
REACT
REVISION
PART-1
WRITTEN BY
ARVIND PANDIT PRAJAPATI

Day 1 of revising React

Hello, I'm Arvind Pandit Prajapati, a frontend developer with a passion for React. I am currently on a revision journey to explore the power, beauty, and ability of React.

Connect with me

- LinkedIn: https://www.linkedin.com/in/arvindpndit/
- GitHub: https://github.com/arvindpndit

I'm excited to share my progress with you. Let's dive in and explore the world of React together!

Creating a "Hello World" Heading

First things first, let's create a simple heading with "Hello World" text. Since you probably already know HTML, here's the code:

```
<div id="root">
  <h1>Hello World</h1>
</div>
```

But let's do it using JavaScript for practice:

```
const root = document.querySelector("#root");
const heading = document.createElement("h1");
heading.textContent = "Hello World";
root.appendChild(heading);
```

Understanding CDNs

Now, let's talk about Content Delivery Networks (CDNs). What are they? Simply put, they are a system of distributed servers that deliver web content to users based on their geographic location.

In React, we use CDNs to include the necessary scripts for React in our project. In most cases, you'll see a `crossorigin` attribute in the CDN links, which tells the browser to allow loading of scripts even if they are hosted on another domain.

Setting up React

Now that we have a basic understanding of CDNs, let's set up React in our project. We'll start by copying the CDN links from the React documentation and pasting them in the `head` section of our HTML file.

Once we have the necessary scripts, we can start writing React code. To create the same "Hello World" heading using React, we can use the `createElement` API provided by React:

```
const heading = React.createElement("h1", { id: "heading" }, "Hello world");

const root = ReactDOM.createRoot(document.querySelector("#root"));

root.render(heading);
```

Conclusion

And that's it for day 1 of my React journey! We learned how to create a simple heading using JavaScript, the basics of CDNs, and how to set up React in our project. In the next session, we'll learn about JSX and how it makes writing complex HTML structures easier in React. Stay tuned!

Day 2 of revising React

Today, I separated the HTML, JS, and CSS files to maintain separation of concerns. I also created my own "create_react_app" to make my app production-ready.

To make my app production-ready, I needed to minify the code, optimize the app, bundle it, and run it on a server. I learned that React is a powerful tool, but it cannot do everything alone. That's why I use bundlers like Vite, Parcel, and Webpack to get the desired functionality.

In this project, I used Parcel as my bundler because it is zero-config and provides a lot of features for my app. To use Parcel in my app, I needed a package manager, and I chose npm. npm and yarn are the most commonly used package managers in the market, but I chose npm for this project.

Installing npm and Parcel

To install npm in my app, I used the command `npm init` or `npm init -y` to skip all the questions. Once npm was installed, I saw a `package.json` file along with my existing files.

The `package.json` file contains metadata about my project, including the dependencies required for my app. I used this file to install external packages in my application.

To install Parcel in my app, I used the command `npm install -D parcel`. The `-D` flag installed the package as a dev dependency.

When I installed a package, I got a `package-lock.json` file in my app, which ensured that my app used the exact version of the package.

The caret (^) and tilde (~) signs in front of version numbers indicate which versions of a package are compatible with our app.

The `node_modules` file is like a database for npm. It contains all the packages and dependencies installed in our app. It's essential to put the `node_modules` file in the `.gitignore` folder to avoid uploading it to version control.

Using React and React-DOM with npm

To install React and React-DOM, I used the command `npm install react react-dom`. I removed the CDN links that I used before and used these installed packages.

Running the app with Parcel

To run my app, I used the command `npx parcel index.html`, which created a mini server to run my app on `localhost:1234`.

The `dist` folder contained the production-ready build of my app, and the `parcel_cache` folder contained cached files generated by Parcel during development.

I can make a production build using `npx parcel build index.html` and a development build using `npx parcel index.html`. It's essential to put the `dist` and `parcel_cache` folders in the `.gitignore` folder to avoid uploading them to version control.

Using Browserlist to specify browser support

I also learned about Browserlist, which is a feature in package.json that allows me to specify which browsers I want to support for my project. By default, it uses a list of commonly used browsers and their versions. However, I can customize this list to better suit my needs. This is important because different browsers may have different capabilities and may require different features or polyfills to work properly. By specifying which browsers I want to support, I can ensure that my project will work correctly for my target audience.

Here's an example of how to specify a Browserlist in my package.json file:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "browserslist": ["last 2 versions", "not dead"],
  "dependencies": {
    "react": "^17.0.2",
    "react-dom": "^17.0.2"
  }
}
```

In this example, I am specifying that I want to support the last two versions of all major browsers, except for any that are considered "dead" (i.e. no longer supported). I can customize the list to fit my specific needs by specifying different versions or browser names. The Browserslist documentation provides more information on how to use this feature.

Parcel is a Beast:

Finally, I learned all the features of Parcel, including -

- HMR (Hot Module Replacement): Enables real-time changes to your code without needing to refresh the entire page, making development faster and more efficient.
- File watcher algorithm in C++: A performant file watcher that efficiently tracks changes to your code and rebuilds only what's necessary.
- Bundling: Combines all of your code and dependencies into a single file, making it easier to deploy and reducing page load times.

- Code minification: Reduces the size of your code by removing unnecessary whitespace and renaming variables, making it faster to download and execute.
- Image optimization: Automatically optimizes images to reduce their size without sacrificing quality, improving page load times.
- Caching: Speeds up development by caching assets and only rebuilding when necessary, improving build times.
- Compression: Reduces the size of your assets, making them faster to download and improving page load times.
- Tree shaking: Eliminates unused code from your final build, reducing its size and improving performance.

Day 3 of revising React

In the previous day, I learned about the bundler and how it makes our app faster. Our app is dependent on the bundler which further depends upon other packages, and those can have dependencies on other packages, which is known as transitive dependencies.

I discussed browserlist yesterday, and when we write browserlist in the package.json file, what happens is our code is converted so that it can be run in older JavaScript engines.

But the question is: who does that? Does React do that or does Parcel do that? Interesting question, right? The answer is neither of those two does that. Babel does that!

Now, I'm confused. I didn't install Babel yesterday. Do I have to install it? No, I don't because it came along with Parcel. When I installed Parcel, Babel and a lot more packages came along with it.

Babel

What is Babel?

Babel is a popular tool used in web development to convert ECMAScript 2015+ code (also known as ES6+) into backwards-compatible versions of JavaScript that can run on older browsers or environments that don't support the latest syntax and features.

Babel works by analyzing and transforming your code, converting any modern JavaScript features that might not be supported by all browsers or environments into equivalent code that is widely compatible. This allows developers to write cutting-edge, modern JavaScript while still being able to support older systems.

In addition to converting modern syntax, Babel can also perform other tasks such as transforming JSX syntax used by React into regular JavaScript, and adding polyfills for features that are missing from certain browsers or environments.

To run my project, I have to run `npx parcel index.html`, which takes time to write. So I'm going to build a script inside `package.json`:

```
"scripts": {
  "start": "parcel index.html",
  "build": "parcel build index.html"
}
```

Now, to run, I'll use `npm run start` or `npm start`, which is faster to write.

At the end of the project, it is necessary that I'll remove all the `console.log()`. To do so, there is a package `babel-plugin-transform-remove-console`, which removes all the `console.log()` statements.

After installing this package, I created a folder `.babelrc`, and in this folder, I'll write the given config:

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

Now all the `console.log()` statements are disappeared just like magic.

Introduction of keys

When a React element has multiple children, each child should have a unique key that is used by React for efficient tree conversion.

Here's an example code snippet that creates two headings using `React.createElement()` and attaches them to a root element:

```
import React from "react";
import ReactDOM from "react-dom/client";

const root = ReactDOM.createRoot(document.querySelector("#root"));
const heading1 = React.createElement(
  "h1",
  {
    id: "heading1",
    key: "1",
  },
  "Heading one"
);

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

const container = React.createElement("div", {}, [heading1, heading2]);
```

```
root.render(container);
```

The `React.createElement()` function returns an object that represents the HTML code and puts it upon the DOM. I've come across a common problem where I need to create a large HTML structure using React's `createElement()` method. While this method works fine for small structures, it quickly becomes unmanageable for larger ones.

So now, instead of writing tedious code with `createElement()`, I can write HTML-like code in my JavaScript files and let JSX do the heavy lifting. It's a game-changer for my React development!

Day 4 of revising React

What is JSX?

JSX (JavaScript XML) is an extension to the JavaScript language, which allows you to write HTML-like syntax directly in your JavaScript code. It is not a requirement for writing React applications, but it is a popular choice among developers because it offers a lot of benefits.

Why JSX was introduced?

JSX was introduced to make it easier to create and manipulate the DOM elements in a React application. Instead of using plain JavaScript to create and manipulate HTML elements, developers can use JSX to write code that is easier to read and maintain.

Converting from React.createElement to JSX

Let's take a look at the following code that we are already familiar with:

```
const heading = React.createElement(
  "h1",
  {
    id: "title",
  },
  "Namaste Everyone"
);
```

Here, we are creating a heading element using `React.createElement()`. We are passing three arguments to this function: the type of element we want to create (`h1`), its attributes (`id: "title"`), and its content (`"Namaste Everyone"`).

Now, let's convert this code into JSX:

```
const heading = <h1>Namaste Everyone</h1>;
```

As you can see, writing JSX is very simple and allows us to write HTML-like syntax directly in our JavaScript code.

Remark: JSX is not HTML inside JavaScript, it is HTML-like syntax.

The Magic Behind JSX

Under the hood, JSX is converted into calls to `React.createElement()` by a tool called Babel. When you write JSX code, Babel compiles it into plain JavaScript that can be understood by the browser.

Here's what happens behind the scenes when we write JSX:

JSX => Babel => React.createElement() => Object => HTML(DOM)

In other words, Babel compiles JSX into calls to `React.createElement()`, which returns an object that represents the DOM element. This object is then converted into HTML and rendered in the browser.

Advantages of JSX

JSX offers several advantages over plain JavaScript for creating and manipulating DOM elements:

1. **Readability**: JSX code is much easier to read than plain JavaScript, especially when dealing with complex DOM structures.
2. **Syntactical Sugar**: JSX is a syntactical sugar over `React.createElement()`, which makes it easier to write and understand code.
3. **Less Code**: JSX allows you to write less code than plain JavaScript, which can save time and reduce the chances of making errors.
4. **Developer Experience**: JSX provides a better developer experience by allowing developers to write code that is more familiar to them.
5. **Maintainability**: JSX code is easier to maintain than plain JavaScript, as it is more declarative and easier to read and understand.

In summary, JSX is a powerful tool that allows developers to write HTML-like syntax directly in their JavaScript code, making it easier to create and manipulate DOM elements in React applications.

React Components

In React, everything is a component - from simple UI elements like buttons and forms to entire pages and applications. Components are reusable, modular pieces of code that can be easily combined to build complex UIs.

There are two types of React components - functional and class-based components. Functional components are simply JavaScript functions that return JSX code. They are the newer and more preferred way of writing React components.

An example of a functional component is:

```
const HeadingComponent = () => {
  return <h1>Namaste Everyone</h1>;
};
```

This component simply returns an `h1` element with the text "Namaste Everyone".

Functional components can be easily composed to build more complex UIs. For example, we can define another functional component called `Heading2`:

```
const Heading2 = () => {
  return <h2>This is second functional component</h2>;
};
```

We can then use `Heading2` inside `HeadingComponent`:

```
const HeadingComponent = () => {
  return (
    <>
      <h1>Namaste Everyone</h1>
      <Heading2 />
    </>
  );
};
```

This code renders both the `h1` element and the `Heading2` component.

We can also use a functional component inside another functional component by calling it as a function. For example:

```
const Heading = () => {
  return <h3>Hi from Heading!</h3>;
};

const HeadingComponent = () => {
  return (
    <>
      <h1>Namaste Everyone</h1>
      {Heading()}
    </>
  );
};
```

This code also renders both the `h1` element and the `Heading` component.

Finally, component composition is the concept of combining multiple smaller components to create larger and more complex components. It is a fundamental concept in React development and allows for greater reusability and maintainability of code.

Day 5 of revising React

Hi there, it's me, Arvind! Today is day 5 of my React revision, and I'm excited to start building a food ordering application. To begin with, I created the basic layout of my application using the following components:

- AppLayout
  - Navbar
  - Body
    - SearchBar
    - RestaurantList
      - RestaurantCard
  - Footer

To style my components, I decided to use Tailwind CSS and focus more on React than on CSS.

After creating the basic layout, I started creating functional components that we learned about yesterday. I first created the `AppLayout` component:

```
const AppLayout = () => {
  return (
    <div>
      <Navbar />
      <Body />
      <Footer />
    </div>
  );
};
```

Then I created the `Navbar` component:

```
const Navbar = () => {
  return (
    <div className="bg-green-50">
      <div className="flex justify-between items-center py-3 bg-green-50 w-9/12 mx-auto rou
        <h1 className="text-4xl font-bold text-green-600">FoodZilla</h1>
        <div className="flex gap-6 text-green-700">
          <div>Home</div>
          <div>Service</div>
          <div>Cart</div>
        </div>
      </div>
    </div>
  );
};
```

Next, I created the `Footer` component:

```
const Footer = () => {
```

```
  return (
    <div className="flex justify-center bg-slate-600 py-3 text-white">
      <div>This is React-Revision by Arvind Pandit</div>
    </div>
  );
};
```

Then, I focused on the `Body` component, which contains a search bar and `RestaurantList` component. I first created the `RestaurantList` component:

```
const RestaurantList = () => {
  return (
    <div className="mt-7 mb-10 flex flex-wrap justify-between gap-5 w-9/12 mx-auto">
      {cards.map((card) => {
        return <RestaurantCard restaurant={card} />;
      })}
    </div>
  );
};
```

Finally, I created the `RestaurantCard` component:

```
const RestaurantCard = ({ restaurant }) => {
  const { name, cuisines, cloudinaryImageId, costForTwoString } =
    restaurant.data;
  return (
    <div className="w-72 h-72 border shadow-sm hover:shadow-md p-2">
      <img
        src={
          "https://res.cloudinary.com/swiggy/image/upload/fl_lossy,f_auto,q_auto,w_508,h_320
          cloudinaryImageId
        }
        alt="food"
      />
      <div className="font-semibold py-1">{name}</div>
      <div className="text-sm pb-2">{cuisines.join(", ")}</div>
      <div className="text-gray-500">{costForTwoString}</div>
    </div>
  );
};
```

I got the data from the `cards` array, which I fetched from the Swiggy API.

Now, let's discuss some important concepts in React:

1. Config Driven UI

Config-driven UI means that the UI or the frontend is driven by the backend. In other words, the UI updates according to the data sent by the backend. The

configuration is defined in the backend and is sent to the frontend. The frontend renders the UI based on this configuration. This approach makes it easier to maintain and update the UI.

2. Virtual DOM

Virtual DOM is the representation of our actual DOM. It is required for reconciliation. Reconciliation is the process by which React updates the UI. It involves comparing the current virtual DOM with the previous one and updating the real DOM with the differences. The virtual DOM is a lightweight copy of the actual DOM, and changes made to it are not immediately reflected in the browser. Instead, React batches these changes and updates the actual DOM in an optimized way.

3. Props

Props are short for properties. They are passed to the children of a component. Props are used to pass data from the parent component to its children. This allows for more modular and reusable code. Props are immutable, which means they cannot be changed by the component that receives them.

4. Reconciliation/Diff Algorithm

Reconciliation is the process by which React updates the UI. It involves comparing the current virtual DOM with the previous one and updating the real DOM with the differences. React uses a diff algorithm to determine what has changed in the virtual DOM. The diff algorithm is a process of comparing two trees and returning the differences between them. The algorithm is optimized to minimize the number of updates to the actual DOM.

5. React Fiber

React Fiber is a new core algorithm in React that is responsible for scheduling and rendering updates. It is a reimplementation of the reconciliation algorithm that allows for better performance and more granular control over rendering. React Fiber introduces the concept of priorities, which allows React to prioritize certain updates over others, improving the perceived performance of the application. React Fiber also allows for interruption and resumption of the rendering process, which can help prevent the application from becoming unresponsive.

After working on the application, I realized that my `App.js` file was getting messy, and everything was inside it. So tomorrow, I plan to modularize my code by separating the components and constants files from the `App.js` file. Overall, I'm excited to continue working on my food ordering application using React.

Day 6 of revising React

Creating a File Structure

Yesterday I created the layout of my application but I wrote everything inside the `App.js` file and my `App.js` file was looking so messy. So today, I decided to make different files for different components which will make our code more modular and easier to navigate.

To achieve this, I created a `src` folder and added the `App.js` inside it. Then, I created a `component` folder and put all the components inside it. For my `cards` array, I created a `constants.js` file and put the cards array inside it.

Named and Default Export

To make our code more modular, we need to understand named and default exports.

Let's take an example to understand this:

```
const NavbarComponent = () => {
  return <div>NavbarComponent</div>;
};

export default NavbarComponent;
```

This is a default export. To import this, we write:

```
import NavbarComponent from "./components/NavbarComponent";
```

For named export, we do:

```
export const NavbarComponent = () => {
  return <div>NavbarComponent</div>;
};
```

To import this, we write:

```
import { NavbarComponent } from "./components/NavbarComponent";
```

Learning Hooks: useState Hook

Hooks are basically normal JS functions. Yes, you read it right - hooks are just normal JS functions. Now, let's learn our first React hook - `useState` hook.

`useState` is used to create a state variable. Here's the syntax:

```
const [name, setName] = useState("Arvind");
```

Here, `name` is the state variable and `setName` is a function that `useState` returns. To change the value of the `name` variable, you cannot simply do:

```
name = "DevArvind";
```

To change the state variable `name` to `"DevArvind"`, you need to use the `setName` function, like this:

```
setName("DevArvind");
```

Sure, here's the updated README.md file written in first person:

React SearchBar Component

I've created a simple React component that allows users to search for restaurants based on their names. I've used the `useState` hook from the React library to manage the state of the search input field.

Why did I use the `useState` hook?

I used the `useState` hook to create a state variable `searchInput` and a function `setSearchInput` to manage the value of the search input field. Whenever the user types something in the search input field, the `onChange` event handler updates the value of `searchInput` using the `setSearchInput` function. This triggers a re-render of the component with the new value of `searchInput`. The `filteredRestaurantList` function uses the current value of `searchInput` to filter the list of restaurants based on their names.

How to use the component?

To use this component, simply import it into your React application and pass in the following props:

- `cards`: an array of restaurant data
- `setNewRestaurant`: a function that updates the list of restaurants to be rendered based on the search input

The component renders a search input field and a search button. Whenever the user types something in the search input field and clicks the search button, the component filters the list of restaurants based on the search input and updates the list of restaurants to be rendered using the `setNewRestaurant` function.

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

const SearchBar = ({ cards, setNewRestaurant }) => {
  const [searchInput, setSearchInput] = useState("");

  const filteredRestaurantList = () => {
    const filteredRestaurantListData = cards.filter((card) =>
      card.data.name.includes(searchInput)
    );
    return filteredRestaurantListData;
  };
```

```
  return (
    <div className="text-center mt-3 outline-none ">
      <input
        type="text"
        placeholder="Search Food"
        value={searchInput}
        onChange={(e) => setSearchInput(e.target.value)}
        className="bg-green-100 px-3 py-2 "
      />

      <button
        className="bg-green-200 px-3 py-2"
        onClick={() => {
          //filter the restaurant list
          const filteredRestaurantListData = filteredRestaurantList();
          //set the new restaurant data which gets rendered
          setNewRestaurant(filteredRestaurantListData);
        }}
      >
        Search
      </button>
    </div>
  );
};

export default SearchBar;
```