A Guide to React Concepts

This document covers the fundamental concepts of React, a JavaScript library for building modern, interactive user interfaces.

1. What is React?

- **A JavaScript Library:** React is not a full framework (like Angular). It's a library focused on one thing: building user interfaces (UIs).
- **Component-Based:** The core idea of React is to break your UI into small, reusable pieces called **components**. A button, a form, a user profile card—these are all components.
- **Declarative:** You "declare" what your UI should look like based on its current state, and React handles the complex job of updating the actual DOM (Document Object Model) efficiently.

2. Core Concept: JSX (JavaScript XML)

JSX is a syntax extension for JavaScript that looks very similar to HTML. It's the primary way you will write your UI in React.

- It's not HTML. It's "syntactic sugar" that gets converted into regular JavaScript (React.createElement()) by a build tool.
- It allows you to write HTML-like structures directly inside your JavaScript code.

```
// This is JSX
const element = <h1>Hello, world!</h1>;

// You can embed JavaScript expressions inside {}
const name = "Alice";
const greeting = <p>Hello, {name}!</p>;

// HTML attributes become camelCase
// 'class' becomes 'className'
// 'for' becomes 'htmlFor'
const myElement = <div className="container">My Content</div>;
```

3. Core Concept: Components

Components are the building blocks of any React application. They are like JavaScript functions that return JSX.

- **Functional Components:** The modern and standard way to write components. They are literally just JavaScript functions.

- **Component Names:** Must always start with a capital letter (e.g., MyButton, not myButton).

```
// A simple functional component
function Welcome() {
  return <h1>Hello, this is a component!</h1>;
}

// You "use" a component like an HTML tag
// <Welcome />
```

4. Core Concept: Props (Properties)

Props (short for "properties") are how you pass data *from a parent component down to a child component*. Props are **read-only**; a component can *never* change its own props.

```
// 1. Define a component that accepts 'props'
function Welcome(props) {
  // props is an object: { name: "Bob" }
  return <h1>Hello, {props.name}!</h1>;
}

// You can "destructure" props for cleaner code
function Welcome({ name, age }) {
  return <p>Hello, {name}! You are {age}.</p>;
}

// 2. Use the component and "pass props" like HTML attributes
function App() {
  return (
    <div>
      <Welcome name="Alice" age={25} />
      <Welcome name="Bob" age={30} />
    </div>
  );
}
```

5. Core Concept: State (useState Hook)

If props are data passed *in*, **state** is a component's *own internal memory*. It's data that can change over time in response to user events.

The useState hook is the standard way to add state to a functional component.

- useState returns an array with two things:
    1. The current state value.
    2. A function to update that value.

```
import React, { useState } from 'react';

function Counter() {
  // 1. Declare a state variable "count"
  //    '0' is the initial value.
  //    'setCount' is the function to update it.
  const [count, setCount] = useState(0);

  // 2. When you call setCount, React "re-renders"
  //    this component with the new count value.
  return (
    <div>
      <p>You clicked {count} times</p>
      {/* We'll cover events next! */}
    </div>
  );
}
```

6. Handling Events

React elements can listen for events, just like in HTML, but the syntax is camelCase (e.g., onClick, onChange).

You pass a function (an "event handler") to the event attribute.

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  // This is an event handler function
  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
```

```
      <p>You clicked {count} times</p>
      {/* Pass the function to the onClick prop */}
      <button onClick={handleClick}>
        Click me
      </button>

      {/* You can also use an inline arrow function */}
      <button onClick={() => setCount(count - 1)}>
        Decrement
      </button>
    </div>
  );
}
```

7. Core Concept: Lifecycle (useEffect Hook)

Components have a "lifecycle": they are "born" (mounted), they "live" (update), and they "die" (unmounted).

The useEffect hook lets you run "side effects" (like fetching data, setting timers, or manually changing the DOM) at different points in this lifecycle.

```
import React, { useState, useEffect } from 'react';

function DataFetcher({ id }) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  // useEffect runs *after* the component renders
  useEffect(() => {
    // This is the "effect"
    setLoading(true);
    fetch(`https://api.example.com/data/${id}`)
      .then(res => res.json())
      .then(json => {
        setData(json);
        setLoading(false);
      });

    // Optional: Return a "cleanup" function
    // This runs when the component unmounts
    return () => {
      console.log("Component is unmounting or 'id' changed");
    };

  }, [id]); // <-- This is the "dependency array"

  // Rules of the dependency array:
  // 1. [id]: The effect runs when 'id' changes.
  // 2. []:   An empty array means the effect runs *only once* when the component mounts.
  // 3. (no array): The effect runs on *every single render* (usually a bug!).

  if (loading) {
    return <p>Loading...</p>;
  }

  return <div>{data.name}</div>;
}
```

8. Conditional Rendering

You often need to show different JSX based on a condition. You can't use if/else *inside* JSX, but you can use:

1. **Ternary Operator:** condition ? <A /> : <B />
2. **Logical &&:** condition && <A /> (Renders <A /> if condition is true, renders nothing otherwise)
3. **if statements:** You can use a regular if *before* your return statement to decide what to render.

```
function Greeting() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  return (
    <div>
      {/* 1. Ternary Operator */}
      {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}

      {/* 2. Logical && Operator */}
      {isLoggedIn && <button>Log Out</button>}
    </div>
  );
}
```

9. Lists and Keys

You can't just render an array of components. You must loop over the array (using .map()) and return a JSX element for each item.

Crucial Rule: When rendering a list, you *must* provide a unique key prop to each item. This helps React identify which items have changed, been added, or been removed.

```
const products = [
  { id: 'p1', name: 'Laptop' },
  { id: 'p2', name: 'Mouse' },
  { id: 'p3', name: 'Keyboard' },
];

function ProductList() {
  return (
    <ul>
      {products.map((product) => (
        // 'product.id' is a perfect unique key.
        // *Never* use the array index as a key if the list can change!
        <li key={product.id}>
          {product.name}
        </li>
      ))}
    </ul>
  );
}
```

10. Forms (Controlled Components)

In React, form elements like <input> and <textarea> are typically "controlled." This means:

1. The element's value is set from a useState variable.
2. Its onChange event handler updates that useState variable.

This "single source of truth" (the state) controls the form.

```
import React, { useState } from 'react';

function NameForm() {
  // 1. Create state to hold the input's value
  const [name, setName] = useState("");

  function handleSubmit(event) {
    event.preventDefault(); // Stop page refresh
    alert("A name was submitted: " + name);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Name:</label>
      <input
        type="text"
        // 2. Value is tied to state
        value={name}
        // 3. onChange updates the state
        onChange={(event) => setName(event.target.value)}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

11. Other Important Hooks

useContext

- **Solves:** "Prop Drilling" (passing props down through many nested components that don't need them).
- **How:** Lets you create a global-like "context" (e.g., ThemeContext, UserContext) that any component in the tree can "subscribe" to.

useRef

- **Use Case 1:** Accessing a DOM element directly (e.g., to focus an input).
- **Use Case 2:** Storing a mutable value that *does not* cause a re-render when it changes (like a timer ID).

```
import React, { useRef, useEffect } from 'react';

// Use Case 1: Focusing an input
function MyInput() {
  // 1. Create a ref
  const inputRef = useRef(null);

  useEffect(() => {
    // 2. On mount, focus the input
    // 'inputRef.current' points to the DOM node
    inputRef.current.focus();
  }, []);

  // 3. Attach the ref to the DOM element
  return <input ref={inputRef} type="text" />;
}
```