⚛️ React Essentials: From Console Removal to Config Driven UI

🚀 `npx` vs `npm run`

> ❓ **Did you know?** `npx` is used to run binaries from `node_modules` directly. 💡 It's similar to `npm run` but more powerful!

```
npx create-react-app myApp
```

➡️ **Internally:** `npx` = `npm exec` = Executes a CLI tool without globally installing it.

🧹 Remove Console Logs in Production

🗑️ **Console logs slow down your app in production! Clean them!**

🔧 Setup: Remove `console.log`

1. Install plugin:

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

2. Create `.babelrc` file:

```
{
  "plugins": ["transform-remove-console"]
}
```

3. Build it:

```
npm run build
```

☑️ Result: All `console.log` are **gone** from the production bundle.

🖼️ Render vs Reconciliation

⚙️ Render

📌 **Render = Updating the DOM.** Every React component's `render()` returns the UI tree → which React uses to update the DOM.

🔄 Reconciliation

💡 React compares the **previous** and **new** element tree to minimize changes.

Example:

```
<div>
  <Child1 />
  <Child2 />
</div>
```

🔁 On update:

- React compares **Child1** & **Child2**
- If no change → skips unnecessary updates ☑️

😞 But if you insert `<Child0 />` at the top:

```
<div>
  <Child0 />
  <Child1 />
  <Child2 />
</div>
```

❌ React **re-renders everything** = Bad for performance.

🔑 Key to Optimization = `key` Prop

💡 Solution:

- React uses `key` to uniquely identify list items.
- Helps in **matching elements** instead of re-rendering all.

```
{items.map((item) => (
  <Card key={item.id} data={item} />
))}
```

☑️ Helps React to smartly re-render **only what changed**.

🛠️ React.createElement() & JSX

⚙️ `React.createElement()`:

Creates a **React Element** object (not actual DOM node yet).

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

Becomes:

```
<h1 id="heading">Namaste React</h1>
```

💡 JSX = JavaScript + XML

> JSX is a **syntactic sugar** on top of `React.createElement()`.

```
<h1 id="heading">Namaste React</h1>
```

🛠️ Compiled by **Babel** into:

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

📦 Babel is bundled automatically with **Parcel** (zero-config bundler).

💎 JSX Advantages

☑ Developer Experience ☑ Readability ☑ Cleaner Code ☑ Easy Maintenance ☑ Secure (Sanitized automatically)

```
const name = "<script>alert('XSS')</script>";
<h1>{name}</h1>  // Will be sanitized automatically
```

🧱 Everything is a Component

2 Types:

1. 🔧 **Functional Component** (Modern)
2. 🏛️ **Class Component** (Legacy)

🧠 Functional Component

Just a **JavaScript Function** returning JSX.

```
function HeaderComponent() {
  return <h1>Hello from Header</h1>;
}
```

Use:

```
<HeaderComponent />
```

🔁 Component Composition

🧱 Nesting components = Composition.

```
function App() {
  return (
    <div>
      <Header />
      <Body />
      <Footer />
    </div>
  );
}
```

☑ 3 Ways to compose:

- `{Header()}`
- `<Header />` ☑ (preferred)
- `<Header></Header>`

🧩 React Fragment

Problem: JSX must return **only one** parent element.

Solution: `React.Fragment` or shorthand `<> </>`

```
<>
  <ChildA />
  <ChildB />
</>
```

☑ Groups multiple children without adding extra node.

🚫 Cannot style shorthand fragments directly.

🎨 Styling in React

Three common ways:

1. **Inline Styles (object):**

```
<h1 style={{ color: "red" }}>Hello</h1>
```

2. **CSS Modules/Files:**

```
<div className="heading">Hello</div>
```

3. **Utility Libraries:** Tailwind CSS, Bootstrap, Material UI, etc.

🧠 Config Driven UI

🎯 Modern UIs like **Swiggy** use config from backend to render UI dynamically.

📦 Config = JSON/Array of Objects

```
{
  "type": "restaurant_list",
  "cards": [ { "name": "KFC" }, { "name": "Burger King" } ]
}
```

Frontend dynamically renders based on this data = 🔥 highly scalable UI.

❓ Optional Chaining (ES2020)

No more null/undefined checks!

```
restaurantList[0]?.data?.name
```

☑️ If anything is `undefined` in the chain, result is `undefined` instead of error.

📦 Props in React

🧳 Props = Arguments passed to components

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

Access via:

```
function RestaurantCard(props) {
  return <h2>{props.resData.name}</h2>;
}
```

🎯 Use destructuring for cleaner code:

```
function RestaurantCard({ resData }) {
  return <h2>{resData.name}</h2>;
}
```

🧪 Spread Operator `...`

Used for:

☑ Spreading arrays ☑ Merging objects ☑ Passing arguments

```
const obj1 = { name: "React" };
const obj2 = { ...obj1, version: 18 };

// Passing props
<Component {...props} />
```

🏁 Final Project Structure Example: Food Delivery UI

```
App
├── Header (Logo, Nav)
├── Body (Search, RestaurantList)
├── Footer
```

✨ BONUS: Example JSX Tree

```
<>
  <Header />
```

```
    <SearchBar />
    <RestaurantList />
    <Footer />
  </>
```

💡 Summary Cheatsheet

🧠 Concept	📘 Quick Notes
`npx`	Run CLI tools without global install
Babel Plugin	Removes `console.log` in prod
Reconciliation	Efficient DOM update algorithm
`key` in list	Optimize diffing process
JSX	Syntax to write HTML-like JS
`React.createElement`	JSX compiles into this
Functional Component	Basic building block
Fragment	Avoid extra divs
Config Driven UI	Backend controls UI
Props	Data passed to components
Optional Chaining	Safe property access
Spread Operator	Merge and pass data cleanly
