The Blue Print

A Journey Into Web Application Development with React

Thomas Ochman

Content

Introduction and fundamentals

The Blue Print - Alpha Edition

ISBN: —

Library of Congress Control Number: —

To request permissions, contact the author at thomas@agileventures.org

Rationale

React gets plenty of attention in programming resources, but most books and tutorials focus on the happy path. You'll find countless introductions to JSX and state management, but when it comes to building maintainable React applications that scale, the guidance gets thin fast.

This book focuses entirely on that gap. Real React architecture patterns, practical strategies for handling complex state, and techniques that work when you're dealing with production applications instead of todo list examples. I wrote this because I couldn't find a comprehensive resource that treated React development as a serious discipline rather than a collection of scattered tutorials. Most books cover the basics, then jump to advanced topics without bridging the gap. This one stays put and goes deep.

For developers who need to build React applications that last and teams who want to establish solid development practices, you'll find strategies that work in production environments, not just demos.

This book assumes you're smart enough to take what works and leave what doesn't. Read it cover to cover or jump to the chapters that solve your immediate problems. Your choice.

Thomas

Gothenburg, June 2025

Rationale

Preface

Welcome to “The Blue Print: A Journey Into Web Application Development with React”. This comprehensive guide equips you with the knowledge and skills needed to create scalable, maintainable React applications using modern development practices.

Each chapter builds on the previous ones, providing you with a complete education in modern React development—from React fundamentals to advanced topics like performance optimization, testing strategies, state management patterns, and production deployment.

Happy coding!

Thomas

Tip

Why read this book?

This book offers:

- A systematic approach to learning React from fundamentals to production-ready applications
- Real-world examples and practical patterns to solve common development challenges
- Solutions to scaling and architecture problems in modern React development
- Strategies for integrating React into your development workflow effectively

The blueprint approach

Setting the stage

In React development, there's a simple truth: *good architecture is invisible*. When your React application is well-structured, components feel natural, state flows predictably, and new features integrate seamlessly. Conversely, poor architecture makes itself known through difficult debugging sessions, unpredictable behavior, and the dreaded "works on my machine" syndrome.

No matter your experience level with JavaScript or web development, remember that everyone begins as a beginner with React. I too started my journey with many struggles and questions about this seemingly magical library.

One early challenge I faced was understanding the distinction between React's declarative paradigm and the imperative JavaScript I was used to writing. I often wondered why I couldn't simply manipulate the DOM directly and call it a day. Later, we'll explore the advantages of React's component-based architecture, which will clarify this fundamental shift in thinking.

Learning React was already demanding, and understanding its ecosystem made it even more daunting. The new concepts like JSX, virtual DOM, and unidirectional data flow seemed like entering a different world. Moreover, the boundary between React code and plain JavaScript was initially blurry, making it difficult to know when I was "thinking in React" versus falling back to old patterns.

In retrospect, I was fortunate to learn React alongside modern JavaScript fundamentals. Though challenging, this parallel learning provided me with a solid foundation. Early exposure to React's patterns helped me recognize faster than many colleagues how

essential it is to structure applications in a way that promotes reusability, testability, and maintainability. Furthermore, React has consistently helped me organize my thoughts, break down complex UIs into manageable pieces, and plan and prioritize feature development.

The terminology in React development was another source of confusion. Various concepts exist—components, props, state, hooks, context, reducers, and more. Sometimes components are called functional components, other times class components. Some advocate for keeping everything in state, while others recommend lifting state up. Then there's the modern hooks paradigm versus older class-based patterns. And what exactly are higher-order components, render props, and custom hooks? Later chapters will demystify these concepts, providing you with a comprehensive understanding of the React landscape.

If your head is spinning, you're not alone. I recognize you as a curious person; otherwise, you wouldn't have picked up this book despite its technical focus. I pledge to use minimal jargon, and when necessary, explain concepts in the simplest terms possible. However, React development is inherently complex when building real applications, and sometimes the solutions are too. Bear with me as we journey together to elevate your React development skills.

The paradigm shift: imperative to declarative

Before we explore React's technical aspects, it's crucial to understand the fundamental mental shift that React requires. This shift from imperative to declarative thinking is perhaps the most important concept to grasp, and understanding it conceptually will make everything else in this book much clearer.

How we traditionally think about interfaces

Most developers come to React with experience in imperative programming—writing code that explicitly describes *how* to accomplish tasks step by step. When building user interfaces, this typically means:

- Selecting DOM elements directly
- Modifying their properties one by one
- Orchestrating complex sequences of changes
- Managing the current state of each element manually

This approach feels natural because it mirrors how we think about tasks in real life: "First do this, then do that, then check if something happened, and respond accordingly."

React's declarative approach

React asks you to think differently. Instead of describing *how* to change the interface, React wants you to describe *what* the interface should look like at any given moment based on your application's current state.

Tip

Think in snapshots, not steps

Rather than writing instructions for how to transform your interface from one state to another, you describe what your interface should look like for each possible state. React handles the transformation details for you.

This mental shift takes time to internalize, but once mastered, it leads to more predictable and maintainable applications. Instead of tracking all the possible ways your interface might change, you simply describe the desired end result for each scenario.

Why this matters

The declarative approach offers several advantages that become apparent as your applications grow:

Predictability: When you describe what your interface should look like rather than how to change it, it becomes much easier to reason about what will happen in any given situation.

Maintainability: Declarative code is typically easier to understand and modify because each piece describes a clear relationship between data and interface, rather than a complex sequence of transformations.

Debugging: When something goes wrong, you can focus on *what* the interface should show rather than trying to trace through all the *how* instructions that led to the current state.

Reusability: Declarative components naturally become more reusable because they focus on the relationship between input and output rather than specific implementation details.

In Chapter 2, we'll explore this concept in depth with concrete examples that demonstrate the difference between imperative DOM manipulation and React's declarative component approach. For now, keep this fundamental shift in mind as we continue building our conceptual foundation.

The thinking framework

Before we dive into the technical details, it's worth establishing the mental framework that will guide our approach throughout this book. React development isn't just about learning syntax and APIs—it's about developing a way of thinking that leads to maintainable, scalable applications.

Important

Architecture first, implementation second

The most successful React applications start with thoughtful planning, not rushed coding. Taking time to think through component relationships, data flow, and user interactions before writing code will save countless hours of refactoring later.

At its core, effective React development revolves around several key principles that we'll explore throughout this book:

Visual planning: Before writing a single line of code, successful React developers map out their component hierarchy and data flow. This connects directly to declarative thinking—instead of planning *how* to build features step by step, you plan *what* your interface should look like and let React handle the implementation details.

Data flow strategy: Understanding where data lives and how it moves through your application is crucial. React's unidirectional data flow isn't just a technical constraint—it's a design philosophy that makes applications predictable and debuggable.

Component boundaries: Learning to identify the right boundaries for your components is perhaps the most important skill in React development. Components that are too small become unwieldy, while components that are too large become unmaintainable.

Composition over inheritance: React favors composition patterns that allow you to build complex UIs from simple, reusable pieces. This approach leads to more flexible and maintainable code than traditional inheritance-based architectures.

Progressive complexity: Starting simple and adding complexity gradually is not just a learning strategy—it's a development strategy. Even experienced developers benefit from building applications incrementally, validating each layer before adding the next.

These principles will resurface throughout our journey, each time with deeper exploration and practical examples. In Chapter 2, we'll put these concepts into practice with hands-on exercises in component design and architecture planning.

A journey in 10 acts

Setup

Book structure

1. **Introduction and fundamentals** - Core React concepts and development philosophy
2. **Component thinking** - Breaking down UIs into reusable, composable pieces
3. **State and props** - Managing data flow and component communication
4. **Hooks and lifecycle** - Modern React patterns and component behavior
5. **Advanced patterns** - Higher-order components, render props, and composition
6. **Performance optimization** - Making React applications fast and efficient
7. **Testing React components** - Ensuring reliability through comprehensive testing
8. **State management** - Handling complex application state with various tools
9. **Production deployment** - Taking React applications from development to production
10. **The journey continues** - Future directions and continuous learning in React

We'll embark on a journey to enhance your React development skills and empower you to build more maintainable, scalable web applications. React development is a broad topic, and teaching someone new to it presents challenges. It's difficult to discuss one aspect of React without touching on others that might still be beyond your current skillset.

I believe in structure and that practice makes perfect. There's only one way to learn to write good React applications—by building them yourself, not just reading about them or watching tutorials. For this reason, I've divided this book into chapters that guide you through various aspects of React development step-by-step, with each chapter containing examples and exercises I strongly encourage you to complete.

First, we'll explore React's core concepts and their benefits during development. This section contains theory and patterns that need clarification. Though potentially challenging, understanding this foundation is crucial. The React ecosystem is full of specific terminology that often carries different meanings depending on context. I'll do my best to clarify ambiguities and establish a consistent framework for this book.

As we focus on building user interfaces and managing application state, you'll learn the capabilities and limitations of React. With this foundation, we'll dive into the practical aspects of structuring and building React applications for various scenarios. We'll start with simple components with limited complexity, gradually increasing difficulty to tackle more complex applications and architectural challenges.

Along the way, we'll cover a wide range of topics. Chapter 2, "Component Thinking," introduces the fundamental mindset shift required for effective React development. Building on the imperative-to-declarative paradigm shift introduced in this chapter, you'll see concrete examples of how this thinking applies to real interface problems and learn to break down complex user interfaces into small, reusable components that work together harmoniously.

Chapter 3, "State and Props," dives deep into React's data flow patterns. We'll explore how to manage component state effectively, establish clear communication patterns between components through props and callbacks, and handle data fetching and network requests in React applications.

In Chapter 4, "Hooks and Lifecycle," you'll master modern React patterns through hooks while understanding component lifecycle concepts. Through guided exercises, you'll learn to handle side effects, manage complex state, optimize component behavior using React's powerful hooks system, and integrate API calls seamlessly into component lifecycles.

Chapter 5, "Advanced Patterns," takes your skills to the next level with sophisticated techniques for building flexible, reusable components. We'll cover higher-order components, render props, compound components, and composition patterns that enable you to build truly scalable React applications.

Chapter 6, “Performance Optimization,” addresses the challenges of building fast, responsive React applications. You’ll learn to identify performance bottlenecks, implement effective optimization strategies, and ensure your applications remain snappy as they grow in complexity.

Chapter 7, “Testing React Components,” focuses on building confidence in your React applications through comprehensive testing strategies. We’ll cover unit testing, integration testing, and end-to-end testing approaches that ensure your components work correctly in isolation and as part of larger systems.

Chapter 8, “State Management,” explores solutions for handling complex application state that goes beyond what React’s built-in state can handle effectively. We’ll examine various state management libraries and patterns, helping you choose the right approach for your specific needs.

Chapter 9, “Production Deployment,” covers the essential steps for taking your React applications from development to production environments. We’ll discuss build optimization, deployment strategies, monitoring, and maintenance practices that ensure your applications run reliably for users.

Finally, we’ll conclude our journey in Chapter 10, “The Journey Continues.” While our time together ends here, your journey in React development continues. We’ll reflect on the knowledge you’ve gained and discuss the future of React, offering guidance on expanding your expertise in this rapidly evolving ecosystem.

A word of caution

Caution

Different approaches to React development

The React community is diverse, with many valid approaches to building applications. While this book presents patterns that have proven successful in my experience, they are not the only valid approaches. Take what works for you, adapt

techniques to your context, and remember that the ultimate goal is creating applications that deliver value to users.

It's important to acknowledge that React development is a diverse field where professionals employ varied strategies and architectural patterns. Some approaches may align with mine, while others diverge significantly. These variations are natural, as each person and team brings unique experiences, constraints, and requirements.

This book offers a structured approach to a complex topic, allowing you to build on existing knowledge and discover techniques that work for your specific context. However, I emphasize that the perspectives shared here aren't derived from scientific expertise or universal truth, but from my personal experiences and knowledge gained across various React projects and teams. My approach has consistently led to increased developer productivity, better code maintainability, improved team collaboration, and more successful project outcomes.

Remember that every developer's journey is unique. While these strategies have succeeded for me and the teams I've worked with, it's essential to adapt them to your context, project requirements, and team dynamics. As you explore React development, select the best elements from various approaches and incorporate them into your workflow in ways that benefit you and your users most.

The React ecosystem evolves rapidly, and what works today may be superseded by better approaches tomorrow. Stay curious, keep learning, and always be willing to reconsider your assumptions as new patterns and tools emerge.

Component thinking

The shift from imperative to declarative thinking represents one of the most fundamental changes developers must make when learning React. In Chapter 1, we introduced this paradigm shift conceptually. Now we'll explore it through concrete examples and see how it applies to building real React applications.

This chapter focuses on developing the architectural mindset that separates effective React developers from those who struggle with component design and application structure. By the end of this chapter, you'll understand how to break down complex user interfaces into manageable, reusable components and plan data flow patterns that scale.

Tip

What you'll learn in this chapter

- How to transition from imperative DOM manipulation to declarative component composition
- Techniques for identifying optimal component boundaries and responsibilities
- Strategies for planning data flow and component communication patterns
- Methods for organizing code to separate concerns effectively
- A systematic approach to architectural planning before implementation

From DOM manipulation to component composition

Most developers come to React with experience in direct DOM manipulation—selecting elements, modifying their properties, and orchestrating complex interactions through event handlers scattered across their codebase. React asks you to think differently.

Important

Key concept: Imperative vs Declarative programming

Imperative programming describes *how* to accomplish a task step by step. You write explicit instructions: "First do this, then do that, then check this condition, then do something else."

Declarative programming describes *what* you want to achieve, letting the framework handle the *how*. You describe the desired end state and let React figure out how to get there.

Understanding the mental shift

The transition from imperative to declarative thinking affects how you approach every aspect of interface development:

Traditional imperative approach: - Manually select DOM elements - Explicitly modify element properties - Track current state in variables - Write step-by-step transformation logic - Handle all edge cases manually

React's declarative approach: - Describe what the interface should look like for each state - Let React handle DOM updates automatically - Define state as data, not DOM properties - Express transitions through state changes - Trust React to handle edge cases consistently

Important

The declarative shift

Instead of describing *how* to change the interface, React asks you to describe *what* the interface should look like at any given moment. This fundamental shift in thinking takes time to internalize, but once mastered, it leads to more predictable and maintainable applications.

Consider a simple example: showing and hiding a modal dialog. This everyday interaction demonstrates the profound difference between these two approaches.

The imperative way requires you to manually orchestrate each step:

Example

```
// Imperative approach - describing HOW to change the interface
function showModal() {
  document.getElementById("modal").style.display = "block";
  document.getElementById("overlay").classList.add("active");
  document.body.classList.add("no-scroll");
}

function hideModal() {
  document.getElementById("modal").style.display = "none";
  document.getElementById("overlay").classList.remove("active");
  document.body.classList.remove("no-scroll");
}

// Usage requires manual state tracking
let modalIsOpen = false;
button.addEventListener('click', () => {
  if (modalIsOpen) {
    hideModal();
    modalIsOpen = false;
  } else {
    showModal();
    modalIsOpen = true;
  }
});
```

Component thinking

Notice how the imperative approach requires you to: - Manually track the current state (`modalIsOpen`) - Write explicit functions for each transformation - Remember to update all related elements (modal, overlay, body) - Handle the state tracking logic separately from the UI updates

The declarative way describes what the interface should look like:

Example

```
// Declarative approach - describing WHAT the interface should look like
function App() {
  const [isModalOpen, setIsModalOpen] = useState(false);

  return (
    <div className={`app ${isModalOpen ? "no-scroll" : ""}`}>
      <button onClick={() => setIsModalOpen(!isModalOpen)}>
        {isModalOpen ? "Close Modal" : "Open Modal"}
      </button>

      <Modal isOpen={isModalOpen} onClose={() => setIsModalOpen(false)} />
      <Overlay active={isModalOpen} />
    </div>
  );
}

// The Modal component describes what it looks like when open or closed
function Modal({ isOpen, onClose }) {
  if (!isOpen) return null;

  return (
    <div className="modal">
      <div className="modal-content">
        <h2>Modal Title</h2>
        <p>Modal content goes here...</p>
        <button onClick={onClose}>Close</button>
      </div>
    </div>
  );
}
```

In the declarative approach: - State is explicit and centralized (`isModalOpen`) - Components describe their appearance for each state - React handles all DOM updates automatically - The relationship between state and UI is clear and predictable

Tip

Why this matters

As your applications grow, the imperative approach becomes exponentially more complex. You end up with scattered state mutations, unclear dependencies between UI elements, and bugs that are hard to track down. The declarative approach scales naturally because each component simply describes what it should look like given its current props and state.

The compounding benefits

This shift requires rewiring how you think about user interfaces, but the benefits compound quickly as applications grow in complexity:

Predictability: When every component describes what it should look like rather than how to change, it becomes much easier to reason about what will happen in any given situation.

Debugging: Instead of tracing through complex sequences of DOM manipulations, you can look at the current state and immediately see what the interface should display.

Testing: Declarative components are easier to test because you can provide props and state, then verify the rendered output, without needing to simulate complex user interactions.

Reusability: Components that describe their appearance based on props naturally become more reusable across different contexts.

A word about “best practices” and patterns

Before we dive deeper into specific techniques, it’s important to address the elephant in the room: the concept of “best practices” in React development.

Important

Understanding “best practices”

“Best practices” are approaches that many developers commonly use and have found effective in their contexts. However, React itself is deliberately unopinionated—it gives you the tools but doesn’t dictate how to use them. This means there are many valid ways to structure React applications, and what’s “best” depends heavily on your specific situation.

The evolving nature of practices

The React ecosystem evolves rapidly. Practices that were considered “best” two years ago might be discouraged today:

- **Class components** were the standard, then **functional components with hooks** became preferred
- **Higher-order components (HOCs)** were popular, then **render props** emerged, then **custom hooks** became the go-to solution
- **Redux** was the default state management choice, now **Context API** and **Zustand** are often preferred for many use cases

Tip

What this means for you

Don't get too attached to any specific practice as the "one true way." Instead, focus on understanding the underlying principles that make practices effective: clarity, maintainability, performance, and team collaboration.

Patterns vs. principles

In software development, we often encounter two important concepts:

Patterns are reusable solutions to commonly occurring problems. In React, examples include: - Container/Presentation component pattern - Compound components pattern - Higher-order components pattern - Custom hooks pattern

Principles are fundamental guidelines that help us make good decisions. Examples include: - Single Responsibility Principle (each component should do one thing well) - Don't Repeat Yourself (DRY) - Separation of Concerns - Composition over Inheritance

Note

Patterns change, principles endure

While specific patterns may fall in and out of favor, solid principles tend to remain valuable across different technologies and time periods. Understanding why a pattern works (the principle behind it) is more valuable than memorizing the pattern itself.

Context matters: choosing the right approach

Your architectural choices should be influenced by your specific context:

Working alone vs. team development: - Solo projects allow for more personal preferences and rapid iteration - Team projects require consistent conventions and clear communication patterns - Team size affects how much abstraction and documentation you need

Project timeline and scope: - **Prototypes/MVPs**: Prioritize speed and flexibility over perfect architecture - **Long-term applications**: Invest in maintainability, testing, and clear patterns - **Short-term projects**: Simpler approaches often work better than complex architectures

Team experience level: - **Beginner teams**: Benefit from well-established patterns and conventions - **Experienced teams**: Can handle more sophisticated architectures and custom solutions - **Mixed experience**: Need clear documentation and consistent patterns

Application complexity: - **Simple applications**: Don't over-engineer with complex state management - **Complex applications**: Invest in proper architecture and tooling - **Growing applications**: Plan for scalability but don't optimize prematurely

Example

Context-driven decisions in practice

Scenario 1: Solo developer, 2-week prototype - Use simple local state and prop drilling - Minimal abstraction, focus on functionality - Direct API calls in components are fine

Scenario 2: 5-person team, 2-year product - Establish clear component patterns and naming conventions - Implement proper separation between data fetching and presentation - Use consistent error handling and loading states - Document architectural decisions

Scenario 3: Large team, enterprise application - Implement strict component patterns and code organization - Use TypeScript for better collaboration and maintainability - Establish testing standards and CI/CD processes - Create reusable component libraries

What this book provides

The approaches presented in this book represent **one effective way** to structure React applications—approaches that have proven successful in various professional contexts. They're not the only way, and they may not be the best way for your specific situation.

Caution

Take what works, leave what doesn't

As you read through the patterns and techniques in this book, consider them through the lens of your own context. Some practices might be perfect for your situation, others might be overkill, and some might not fit your constraints at all. The goal is to build your understanding so you can make informed decisions about what works for you.

Our focus: We emphasize approaches that tend to work well for: - Teams building applications that need to be maintained over time - Developers who want clear, predictable patterns - Applications that are expected to grow in complexity - Contexts where code quality and maintainability matter

If you're building a quick prototype or working in a very different context, you might choose different approaches—and that's perfectly valid.

Building your judgment

The real skill in React development isn't memorizing the "right" patterns—it's developing the judgment to choose appropriate solutions for your context. This comes from:

1. **Understanding the principles** behind different approaches
2. **Experiencing the consequences** of different architectural decisions
3. **Learning from the community** while forming your own opinions
4. **Staying curious** about new approaches without chasing every trend

As we explore the techniques in this book, we'll try to explain not just *what* to do, but *why* these approaches work and *when* you might choose alternatives.

The architecture-first mindset

Before writing any component code, successful React developers engage in what we call "architectural thinking." This involves mapping out component relationships, data flow, and interaction patterns before implementation begins.

Important

Definition: Architectural thinking

Architectural thinking in React development means deliberately planning your component structure, data flow, and interaction patterns before writing code. It's the practice of designing your application's structure at a high level to ensure scalability, maintainability, and clear separation of concerns.

Many developers skip this planning phase and jump straight into coding, which often leads to: - Components that are too large and try to do too much - Confusing data flow patterns that are hard to debug - Tight coupling between components that should be independent - Difficulty adding new features without breaking existing functionality

Why architecture-first matters

The architecture-first approach provides several critical advantages:

Prevents refactoring cycles: Good upfront planning eliminates the need for major structural changes later when requirements become clearer.

Reveals complexity early: Planning exposes potential problems when they're cheap to fix, not after you've written thousands of lines of code.

Enables team collaboration: Clear architectural plans help team members understand how pieces fit together and where to make changes.

Improves code quality: When you know where each piece of functionality belongs, you write more focused, single-purpose components.

Visual planning exercises

The most effective way to develop architectural thinking is through visual planning. Take a whiteboard, paper, or digital tool and practice breaking down interfaces into components.

Tip

Recommended tools for visual planning

- **Physical tools**: Whiteboard, paper and pencil, sticky notes
- **Digital tools**: Figma, Sketch, draw.io, Miro, or even simple drawing apps
- **The key**: Use whatever feels natural and allows quick iteration

Example

Exercise: component identification

Visit a popular website (like GitHub, Twitter, or Medium) and practice identifying potential React components. Draw boxes around distinct pieces of functionality and consider:

- What data does each component need?
- How do components communicate with each other?
- Which components could be reused in other parts of the application?
- Where should state live for each piece of data?

Example walkthrough: Looking at a Twitter-like interface, you might identify: - `Header` component (logo, navigation, user menu) - `TweetComposer` component (text area, character count, post button) - `Feed` component (container for tweet list) - `Tweet` component (avatar, content, actions, timestamp) - `Sidebar` component (trends, suggestions, ads)

Each component has clear boundaries and responsibilities, making the overall application easier to understand and maintain.

The component hierarchy principle

Every React application is a tree of components, and understanding this hierarchy is crucial for effective architecture. When planning your component structure, consider these guidelines:

Note

Definition: Component hierarchy

The component hierarchy is the tree-like structure that describes how components are nested within each other. Just like HTML elements form a DOM tree, React components form a component tree where parent components contain and manage child components.

Single responsibility principle: Each component should have one clear purpose. If you find yourself struggling to name a component or describing its purpose requires multiple sentences, it likely needs to be broken down further.

Example

Good component responsibilities: - `UserCard` - displays user information - `SearchBar` - handles search input and triggers search - `ProductList` - renders a list of products - `ShoppingCart` - manages cart items and checkout

Poor component responsibilities: - `UserDashboard` - displays user info, handles search, shows products, manages cart, and processes checkout (too many responsibilities)

Data ownership: Components should own the data they need to function. When data needs to be shared between components, it should live in their closest common ancestor.

Tip

The principle of data ownership

Data should live in the component that: 1. Needs to modify that data, OR 2. Is the closest common ancestor of all components that need that data

This principle helps prevent prop drilling (passing props through many levels) and keeps your data flow predictable.

Reusability consideration: While not every component needs to be reusable, thinking about reusability during design often leads to better component boundaries and cleaner interfaces.

Common hierarchy patterns

Successful React applications often follow similar hierarchical patterns:

Container/Presentation pattern: Separate components that manage data (containers) from components that display data (presentational).

Feature-based grouping: Group related components together that work toward the same user goal.

Composition over inheritance: Build complex components by combining simpler ones rather than extending base classes.

Component boundaries and responsibilities

Identifying the right boundaries for your components is perhaps the most critical skill in React development. Components that are too small create unnecessary complexity, while components that are too large become difficult to understand and maintain.

Important

The Goldilocks principle for components

Good components are “just right” - not too big, not too small. They handle a cohesive set of functionality that makes sense to group together, without trying to do too much or too little.

Signs of poor component boundaries

Components that are too large exhibit these warning signs: - Difficult to name clearly and concisely - Handle multiple unrelated concerns - Have too many props (typically more than 5-7) - Are hard to test because they do too much - Require significant scrolling to read through the code

Components that are too small create these problems: - Excessive prop drilling between parent and child - No clear benefit from the separation - Difficult to understand the overall functionality - Create unnecessary rendering overhead

Example

Too large - UserDashboard component:

```
function UserDashboard() {
  // Manages user profile data
  const [user, setUser] = useState(null);
  // Manages notification settings
  const [notifications, setNotifications] = useState([]);
  // Handles billing information
  const [billingInfo, setBillingInfo] = useState(null);
```

```
  // Manages account settings
  const [settings, setSettings] = useState({});

  // 200+ lines of mixed functionality...

  return (
    <div>
      {/* Profile section */}
      {/* Notifications section */}
      {/* Billing section */}
      {/* Settings section */}
    </div>
  );
}
```

Better - Separated components:

```
function UserDashboard() {
  return (
    <div className="dashboard">
      <UserProfile />
      <NotificationCenter />
      <BillingPanel />
      <AccountSettings />
    </div>
  );
}
```

The rule of three levels

A useful heuristic for component boundaries is the “rule of three levels”:

1. **Presentation level**: Components that focus purely on rendering UI elements
2. **Container level**: Components that manage state and data flow

3. **Page level**: Components that orchestrate entire application sections

Note

Understanding the three levels

Presentation components (also called “dumb” or “stateless” components): - Receive data via props - Focus on how things look - Don’t manage their own state (except for UI state like form inputs) - Are highly reusable

Container components (also called “smart” or “stateful” components): - Manage state and data fetching - Focus on how things work - Provide data to presentation components - Handle business logic

Page components: - Coordinate multiple features - Handle routing and navigation - Manage application-level state - Compose container and presentation components

Example

Three-level example - User profile feature:

```
// PAGE LEVEL - coordinates the entire profile page
function UserProfilePage({ userId }) {
  return (
    <div className="profile-page">
      <Header />
      <UserProfileContainer userId={userId} />
      <UserActivityContainer userId={userId} />
      <Footer />
    </div>
  );
}

// CONTAINER LEVEL - manages data and state
function UserProfileContainer({ userId }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchUser(userId)
      .then(setUser)
      .finally(() => setLoading(false));
  }, [userId]);

  if (loading) return <LoadingSpinner />;

  return <UserProfile user={user} onUpdate={setUser} />;
```

```
}

// PRESENTATION LEVEL - focuses on display
function UserProfile({ user, onUpdate }) {
  return (
    <div className="user-profile">
      <Avatar src={user.avatar} alt={user.name} />
      <h1>{user.name}</h1>
      <ContactInfo email={user.email} phone={user.phone} />
      <EditButton onClick={() => onUpdate(user)} />
    </div>
  );
}
```

Data flow patterns

Understanding how data flows through your component hierarchy is essential for good architecture. React's unidirectional data flow means data flows down through props and actions flow up through callbacks.

Important

Definition: Unidirectional data flow

In React, data flows in one direction: from parent components to child components through props. When child components need to communicate with parents, they do so through callback functions passed down as props. This creates a predictable pattern where data changes originate at a known source and flow downward.

Data flows down: Parent components pass data to children through props. **Actions flow up**: Children communicate with parents through callback functions.

Example

Data flow in action:

```
// Parent component - owns the data
```

```
function ShoppingCart() {
  const [items, setItems] = useState([]);
  const [total, setTotal] = useState(0);

  const addItem = (item) => {
    setItems([...items, item]);
    setTotal(total + item.price);
  };

  const removeItem = (itemId) => {
    const newItems = items.filter(item => item.id !== itemId);
    setItems(newItems);
    setTotal(newItems.reduce((sum, item) => sum + item.price, 0));
  };

  return (
    <div>
      {/* Data flows down via props */}
      <CartItems
        items={items}
        onRemoveItem={removeItem}  // Callback flows down
      />
      <CartTotal total={total} />
      <AddItemForm onAddItem={addItem} />  // Callback flows down
    </div>
  );
}

// Child component - receives data and callbacks
function CartItems({ items, onRemoveItem }) {
  return (
    <div>
      {items.map(item => (
        <CartItem
          key={item.id}
          item={item}
          onRemove={() => onRemoveItem(item.id)}  // Action flows up
        />
      ))}
    </div>
  );
}
```

Tip

The 2-3 level rule

If you find yourself passing props more than 2-3 levels deep, consider whether your component hierarchy needs restructuring or whether you need state management tools like Context or external libraries.

Prop drilling occurs when you pass props through multiple component levels just to get data to a deeply nested child. This is a sign that your component structure might need adjustment.

When to break the rules

While unidirectional data flow is React's default pattern, there are legitimate cases where you might need alternative approaches:

Context API: For data that many components need (like user authentication, theme settings) **State management libraries**: For complex applications with intricate state relationships **Custom hooks**: For sharing stateful logic between components **Refs**: For imperative DOM access (though use sparingly)

Thinking about data sources

No modern React application exists in isolation. Your components will need to fetch data from APIs, submit forms to servers, and handle real-time updates. Understanding how to architect data fetching early in your component planning process is crucial.

The resource pattern

When designing React applications that interact with APIs, thinking in terms of "resources" provides a clean abstraction. A resource represents a collection of related data and the operations you can perform on it.

Consider this resource pattern for managing user data:

Example

```
// src/resources/User.js
import { protectedRoute } from "../network/apiConfig";

const User = {
  // Fetch all users
  async index() {
    try {
      const response = await protectedRoute.get("/users");
      return response.data;
    } catch (error) {
      console.error("Failed to fetch users:", error.message);
      throw new Error("Unable to fetch user data. Please try again later.");
    }
  },

  // Fetch a specific user by ID
  async show(id) {
    try {
      const response = await protectedRoute.get(`/users/${id}`);
      return response.data;
    } catch (error) {
      console.error(`Failed to fetch user with ID ${id}:`, error.message);
      throw new Error(`Unable to fetch user data for ID ${id}.`);
    }
  },

  // Create a new user
  async create(userData) {
    try {
      const response = await protectedRoute.post("/users", userData);
      return response.data;
    } catch (error) {
      console.error("Failed to create user:", error.message);
      throw new Error("Unable to create user. Please try again later.");
    }
  },

  // Update an existing user
  async update(id, userData) {
    try {
      const response = await protectedRoute.put(`/users/${id}`, userData);
```

```
      return response.data;
    } catch (error) {
      console.error(`Failed to update user with ID ${id}:`, error.message);
      throw new Error(`Unable to update user with ID ${id}.`);
    }
  },

  // Delete a user
  async destroy(id) {
    try {
      await protectedRoute.delete(`/users/${id}`);
    } catch (error) {
      console.error(`Failed to delete user with ID ${id}:`, error.message);
      throw new Error(`Unable to delete user with ID ${id}.`);
    }
  },
};

export default User;
```

Organizing network code

A well-structured React application separates network concerns into dedicated modules:

Example

```
src/
|-- components/
|-- resources/
|   |-- User.js
|   |-- PracticeSession.js
|   '-- Repertoire.js
|-- network/
|   |-- apiConfig.js
|   |-- interceptors.js
|   '-- errorHandling.js
'-- utils/
    |-- timing.js
    '-- musicNotation.js
```

This organization keeps API logic separate from component logic, making both easier to test and maintain.

Data fetching in components

When planning component architecture, consider how each component will interact with data:

- **Data-fetching components**: Components responsible for loading data from APIs
- **Data-displaying components**: Pure presentation components that receive data via props
- **Data-mutating components**: Components that handle form submissions and data updates

Example

```
// Data-fetching component
function SessionList() {
  const [sessions, setSessions] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    PracticeSession.index()
      .then(setSessions)
      .finally(() => setLoading(false));
  }, []);

  if (loading) return <LoadingSpinner />;

  return (
    <div className="session-list">
      {sessions.map((session) => (
        <SessionCard key={session.id} session={session} />
      ))}
    </div>
  );
}

// Data-displaying component
function SessionCard({ session }) {
```

```
  return (
    <div className="session-card">
      <h3>{session.piece}</h3>
      <p>Duration: {session.duration} minutes</p>
      <p>Focus: {session.focus}</p>
      <span className="practice-date">{session.date}</span>
    </div>
  );
}
```

Important

Separation of concerns

Keep network logic separate from component logic. Components should focus on rendering and user interaction, while resource modules handle API communication. This separation makes your code more testable and maintainable.

We'll explore data fetching patterns, error handling, and state management for network requests in greater detail in Chapter 3, "State and Props," and Chapter 8, "State Management."

Building your first component architecture

Let's put these principles into practice by architecting a real application. We'll design a music practice tracker that demonstrates proper component thinking.

Important

Focus on thinking, not implementation

In this section, we're focusing purely on architectural planning and component design thinking. We won't be writing code to build this application—instead, we're

practicing the mental framework that comes before implementation. This same music practice tracker example may reappear in later chapters when we explore specific implementation techniques.

Planning phase

Before writing code, let's map out our application:

User interface requirements:

- Practice session log with filtering and search
- Session creation form with timer functionality
- Individual practice entries with editing capabilities
- Progress dashboard and statistics
- Repertoire management (pieces being practiced)
- Goal setting and tracking

Data requirements:

- Fetch practice sessions from API
- Create new practice sessions
- Update existing sessions and progress notes
- Delete practice entries
- Manage repertoire (add/remove pieces)
- Track practice goals and achievements

Component identification exercise:

1. Draw the complete interface
2. Identify distinct functional areas
3. Determine data requirements for each area
4. Map component relationships
5. Plan data flow paths
6. Identify which components need network access

Tip

Practice makes perfect

The goal here is to develop your architectural thinking skills. Try sketching out the interface on paper or using a digital tool. Focus on breaking down the problem into logical pieces rather than worrying about perfect solutions.

Component responsibility mapping

For our music practice tracker, we might identify these components:

Example

```
PracticeApp
|-- Header
|   |-- Logo
|   '-- UserProfile
|-- PracticeDashboard
|   |-- PracticeStats (fetches statistics)
|   '-- PracticeFilters
|-- SessionList (fetches practice sessions)
|   '-- SessionItem[] (updates individual sessions)
|-- RepertoirePanel
|   |-- PieceCard[] (displays practice pieces)
|   '-- AddPieceForm
'-- SessionForm (creates new practice sessions)
```

Each component has clear responsibilities:

- `PracticeApp`: Application state and data coordination
- `PracticeDashboard`: Filtering, statistics, and goal tracking
- `SessionList`: Session rendering, list management, and data fetching
- `SessionItem`: Individual session behavior and progress updates
- `RepertoirePanel`: Managing pieces being practiced
- `SessionForm`: Practice session creation with timer functionality

Resource planning:

Example

```
// src/resources/PracticeSession.js
const PracticeSession = {
  async index(filters = {}) {
    /* fetch filtered practice sessions */
  },
  async show(id) {
    /* fetch single practice session */
  },
  async create(sessionData) {
    /* create new practice session */
  },
  async update(id, sessionData) {
    /* update session notes and progress */
  },
  async destroy(id) {
    /* delete practice session */
  },
  async getStats(dateRange) {
    /* fetch practice statistics */
  },
};

// src/resources/Repertoire.js
const Repertoire = {
  async index() {
    /* fetch user's repertoire */
  },
  async create(pieceData) {
    /* add new piece to repertoire */
  },
  async update(id, pieceData) {
    /* update piece details or progress */
  },
  async destroy(id) {
    /* remove piece from repertoire */
  },
};
```

Note

Architectural thinking in action

Notice how we've broken down a complex application into manageable pieces without writing a single line of React code. This planning phase is where good React applications are really built—the implementation is just translating these architectural decisions into code. We'll explore how to implement these patterns in subsequent chapters.

Common architectural patterns

As you develop more React applications, certain patterns emerge repeatedly. Understanding these patterns helps you make better architectural decisions.

Container and presentation pattern

Separating components that manage state (containers) from components that render UI (presentation) creates cleaner, more testable code.

Compound components pattern

For complex UI elements like modals, dropdowns, or tabs, compound components allow you to create flexible, composable interfaces.

Higher-order component pattern

When you need to share logic between components, higher-order components provide a powerful abstraction mechanism.

Practical exercises

::: setup

Setup requirements

For the following exercises, you'll need:

- Node.js installed on your system
- A code editor (VS Code recommended)
- Basic familiarity with ES6+ Java

State and props

Note

Empty in preview version

Hooks and lifecycle

Note

Empty in preview version

Advanced patterns

Note

Empty in preview version

Performance optimization

Note

Empty in preview version

Testing React components

Note

Empty in preview version

State management

Note

Empty in preview version

Production deployment

> **Note**
>
> Empty in preview version

The journey continues

> **Note**
>
> Empty in preview version