1

**1. React**

React is a JavaScript library for building user interfaces.

* **Core Ideas**:
  + Component-based architecture: UI is split into reusable pieces (components).
  + Declarative: You define what the UI should look like, and React manages updates.
  + Virtual DOM: Efficiently updates the UI by calculating changes.

**2. Thinking in React**

* Break the UI into components.
* Build a static version in React (props only, no state).
* Identify the minimal representation of UI state.
* Determine where the state should live.
* Add inverse data flow (child to parent communication).

**3. Import/Export Components**

* **Import**: Use components in other files.

javascript

Copy code

import ComponentName from './ComponentName';

* **Export**: Make a component available.

javascript

Copy code

export default ComponentName;

**4. Writing JSX**

JSX is a syntax extension that allows mixing HTML-like markup with JavaScript.

* **Example**:

javascript

Copy code

const element = <h1>Hello, world!</h1>;

* Follows XML-like syntax.

**5. JavaScript in JSX**

* Use curly braces {} to embed JavaScript in JSX.
* **Example**:

javascript

Copy code

const name = "John";

const element = <h1>Hello, {name}!</h1>;

**6. Keeping Components Pure**

* A **pure component** is one that:
  + Returns the same output for the same input (props).
  + Does not modify props or global variables.
  + Avoids side effects (e.g., making HTTP requests).

**7. State Variables**

* State is a built-in object that allows components to remember data between renders.
* Use useState to define a state variable:

javascript

Copy code

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

**8. Lists (Dynamic or Static)**

* Render lists using map() in JSX.
* Add a unique key prop to each list item.
* **Example**:

javascript

Copy code

const items = ["Apple", "Banana", "Cherry"];

const list = items.map((item) => <li key={item}>{item}</li>);

**9. Updating State Variables (Objects)**

* React state updates must be immutable.
* Use the spread operator to update objects:

javascript

Copy code

setState(prevState => ({

...prevState,

key: newValue

}));

**10. Events**

* Handle user actions with event listeners (e.g., onClick).
* Pass functions to event attributes:

javascript

Copy code

<button onClick={handleClick}>Click me</button>

* Event handlers receive an event object.

**11. Creating React Root**

* Create a React root to render your React app.
* **Example**:

javascript

Copy code

import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));

root.render(<App />);

**12. React DOM**

* React DOM renders components to the actual DOM.
* Represents the UI as a tree structure.

**13. useState React Hook**

* Allows adding state to functional components.
* **Example**:

javascript

Copy code

const [name, setName] = useState('');

setName('Alice');

* Updates trigger re-rendering of the component.

2

**1. Describing the UI**

* React uses components to describe the UI in a declarative way.
* A component is a JavaScript function that returns JSX.
* Example:

javascript

Copy code

function Greeting() {

return <h1>Hello, World!</h1>;

}

**2. Passing Props**

* Props are used to pass data from a parent component to a child component.
* Props are immutable within the child.
* Example:

javascript

Copy code

function Welcome(props) {

return <h1>Hello, {props.name}!</h1>;

}

<Welcome name="Alice" />;

**3. Understanding Your UI as a Tree**

* React components form a tree structure where:
  + Each component is a node.
  + Parent components pass data (props) to child components.
* Visualize the hierarchy to manage complex UIs.

**4. Conditional Rendering**

* Display different elements based on conditions.
* Example:

javascript

Copy code

function Greeting({ isLoggedIn }) {

return isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please log in.</h1>;

}

**5. Adding Interactivity**

* Use event listeners and state to make components interactive.
* Example:

javascript

Copy code

function Counter() {

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

return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;

}

**6. Render and Commit**

* **Render Phase**: React prepares updates without making visible changes.
* **Commit Phase**: React updates the DOM and runs side effects (e.g., animations).

**7. Changing Mutable State – Part I (Objects)**

* Use the spread operator to update object state.
* Example:

javascript

Copy code

setState(prevState => ({ ...prevState, key: newValue }));

**8. Changing Mutable State – Part II (Arrays)**

* Use methods like map(), filter(), and the spread operator to update arrays immutably.
* Example:

javascript

Copy code

setArray(prevArray => [...prevArray, newItem]);

**9. Managing State**

* Determine where state should live (e.g., closest common parent).
* Avoid redundant state to prevent synchronization issues.

**10. Concept of State (Machine)**

* State represents the "current situation" of a component.
* It updates in response to user input or other events.
* Example:

javascript

Copy code

const [isOn, setIsOn] = useState(false);

**11. Lifting State Up**

* Share state between components by moving it to their common ancestor.
* Example:

javascript

Copy code

function Parent() {

const [sharedState, setSharedState] = useState(value);

return (

<>

<ChildA state={sharedState} />

<ChildB setState={setSharedState} />

</>

);

}

**12. State vs Component**

* **State**: Data managed by a component. Changes trigger re-renders.
* **Component**: Reusable building block of the UI.

**13. State as a Snapshot – Part I**

* State reflects the moment when it was last set. Updates are independent.
* Example:

javascript

Copy code

setState(count + 1); // Based on the current count value

**14. State as a Snapshot – Part II**

* Use functional updates for queued state changes.
* Example:

javascript

Copy code

setState(prevState => prevState + 1);

**15. Vanilla JavaScript vs React**

* **Vanilla JavaScript**:
  + Direct DOM manipulation.
  + Manual updates for UI changes.
* **React**:
  + Virtual DOM for efficient updates.
  + Component-based and declarative approach.

**16. Declarative vs Imperative Programming**

* **Declarative**: Describe *what* you want to happen (e.g., React).
  + Example:

javascript

Copy code

const items = list.map(item => <li>{item}</li>);

* **Imperative**: Describe *how* to achieve it (e.g., Vanilla JS).
  + Example:

javascript

Copy code

const ul = document.createElement('ul');

list.forEach(item => {

const li = document.createElement('li');

li.textContent = item;

ul.appendChild(li);

});

3

**1. Referencing Values with Refs**

* Refs in React store a mutable value that persists across renders.
* Created using useRef. Example:

javascript

Copy code

const inputRef = useRef(null);

console.log(inputRef.current); // Access the ref value

* Use cases include:
  + Accessing a DOM element.
  + Storing mutable state without causing re-renders.

**2. Manipulating the DOM with Refs**

* Directly manipulate DOM nodes using refs, bypassing React's declarative model.
* Example:

javascript

Copy code

function FocusInput() {

const inputRef = useRef(null);

return (

<input

ref={inputRef}

onClick={() => inputRef.current.focus()}

placeholder="Click to focus"

/>

);

}

**3. Making Use of useReducer Hook**

* An alternative to useState for managing complex state logic.
* useReducer accepts a reducer function and an initial state.
* Example:

javascript

Copy code

const reducer = (state, action) => {

switch (action.type) {

case 'increment': return { count: state.count + 1 };

case 'decrement': return { count: state.count - 1 };

default: throw new Error();

}

};

const [state, dispatch] = useReducer(reducer, { count: 0 });

**4. Making Use of useContext Hook**

* Simplifies passing data to deeply nested components without prop drilling.
* Example:

javascript

Copy code

const ThemeContext = React.createContext();

function App() {

return (

<ThemeContext.Provider value="dark">

<Child />

</ThemeContext.Provider>

);

}

function Child() {

const theme = useContext(ThemeContext);

return <div>Theme: {theme}</div>;

}

**5. Application Scalability**

* Use useReducer for managing complex state logic.
* Combine it with useContext to scale state across multiple components.
* Example: Global state management in a larger app:

javascript

Copy code

const AppContext = React.createContext();

function AppProvider({ children }) {

const [state, dispatch] = useReducer(reducer, initialState);

return (

<AppContext.Provider value={{ state, dispatch }}>

{children}

</AppContext.Provider>

);

}

**6. useRef – React Hook**

* useRef stores a reference to a DOM element or mutable value.
* Example (DOM element):

javascript

Copy code

const buttonRef = useRef(null);

useEffect(() => {

buttonRef.current.focus();

}, []);

**7. useImperativeHandle**

* Customizes the instance value exposed when using ref.
* Works with forwardRef.
* Example:

javascript

Copy code

const Input = forwardRef((props, ref) => {

const inputRef = useRef();

useImperativeHandle(ref, () => ({

focus: () => inputRef.current.focus(),

}));

return <input ref={inputRef} />;

});

const App = () => {

const inputRef = useRef();

return (

<div>

<Input ref={inputRef} />

<button onClick={() => inputRef.current.focus()}>Focus Input</button>

</div>

);

};

**8. forwardRef API**

* Enables passing refs to child components.
* Example:

javascript

Copy code

const Button = forwardRef((props, ref) => (

<button ref={ref}>{props.children}</button>

));

function App() {

const buttonRef = useRef();

return <Button ref={buttonRef}>Click Me</Button>;

}

**9. useState vs. useReducer**

* **useState**:
  + Best for simple state logic.
  + Example: Toggling a boolean or incrementing a counter.
* **useReducer**:
  + Best for complex state with multiple sub-values or actions.
  + Example: Managing a form or a todo list.
* **Comparison**:
  + useState has minimal setup and is more straightforward.
  + useReducer centralizes state logic and makes actions explicit.

4

**1. React Router DOM – Tutorial**

* **Purpose**: Handles navigation in a React app without reloading the page (Single Page Application).
* **Core Components**:
  + <BrowserRouter>: Wraps the app and enables routing.
  + <Routes>: Contains multiple <Route> elements.
  + <Route>: Maps a URL path to a component. Example:

javascript

Copy code

import { BrowserRouter, Routes, Route } from 'react-router-dom';

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

</Routes>

</BrowserRouter>

);

}

* + useNavigate: Programmatically navigate. Example:

javascript

Copy code

const navigate = useNavigate();

navigate('/about');

**2. Using useEffect Hook**

* **Purpose**: Synchronize the component with external systems (like APIs, subscriptions).
* Runs after the component renders and optionally cleans up when it unmounts.
* Syntax:

javascript

Copy code

useEffect(() => {

console.log("Component mounted or state/prop changed.");

return () => console.log("Cleanup logic here.");

}, [dependency]);

* **Key Points**:
  + []: Runs only once (on mount).
  + [dependency]: Runs whenever the dependency changes.

**3. Removing Effect Dependencies**

* **Issue**: Over-dependence on useEffect can make the code unnecessarily complex.
* **Approach**:
  + Avoid effects for synchronous calculations or derived state.
  + Use React’s state updates or derived values in the render logic.
  + Example:

javascript

Copy code

// Instead of this:

useEffect(() => {

setDerivedValue(propValue \* 2);

}, [propValue]);

// Do this:

const derivedValue = propValue \* 2;

**4. React: You Might Not Need an Effect**

* **Key Idea**: Many use cases for useEffect can be replaced with React’s state or props.
* Example: Avoid fetching data in useEffect if it can be derived from props:

javascript

Copy code

// Bad: Overuse of useEffect

useEffect(() => {

setFilteredData(data.filter(item => item.active));

}, [data]);

// Good: Compute filtered data directly in the render function

const filteredData = data.filter(item => item.active);

**5. Context Module Functions/Pattern**

* **Concept**: Organizes and manages state globally using React Context.
* **Pattern**:
  + Export both the provider and custom hooks for easier usage.
  + Example:

javascript

Copy code

const MyContext = React.createContext();

function MyProvider({ children }) {

const [state, setState] = useState(initialState);

return (

<MyContext.Provider value={{ state, setState }}>

{children}

</MyContext.Provider>

);

}

function useMyContext() {

const context = useContext(MyContext);

if (!context) {

throw new Error("useMyContext must be used within MyProvider");

}

return context;

}

export { MyProvider, useMyContext };

**6. Context Module Pattern (Variant Blog Example)**

* **Use Case**: Encapsulates complex state logic in a single module.
* **Structure**:
  1. Define Context.
  2. Define Provider.
  3. Define Custom Hooks.
  4. Expose everything in a single module.
  5. Example:

javascript

Copy code

// context.js

import { createContext, useContext, useState } from 'react';

const AuthContext = createContext();

function AuthProvider({ children }) {

const [user, setUser] = useState(null);

const login = (userInfo) => setUser(userInfo);

const logout = () => setUser(null);

return (

<AuthContext.Provider value={{ user, login, logout }}>

{children}

</AuthContext.Provider>

);

}

const useAuth = () => useContext(AuthContext);

export { AuthProvider, useAuth };

5

**1. Callbacks**

* **Concept**: A function passed as an argument to another function, executed after the completion of the parent function.
* **Use Case**: Handle asynchronous operations (e.g., file reading, API calls).
* Example:

javascript

Copy code

function fetchData(callback) {

setTimeout(() => {

console.log("Data fetched");

callback();

}, 1000);

}

fetchData(() => console.log("Callback executed"));

**2. Asynchronous Programming in JavaScript**

* **Concept**: Allows non-blocking code execution, enabling other operations to run while waiting for long tasks to complete.
* **Techniques**:
  + **Callbacks**: Execute a function after another finishes.
  + **Promises**: Represent eventual completion or failure of an asynchronous operation.
  + **Async/Await**: Syntax sugar over Promises for cleaner asynchronous code.
* Example:

javascript

Copy code

console.log("Start");

setTimeout(() => console.log("Async operation"), 1000);

console.log("End");

**3. Promises to Replace Callbacks**

* **Concept**: Provide a cleaner way to handle asynchronous operations without "callback hell."
* **States**:
  + **Pending**: Initial state.
  + **Fulfilled**: Operation completed successfully.
  + **Rejected**: Operation failed.
* Example:

javascript

Copy code

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => resolve("Data fetched"), 1000);

});

};

fetchData()

.then(data => console.log(data))

.catch(error => console.error(error));

**4. Async-Await to Replace Promises**

* **Concept**: Simplifies working with Promises by writing asynchronous code that looks synchronous.
* **Syntax**:
  + async keyword makes a function return a Promise.
  + await pauses execution until the Promise resolves.
* Example:

javascript

Copy code

const fetchData = async () => {

try {

const data = await new Promise(resolve =>

setTimeout(() => resolve("Data fetched"), 1000)

);

console.log(data);

} catch (error) {

console.error(error);

}

};

fetchData();

**5. The JavaScript Event Loop**

* **Concept**: Manages the execution of asynchronous code in JavaScript.
* **Key Points**:
  + JS is single-threaded but can handle asynchronous operations via the **Event Loop**.
  + Tasks are divided into:
    - **Call Stack**: Executes functions.
    - **Task Queue**: Stores callback functions from asynchronous tasks.
    - **Microtask Queue**: Stores resolved Promise callbacks (higher priority than Task Queue).
* **Video Summary**:
  + Demonstrates how asynchronous tasks are scheduled and executed.

**6. Callback Hell**

* **Concept**: Nested callbacks that make code difficult to read and maintain.
* **Problem**: Increasing levels of nesting lead to hard-to-debug code.
* **Solution**: Use Promises or async/await.
* Example of Callback Hell:

javascript

Copy code

getData1((data1) => {

getData2(data1, (data2) => {

getData3(data2, (data3) => {

console.log(data3);

});

});

});

* **Solution with Promises**:

javascript

Copy code

getData1()

.then(data1 => getData2(data1))

.then(data2 => getData3(data2))

.then(data3 => console.log(data3))

.catch(error => console.error(error));

6

**1. REST API**

* **Concept**: Representational State Transfer (REST) is an architectural style for designing networked applications.
* **Key Features**: Statelessness, Client-Server architecture, Cacheability, Uniform Interface.
* **Use Case**: Exposing services over HTTP for CRUD operations using standard methods like GET, POST, PUT, and DELETE.

**2. HTTP**

* **Concept**: Hypertext Transfer Protocol, the foundation of communication on the web.
* **Key Features**: Stateless protocol using request/response model.
* **Common Methods**: GET (retrieve data), POST (send data), PUT (update data), DELETE (remove data).

**3. HTTP Connection**

* **Persistent Connections**: Allow multiple requests and responses over a single TCP connection.
* **Session Management**: Includes mechanisms like cookies, tokens, and session storage for maintaining user state.

**4. HTTP Messages**

* **Request Message**: Contains method, URL, headers, and optional body (for POST, PUT).
* **Response Message**: Contains status code, headers, and optional body (e.g., HTML, JSON).
* **Status Codes**:
  + 200: OK
  + 404: Not Found
  + 500: Internal Server Error

**5. Ajax (Asynchronous JavaScript and XML)**

* **Concept**: Technique for updating parts of a webpage without reloading the entire page.
* **Key Methods**:
  + XMLHttpRequest or modern Fetch API.
* Example:

javascript

Copy code

const xhr = new XMLHttpRequest();

xhr.open("GET", "https://api.example.com/data");

xhr.onload = () => console.log(xhr.responseText);

xhr.send();

**6. Fetch API**

* **Concept**: Modern replacement for XMLHttpRequest.
* **Features**: Promise-based, easier syntax, better readability.
* Example:

javascript

Copy code

fetch("https://api.example.com/data")

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error("Error:", error));

**7. Using Fetch API**

* **Advanced Usage**:
  + Custom headers using headers property.
  + Configuring HTTP methods using method.
  + Handling errors with .catch().

**8. Promise**

* **Concept**: Object representing the eventual completion or failure of an asynchronous operation.
* **States**: Pending, Fulfilled, Rejected.
* Example:

javascript

Copy code

const promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("Success"), 1000);

});

promise.then(data => console.log(data));

**9. Using Promises**

* **Chain Promises**:

javascript

Copy code

fetch("https://api.example.com/data")

.then(response => response.json())

.then(data => process(data))

.catch(error => console.error("Error:", error));

**10. JSON Syntax**

* **Key Points**:
  + Data is represented as key-value pairs.
  + Values can be strings, numbers, arrays, objects, or null.
  + Example:

json

Copy code

{

"name": "John",

"age": 30,

"hobbies": ["reading", "gaming"]

}

**11. XMLHttpRequest API**

* **Concept**: Legacy API for making HTTP requests.
* **Key Features**:
  + Allows handling of response statuses, progress, and headers.
* Example:

javascript

Copy code

const xhr = new XMLHttpRequest();

xhr.open("GET", "https://api.example.com/data");

xhr.onreadystatechange = () => {

if (xhr.readyState === 4 && xhr.status === 200) {

console.log(xhr.responseText);

}

};

xhr.send();

**12. Using XMLHttpRequest**

* **Steps**:
  1. Create an XMLHttpRequest object.
  2. Open a connection (GET or POST).
  3. Set request headers (if necessary).
  4. Send the request.

**13. Axios to Replace Fetch API**

* **Concept**: A Promise-based library for making HTTP requests, with additional features like interceptors and default settings.
* **Example**:

javascript

Copy code

axios.get("https://api.example.com/data")

.then(response => console.log(response.data))

.catch(error => console.error(error));

**14. REST vs SOAP**

* **REST**: Lightweight, uses JSON, stateless.
* **SOAP**: More secure, uses XML, stateful.
* **Comparison**:
  + REST is simpler and faster.
  + SOAP is better for complex, transactional scenarios.

**15. JSON Datatypes**

* **Supported Types**: String, Number, Boolean, Array, Object, null.
* **Example**:

json

Copy code

{

"name": "Alice",

"age": 25,

"isStudent": false,

"courses": ["Math", "Physics"]

}

**16. Converting JSON to JS Object**

* **Using JSON.parse()**:

javascript

Copy code

const jsonData = '{"name": "Alice", "age": 25}';

const obj = JSON.parse(jsonData);

console.log(obj.name); // Alice

**17. Converting JS Object to JSON**

* **Using JSON.stringify()**:

javascript

Copy code

const obj = { name: "Alice", age: 25 };

const jsonData = JSON.stringify(obj);

console.log(jsonData); // '{"name":"Alice","age":25}'