Introduction to React

Objectives

- Describe what React is and what problem it solves
- Can explain the basic React component architecture
- Build and connect React components that correctly utilize React's Application Programming Interface (API)
- Learn React.js best practices

What Is React.js?

React.js is a flexible JavaScript library developed by Facebook for building interactive user interfaces. By focusing purely on the “view” layer (in the MVC sense), React lets you create encapsulated UI components that handle their own state and logic, making it easy to compose complex interfaces from smaller, reusable building blocks. With React, every piece of your UI is treated as a component, drastically simplifying debugging and refactoring as your application grows.

Unlike more monolithic frameworks, React is primarily concerned with rendering data to the DOM and handling updates in an efficient manner via its virtual DOM system. This approach provides performance benefits and a declarative style of coding: rather than directly manipulating the DOM, you describe how the UI should look for a given state, and React figures out the minimal set of changes needed to update the UI.

Resources:

https://react.dev/

https://react.dev/learn

React API

React’s Application Programming Interface (API) encompasses all the methods, properties, and structures that React provides for building user interfaces. At its core, this includes the ability to create and manage components—either as class-based components with lifecycle methods or, more commonly nowadays, as functional components with Hooks (like useState, useEffect, and others). When React renders these components, it uses a virtual DOM (Document Object Model) to efficiently

compare new UI states with previous ones, then applies only the minimal necessary changes to the actual browser DOM.

Beyond components and Hooks, React's API also features utilities for special tasks like context management (createContext), memoization (React.memo), and concurrency features (startTransition and useTransition) that enable smoother rendering under heavy loads. It includes elements for structuring your application (React.Fragment) and debugging or development mode aids (React.StrictMode). Additionally, React DOM (often imported separately as react-dom) offers methods like createRoot to start rendering your components in the browser and specialized APIs for server rendering. Taken together, these capabilities form React's core toolkit for constructing, updating, and reasoning about interactive, component-based interfaces.

Resources:

https://react.dev/reference/react

What Problem Does It Solve?

Modern web apps have become increasingly dynamic, requiring frequent data updates and user interactions. Directly managing the browser DOM can lead to messy code, performance bottlenecks, and complicated state management. React solves these problems by using a virtual DOM that updates the interface only where necessary, and by offering a straightforward, component-based architecture that keeps application logic organized.

In practice, this means you no longer need to write complex event listeners or DOM queries. Instead, you use React's declarative style: define what your components should look like given certain data, and React automatically re-renders them whenever that data changes. This results in UIs that are easier to maintain, debug, and scale over time.

Development Eco-System

React has a vibrant ecosystem of supporting tools. For instance, you can quickly scaffold React apps using Vite, which offers fast bundling and a dev server with hot module replacement. You'll also find libraries dedicated to routing (React Router), server-side rendering (Next.js), state management (Redux, MobX), and UI components (Material UI, Chakra UI)—all of which can be mixed and matched as needed.

This modularity lets you start simple with React's core features and then introduce additional libraries only when they become relevant for your project. Whether you need an advanced state manager, complex routing setup, or a slick UI library, there's a well-supported solution in the React ecosystem.

Resources:

https://code.visualstudio.com/

https://nextjs.org/

React versus Other Frameworks

Compared to Angular and Vue, React deliberately remains minimal by focusing on just rendering UIs with components. This gives you the freedom to select the additional tools and patterns that best fit your project's requirements. If you prefer a flexible approach that doesn't lock you into a specific architecture, React's lightweight core can be a perfect match.

Additionally, React's unidirectional dataflow is often simpler to understand than two-way binding. By funneling data in a single direction, it becomes easier to track and debug, making React popular for teams prioritizing both performance and code clarity.

Resources:

https://vuejs.org/

https://angular.dev/

Key React Components

Editor and Web Server

For a smooth React workflow, you'll often use a local development server (e.g., via Vite) that bundles your files, handles hot reloading, and serves your app in the browser. This setup makes for a rapid feedback loop as you edit your code.

A modern code editor such as Visual Studio Code further enhances development with IntelliSense, auto-formatting, Git integration, and an integrated terminal. Together, these tools reduce context switching and help you build applications faster.

Resources:

https://vite.dev/

Browser Development Tools

React debugging is greatly simplified by the React DevTools extension, which you can add to Chrome or Firefox. It provides a visual tree of your React components, letting you inspect props, state, and performance metrics.

Meanwhile, standard browser DevTools (Elements, Network, Console, etc.) still apply. Together, these tools help you pinpoint unnecessary re-renders, track data flow, and debug layout issues—vital skills for maintaining efficient, large-scale React apps.

Resources:

https://chromewebstore.google.com/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi

Components

Components are the fundamental building blocks of a React application. Each component is responsible for rendering a specific part of the UI, which keeps your codebase modular and organized. You can define components as functions (with Hooks) or classes (legacy approach).

By combining multiple components, you create complex UIs out of smaller, reusable pieces. This compositional model not only promotes code reuse but also makes it easier to reason about each segment of your UI in isolation.

Resources:
https://react.dev/learn/your-first-component

Virtual DOM

The virtual DOM is an internal data structure React uses to optimize updates. Whenever your app's state changes, React creates a new virtual DOM tree and compares it against the previous one. Only the differences—determined through a process called reconciliation—are then applied to the browser's actual DOM, cutting down on expensive direct manipulations.

This approach ensures that your UI re-renders efficiently, even as your application grows more complex. Developers don't have to worry about manually updating DOM elements or writing large amounts of boilerplate—React quietly handles those updates behind the scenes, letting you focus on the overall logic and design of your app.

Resources:
https://legacy.reactjs.org/docs/faq-internals.html
https://legacy.reactjs.org/docs/reconciliation.html

Dataflow

React employs a one-way (unidirectional) dataflow, where parent components pass data down to children via props. If a child component needs to update the parent, it typically does so using callback functions or external state managers like Redux.

This predictable flow simplifies debugging and reasoning about your app. You always know where data originates (the top) and how it propagates (downward), helping keep each component more self-contained and maintainable.

JSX

JSX is a syntax extension for JavaScript that lets you write HTML-like structures inside your scripts. This tight integration of markup and logic keeps your component code more intuitive and readable. Tools like Babel transpile JSX into standard JavaScript calls so that browsers can render it seamlessly.

Resources:

https://react.dev/learn/javascript-in-jsx-with-curly-braces

https://vite.dev/guide/why

Your First React UI

Understand the Components

When you begin building a new React UI, start by identifying which elements naturally group together into distinct components. Each component has its own render logic and can have its own internal state or rely on props passed down from a parent.

By nesting child components within parent components, you form a hierarchy that maps closely to the structure of your UI. This architecture also fosters reusability, since any component can be dropped into different parts of your app with minimal changes.

Build a Hello World Component

A simple “Hello World” example demonstrates how React and JSX work together. Typically, you’ll write a functional component—like function HelloWorld() { return <h1>Hello World</h1>; }—and import that component into a main file (e.g., App.jsx) to render.

When this renders in your Vite environment, it showcases how React efficiently manages updates. Although trivial, this initial step lays the foundation for building more dynamic interfaces, letting you see firsthand how changes in data automatically re-render your UI.

JSX

What Is JSX?

JSX stands for JavaScript XML, blending HTML-like syntax directly within your JavaScript. It allows you to embed expressions (e.g., {user.name}) and conditionals ({isLoggedIn ? <Dashboard /> : <Login />}) as part of your layout, making the code expressive and compact. Under the hood, JSX compiles into plain JavaScript function calls, leveraging React.createElement to construct DOM nodes.

This seamless blending of layout and logic is one of the reasons React quickly became popular. Instead of juggling separate files for templates and scripts, you declare the desired UI structure and the associated interaction code together in one place. JSX also helps React's one-way dataflow feel more natural, as data can flow in from props and state without leaving the scope of your component's logic.

How to use JSX

Using JSX means you can write expressions such as Hello, {name} instead of dealing with verbose React.createElement() calls. This significantly reduces boilerplate and keeps the DOM representation close to the underlying logic.

Most React setups rely on a build tool like Vite or a bundler with Babel to automatically transpile your JSX to browser-compatible JavaScript. Consequently, you can stick to writing modern JavaScript without manually worrying about older syntax or the intricacies of DOM manipulation.

Utilize React with JSX

When paired with React's component model, JSX makes it straightforward to convey how each part of your UI should respond to data changes. If your component receives new props or the state changes, React triggers a re-render of the relevant JSX, ensuring the visible UI stays current.

This unified approach to structure and logic also aids debugging. You only need to look in one place—the component's JSX definition—to see how UI elements and data flows tie together. It's a key advantage over the template-and-script separation in some frameworks.

Resources:

https://react.dev/learn/writing-markup-with-jsx

Use React without JSX

Although possible, using React without JSX can quickly become cumbersome. You'd have to write multiple React.createElement() calls for every piece of your interface, nesting them in a way that mirrors a tree-like structure. This is both less readable and more error-prone.

Still, certain use cases or build constraints might require skipping JSX. In those scenarios, the React API works just fine in raw form, though you lose the syntactic elegance that JSX provides. Most modern React projects, however, use JSX by default.

Resources:

https://react.dev/reference/react/createElement#creating-an-element-without-jsx

Precompiled JSX with Babel

Babel is a popular transpiler that converts next-generation JavaScript features, including JSX syntax, into widely supported JavaScript. When you use Vite or similar tools, the transpiler runs automatically, so you can write modern code (with JSX) without worrying about browser compatibility.

This preprocessing step allows React to function seamlessly in all major browsers. It also means that any time you add new language features (like optional chaining or class fields), Babel can handle them, keeping your workflow smooth and cutting-edge.

Working with Components

Component Life-Cycle

Class components(Legacy) rely on lifecycle methods—such as componentDidMount or componentWillUnmount—to manage logic when a component enters or leaves the DOM. In functional components, you achieve similar results with Hooks, primarily useEffect, which triggers side effects based on component mount, unmount, or prop changes.

This lifecycle logic is essential for tasks like fetching data, setting up event listeners, or cleaning up resources. By placing these effects within clearly defined methods or Hooks, you maintain clean, predictable control over when and how side effects run.

Properties and State

React divides data into props and state. Props (short for properties) are passed into a component from its parent, remaining read-only within that component. They allow you to configure or parameterize your component from the outside—like passing in a “title” or “onClick” handler.

State, by contrast, represents data within the component itself—something that can change over time. Calling a state setter (e.g., setCounter) triggers a re-render, ensuring the UI keeps up with updates in real time. Understanding this distinction between props and state is central to mastering how React apps synchronize data and visuals.

Virtual DOM

Although discussed earlier, it bears repeating that the virtual DOM underpins React’s efficient updates. When props or state change, React re-renders the component into a virtual DOM and compares it to the previous version.

By applying only the minimal changes to the actual DOM, React avoids expensive updates and keeps your interface responsive. This is a key reason why React-based apps can scale to handle intricate UIs without massive performance penalties.

Events

React normalizes events across browsers, letting you attach listeners in a consistent way. For example, <button onClick={handleClick}> will trigger handleClick in all supported browsers without any extra polyfills.

This event system also keeps concerns localized. Instead of adding global listeners or multiple queries to attach events, you define them in the component's JSX. This keeps all UI-related code in one place, maintaining clarity as your project grows.

Compositions

Composition in React refers to creating more complex UIs by nesting or embedding components inside each other. This can be as simple as placing a <Header /> component in your main App layout or as sophisticated as building higher-order components.

This approach aligns nicely with React's philosophy of breaking an app into small, focused parts. Each component fulfills a unique role, and you can flexibly combine them without repeating yourself. Composition thus fosters reusable patterns and clearer data flows.

Reusable Components

When each component does one thing well, you can easily reuse it across different parts of your application. Common UI elements—like buttons, input fields, or layout wrappers—become building blocks that speed development and maintain design consistency.

Moreover, by separating a component's logic from its presentation, you can adapt its appearance or behavior with minimal code changes. This modular approach cuts down on duplicated logic and simplifies long-term maintenance in evolving applications.

Forms

Controlled Components

In a controlled component, form fields (e.g., <input>, <textarea>) are “controlled” by React’s state. The value displayed is always derived from this.state or a Hook like useState, and on every keystroke, React updates that state. This gives you precise control over input changes and makes real-time validation or dynamic enabling/disabling of fields straightforward.

Because React controls the input values, you also get an easy way to reset the form or apply transformations to user inputs before they appear in the UI. While this pattern can be slightly more verbose, it delivers powerful benefits in larger apps where tracking and validating user data is essential.

Resources:

https://react.dev/reference/react-dom/components/form

Uncontrolled Components

Uncontrolled components let the DOM itself manage form values, using a ref to access the input’s current value when necessary. This approach can simplify code for smaller forms, where constant synchronization between React state and input fields may not be essential.

However, this also means you have less immediate visibility into the data as a user types. Real-time validation or complex interactions become trickier, as the logic resides partly in the DOM, not just in your React components. Use uncontrolled forms when you want minimal overhead and fewer updates.

Resources:

https://legacy.reactjs.org/docs/uncontrolled-components.html

React Best Practices

Writing maintainable React code typically means embracing functional components with Hooks, keeping components small, and following a clear file structure. If an app grows complex, you might “lift state up” to parent components or use global stores (e.g., Redux, Recoil) to manage data shared across multiple parts of the UI.

Additionally, integrating tools like linters, TypeScript, or robust testing frameworks (Jest, React Testing Library) further streamlines development. By consistently applying patterns and guidelines—such as using memoization only where needed and avoiding large, deeply nested component trees—you ensure your React applications remain both performant and easy to extend.

Resources:

https://react.dev/learn/thinking-in-react

Using React with TypeScript

Using TypeScript with React offers strong typing for props, state, and component behavior, reducing runtime errors by catching issues at compile time. You can define component interfaces that describe exactly what shape your props should have and rely on TypeScript’s compiler to enforce these constraints across the project.

Setting up a React + TypeScript project is straightforward with tools like Vite, which provides a template for React TypeScript apps out of the box. As you build, TypeScript’s type definitions for React (maintained under the DefinitelyTyped organization) provide intellisense and comprehensive autocomplete, making your development workflow more reliable and productive.

React 18 vs React 19

React 18 introduced significant under-the-hood improvements such as concurrent rendering features (e.g., the startTransition API), automatic batching, and a revamped Suspense mechanism for data fetching. These changes laid the groundwork for more responsive apps, especially under heavy rendering loads, and made server-side rendering (via frameworks like Next.js) more efficient.

With the stable release of React 19 (announced on December 5, 2024), these concurrency features have been refined further, offering more predictable scheduling of updates and smoother transitions. Additionally, React 19 brings enhancements to Suspense for data fetching, incremental rendering improvements, and various performance optimizations—continuing the forward momentum from React 18 while maintaining backward compatibility for most applications.

Resources:

https://react.dev/blog/2024/12/05/react-19

Code Alongs

Hello World

https://vite.dev/guide/

1. Review Vite generator: Create a new React project (npm create vite@latest my-app –template react).
2. Starting the server: Run npm run dev and verify in the browser.
3. Understand the directory system: Examine main.jsx/App.jsx, public folder, and component organization.
4. First component: Build a basic “Hello World” and ensure React is working.
5. Multi-component (in-file): Show multiple component definitions in one file.
6. With vs without JSX: Compare JSX usage to React.createElement for clarity.
7. Basic state/props usage: Introduce useState and passing props.
8. React DevTools: Install the extension, inspect components, props, and state.
9. Build/Deploy: Generate a production build (npm run build) and discuss deployment options.

Vehicle/Fleet CRUD

1. Overview of backend service with fleet data: Explain endpoints or data structure for vehicles.
2. Local JSON data: Show how to serve and load local JSON data in dev.
3. State management best practices (do not mutate state directly).
4. Props / unidirectional data flow: Pass data down, use callbacks to send changes up.
5. Raising (lifting) state: When to move local state up so multiple children can share it.
6. Data lifecycle: Show how data flows from initial load to user interactions.
7. Components in separate files: Organize a larger codebase by splitting components.
8. Class (legacy) vs functional (modern) components: Compare old class-based approach to newer Hooks-based approach.
9. External API: Integrate a real or publicly available service to show real-world data fetching.

Additional Resources:

Routing:
https://reactrouter.com/start/declarative/installation

Component Library:

MaterialUI
https://www.npmjs.com/package/@mui/material
https://mui.com/

State Management:

Redux
https://react-redux.js.org/
https://redux-toolkit.js.org/introduction/getting-started

Additional Reading:

https://swc.rs/