "Heading 2"
);

const container = React.createElement(
```

```
  "div",
  {
    id: "container",
  },
  [heading, heading2]
);
```

💡 JSX is HTML-like syntax. As coding using “React.createElement” makes our code lengthy, difficut to manage, Facebook developers introduced JSX, using which we can write our code using HTML-like syntax. Babel converts HTML-like JSX code into React.createElement.

```
// This is known as a React Element
const heading2 = (
  <h1 id="title" key="h2">
    Namaste React
  </h1>
);
```

💡 **What is a React Component?**
Components are **independent and reusable bits of code**. They serve the same purpose as JavaScript functions, but work in isolation and return HTML. Components come in two types, Class components and Function components.

- Class component is a old way
- Functional Component is the new way. Functional component is nothing but a javascript function.
- Component name start with a capital letter. (Not mandatory, but good practice)

```
// Functional Component Example
const HeaderComponent = () => {
  return (
    <div>
      <h1>This is functional component</h1>
      <h2>This is a h2 tag</h2>
    </div>
  );
};
```

```
// Above code and this code is same
// We can omit writing "return" like this
const HeaderComponent2 = () => (
    <div>
      <h1>This is functional component</h1>
      <h2>This is a h2 tag</h2>
    </div>
);
```

```
How to write React Element & React Component, when we want to render it

// This is React Element
const heading = (
  <h1 id="title" key="h2">
    Namaste React
  </h1>
);

// This is React Component
const HeaderComponent = () => {
  return (
    <div>
      <h1>This is functional component</h1>
      <h2>This is a h2 tag</h2>
    </div>
  );
};


// We can call React Element like this
root.render(heading);

// We can call React Component like this
root.render(<HeaderComponent />);



// We can use React Element inside React Component, using {}
const HeaderComponent = () => {
  return (
    <div>
      {heading}
      <h1>This is functional component</h1>
      <h2>This is a h2 tag</h2>
    </div>
  );
};


// We can also use React Component inside another React Component,
// Component inside component is known as "Component Composition"

const Title = () => (
  <h1 id="title" key="h2">
    Namaste React
  </h1>
);
```

```
const HeaderComponent = () => {
  return (
    <div>
      <Title />      // another way we can use like {Title()}

      <h1>This is functional component</h1>
      <h2>This is a h2 tag</h2>
    </div>
  );
};
```

💡 We can write any Javascript code inside { }

▼ Chapter 4: Show me the Code

💡 “React Fragment” is a component exported by React. JSX can only have one parent, hence we need **React Fragment**. We can use "React Fragment in Two Ways, using "<React.Fragment>" or by "<>".

```
// "React Fragment"

// JSX should have only one parent
// we can't have <Header/>, <Body/> & <Footer /> at same level without parent
// Hence we can add one parent div or can use "React Fragment"

const AppLayout = () => {
  return (
      <Header />
      <Body />
      <Footer />
  );
};

// We can use "React Fragment in Two Ways
// using "<React.Fragment>" or by "<>"
const AppLayout = () => {
  return (
    // <React.Fragment>   // OPTION 1
    <>                    // OPTION 2 (recommended)
      <Header />
      <Body />
      <Footer />
```

```
    </>
    // </React.Fragment>
  );
};
```

```
// Inline Style in React

const styleObj = {
  backgroundColor: "red",
};

const jsx = (
//  <div style={{backgroundColor: "red"}}>          // OPTION 1, direct inline
    <div style={styleObj}>                          // OPTION 2, with object
      <h1>JSX</h1>
      <h1>Second JSX</h1>
    </div>
);
```

"**Props" in React**: Props are arguments passed into React components.

`props` stands for properties. React Props are like function arguments in JavaScript *and* attributes in HTML.

```
// props in below code is user-defined word, we can name it anything we want
// It is same as receiving parameters in function in Javascript

const RestaurantCard = (props) => {
  const imgURL = "https://res.cloudinary.com/swiggy/image/upload/";
  return (
    <div className="card">
      <img src={imgURL + props.restaurant.data?.cloudinaryImageId}  />
      <h2>{props.restaurant.data?.name}</h2>
      <h3>{props.restaurant.data?.cuisines.join(", ")}</h3>
      <h4>{props.restaurant.data?.lastMileTravelString} minutes</h4>
    </div>
  );
};


const Body = () => {
  return (
    <div className="restaurant-list">

      // restaurant used below is user-defined word, we can name it anything we want
      // we are passing "restaurant" props to "RestaurantCard" component
      // it is same as passing arguments to function in Javascript
      // <RestaurantCard(restaurantList[0]) }/>

      <RestaurantCard restaurant={restaurantList[0]} />
```

```
      <RestaurantCard restaurant={restaurantList[1]} />
      <RestaurantCard restaurant={restaurantList[2]} />
      <RestaurantCard restaurant={restaurantList[3]} />
      <RestaurantCard restaurant={restaurantList[4]} />
      <RestaurantCard restaurant={restaurantList[5]} />
    </div>
  );
};
```

```
// We can write "RestaurantCard" in better way
// instead of receiving arguments in "props" parameter,
// we can use object destructuring, as ({restaurant})
// Now, we can avoid writing "props" each time to use received data

const RestaurantCard = ({restaurant}) => {
  const imgURL = "https://res.cloudinary.com/swiggy/image/upload/";
  return (
    <div className="card">
      <img src={imgURL + restaurant.data?.cloudinaryImageId}  />
      <h2>{restaurant.data?.name}</h2>
      <h3>{restaurant.data?.cuisines.join(", ")}</h3>
      <h4>{restaurant.data?.lastMileTravelString} minutes</h4>
    </div>
  );
};
```

```
// We can further destructure ({restaurant}) to make our code look clean

const RestaurantCard = ({restaurant}) => {
  const {name, cuisines, cloudinaryImageId, lastMileTravelString} = restaurant.data;
  const imgURL = "https://res.cloudinary.com/swiggy/image/upload/";
  return (
    <div className="card">
      <img src={imgURL + cloudinaryImageId}  />
      <h2>{name}</h2>
      <h3>{cuisines.join(", ")}</h3>
      <h4>{lastMileTravelString} minutes</h4>
    </div>
  );
};


// now, we can further modify and remove "Restaurant" from "RestaurantCard"

const RestaurantCard = ({name, cuisines, cloudinaryImageId, lastMileTravelString}) => {
  const imgURL = "https://res.cloudinary.com/swiggy/image/upload/";
  return (
    <div className="card">
      <img src={imgURL + cloudinaryImageId}  />
      <h2>{name}</h2>
      <h3>{cuisines.join(", ")}</h3>
      <h4>{lastMileTravelString} minutes</h4>
    </div>
  );
};
```

```
// Earlier

const Body = () => {
  return (
    <div className="restaurant-list">

      // restaurant used below is user-defined word, we can name it anything we want
      // we are passing "restaurant" props to "RestaurantCard" component
      // it is same as passing arguments to function in Javascript
      // <RestaurantCard(restaurantList[0]) }/>

      <RestaurantCard restaurant={restaurantList[0]} />
      <RestaurantCard restaurant={restaurantList[1]} />
      <RestaurantCard restaurant={restaurantList[2]} />
      <RestaurantCard restaurant={restaurantList[3]} />
      <RestaurantCard restaurant={restaurantList[4]} />
      <RestaurantCard restaurant={restaurantList[5]} />
    </div>
  );
};


// We can use "Spread Operator" to make our "<RestaurantCard/>" look clean

const Body = () => {
  return (
    <div className="restaurant-list">

      // restaurant used below is user-defined word, we can name it anything we want
      // we are passing "restaurant" props to "RestaurantCard" component
      // it is same as passing arguments to function in Javascript
      // <RestaurantCard(restaurantList[0]) }/>

      <RestaurantCard {...restaurantList[0].data} />
      <RestaurantCard {...restaurantList[1].data} />
      <RestaurantCard {...restaurantList[2].data} />
      <RestaurantCard {...restaurantList[3].data} />
      <RestaurantCard {...restaurantList[4].data} />
      <RestaurantCard {...restaurantList[5].data} />
    </div>
  );
};

//If there 50 restaurants, then it's difficult to mention each restaurant like this
// To get details of all Restaurants, we need to loop our data
// We use "Map", instead of for loop or for each loop as shown below

const Body = () => {
  return (
    <div className="restaurant-list">
      {restaurantList.map((restaurant) => {
        return <RestaurantCard {...restaurant.data} />;
      })}
    </div>
  );
};
```

```
// We need to give unique Key, without which we will get warning in console
// Warning: Each child in a list should have a unique "key" prop.

const Body = () => {
  return (
    <div className="restaurant-list">
      {restaurantList.map((restaurant) => {
        return <RestaurantCard {...restaurant.data} key={restaurant.data.id} />;
      })}
    </div>
  );
};
```

Q: What is `Reconciliation` in React?

A: `Reconciliation` is the process through which React updates the Browser DOM and makes React work faster. React use a `diffing algorithm` so that component updates are predictable and faster. React would first calculate the difference between the real DOM and the copy of DOM (Virtual DOM) when there's an update of components. React stores a copy of Browser DOM which is called `Virtual DOM`. When we make changes or add data, React creates a new Virtual DOM and compares it with the previous one. Comparison is done by `Diffing Algorithm`. React compares the Virtual DOM with Real DOM. It finds out the changed nodes and updates only the changed nodes in Real DOM leaving the rest nodes as it is. This process is called Reconciliation.

Q: Difference between `Virtual DOM` and `Real DOM`?

A: DOM stands for `Document Object Model`, which represents your application UI and whenever the changes are made in the application, this DOM gets updated and the user is able to visualize the changes. DOM is an interface that allows scripts to update the content, style, and structure of the document.

- `Virtual DOM`
  - The Virtual DOM is a light-weight abstraction of the DOM. You can think of it as a copy of the DOM, that can be updated without affecting the actual DOM. It has all the same properties as the real DOM object, but doesn’t have the ability to write to the screen like the real DOM.
  - Virtual DOM is just like a blueprint of a machine, can do the changes in the blueprint but those changes will not directly apply to the machine.
  - Reconciliation is a process to compare and keep in sync the two files (Real and Virtual DOM). Diffing algorithm is a technique of reconciliation which is

used by React.

- `Real DOM`
  - The DOM represents the web page often called a document with a logical tree and each branch of the tree ends in a node and each node contains object programmers can modify the content of the document using a scripting language like javascript and the changes and updates to the dom are fast because of its tree-like structure but after changes, the updated element and its children have to be re-rendered to update the application UI so the re-rendering of the UI which make the dom slow all the UI components you need to be rendered for every dom update so real dom would render the entire list and not only those item that receives the update .

`Real DOM`	`Virtual DOM`
DOM manipulation is very expensive	DOM manipulation is very easy
There is too much memory wastage	No memory wastage
It updates Slow	It updates fast
It can directly update HTML	It can't update HTML directly
Creates a new DOM if the element updates.	Update the JSX if the element update
It allows us to directly target any specific node (HTML element)	It can produce about 200,000 Virtual DOM Nodes / Second.
It represents the UI of your application	It is only a virtual representation of the DOM

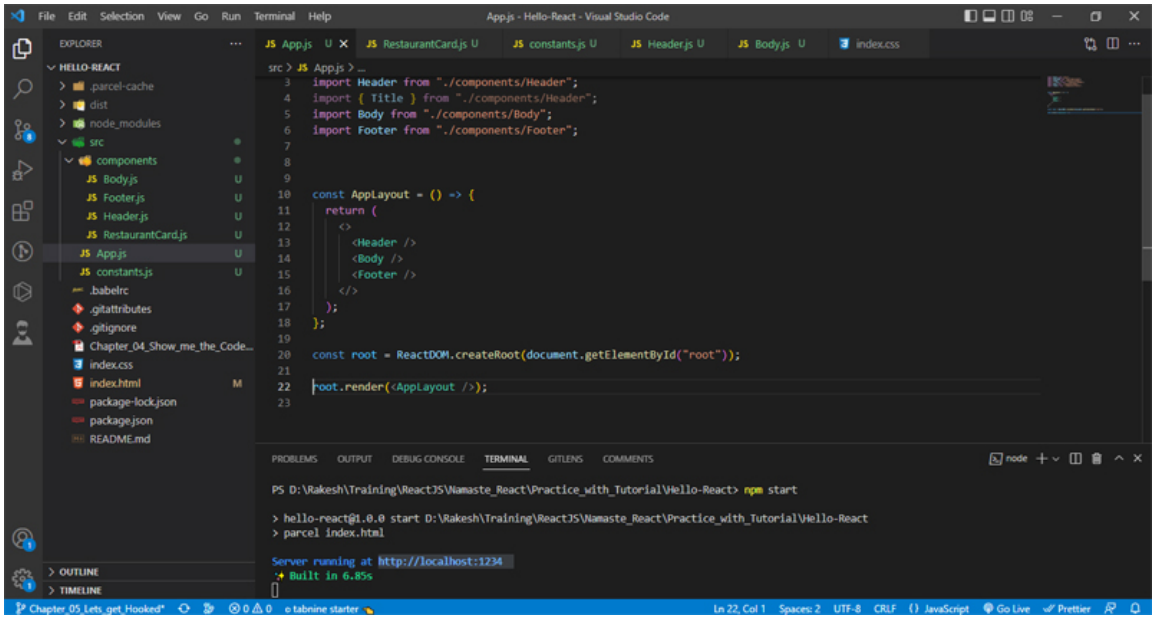
▼ Chapter 5: Let's get Hooked

created "src" folder and moved all project related files inside it.

```
// Import & Export in React

// We can export like this from any file
export default Header;

// As we can export only 1 thing, we can export other parts while writing components
export const Title = () => (
     . . . . . .
  );

// Default import
// When importing default, we can change name of component, and use new name given
import Header from "./components/Header";
```

```
// Named import
import { Title } from "./components/Header";
// We can use multiple named import
import { Title, Header } from "./components/Header";

// If we want to import everything then we can write as below
import * as Obj from "./components/Header";
```

Create “src” folder in project. Create “components” folder inside src. Inside components folder, cut paste all components from “App.js” & create separate .js file for each component. Create “constants.js” file inside “src” folder, put all hard coded data inside it.



Hooks

useState Hook: Every component in react maintains a state. useState is used to create state variable. It is used to manage states. It returns a “State variable” and an “updater function” to update that state variable.

```
// Importing useState
import { useState } from "react";

// How to use useState in our code

// in below code "searchText" is a state variable
// "setSearchText" is a function to update "searchText"
// we can provide default value to variable, for e.g., "KFC" in below case
const [searchText, setSearchText] = useState("KFC");

const [restaurants, setRestaurants] = useState(restaurantList);



// We kept value as "searchText" which is state variable
// onChange of input we called "setSearchText" function and passed user input
// "e.target.value" gives whatever entered by user in below input box
<input type="text" className="search-input" placeholder="search" value={searchText}
          onChange={(e) => {setSearchText(e.target.value);}}
        />


// We need to search for restaurant when clicked on Search button

<button className="search-btn" onClick={() => {
            const data = filterData(searchText, restaurants);
            setRestaurants(data);
          }}
        >Search</button>


function filterData(searchText, restaurants) {
  const filterData = restaurants.filter((restaurant) =>
    restaurant.data.name.includes(searchText)
  );
  return filterData;
}
```