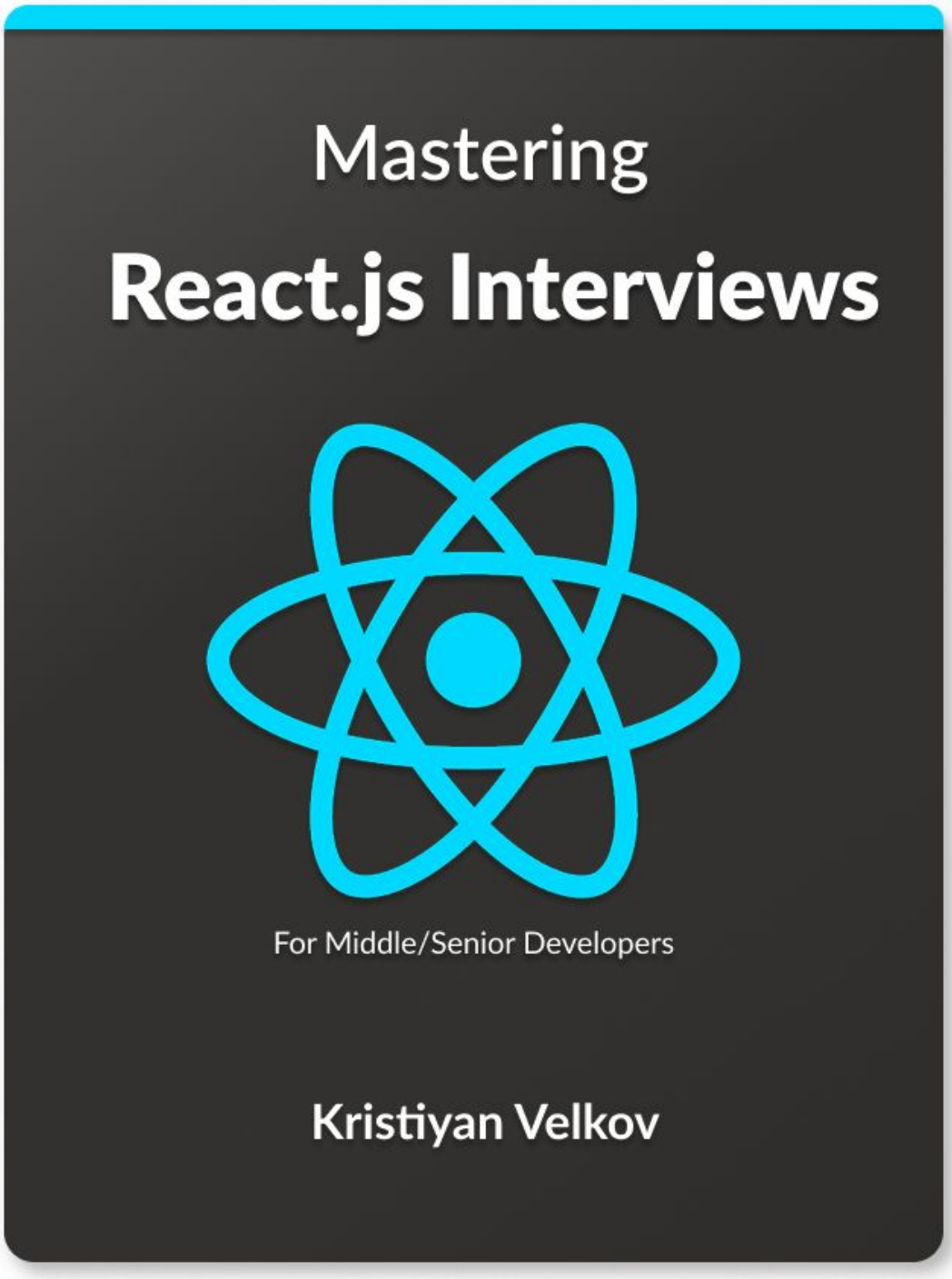


Part 1: Security

Battle-tested strategies and real-world insights from a Technical Lead who has conducted 350+ front-end interviews.

Table of content:

About the Author **5**

My Books **6**

Mastering React.js Interviews: From Zero to Hero 7

Mastering TypeScript Core Utility Types 9

Front-end world - Newsletter **11**

Connect with me **12**

Introduction **13**

Why This Book? 14

What Makes This Book Different? **15**

What You'll Learn? 16

Who This Book Is For ? 21

How To Use This Book ? 22

What's Inside Each Question? 23

Part I: Security **24**

1. What is Cross-Site Scripting (XSS), and why is it a security concern in web applications? 25

2. How does React handle XSS attacks in JSX by default? 29

3. Why is it important to sanitize user inputs in a React app? 33

4. What is the dangerouslySetInnerHTML attribute in React, and why is it considered dangerous? 35

5. Where would you store sensitive data, like API keys or tokens, in a React app? 38

6. Why is storing sensitive data in localStorage or sessionStorage potentially insecure? 41

7. Why is using HTTPS important in a React application? 44

8. What is Cross-Site Request Forgery (CSRF), and why is it a concern for web applications? 46

9. How can you restrict access to certain routes in a React application? 49

- About the Author
- My Books
  - Mastering React.js Interviews: From Zero to Hero
  - Mastering TypeScript Core Utility Types
- Front-end world - Newsletter
- Connect with me
- Introduction
  - Why This Book?
- What Makes This Book Different?
  - What You'll Learn?
  - Who This Book Is For ?
  - How To Use This Book ?
  - What's Inside Each Question?
- Part I: Security
  - 1. What is Cross-Site Scripting (XSS), and why is it a security concern in web applications?
  - 2. How does React handle XSS attacks in JSX by default?
  - 3. Why is it important to sanitize user inputs in a React app?
  - 4. What is the dangerouslySetInnerHTML attribute in React, and why is it considered dangerous?
  - 5. Where would you store sensitive data, like API keys or tokens, in a React app?
  - 6. Why is storing sensitive data in localStorage or sessionStorage potentially insecure?
  - 7. Why is using HTTPS important in a React application?
  - 8. What is Cross-Site Request Forgery (CSRF), and why is it a concern for web applications?
  - 9. How can you restrict access to certain routes in a React application?

Copyright © 2025 Kristiyan Velkov. All rights reserved.

No part of this book may be copied, reproduced, distributed, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written consent of the copyright holder, except for brief quotations used in critical reviews or noncommercial purposes permitted by copyright law.

Every effort has been made to ensure the accuracy and reliability of the information presented in this book. However, the content is provided "as is," without warranties of any kind, either express or implied. The author assumes no responsibility for any errors, omissions, or discrepancies, nor for any damages or losses arising directly or indirectly from the use or reliance on this information.

All trademarks, company names, and product names mentioned in this book are the property of their respective owners and are used solely for identification purposes. Their inclusion does not imply endorsement or affiliation by the author.

The advice, examples, and strategies provided in this book may not be applicable to every individual or situation. Readers are encouraged to consult with a professional when appropriate. The author is not liable for any loss, injury, or damage resulting from the application of the information or for any errors or omissions in the content.

For inquiries or permission requests, please contact the author at:

Email: kristiqn.velkov@gmail.com
LinkedIn: Kristiyan Velkov

About the Author

Kristiyan Velkov is a software developer, technical leader, Docker Captain, author, blogger, and mentor with over a decade of experience in front-end development and open-source technologies.

His comprehensive skill set spans:

- **JavaScript and TypeScript**
- **React.js, Angular, and Next.js**
- **DevOps, Web Accessibility, and Security**
- **People Management and Leadership**
- **Over 350 Front-End Technical Interviews**: Expertise in assessing and mentoring developers, ensuring alignment with industry standards.

As a lifelong learner, Kristiyan has earned more than 100+ certifications from prestigious organizations and institutions, including:

- **AWS Cloud Solutions Architect – Amazon Web Services (AWS)**
- **Advanced React.js Developer – Meta**
- **Developing Cloud Apps with Node.js and React – IBM**
- **Cybersecurity for Everyone – University of Maryland**
- **Foundations of Cybersecurity – Google**
- **Leadership and Influence – University of Michigan**

Kristiyan has conducted more than **350 technical interviews**, bringing a wealth of insights into what top companies look for in candidates. He has also participated in several conferences, sharing his expertise on mastering tech interviews and advancing as a front-end developer.

My Books

I write to share what I've learned the hard way—through building, leading, mentoring, and interviewing. My goal is to bridge the gap between theory and real-world practice, helping front-end developers not just learn, but grow with clarity and confidence. If my books, articles, or talks have helped even one developer move forward, then the work has been worth it.



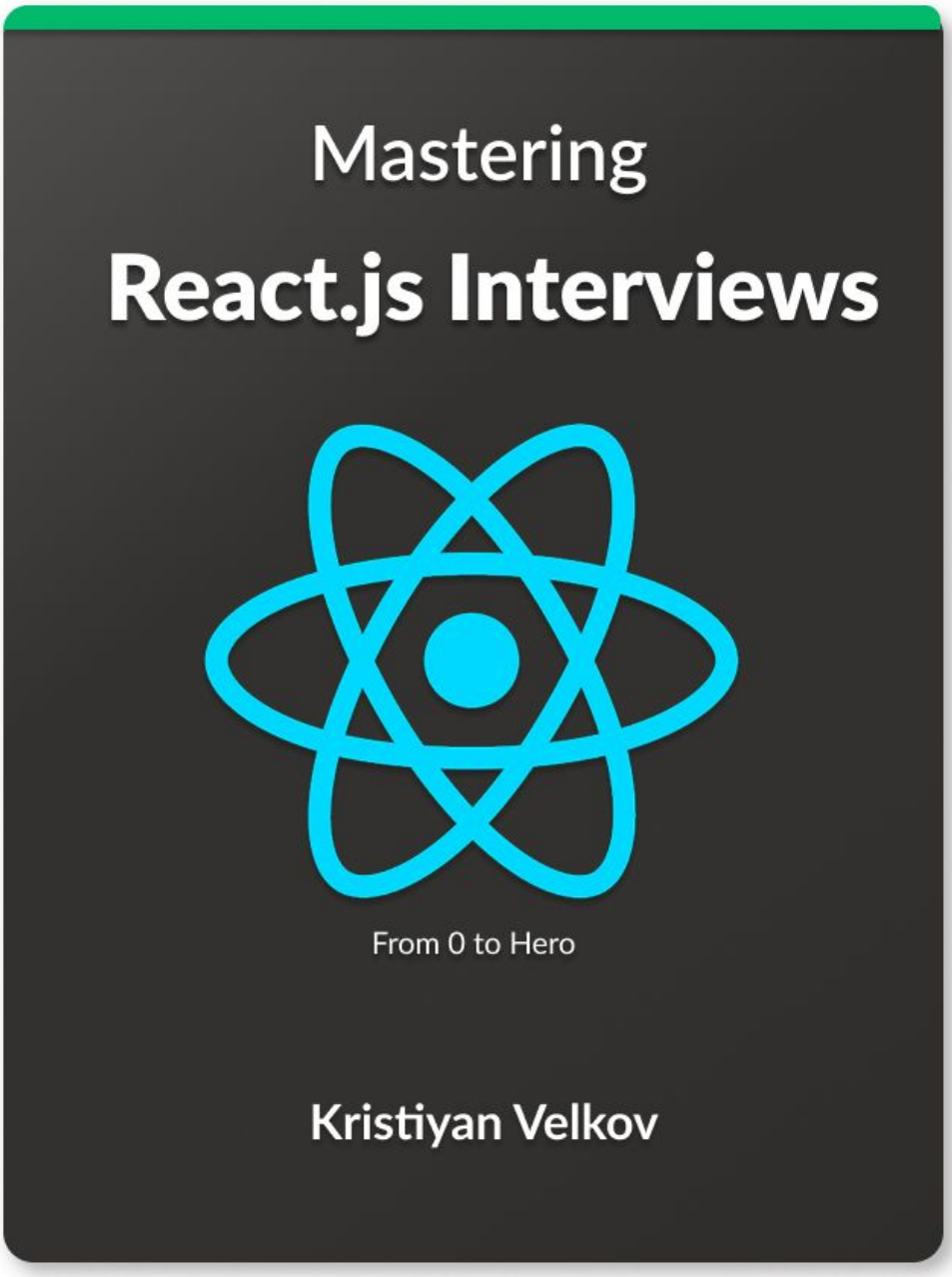
📖 **Find all my books:** https://leanpub.com/u/kristiyan-velkov

For questions, please get in touch with the author at:

Email: kristiqn.velkov@gmail.com
LinkedIn: Kristiyan Velkov

Mastering React.js Interviews: From Zero to Hero

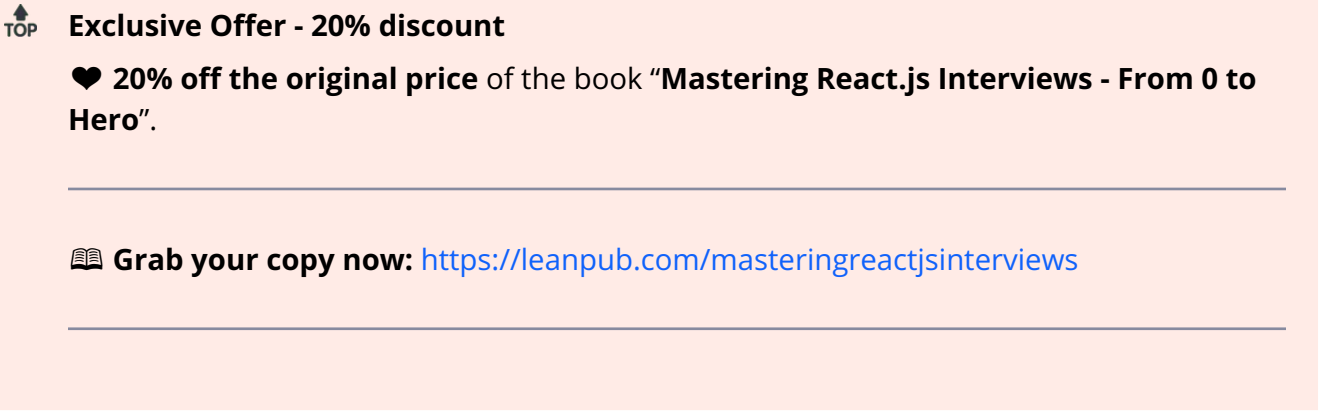
This book is your complete roadmap to crushing React.js interviews and rising above the competition.



Based on **350+ real-world front-end interviews**, it delivers the most relevant technical questions asked by top companies—along with clear, practical answers that help you build confidence, deepen your React.js knowledge, and prove your skills where it matters most.

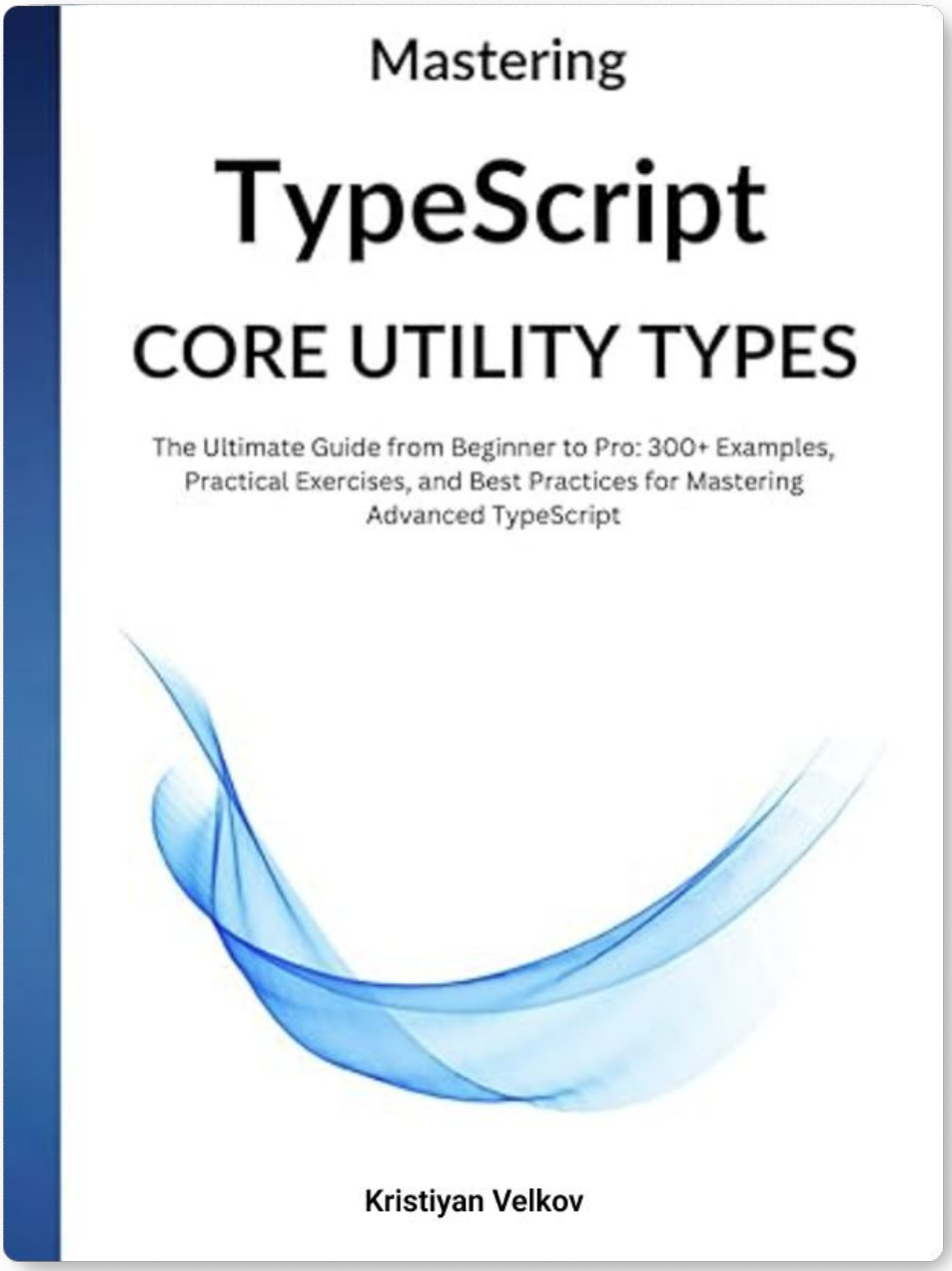
What's Inside

- Core React **Concepts –** JSX, Virtual DOM, props, state, and component structure (**React.js v19)**
- Hooks Dee**p Dive –** useState, useEffect, useReducer, useRef, useMemo, and more (**React.js v19**)
- Per**formance Optimization –** Avoid re-renders, lazy loading, memoization, and profiling (**React.js v19**)
- Componen**t Design & Reusability –** HOCs, render props, children, and Fragments (**React.js v19**)
- Forms & Event Handling – Controlled/uncontrolled components, validation, synthetic events (**React.js v19**)
- Styling Techniques – CSS Modules, Tailwind CSS (**v4.0), clsx (v2.1), and scalable styling strategies**
- **Routing –** Nested routes, useParams, useSearchParams, and route protection (**React Router v7.4)**
- Error **Handling –** Error boundaries, debugging practices, graceful fallbacks (**React.js v19**)
- Globa**l State Management –** Context API (**React 19**), Redux Toolkit (**v2.6), Zustand (v5.0), and best practices**



Mastering TypeScript Core Utility Types

Kristiyan is also the author of "**Mastering TypeScript Core Utility Types**".
The Ultimate Guide from Beginner to Pro: 300+ Examples, Practical Exercises, and Best Practices for Mastering Advanced TypeScript.



This book is your ultimate guide to **TypeScript core utility types**, equipping you with the knowledge to write cleaner, more efficient, and scalable code.

In this guide, you'll master the most essential TypeScript core utility types:

- **Partial**: Simplify workflows by making all properties optional.
- **Required**: Enforce mandatory properties for robust type safety.
- **Readonly**: Protect your data by preventing unwanted changes.
- **Record**: Effortlessly map keys to consistent value types.
- **Pick**: Extract only the properties you need for focused use cases.
- **Omit**: Exclude specific properties to streamline your types.



Exclusive Offer - 20% discount

❤ **20% off the original price** of the book "**Mastering TypeScript Core Utility Types**".

📖 **Grab your copy now:** https://leanpub.com/masteringtypescriptcoreutilitytypes

---

Front-end world - Newsletter

📥 Subscribe to my newsletter for exclusive, in-depth insights on topics covering :

React.js, Next.js, Angular, JavaScript, TypeScript, Web Security, Web Accessibility, Best practices, People management, DevOps.

🔝 **Exclusive Offer - 20% discount**

Every **paid subscriber** receives a **lifetime 20% discount** on all my upcoming books!

Subscribe now: https://kristiyanvelkov.substack.com/subscribe

---

☕ For less than the price of one coffee per month, you'll get: ☕

- **Lifetime 20% Discount** on all my upcoming books, including https://leanpub.com/masteringtypescriptcoreutilitytypes, Mastering React.js Interviews - from 0 to Hero and future releases.
- **Exclusive Early Access** to new articles, guides, and book drafts before they go public.
- **Behind-the-Scenes Insights** into my writing process, research, and in-depth coding breakdowns.
- **Private Q&A Sessions** where you can ask me anything about React, Next.js, TypeScript, Angular, DevOps, Security, Performance optimization, Career growth and more.
- **Access to 160+ In-Depth Blog Posts** covering JavaScript, TypeScript, React.js, Next.js, Angular, DevOps, Security, Web Accessibility, Unit Testing, and more.
- **Premium Content & Tutorials** that won't be available to free subscribers.
- **Exclusive Discounts on Courses & Workshops** – Special pricing on upcoming courses, training sessions, or webinars.
- **Curated Industry Insights & Tech Trends** – Stay ahead with hand-picked resources, emerging trends, and must-know updates in the front-end ecosystem.

Connect with me

Kristiyan Velkov has cultivated a strong community of developers through his insights and expertise in Front-end and DevOps:

- **30,000 LinkedIn connections**
- **6,500 Medium subscribers**
- **250,000+ annually readers**

Through his blogs and articles, Kristiyan delivers valuable content on advanced coding techniques, best practices, productivity strategies, and leadership. His work simplifies complex topics, making them accessible and actionable for developers worldwide.

📖 **Follow Kristiyan Velkov on social media platforms.**

- **Medium Blog**
- **Substack**
- **LinkedIn**
- **X (Twitter)**

Introduction

Welcome to "Mastering React.js Interviews”

With over **350 front-end technical interviews** under my belt, I’ve gained deep insights into what separates successful candidates from the rest. From assessing technical skills to evaluating problem-solving abilities, I’ve seen it all. This experience has equipped me to guide you through the React.js interview process, helping you understand what interviewers are looking for and how to showcase your skills effectively.

React.js has solidified its place as an essential tool for creating modern, dynamic user interfaces. Its component-based structure, unparalleled flexibility, and vast ecosystem make it indispensable for front-end developers aiming to excel in their careers.

Securing a React.js role means mastering more than just the basics. Interviewers often probe into topics like state management, hooks, performance optimization, web accessibility, unit testing, security, and more. This book is designed to unravel the complexities of the interview process, equipping you with the knowledge and confidence to excel, no matter the difficulty of the questions you face.

Whether you're a **mid-level React.js developer** or a **seasoned senior engineer aiming for your next big opportunity**, this book is built for you. It goes beyond the basics, covering both **advanced topics and real-world scenarios** that top companies expect you to know.

Get ready to **master the interview process**, showcase your expertise with confidence, and take your React.js career to the next level.

Why This Book?

The React.js interview process is demanding. It goes far beyond textbook knowledge—testing your grasp of **real-world scenarios**, from JSX fundamentals to advanced patterns, performance optimization, security, and architecture.

Here's the hard truth: most developers don't stumble because they lack knowledge.
They stumble because they're **unprepared for the depth, structure, and practical focus** of today's front-end interviews.

That's exactly why this book exists.

After leading over **350 technical interviews** for front-end roles, I've seen firsthand what top companies look for—and what consistently separates standout candidates from the rest. This book is a **battle-tested, no-fluff guide** built to prepare mid and senior-level developers to approach any React.js interview with clarity, confidence, and a sharp competitive edge.

What Makes This Book Different?

✓ **Real Interview Insight**

Gain knowledge directly from the hiring side—based on actual interviews, recurring patterns, and real expectations for React.js roles.

✓ **Full-Coverage, No Gaps**

From core principles to advanced concepts, this book prepares you for both **mid-level** and **senior-level** interviews—nothing skipped, nothing watered down.

✓ **Hands-On Practice**

Sharpen your skills with real-world exercises, interview-style questions, and mock challenges that mirror what you'll face in top interviews.

✓ **Built for Practical Applicatio**n

Knowing React isn't enough—you'll learn how to demonstrate expertise through state management, performance optimization, accessibility, testing, and more.

This isn't just another study guide.
It's your **strategic advantage** in the React.js job market—built to help you stand out, communicate with confidence, and land the role you've worked hard to earn.

What You'll Learn?

This book is designed to equip you with the tools and confidence needed to excel in React.js interviews, no matter your experience level. By the end, you'll be able to:

1. Security in React Applications

- What is Cross-Site Scripting (XSS), and why is it a security concern in web applications?
- How does React handle XSS attacks in JSX by default?
- Why is it important to sanitize user inputs in a React app?
- What is the `dangerouslySetInnerHTML` attribute in React, and why is it considered dangerous?
- Where would you store sensitive data, like API keys or tokens, in a React app?
- Why is storing sensitive data in `localStorage` or `sessionStorage` potentially insecure?
- Why is using HTTPS important in a React application?
- What is Cross-Site Request Forgery (CSRF), and why is it a concern for web applications?
- How can you restrict access to certain routes in a React application?

2. Web Accessibility (a11y)

- What is web accessibility, and why is it important?
- What are WCAG guidelines, and can you name the four principles of accessibility?
- What is the purpose of ARIA roles, and when should you use them?
- Can you explain the difference between semantic HTML and non-semantic HTML?
- What are some common accessibility tools for testing a website? (e.g., Lighthouse, Axe, NVDA, JAWS, VoiceOver)
- What are some examples of implementing Web accessibility in React.js?
- How do you ensure color contrast ratios meet accessibility standards?
- Implement WCAG accessibility guidelines in React applications.
- Use ARIA roles and attributes effectively.
- Optimize keyboard navigation and ensure proper color contrast.

3. React and TypeScript

- How do you define types for React components?
- How do you handle props with TypeScript?
- What are `React.FC`, `PropsWithChildren`, and `ComponentProps`?
- How do you type event handlers in TypeScript?

- How do you type custom hooks in TypeScript?
- How do you type `useState` in React with TypeScript?
- What is the best way to type component props that extend HTML attributes?
- How do you type a React Context API provider with TypeScript?
- What is a mapped type, and how can it be useful in a React project?
- How do you use TypeScript utility types (`Partial` , `Pick` , `Omit` , `Record`) in React?
- When should you use `readonly` in React.js props?
- Define types for React components, props, and events.
- Use utility types (`Partial` , `Pick` , `Omit` , `Record`) in React applications.
- Learn how to type `useState` , `useContext` , and custom hooks.

4. Routing (React Router v7 / TanStack Router)

- What is React Router?
- How do you pass parameters in the URL with React Router? (e.g., `/users/:id`)
- What is the difference between `<Link>` and `<NavLink>` ?
- How do you protect a route in React Router v7 using authentication logic?
- What are layout routes, and how do they work?
- What is the difference between `useParams()` and `useSearchParams()` ?
- What is TanStack Router?
- How do you define route parameters in TanStack Router?
- How do you protect routes for authenticated users only in TanStack Router?
- How does TanStack Router handle search params, and when should they be used?
- Learn advanced navigation techniques with React Router.
- Implement protected routes and authentication handling.
- Explore differences between `useParams` and `useSearchParams` .

5. Global State Management

- What is global state, and how does it differ from local component state in React?
- When is using a global state management solution unnecessary?
- What are the trade-offs between using `useContext` + `useReducer` ?
- What are the trade-offs between using `useContext` + `useReducer` versus a dedicated state management library like Redux, Zustand, MobX, or Recoil?
- Why is Redux Toolkit (RTK) recommended over vanilla Redux?
- How does `useSelector` work, and how can you prevent unnecessary re-renders in Redux?
- What is Redux middleware, and how do Thunks help with async state management?
- How does Redux Persist work, and when would you use it?
- How does Zustand differ from Redux in terms of API and performance?

- How does Zustand handle asynchronous state updates?
- Understand when global state is necessary and compare solutions.
- Learn state management techniques using Redux, Zustand, MobX, and Recoil.
- Optimize global state to prevent unnecessary re-renders.

6. Data Fetching

- What are the key differences between `fetch()`, Axios, SWR, and TanStack Query for fetching data in React?
- How do you fetch data using the `fetch()` API inside `useEffect()`?
- What are the advantages of using Axios over `fetch()` in React?
- How do you handle errors properly in `fetch()`, Axios, and React Query?
- How does SWR handle automatic caching and revalidation compared to TanStack Query?
- How do you send `POST`, `PUT`, and `DELETE` requests using `fetch()` and Axios?
- How do you set up request headers (e.g., authentication tokens) with `fetch()` and Axios?
- How can you cancel an API request using `fetch()` and Axios?
- What is the difference between `fetch().then().catch()` and `async/await` when handling API requests?
- How do you fetch and update paginated data using Axios?
- How do you fetch data using SWR in React? Provide an example.
- How do you use TanStack Query (React Query) to fetch and cache API responses?
- What are the key differences between SWR and TanStack Query when handling real-time data?
- How do you perform optimistic updates using SWR or TanStack Query?
- How do you handle background data refetching in SWR and TanStack Query?
- Compare different data-fetching approaches: `fetch()`, Axios, SWR, and TanStack Query.
- Handle API requests efficiently with caching, optimistic updates, and background re-fetching.
- Learn best practices for managing paginated and real-time data.

7. GraphQL and API Integration

- What is GraphQL, and how does it compare to REST APIs?
- What are GraphQL queries, mutations, and subscriptions?
- How do you integrate Apollo Client with React?
- What are resolvers in GraphQL, and how do they work?
- How does GraphQL handle over-fetching and under-fetching of data compared to REST?
- How do you handle errors in Apollo Client?
- What is a GraphQL schema, and how do you define one?
- How do you optimize GraphQL queries for performance?
- What are the security concerns with GraphQL, and how do you mitigate them?

- How does GraphQL handle authentication and authorization?
- Understand how GraphQL differs from REST APIs.
- Learn GraphQL queries, mutations, and subscriptions.
- Integrate Apollo Client with React applications.

8. Unit Testing

- What are the different types of testing in React? (Unit, Integration, E2E)
- What are the most common libraries used for unit testing in React?
- What is the pyramid of testing?
- What is the difference between shallow and mount rendering in testing?
- How do you mock API requests when testing a React component?
- How do you test a component's props in React?
- How do you test state updates inside a React component?
- How do you test event handlers such as `onClick` or `onChange`?
- How do you test a component that fetches data from an API?
- How do you ensure that a React component renders the correct elements?
- What is the difference between `screen.getByText` and `screen.findByText` in React Testing Library?
- How do you handle asynchronous tests in React?
- Learn different types of testing: Unit, Integration, and E2E.
- Work with Jest, React Testing Library, and Cypress.
- Mock API requests and interactions for better test coverage.

9. React.js & Server-Side Rendering (SSR)

- What is Server-Side Rendering (SSR), and how does it work in React?
- How does SSR improve performance compared to Client-Side Rendering (CSR)?
- What are the key differences between SSR and Static Site Generation (SSG)?
- How does SSR impact SEO, and why is it beneficial for search engines?
- What are React Server Components (RSC), and how do they differ from traditional client components?
- What are React Server Functions?
- What is Next.js, and how does it improve React applications?
- Explain the difference between SSR, SSG, ISR, and CSR in Next.js.
- What is the role of React Suspense in Server-Side Rendering, and how can it improve performance?
- What caching strategies can be used in Next.js?
- What are API routes in Next.js, and how do they work?
- How do dynamic routes work in Next.js? Provide an example.
- Understand the differences between CSR, SSR, SSG, and ISR.
- Work with Next.js for server-side rendering and API routes.

- Optimize caching strategies for SSR applications.

10. Micro Frontends & Component-Driven Architecture

- What is Micro Frontend Architecture, and how does it relate to React?
- When should you consider using Micro Frontends?
- How do you share components between multiple React applications?
- How do you share state in Micro Frontend Architecture?
- How does routing work in a Micro Frontend architecture?
- What are the security concerns in Micro Front-End applications, and how do you mitigate them?
- Learn how Micro Frontends work in large-scale React applications.
- Share components and state across multiple React projects.
- Implement routing and security strategies for Micro Frontend applications.

11. Deployment & CI/CD

- How do you bundle and optimize React applications for production?
- What are environment variables, and how do you manage them in React?
- What is Docker?
- How do you containerize a React application for production?
- How do you configure a CI/CD pipeline for a React app using GitHub Actions or GitLab?
- How do you handle versioning and rollback for a React application deployment?
- How do you ensure zero-downtime deployment for a React application using Kubernetes?
- How do you deploy a React app to Vercel, Netlify, or AWS, and what are the key differences?
- Learn how to bundle and optimize React applications for production.
- Work with Docker to containerize React applications.
- Set up CI/CD pipelines using GitHub Actions or GitLab for automated deployments.
- Deploy React applications on Vercel, Netlify, and AWS.

Who This Book Is For ?

This book is designed for developers at all levels who want to **master React.js interviews** and advance their careers.

- **Intermediate Developers** – If you already have experience with React.js, this book will deepen your knowledge, reinforce best practices, and prepare you for mid-level interviews.
- **Senior Developers & Experts** – If you're aiming for senior or leadership roles, this book covers advanced concepts, performance optimizations, architecture patterns, and real-world problem-solving techniques.

Whether you're hunting for your next opportunity, preparing for a promotion, or planning a strategic career move, this book delivers the knowledge, clarity, and confidence to help you shine in any React.js interview and stand out from the crowd.

How To Use This Book ?

This book is structured to provide a **step-by-step** learning experience, covering React.js**advanced topics** commonly tested in technical interviews. Each chapter builds upon the previous one, increasing in complexity.

- **If you're preparing for your first senior-level interview**, start from the beginning. This approach ensures you reinforce key fundamentals while building toward complex topics.
- **If you're already experienced in specific areas**, feel free to jump straight to the sections that challenge you most. Each question stands on its own, offering practical, real-world insights without needing to read previous chapters.

What's Inside Each Question?

Every question is crafted to deliver **real-world clarity** and **interview precision**, structured for quick reference and deep understanding:

- ✓ **Detailed Answer** – In-depth explanations that go beyond surface-level theory.
- ✓ **Short Answer** – Quick summaries ideal for rapid review.
- ✓ **Code Examples** – TypeScript-based examples reflecting real-world usage.
- ✓ **Common Pitfalls & Best Practices** – Learn how to avoid common mistakes and write cleaner, more efficient code.
- ✓ **Fun Facts** – Technical trivia and historical insights to enrich your learning.
- ✓ **Further Resources** – Curated links to trusted documentation and advanced reading.

This book is not just a reference—it's a **practical roadmap** designed to equip you with the skills, confidence, and strategic thinking needed to **excel in React.js interviews and secure your next role**.

Part I: Security

1. What is Cross-Site Scripting (XSS), and why is it a security concern in web applications?

Cross-site scripting (XSS) is a type of security vulnerability that allows an attacker to inject malicious scripts (usually JavaScript) into a web application. These scripts are then executed in the browser of unsuspecting users, often without their knowledge. XSS can be used to steal sensitive information, impersonate users, or perform malicious actions on their behalf.

How XSS Can Happen in React.js

XSS vulnerabilities in React typically occur when:

1. **Using** `dangerouslySetInnerHTML` without sanitization.
2. Dynamically inserting user-generated content into the DOM without escaping it.
3. Handling unsafe third-party libraries or APIs.

Code Examples

1. **Vulnerable Example with** `dangerouslySetInnerHTML`

```
function App({ userContent }) {
    return <div dangerouslySetInnerHTML={{ __html: userContent }} />;
}
```

Attack Scenario:

- If `userContent` contains malicious input like `<script>alert('XSS')</script>`, it will execute in the browser.

2. **Safe Example: Sanitizing User Input**

Use a library like **DOMPurify** to sanitize user-generated content before rendering it.

```
npm install dompurify
```

```
import DOMPurify from 'dompurify';

function App({ userContent }) {
    const sanitizedContent = DOMPurify.sanitize(userContent);

    return <div dangerouslySetInnerHTML={{ __html: sanitizedContent }} />;
}
```

How It Works:

- `DOMPurify.sanitize` cleans the input, removing any malicious scripts.

3. **Safe Rendering Without** `dangerouslySetInnerHTML`

When possible, avoid using `dangerouslySetInnerHTML` entirely. Instead, use React's built-in escaping mechanisms by rendering content as plain text.

```
function App({ userContent }) {
    return <div>{userContent}</div>;
}
```

Why It's Safe:

- React automatically escapes dynamic content rendered this way, ensuring it cannot execute scripts.

Short Answer

In React.js, XSS occurs when untrusted input is improperly rendered into the DOM, potentially executing malicious scripts. While React automatically escapes content by default, using `dangerouslySetInnerHTML` or external libraries without sanitization can lead to vulnerabilities.

Info

- **Why It Happens**:
  - XSS exploits improperly sanitized input inserted into the DOM.
- **React's Protection**:
  - React escapes strings by default when rendering dynamic content, converting `<script>` into safe characters like `<script>`.
- **When You're Vulnerable**:
  - Using `dangerouslySetInnerHTML` or unsafe libraries.
  - Manipulating the DOM directly (e.g., `document.innerHTML`).

Important:

1. **Avoid** `dangerouslySetInnerHTML`:
   - Use it only when necessary, and always sanitize the input.
2. **Sanitize Input**:
   - Use libraries like DOMPurify to clean user-generated content.
3. **Content Security Policy (CSP)**:
   - Implement a CSP header to restrict scripts to trusted sources.
4. **Trust React**:
   - Rely on React's default escaping for most dynamic content.

Resources

- https://medium.com/kristiyan-velkov/react-js-security-best-practices-37ebe38ee1c2?sk=d7979ccd00b3af8b842be4bddc80ca02
- https://javascript.plainenglish.io/understand-web-security-sop-same-origin-policy-and-cors-cross-origin-resource-sharing-fcf75b2dfe40?sk=baf71800a181e672227c118f5087271f
- https://www.npmjs.com/package/dompurify
- https://react.dev/reference/react-dom/components/common#common-caveats
- https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

---

2. How does React handle XSS attacks in JSX by default?

React protects against **Cross-Site Scripting (XSS)** attacks by default through **automatic escaping** of all dynamic content in JSX. When rendering user-provided or dynamic data, React converts potentially dangerous characters (`<` , `>` , `"` , `'` , `&`) into their HTML entity equivalents. This prevents malicious scripts from executing in the browser.

Key Features of React's XSS Protection

1. **Automatic Escaping**:
    - React escapes all interpolated content in JSX, converting potentially harmful code into harmless text.
2. **Protection Against Script Injection**:
    - Any JavaScript code or HTML injected through JSX is rendered as plain text rather than being executed.
3. **No Direct DOM Manipulation**:
    - By handling DOM updates internally, React minimizes the risk of unsafe code execution.

Code Examples

1. **React Automatically Escapes Dangerous Content**

```
function App() {
    const userInput = "<script>alert('XSS')</script>";
    return <div>{userInput}</div>;
}

export default App;
```

What Happens:

- The `userInput` is rendered as:

```
<div>&lt;script&gt;alert('XSS')&lt;/script&gt;</div>
```

- The browser displays the text `<script>alert('XSS')</script>` instead of executing it.

2. **Safe Rendering of HTML Content**

```
function App() {
    const userContent = '<b>Safe Content</b>';
    return <div>{userContent}</div>;
}

export default App;
```

Output:

- The content is displayed as raw text: `<b>Safe Content</b>`.
- The browser will not interpret it as HTML.

3. **Danger Zone:** `dangerouslySetInnerHTML`

If you use `dangerouslySetInnerHTML`, React disables its default escaping and directly injects the provided HTML into the DOM.

This is a potential XSS risk.

```
function App() {
    const userContent = "<script>alert('XSS')</script>";
    return <div dangerouslySetInnerHTML={{ __html: userContent }} />;
}

export default App;
```

Output:

- The script will execute, triggering an XSS attack.

Solution: Sanitize the content before using `dangerouslySetInnerHTML` with DOMPurify.

```
import DOMPurify from 'dompurify';

function App() {
    const userContent = "<script>alert('XSS')</script>";
    const sanitizedContent = DOMPurify.sanitize(userContent);

    return <div dangerouslySetInnerHTML={{ __html: sanitizedContent }} />;
}

export default App;
```

Short Answer

React prevents XSS attacks by default through the automatic escaping of dynamic content in JSX. It renders user input as plain text, neutralizing malicious scripts before they can execute.

Info

- **How It Works**:
  - Dynamic content in JSX is escaped automatically.
  - Potentially harmful characters are converted into HTML entities.
- **Where React Can't Help**:
  - Using `dangerouslySetInnerHTML` without sanitization can bypass React's built-in protection.
  - Vulnerabilities in third-party libraries or unsafe APIs can introduce risks.

⁉ **Important:**

1. **React's Default Escaping**:
    - Trust React's default behavior when rendering dynamic data.
2. **Avoid** `dangerouslySetInnerHTML`:
    - If absolutely necessary, sanitize the input using libraries like `DOMPurify`.
3. **Validate Inputs**:
    - Perform input validation on both the client and server to ensure data integrity.

Resources

- https://react.dev/reference/react-dom/components/common#common-caveats
- https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
- DOMPurify GitHub

---

3. Why is it important to sanitize user inputs in a React app?

Sanitizing user inputs in a React app is crucial to prevent **security vulnerabilities** such as **Cross-Site Scripting (XSS)**, **SQL Injection**, and **other malicious attacks**. When user input is not properly sanitized, attackers can inject malicious scripts or data into your application, potentially compromising the security of your users and your system.

1. Prevent Cross-Site Scripting (XSS) Attacks

- **What is XSS?** XSS occurs when attackers inject malicious scripts into web pages viewed by other users. These scripts can steal sensitive data, such as cookies, session tokens, or credentials.
- **Example:** If you render raw user input in JSX:

```
const Comment = ({ userInput }) => (
    <div dangerouslySetInnerHTML={{ __html: userInput }}></div>
);
```

An attacker could submit:

```
<script>alert('Hacked!');</script>
```

Resulting in the malicious script being executed in the browser.

Short Answer

Sanitizing user inputs in a React app is essential to prevent security vulnerabilities like XSS attacks, protect data integrity, and maintain user trust.

⁉ **Important:**

- **Never Trust User Inputs:** Always treat all user-provided data as potentially malicious.
- **Sanitize Both Frontend and Backend:** Client-side sanitization improves user experience, but server-side validation is the ultimate safeguard.
- **Regularly Update Dependencies:** Keep libraries like `dompurify` and other security tools up-to-date to stay protected against the latest vulnerabilities.

Resources

- https://medium.com/kristiyan-velkov/react-js-security-best-practices-37ebe38ee1c2
- https://javascript.plainenglish.io/understand-web-security-sop-same-origin-policy-and-cors-cross-origin-resource-sharing-fcf75b2dfe40
- https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
- https://www.cloudflare.com/learning/security/threats/sql-injection/

---

4. What is the dangerouslySetInnerHTML attribute in React, and why is it considered dangerous?

`dangerouslySetInnerHTML` is a special attribute in React that allows you to set the inner HTML of a DOM element directly. It is typically used when you need to render raw HTML content, such as content from a CMS or user-generated input.

Example: Using `dangerouslySetInnerHTML`

```
function RawHTML({ htmlContent }) {
    return <div dangerouslySetInnerHTML={{ __html: htmlContent }} />;
}
```

Here, the `htmlContent` is inserted as raw HTML into the `div`.

Why Is It Considered Dangerous?

1. **Vulnerability to XSS Attacks:**
    - If the content passed to `dangerouslySetInnerHTML` is not sanitized, attackers could inject malicious scripts (e.g., `<script>alert('Hacked!');</script>`).
    - These scripts could steal user data, perform phishing attacks, or compromise the application.
2. **Bypasses React's Escaping Mechanism:**
    - React escapes HTML by default to prevent injection attacks. Using `dangerouslySetInnerHTML` bypasses this protection, introducing potential security risks.
3. **Difficult to Audit:**
    - It's harder to trace where raw HTML is being injected, especially in large applications, making it easier to overlook vulnerabilities.

Short Answer

`dangerouslySetInnerHTML` allows you to set raw HTML content in React components but is considered dangerous because it can introduce security vulnerabilities like XSS if the input is not properly sanitized.

Info

1. **Sanitize Content:** Always sanitize the content before passing it to `dangerouslySetInnerHTML`. Use libraries like `DOMPurify`:

```
import DOMPurify from 'dompurify';

const sanitizedContent = DOMPurify.sanitize(userInput);

return <div dangerouslySetInnerHTML={{ __html: sanitizedContent }} /
>;
```

2. **Avoid Direct User Input:** Never pass user-generated content directly into `dangerouslySetInnerHTML` without validation and sanitization.
3. **Minimize Usage:** Use `dangerouslySetInnerHTML` only when absolutely necessary. Opt for JSX rendering whenever possible:

```
<div>{content}</div> // React escapes this automatically.
```

4. **Backend Validation:** Validate and sanitize the content on the backend to ensure it's safe before sending it to the frontend.

⁉ **Important:**

- The "dangerous" label is a deliberate reminder that this feature must be used with caution.
- Using `dangerouslySetInnerHTML` carelessly can lead to severe security breaches.
- React's default behavior of escaping HTML ensures safety; bypassing it should only be done when you fully control the content.

📚 **Resources**

- https://react.dev/reference/react-dom/components/common#dangerously-setting-the-inner-html
- https://www.npmjs.com/package/dompurify

---

5. Where would you store sensitive data, like API keys or tokens, in a React app?

Sensitive data, like API keys or tokens, must be stored securely to prevent unauthorized access and ensure the safety of your application and users. The storage method depends on the type of data and its intended usage, but here are the best practices for handling sensitive data in a React app.

Options for Storing Sensitive Data

1. Environment Variables

- **Where:** Store sensitive data like API keys in environment files (`.env`) that are not included in your version control system.
- **Why:** Keeps sensitive data out of your codebase and ensures they can be configured for different environments (e.g., development, staging, production).
- **How:**

```
# .env
REACT_APP_API_KEY=your_api_key_here

# or in Vite
VITE_API_KEY=your_secret_api_key
```

Access in your React app:

```
const apiKey = process.env.REACT_APP_API_KEY;

// or in Vite
const apiKey = import.meta.env.VITE_API_KEY;
```

Important: Never expose sensitive keys directly in the client-side code if the key is for a backend service.

2. HTTP-Only Cookies for Tokens

- **Where:** Store tokens (e.g., access tokens, session tokens) in **HTTP-only cookies** to protect them from client-side JavaScript and mitigate XSS attacks.
- **Why:** Prevents the token from being stolen via `document.cookie` or malicious scripts.
- **How:**
  - Set the token on the server using a `Set-Cookie` header with the `HttpOnly` and `Secure` flags.

```
Set-Cookie: token=your_token_here; HttpOnly; Secure; SameSite=Strict;
```

Short Answer

- Use **environment variables** for storing API keys during build-time.
- Store **tokens** in **HTTP-only cookies** to prevent XSS attacks.
- Use **in-memory storage** for temporary sensitive data, avoiding persistent client-side storage like local storage.

Important:

- **API Keys in React (Frontend):**
  - Never expose sensitive API keys (e.g., private API keys) in a frontend application. Instead, proxy requests through a secure backend.
- **Secrets in Version Control:**
  - Use a `.gitignore` file to exclude `.env` or other sensitive files.

```
# .gitignore

*.env
```

- **Token Expiry and Refresh:**
  - Use short-lived tokens and implement a secure **refresh token** mechanism.
- **Server-Side Responsibility:**
  - Sensitive operations should always be handled on the server-side to avoid exposing private data in the client.
- **Use HTTPS:**
  - Always serve your app over HTTPS to encrypt data in transit.

Resources

- https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html

---

6. Why is storing sensitive data in localStorage or sessionStorage potentially insecure?

Storing sensitive data (e.g., tokens, API keys) in `localStorage` or `sessionStorage` is considered insecure because these storage mechanisms are accessible to client-side JavaScript, making them vulnerable to **Cross-Site Scripting (XSS)** attacks. Additionally, they lack features to restrict access, such as the `HttpOnly` and `Secure` flags available with cookies.

Security Risks of `localStorage` and `sessionStorage`

1. Vulnerability to XSS Attacks

- **What Happens:** If an attacker injects malicious scripts into your application, they can access data stored in `localStorage` or `sessionStorage` using `window.localStorage` or `window.sessionStorage`.
- **Example:**

```
// Malicious script injected into your app
console.log(localStorage.getItem('authToken')); // Exposes sensitive token
```

2. No Secure or HttpOnly Flags

- Unlike cookies, `localStorage` and `sessionStorage` do not have security flags like:
  - `HttpOnly`: Prevents access by JavaScript.
  - `Secure`: Ensures data is transmitted only over HTTPS.
- This means sensitive data stored in these mechanisms is always accessible to client-side scripts.

3. Persistent Data in `localStorage`

- Data stored in `localStorage` persists even after the user closes the browser, increasing the exposure window for potential attacks.

- If a device is compromised (e.g., stolen or shared), the stored sensitive data can be accessed easily.

4. Shared Between Tabs/Windows

- `localStorage` is shared across all tabs and windows for the same origin, potentially exposing sensitive data across sessions where it's not needed.

5. CSRF Risks (If Used Incorrectly)

- While `localStorage` and `sessionStorage` themselves don't mitigate Cross-Site Request Forgery (CSRF), improper usage (like storing tokens without proper validation) can compound vulnerabilities.

Short Answer

Storing sensitive data in `localStorage` or `sessionStorage` is insecure because it's accessible to client-side scripts, making it vulnerable to XSS attacks, and lacks security features like `HttpOnly` and `Secure` flags.

Important:

- **Avoid `localStorage` for Sensitive Data:** Use it only for non-sensitive information like user preferences or theme settings.
- **Encrypt If Necessary:** If you must use `localStorage`, encrypt the data, but note that encryption alone does not prevent XSS.
- **Combine Multiple Strategies:** Use a secure backend, in-memory storage, and HTTP-only cookies to manage sensitive data safely.

Resources

- https://owasp.org/www-community/attacks/xss/
- https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

---

7. Why is using HTTPS important in a React application?

Using HTTPS (Hypertext Transfer Protocol Secure) in a React application is critical to ensuring the security and privacy of data transmitted between the client and server. HTTPS encrypts the data, protects against malicious attacks like **Man-in-the-Middle (MITM)**, and ensures that users can trust your application. Without HTTPS, sensitive information such as login credentials, tokens, or API keys could be intercepted by attackers.

Key Reasons for Using HTTPS

1. Data Encryption

- HTTPS uses **SSL/TLS encryption** to ensure that all data transmitted between the browser and server is secure and unreadable to third parties.
- Protects sensitive information like:
  - User credentials
  - Tokens
  - Personally identifiable information (PII)

2. Protection Against Man-in-the-Middle (MITM) Attacks

- Without HTTPS, attackers can intercept and alter data in transit between the client and server.
- HTTPS ensures the data remains intact and cannot be modified or injected with malicious content.

3. Authentication and Trust

- HTTPS ensures that the client is communicating with the legitimate server and not a fake or malicious one.
- Secure websites display a padlock in the browser address bar, giving users confidence in your app's authenticity.

4. Secure Cookies

- Cookies can use the `Secure` flag to ensure they are only transmitted over HTTPS, preventing their interception in unsecured connections.
- Example:

```
Set-Cookie: token=your_token_here; Secure; HttpOnly; SameSite=Strict;http
```

5. Compliance with Modern Web Standards

- Many modern web features, such as **service workers**, **geolocation APIs**, and **Push Notifications**, require HTTPS to function.
- Browsers block or warn users when accessing unsecured websites.

6. SEO Benefits

- HTTPS is a ranking signal for Google, meaning that secure websites are more likely to rank higher in search results.

7. Protection Against Mixed Content Issues

- If your app serves some resources (e.g., images, scripts) over HTTP, it may cause security warnings or break functionality in HTTPS environments.
- HTTPS ensures all resources are loaded securely.

Short Answer

Using HTTPS in a React application encrypts data, prevents attacks like MITM, builds user trust, enables modern web features, and improves SEO. It is essential for protecting sensitive information and maintaining a secure connection.

Resources

- https://developer.mozilla.org/en-US/docs/Glossary/HTTPS
- https://en.wikipedia.org/wiki/Man-in-the-middle_attack

---

8. What is Cross-Site Request Forgery (CSRF), and why is it a concern for web applications?

Cross-Site Request Forgery (CSRF) is a type of attack where an attacker tricks a user into performing unintended actions on a web application where the user is authenticated. The attacker exploits the trust a web application has in the user's browser by sending unauthorized requests from the user's session without their consent.

For example, an attacker can trick a logged-in user into submitting a malicious form or clicking a hidden link, leading to unintended operations such as transferring money, changing account settings, or deleting data.

Why Is CSRF a Concern?

1. **Exploitation of Trust:**
    - Web servers trust the cookies or credentials sent by the browser, even if the request was initiated by a malicious source.
2. **Impact of Attacks:**
    - CSRF attacks can result in:
        - Unauthorized fund transfers.
        - Changing user account details.
        - Deleting or modifying critical data.
3. **Silent Execution:**
    - Users often remain unaware that their actions have triggered malicious requests.
4. **Applicability to Any Web App:**
    - Any application relying on cookies, sessions, or tokens for authentication can be vulnerable to CSRF.

Short Answer

CSRF is an attack where an authenticated user is tricked into performing unintended actions on a web application. It exploits the trust between the server and the browser, making it a serious concern for applications handling sensitive operations.

How to Prevent CSRF Attacks

1. **Use CSRF Tokens**
   - Generate a unique token for each user session and include it in forms or requests.
   - Validate the token server-side to ensure the request is legitimate.
   - Example (Express with CSRF Protection Middleware):

```
const csrf = require('csurf');
app.use(csrf({ cookie: true }));
```

2. **SameSite Cookies**
   - Set the `SameSite` attribute for cookies to prevent them from being sent with cross-origin requests.

```
Set-Cookie: sessionId=abc123; SameSite=Strict; Secure;
```

3. **Enable CORS (Cross-Origin Resource Sharing)**
   - Limit requests from trusted origins only, using strict `Access-Control-Allow-Origin` policies.
4. **Use POST Requests for Sensitive Actions**
   - Avoid using GET requests for sensitive operations (e.g., deleting data) to reduce the risk of CSRF.
5. **User Authentication and Validation**
   - Require re-authentication for critical actions (e.g., password changes or fund transfers).
6. **Content Security Policy (CSP)**
   - Implement a strong CSP to prevent malicious scripts from executing in your application.

Example of a CSRF Attack

1. **Scenario:**
   - A user is logged into their online banking account.
   - The attacker sends the user an email with a hidden malicious link:

```
<img src="https://bank.com/transfer?amount=1000&to=attacker_account" />
```

- When the user opens the email, their browser sends a request to the bank with their authentication cookies.

2. **Result:**
   - The bank processes the unauthorized request, transferring funds to the attacker.

⁉ **Important:**

- CSRF attacks exploit **authenticated users**, making it essential to protect sensitive operations.
- Use **CSRF tokens**, **SameSite cookies**, and strict **CORS policies** as a layered defense against CSRF.
- Educate users about phishing and avoid using GET requests for critical operations.

📚 **Resources**

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie#samesitesamesite-value
- https://javascript.plainenglish.io/understand-web-security-sop-same-origin-policy-and-cors-cross-origin-resource-sharing-fcf75b2dfe40?sk=baf71800a181e672227c118f5087271f
- https://owasp.org/www-community/attacks/csrf

---

9. How can you restrict access to certain routes in a React application?

In a React application, you can restrict access to certain routes by implementing **route guards** using libraries like `react-router-dom`. These guards determine whether a user has the appropriate authentication, role, or permissions to access specific routes. If the user doesn't meet the required criteria, they can be redirected to a login page, an error page, or another appropriate route.

Example: Restricting Routes with `react-router-dom`

1. ProtectedRoute Component

```
const ProtectedRoute = ({ isAuthenticated, userRole, requiredRole, children }) =>
{
    if (!isAuthenticated) {
        return <Navigate to="/login" />;
    }
    if (userRole !== requiredRole) {
        return <Navigate to="/unauthorized" />;
    }
    return children;
};
```

2. Using the ProtectedRoute in App Routes

```
<Route
    path="/admin"
    element={
        <ProtectedRoute isAuthenticated={isAuthenticated} userRole={userRole}
requiredRole="admin">
            <AdminPage />
        </ProtectedRoute>
    }
/>
```

Short Answer

Restrict access to routes in a React application by implementing route guards using `react-router-dom`, authentication context, or role-based access control. Redirect unauthorized users to login or error pages.

Important:

- Protect client-side routes with route guards but also validate permissions on the backend to prevent unauthorized API calls.
- Ensure JWT tokens or session cookies are securely stored (e.g., use HTTP-only cookies for tokens).

Resources

- https://medium.com/@kristiyanvelkov/react-js-protected-routes-a-step-by-step-guide-react-router-tanstack-router-c31ae740fa50
- https://reactrouter.com/
- https://tanstack.com/router/latest
