TEACH2GIVE
INDIVIDUAL FAMILY COMMUNITY

React Docs

This is a comprehesive docs for react

Table of Contents

introduction to react.js

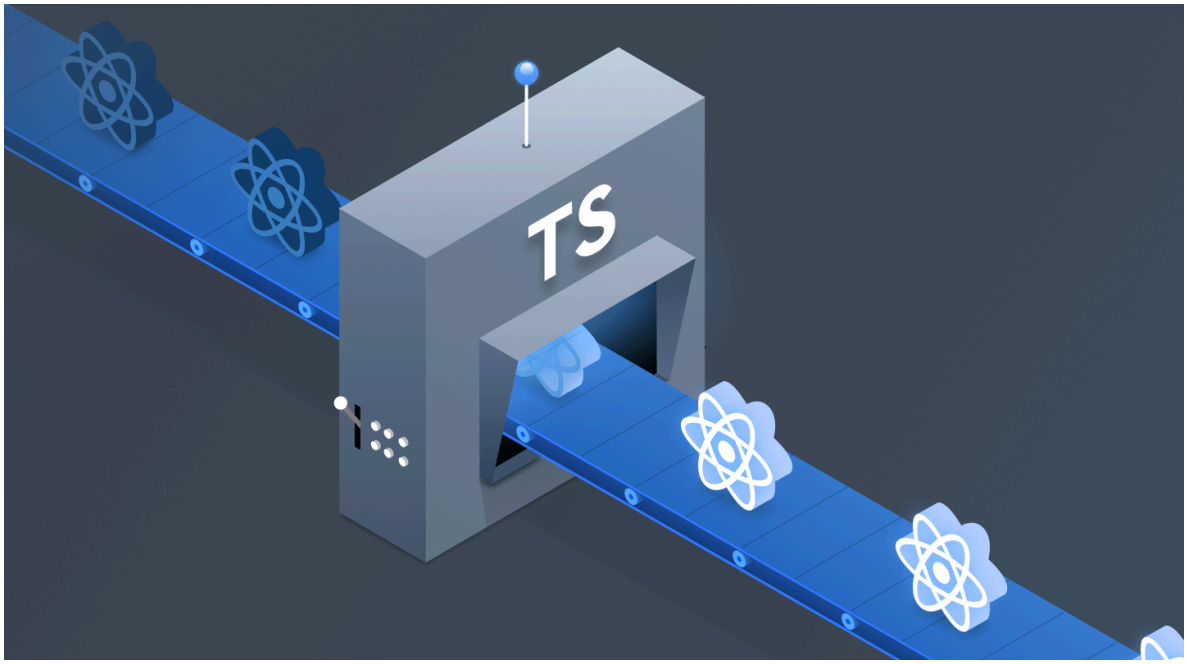


image.png

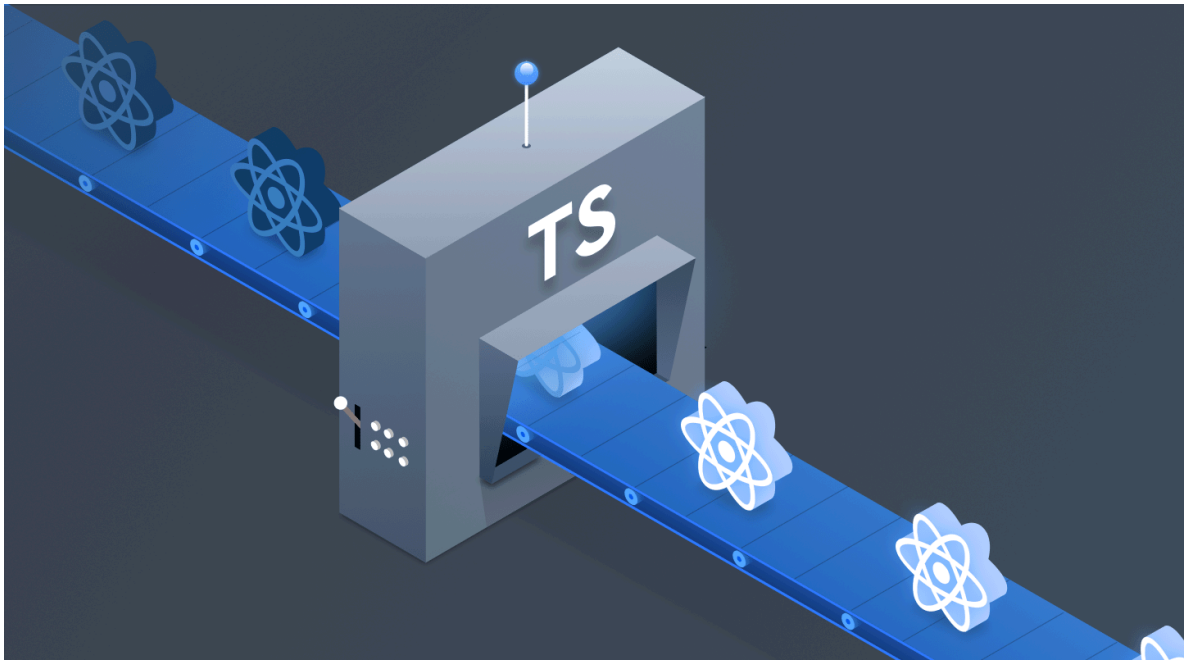
1.1 Overview of React

React Quickstart

Before you start, you should have a basic understanding of:

1. [x] What is HTML
2. [x] What is CSS
3. [x] What is DOM
4. [x] What is ES6
5. [x] What is Node.js
6. [x] What is npm

What is React?



react

- React, sometimes referred to as a frontend JavaScript framework, is a JavaScript library created by Facebook.
- React is a tool for building UI components.
- React has solidified its position as the go-to JavaScript front-end framework in the current tech landscape. It's fascinating to see how it's seamlessly woven itself into the development practices of well-established corporations and budding startups alike.

What is React used for?

1. **Web development :**
    - This is where React got its start and where you'll find it used most often. React is component-based. An example of a component could be a form or even just a form field or button on a website. In React, you build up complete applications using components like these by nesting them.
    - Components in React can manage their own state and communicate that state to child components. By "state," we mean the data that populates the web application.
2. **Mobile app development :**
    - React Native is a JavaScript framework that uses React. With React Native, developers can apply web-based React principles to creating mobile apps for Android and iOS. Here, React is used to connect the mobile user interface of the application to the phone's operating system.
3. **Desktop app development :**
    - Developers can also use React with Electron, another JavaScript library, to create cross-platform desktop apps. Some apps you may know about that are built with Electron include Visual Studio Code, Slack, Skype, Discord, WhatsApp, and WordPress Desktop.

React.JS History

React.js, a popular JavaScript library for building user interfaces, particularly for single-page applications, has a fascinating history that reflects its evolution and growing

adoption in the web development community. Here's a brief history of React.js in bullet points:

- **2011**: React.js created by Facebook's Jordan Walke for internal use.
- **2013**: Open-sourced at JSConf US; introduced virtual DOM.
- **2014**: Facebook introduced Flux, influencing state management in React.
- **2015**: React Native launched, expanding React to mobile apps.
- **2015**: React v0.14 split core into `react` and `react-dom`.
- **2016**: React v15 brought performance improvements and `prop-types`.
- **2017**: React Fiber (v16) restructured core for better responsiveness.
- **2018**: Hooks introduced in v16.8, transforming component design.
- **2019**: Experimental Suspense and Concurrent Mode introduced.
- **2020**: React v17 focused on easier upgrades.
- **2022**: React 18 brought full Concurrent Mode and enhanced UI responsiveness.
- **2023**: React 19 is the latest major release of the React JavaScript library, bringing a range of new features and improvements aimed at enhancing both developer experience and application performance. Some of the key updates include:
  - **React Compiler**: A significant new feature, the React Compiler automates many performance optimizations, like memoization, which were previously handled manually using hooks like `useMemo` and `useCallback`. This simplifies the code and makes React apps faster and more efficient.
- **Actions and Form Handling**: React 19 introduces a new way to handle form submissions and state changes using "Actions." This feature simplifies managing asynchronous operations, making it easier to handle loading states, errors, and successful form submissions.
- **New Hooks**: Several new hooks have been introduced, such as `useOptimistic`, which allows for optimistic UI updates (i.e., updating the UI immediately while awaiting server confirmation), and `use`, which simplifies asynchronous operations within

components. Additionally, the `useFormStatus` and `useActionState` hooks make managing form state more intuitive.

- **Server Components**: React 19 enhances server-side rendering by allowing server components, similar to features in frameworks like Next.js. This can lead to faster page loads and improved SEO.
- **Improved Metadata Management**: Managing document metadata like titles and meta tags is now easier and more integrated into React components, eliminating the need for third-party libraries like `react-helmet`.
- **Background Asset Loading**: React 19 introduces background loading of assets (like images and scripts), which helps improve page load times and overall user experience by preloading resources in the background as users navigate through the app.

Thinking in React

Hooks

Start typing here...

Custom Hooks

8. Custom Hooks

Custom Hooks in React

Custom hooks are a powerful feature in React that allow you to encapsulate and reuse stateful logic across multiple components. They enable you to extract logic into a reusable function, which can then be used just like the built-in hooks provided by React.

What Are Custom Hooks?

A custom hook is essentially a JavaScript function whose name starts with `use` and that can call other hooks. Custom hooks let you combine multiple built-in hooks like `useState`, `useEffect`, and `useRef` to create reusable logic that can be shared across different components. This leads to cleaner, more maintainable, and reusable code.

Why Use Custom Hooks?

1. **Reusability**: Encapsulate logic that is shared across multiple components into a single, reusable hook.
2. **Separation of Concerns**: Keep your components clean by moving complex logic out of them.
3. **Abstraction**: Abstract away implementation details that aren't relevant to the component itself, allowing you to focus on the component's purpose.
4. **Testability**: Custom hooks can be tested independently, improving the overall testability of your application.

Example 1: useLocalStorage Hook

The `useLocalStorage` hook allows you to synchronize a state variable with `localStorage`, so that the state persists even when the page is refreshed.

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

function useLocalStorage<T>(key: string, initialValue: T) {
```

```
    // Retrieve from localStorage or use the initial value
    const [storedValue, setStoredValue] = useState<T>(() => {
        try {
            const item = window.localStorage.getItem(key);
            return item ? JSON.parse(item) : initialValue;
        } catch (error) {
            console.error("Failed to read from localStorage:", error);
            return initialValue;
        }
    });

    // Custom setter function to update state and localStorage
    const setValue = (value: T | ((val: T) => T)) => {
        try {
            // If the new value is a function, call it with the current
state
            const valueToStore = value instanceof Function ?
value(storedValue) : value;
            setStoredValue(valueToStore);
            window.localStorage.setItem(key,
JSON.stringify(valueToStore));
        } catch (error) {
            console.error("Failed to write to localStorage:", error);
        }
    };

    return [storedValue, setValue] as const;
}

// Usage Example
import React from 'react';

const ExampleComponent: React.FC = () => {
    const [name, setName] = useLocalStorage<string>('name', 'John Doe');

    return (
        <div>
            <input
```

```
                type="text"
                value={name}
                onChange={(e) => setName(e.target.value)}
            />
            <p>Stored Name: {name}</p>
        </div>
    );
};

export default ExampleComponent;
```

Explanation:

- **Custom Hook (`useLocalStorage`)**: This hook manages a piece of state that is synchronized with `localStorage`.
- **Generic Type `<T>`**: The hook is generic, meaning it can handle any type of data (`string`, `number`, `object`, etc.).
- **Persisting State**: The hook retrieves the initial value from `localStorage` (if available) and sets up a setter function that updates both the state and `localStorage`.

Example 2: useFetch Hook

The `useFetch` hook abstracts away the logic for making an HTTP request and managing the corresponding loading, error, and data states.

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

interface FetchState<T> {
    data: T | null;
    loading: boolean;
    error: string | null;
}

function useFetch<T>(url: string, options?: RequestInit) {
    const [state, setState] = useState<FetchState<T>>({
        data: null,
```

```
        loading: true,
        error: null,
    });

    useEffect(() => {
        const fetchData = async () => {
            try {
                setState({ data: null, loading: true, error: null });
                const response = await fetch(url, options);
                if (!response.ok) {
                    throw new Error(`Error: ${response.statusText}`);
                }
                const result = await response.json();
                setState({ data: result, loading: false, error: null });
            } catch (error: any) {
                setState({ data: null, loading: false, error:
error.message });
            }
        };

        fetchData();
    }, [url, options]);

    return state;
}

// Usage Example
import React from 'react';

interface User {
    id: number;
    name: string;
}

const UsersComponent: React.FC = () => {
    const { data, loading, error } = useFetch<User[]>
('https://jsonplaceholder.typicode.com/users');
```

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

    return (
        <ul>
            {data?.map(user => (
                <li key={user.id}>{user.name}</li>
            ))}
        </ul>
    );
};

export default UsersComponent;
```

Explanation:

- **Custom Hook (`useFetch`)**: The hook manages the fetching process, including loading, error handling, and storing the fetched data.
- **Generic Type `<T>`**: The hook is generic, allowing it to work with any type of data structure that you fetch.
- **Effect Dependency**: The `useEffect` hook runs the fetch operation whenever the `url` or `options` change.

Best Practices for Custom Hooks

1. **Naming Convention**: Always start the name of your custom hook with `use` (e.g., `useFetch`, `useLocalStorage`). This helps React understand that the function is a hook, and it must follow the Rules of Hooks.
2. **Return Values**: Custom hooks can return any value, but they often return an array (like `useState`) or an object to provide multiple pieces of information (e.g., `data`, `loading`, `error`).
3. **Reusability**: Keep your custom hooks as generic and reusable as possible. This makes it easier to apply them in different parts of your application.

4. **Encapsulation**: Move complex logic out of your components and into custom hooks. This helps keep your components focused on rendering UI and handling user interaction.
5. **Testing**: Because custom hooks are just functions, you can and should test them independently from the components that use them.

Summary

- **Custom Hooks**: Functions that allow you to encapsulate and reuse stateful logic across multiple components.
- **`useLocalStorage` Hook**: Synchronizes state with `localStorage`, persisting the state across page refreshes.
- **`useFetch` Hook**: Manages the data fetching process, including handling loading states and errors.
- **Best Practices**: Use descriptive names, ensure reusability, keep hooks generic, and always test them.