React.js Tutorial for Absolute Beginners

Complete Course Notes — Components, State, Props, Hooks & Building a Real App

SECTION 1: Introduction to React.js

1.1 What is React?

React.js is the world's most popular front-end JavaScript library, developed by Facebook and maintained by Facebook and the open-source community. It is used to build fast, interactive, and dynamic user interfaces.

Feature	Detail
Created by	Facebook (Meta) — released 2013, open-source maintained
Type	Front-end JavaScript library (not a full framework)
Main purpose	Building user interfaces, especially Single Page Applications (SPAs)
Main competitors	Vue.js and Angular — but React leads by a large margin on Google Trends
Platform support	Web apps (React) and native mobile apps (React Native)
Prerequisite	JavaScript — the only prerequisite before learning React

1.2 Why Use React?

- Pages update instantly without a full browser reload — no requesting multiple HTML pages
- Excellent cross-platform support — one skill set covers web and mobile development
- Huge and active community — thousands of libraries, tutorials, and jobs available
- Component-based architecture makes code reusable and easy to maintain
- Used by Facebook, Instagram, Airbnb, Netflix, and hundreds of thousands of developers worldwide

1.3 The Virtual DOM — How React Updates the Page

The Document Object Model (DOM) is the browser's internal representation of a web page. Directly manipulating the real DOM is slow. React solves this with a Virtual DOM.

Concept	Explanation
Real DOM	The actual browser representation of the page. Updating it is slow because the browser must repaint and reflow the entire page.
Virtual DOM	A lightweight JavaScript object — a fast in-memory copy of the real DOM. React performs all calculations here first.
Diffing	When something changes, React compares the new Virtual DOM against the previous one to find exactly what changed.
Selective update	React only updates the specific part of the real DOM that changed — not the entire page. This is extremely efficient.
Key benefit	Developers don't need to manually interact with the DOM API — React handles all updates automatically.

SECTION 2: Components — The Building Blocks of React

2.1 What is a Component?

React is a component-based library. Every part of a user interface is broken into small, independent, reusable components. An entire React application is a tree of components nested inside each other.

- A component is a small, self-contained piece of the user interface
- Example: a movie website might have a Sidebar component, a Search Bar component, and a Movies component made up of individual Movie Card components
- Components can be reused — define once, use anywhere
- Components can accept dynamic data through props and manage their own data through state

2.2 Class-Based vs Functional Components

Type	Status
Class-based components	Legacy — no longer used in modern React
Functional components	Modern standard — use these always

A minimal functional component:

```
import React from 'react';

const MyComponent = () => {
  return <h1>Hello React</h1>;
};

export default MyComponent;
```

- Every component is a JavaScript function that returns JSX
- Arrow functions are the most common way to define functional components
- Every component must be exported (export default) so other files can import and use it
- In newer versions of React you can skip importing React at the top — it is no longer required

2.3 File and Folder Structure

File / Folder	Purpose
package.json	Lists all dependencies (react, react-dom, react-scripts) and npm scripts
node_modules/	Where all installed packages live — never edit manually, very large
public/	Contains one single index.html file — the only HTML file in the entire app
public/index.html	Has a single — React injects the entire app into this div
src/	Where all your React code lives — you spend almost all your time here
src/index.js	The entry point — calls ReactDOM.render() to mount the app into the root div
src/App.js	The root component — the top of the component tree that all others nest inside

2.4 How React Mounts into the Browser

```
// src/index.js — the entry point of every React application
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

- ReactDOM.render() is called only ONCE in the entire application regardless of its size
- It targets the in index.html and injects the entire React app inside it
- Every component you build eventually lives inside the App component

SECTION 3: JSX — JavaScript XML

3.1 What is JSX?

JSX (JavaScript XML) is the special syntax used in React to describe what the user interface should look like. It looks like HTML written inside a JavaScript file, but it is not HTML — it is a JavaScript extension that gets compiled to JavaScript.

- JSX files use the .js or .jsx extension (both work identically — .jsx is preferred for components)
- JSX produces React elements that React renders to the DOM
- JSX is the core syntax of React — understanding it is essential

3.2 JSX vs HTML — Key Differences

HTML	JSX Equivalent
class="app"	className="app"
for="email"	htmlFor="email"
<img> (self-closing)	<img /> (must close)
<!-- comment -->	{/* comment */}
Inline styles as string	Inline styles as object

3.3 Embedding JavaScript Inside JSX

Curly braces { } in JSX let you embed any valid JavaScript expression directly inside the markup. This is what makes JSX so powerful — it combines the structure of HTML with the full power of JavaScript.

```
const name = 'John';
const isLoggedIn = true;

return (
  <div>
    <h1>Hello {name}</h1>              {/* variable */}
    <p>Result: {2 + 2}</p>             {/* expression */}
    <p>{isLoggedIn ? 'Welcome' : 'Please log in'}</p>  {/* ternary */}
  </div>
);
```

- Variables, arithmetic expressions, ternary operators, and function calls all work inside { }
- You cannot use if/else statements directly inside JSX — use ternary expressions instead
- You cannot use for loops directly inside JSX — use array .map() instead

3.4 React Fragments

React requires that a component returns exactly one root element. If you need to return multiple sibling elements, wrap them in a React Fragment — an invisible wrapper that adds no extra DOM node.

```
// Error — two siblings with no wrapper:
return (
  <h1>Title</h1>
  <p>Paragraph</p>
);

// Correct — wrapped in a Fragment:
return (
  <>
    <h1>Title</h1>
    <p>Paragraph</p>
  </>
);
```

- <></> is shorthand for <React.Fragment></React.Fragment>
- Fragments are invisible in the DOM — they do not add an extra div or span
- The error message 'JSX elements must be wrapped in an enclosing tag' means you need a Fragment

SECTION 4: Setting Up a React Application

4.1 Prerequisites

- Node.js must be installed — download from nodejs.org (it is a JavaScript runtime that runs JS outside the browser)
- Node.js also installs npm (Node Package Manager) — used to install React packages
- A code editor — Visual Studio Code is strongly recommended

4.2 Create React App

Create React App (CRA) is the official tool supported by the React team for bootstrapping a new React project. It generates all required files and configuration automatically.

```
# Create a new React app in the current directory
npx create-react-app .

# Or create it in a new named folder
npx create-react-app my-app
```

- npx runs the command without permanently installing it — always uses the latest version
- This installs three core packages: react, react-dom, and react-scripts
- The process takes about 1–2 minutes depending on your internet connection
- When finished, you will see a 'Success!' message

4.3 Running and Stopping the App

Command	What It Does
npm start	Starts the development server at http://localhost:3000 — hot reloads on every file save
Ctrl + C then Y	Stops the development server
npm start (again)	Restarts the server — useful if the app gets into a broken state

4.4 Cleaning Up for a Fresh Start

After CRA creates the app, delete the generated src/ folder and recreate it with just two files:

```
// src/index.js — minimum entry point
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));

// src/App.js — minimum root component
import React from 'react';

const App = () => {
  return <h1>App</h1>;
};

export default App;
```

SECTION 5: Props — Passing Data Between Components

5.1 What are Props?

Props (short for properties) are how you pass dynamic data from a parent component into a child component. They work exactly like arguments passed to a function.

- Props are read-only — a child component cannot modify its own props
- Props are passed as attributes on the component tag, just like HTML attributes
- Any type of data can be passed: strings, numbers, booleans, arrays, objects, functions

5.2 Passing and Receiving Props

```
// Parent component — passing props
const App = () => {
  return (
    <>
      <Person name="John" lastName="Doe" age={25} />
      <Person name="Jane" lastName="Smith" age={30} />
    </>
  );
};

// Child component — receiving props
const Person = (props) => {
  return (
    <>
      <h1>{props.name}</h1>
      <h2>{props.lastName}</h2>
      <h2>{props.age}</h2>
    </>
  );
};
```

- String props can be passed without curly braces: name="John"
- Dynamic values (numbers, variables, expressions) require curly braces: age={25} or age={2+2}

5.3 Destructuring Props

Instead of writing props.name every time, you can de structure props directly in the function parameter — cleaner and less repetitive:

```
// Without destructuring:
const Person = (props) => <h1>{props.name}</h1>;

// With destructuring (recommended):
const Person = ({ name, lastName, age }) => (
  <>
    <h1>{name}</h1>
    <h2>{lastName}</h2>
    <p>{age}</p>
  </>
);
```

5.4 Why Components and Props Are Powerful

- Without components: if you need 100 movie cards, you must copy/paste the JSX 100 times
- With components: define the Movie Card once, render it 100 times in a single .map() call
- Each instance receives different data through props — same structure, different content
- If the component design changes, you update it in one place and all instances update automatically

SECTION 6: State and the useState Hook

6.1 What is State?

State is a plain JavaScript object that stores dynamic data belonging to a component. When state changes, React automatically re-renders the component to reflect the new data — without a page reload.

- State is managed entirely by the component itself (unlike props, which come from outside)
- State is the mechanism that makes React applications interactive and reactive
- When state updates, only the affected component (and its children) re-render — not the entire page

6.2 The useState Hook

useState is a React Hook that lets functional components create and manage state.

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

const Counter = () => {
  const [counter, setCounter] = useState(0);  // initial value = 0

  return (
    <>
      <button onClick={() => setCounter(prev => prev - 1)}>-</button>
      <h1>{counter}</h1>
      <button onClick={() => setCounter(prev => prev + 1)}>+</button>
    </>
  );
};
```

useState Part	What It Is
useState(0)	The Hook call — the argument is the initial value of the state
counter	The state variable — holds the current value. Use this in JSX to display data.
setCounter	The setter function — the ONLY correct way to update state. Naming convention: set + StateName.
[counter, setCounter]	Array destructuring — useState returns an array of exactly two items: [value, setter]
prev => prev - 1	Using the previous state value inside the setter — safe and recommended practice

6.3 The Golden Rule — Never Mutate State Directly

This is the most important rule in React. State variables are not regular JavaScript variables. They must only be updated through their setter function.

```
// WRONG — never do this:
counter = counter + 1;      // React won't detect this change — no re-render
counter = 100;              // Breaks React's update cycle

// CORRECT — always use the setter:
setCounter(100);
setCounter(prev => prev + 1);
```

- Direct mutation breaks React's ability to track changes and update the DOM
- Always use the setter function — React will then automatically re-render the component
- You can have as many state variables per component as you need — just call useState multiple times

6.4 Events in React

Events are actions triggered by user interaction (clicks, typing, etc.). React handles events using camelCase attributes directly on JSX elements.

Event	JSX Syntax
Click	onClick={handler}
Input change	onChange={handler}
Form submit	onSubmit={handler}
Mouse enter	onMouseEnter={handler}
Key press	onKeyDown={handler}

- Event handlers receive an event object (e) as a parameter — use e.target.value to get an input's current value

```
<input onChange={(e) => setSearchTerm(e.target.value)} value={searchTerm} />
```

SECTION 7: The useEffect Hook

7.1 What is useEffect?

useEffect is the second most important React hook. It lets you run code in response to something happening — like the component loading for the first time, or a state variable changing.

- Common uses: fetching data from an API when the page loads, setting up event listeners, updating the document title
- useEffect runs after the component renders, not before
- You can have multiple useEffect hooks in one component — no limit

7.2 useEffect Syntax

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

useEffect(() => {
  // code to run
}, [/* dependency array */]);
```

7.3 The Dependency Array — Controlling When useEffect Runs

Dependency Array	When the Effect Runs	Common Use Case
No array (omitted)	After every single render — including every state update	Rarely used — usually causes infinite loops
Empty array [ ]	Only once, when the component first mounts (loads)	Fetching initial data from an API on page load
[variable]	Once on mount AND every time that variable changes	Re-fetching data when a search term or filter changes
[var1, var2]	Once on mount AND when either variable changes	Reacting to multiple dependencies simultaneously

7.4 Practical useEffect Examples

```
// Run once on load — fetch movies when the page opens
useEffect(() => {
  searchMovies('Spider-Man');
}, []);

// Run every time counter changes
useEffect(() => {
  alert(`Counter changed to ${counter}`);
}, [counter]);
```

- To set state inside useEffect on load: call the setter function — do NOT assign directly

```
// CORRECT inside useEffect:
useEffect(() => {
  setCounter(100);  // fine — runs once, sets counter to 100
}, []);

// DANGEROUS — causes infinite loop:
useEffect(() => {
  setCounter(100);  // setCounter triggers re-render → re-runs effect → infinite
loop
}, [counter]);
```

7.5 async/await Inside useEffect

useEffect's callback cannot be async directly. To use async/await (for API calls), define an async function inside the effect and call it immediately:

```
useEffect(() => {
  const fetchData = async () => {
    const response = await fetch(url);
    const data = await response.json();
    setMovies(data.Search);
  };

  fetchData();   // call the async function
}, []);
```

SECTION 8: Hooks Overview

8.1 What are Hooks?

Hooks are special functions provided by React that let functional components access React features like state, lifecycle methods, and context. Every hook name begins with 'use'.

- Hooks were introduced in React 16.8 and replaced the need for class-based components entirely
- You can only call hooks at the top level of a functional component — not inside loops, conditions, or nested functions
- Custom hooks can be created by combining existing hooks

8.2 Core Hooks Reference

Hook	Purpose
useState	Create and manage local state in a component. Returns [value, setter].
useEffect	Run side effects (API calls, subscriptions, timers) after render. Controlled by dependency array.
useContext	Access data from a React Context — avoids prop drilling through many component levels.
useRef	Hold a mutable reference to a DOM element or a value that persists between renders without causing re-renders.
useMemo	Memoize an expensive computed value — only recalculates when dependencies change.
useCallback	Memoize a function reference — prevents unnecessary re-creation on every render.
useReducer	Alternative to useState for complex state logic — works like a Redux-style reducer.

8.3 The Three Most Important React Concepts

Concept	What It Does
Components	Divide the UI into small, independent, reusable pieces. Build once, use anywhere.
Props	Pass dynamic data from parent to child components. Like function arguments — read-only.
State	Store and manage data that belongs to a component. When it changes, the component re-renders.

SECTION 9: Building the MovieLand App — Step by Step

9.1 Project Overview

MovieLand is a React application that searches for movies using the OMDB API and displays the results as cards. It demonstrates state, props, useEffect, API fetching, component extraction, and dynamic list rendering.

Feature	React Concept Used
Search input controlled by typing	useState (searchTerm)
Movies array populated from API	useState (movies), useEffect, fetch/async-await
Search triggered by button click	onClick event handler
List of movie cards rendered	Array .map() over movies state
Each card receives different movie data	Props passed from App to MovieCard
Placeholder image when poster missing	Ternary expression in JSX
No-results message	Conditional rendering with ternary in JSX

9.2 Setting Up the API

- Register at omdbapi.com to get a free API key
- The base API URL pattern: http://www.omdbapi.com/?apikey=YOUR_KEY&s=SEARCH_TERM

```
const API_URL = 'http://www.omdbapi.com/?apikey=YOUR_API_KEY';
```

- The API returns a JSON object — the movies array is found at data.Search
- The fetch response must be converted to JSON using response.json() before use

9.3 Fetching Movies with useEffect

```
const [movies, setMovies] = useState([]);

useEffect(() => {
  searchMovies('Spider-Man');  // load default movies on startup
}, []);

const searchMovies = async (title) => {
  const response = await fetch(`${API_URL}&s=${title}`);
  const data = await response.json();
  setMovies(data.Search);
};
```

9.4 Extracting the MovieCard Component

Instead of repeating the movie card JSX for every movie, extract it into its own component file:

```
// src/MovieCard.jsx
import React from 'react';

const MovieCard = ({ movie }) => {
  return (
    <div className="movie">
      <div><p>{movie.Year}</p></div>
      <div>
        <img
          src={movie.Poster !== 'N/A' ? movie.Poster :
'https://via.placeholder.com/400'}
          alt={movie.Title}
        />
      </div>
      <div>
        <span>{movie.Type}</span>
        <h3>{movie.Title}</h3>
      </div>
    </div>
  );
};

export default MovieCard;
```

9.5 Rendering the Movie List and Handling Empty State

```
// Inside App.jsx return:
{movies?.length > 0 ? (
  movies.map((movie) => (
    <MovieCard key={movie.imdbID} movie={movie} />
  ))
) : (
  <div className="empty">
    <h2>No movies found</h2>
  </div>
)}
```

- movies?.length — the optional chaining operator (?.) prevents errors if movies is undefined
- The key prop on mapped elements must be a unique identifier — use the movie's imdbID
- The key prop helps React efficiently track which items changed, were added, or were removed

9.6 Controlled Input for Search

```
const [searchTerm, setSearchTerm] = useState('');

<input
  placeholder="Search for movies"
  value={searchTerm}
  onChange={(e) => setSearchTerm(e.target.value)}
/>
<img src={searchIcon} alt="search" onClick={() => searchMovies(searchTerm)} />
```

- A controlled input has its value linked to state — React is the single source of truth
- onChange fires on every keystroke and updates the searchTerm state
- Clicking the search icon calls searchMovies(searchTerm) with the current state value
- This pattern — state-controlled input feeding into an API call — is one of the most common React patterns

9.7 Complete App.js Structure

Code Block	Purpose
const API_URL = '...'	Store the base API URL as a constant outside the component
const [movies, setMovies] = useState([])	State to hold the fetched movies array
const [searchTerm, setSearchTerm] = useState('')	State to hold the current search input value
useEffect(() => { searchMovies(...) }, [])	Fetch default movies when the app first loads
const searchMovies = async (title) => {...}	Async function to call the API and update movies state
<input value={searchTerm} onChange={...} />	Controlled input that tracks what the user types
<img onClick={() => searchMovies(searchTerm)} />	Search button that triggers the API call
movies.map(movie => <MovieCard movie={movie} />)	Render a MovieCard for each movie in the array

SECTION 10: Key React Patterns and Best Practices

10.1 Component Naming Conventions

- Component names must start with a capital letter — React uses this to distinguish components from HTML tags
- Use PascalCase: MovieCard, SearchBar, UserProfile (not movieCard or movie-card)
- File names should match the component name: MovieCard.jsx exports MovieCard
- Use .jsx extension for component files — adds the React logo in VS Code and signals intent

10.2 Import and Export Rules

Pattern	Syntax
Default export	export default MyComponent
Default import	import MyComponent from './MyComponent'
Named export	export const helper = () => {}
Named import	import { helper } from './utils'
Import CSS	import './App.css'
Import SVG	import searchIcon from './search.svg'

10.3 Conditional Rendering Patterns

Pattern	When to Use
Ternary expression	Two alternatives — show A or B
Short-circuit &&	Show or hide one thing
Early return	Render nothing or loading state before data arrives
Optional chaining ?.	Safely access nested properties that might be undefined

10.4 Working with Lists — The .map() Pattern

Any time you need to render a list of items in React, use the .map() array method to transform an array of data into an array of JSX elements:

```
// Map over an array and return a component for each item
movies.map((movie) => (
  <MovieCard key={movie.imdbID} movie={movie} />
))
```

- Always provide a key prop — a unique identifier for each item in the list
- Never use the array index as a key if the list can be reordered — use a stable unique ID
- The key prop is not passed to the child component — it is used internally by React only

10.5 React Developer Workflow Summary

Step	Action
1. Plan UI	Sketch the component tree — identify which parts should be separate components
2. Build static UI	Write the JSX structure without any state or interactivity first
3. Add state	Identify what data changes over time — create useState for each piece
4. Handle events	Add onClick, onChange handlers that call setters to update state
5. Add effects	Use useEffect for API calls, subscriptions, or any side effects
6. Extract components	Once a piece of JSX is reused or complex, move it to its own file
7. Pass props	Connect parent and child components with props to share data

— End of Course Notes —