The Blue Print

A Journey Into Web Application Development with React

Thomas Ochman

Table of Contents

Introduction and Fundamentals

The Blueprint Approach

The Reality of Software Development

There's a saying about Texas Hold'em poker: "It takes five minutes to learn and a lifetime to master." The same could be said about software development in general, and React in particular. You can write your first React component in minutes, but mastering the craft of building maintainable, scalable applications is a journey measured in years, not weeks.

This isn't meant to discourage you—it's meant to set realistic expectations. Software development is fundamentally complex because it's not just about programming languages. It's equally about libraries, tools, methodologies, collaboration, specifications, patterns, principles, and much more. When you're building applications, you're solving complex problems for real people in an ever-changing technological landscape.

The complexity isn't a bug; it's a feature. But here's the crucial point: **hard doesn't mean impossible**. It means we need to approach learning thoughtfully and systematically.

A Learning Philosophy for Complex Systems

Over years of teaching software development and mentoring programmers, I've noticed that successful developers—whether they're learning React, database design, or

system architecture—tend to follow similar patterns in how they approach complex topics.

First, they cultivate genuine interest. Whether you're learning application frameworks, infrastructure concepts, or programming paradigms, motivation makes the difference between superficial copying and deep understanding. That motivation often comes from wanting to solve real problems—maybe you're frustrated with the limitations of your current tools, or you have an application idea that excites you.

Second, they understand scope before diving into details. Before jumping into syntax and APIs, successful learners ask: What is this technology really about? How does it fit into the broader ecosystem? What's the landscape I'm entering? This prevents getting lost in implementation details without understanding the bigger picture.

Third, they engage actively through multiple channels. Learning complex systems requires more than just reading documentation. You need to build things (there's no substitute for hands-on practice), study how others approach problems, and discuss concepts with other developers. Teaching others is particularly powerful—it forces you to understand concepts deeply enough to explain them.

Fourth, they reflect constantly. After learning sessions, they think about how new concepts connect to existing knowledge, ask "what if" questions about different approaches, and consider trade-offs. For every horizon you reach in software development, another one appears. This forward-thinking mindset keeps you growing.

Finally, they embrace deliberate repetition. Not mindless copying, but returning to fundamental concepts with deeper understanding, practicing problem-solving patterns in different contexts, and revisiting challenging topics from new angles as knowledge expands.

Setting the Stage: Architecture as Foundation

In building applications—whether they use React, Vue, Angular, or any other framework—there's a simple truth: *good architecture is invisible*. When your application is well-structured, components feel natural, data flows predictably, and new features integrate seamlessly. Poor architecture makes itself known through difficult debugging sessions, unpredictable behavior, and the dreaded "works on my machine" syndrome.

Most developers come to React with existing web development experience, but React requires a fundamental shift in thinking. The transition from imperative DOM manipulation to declarative component composition represents one of the most challenging—and rewarding—conceptual leaps in modern development.

Understanding the Learning Curve

React introduces several concepts that may feel foreign at first: JSX syntax, component lifecycle, unidirectional data flow, and the virtual DOM. The boundary between React-specific patterns and regular JavaScript can initially feel blurry. This confusion is normal and temporary—by the end of this book, these concepts will feel natural.

React's ecosystem includes a rich vocabulary of terms: components, props, state, hooks, context, reducers, and more. You'll encounter functional components, class components, higher-order components, render props, and custom hooks. Rather than overwhelming you with definitions upfront, we'll introduce these concepts gradually as they become relevant to your understanding.

This book takes a practical approach: we'll explore concepts through concrete examples and build your understanding incrementally. Each chapter assumes you're intelligent enough to adapt the patterns to your specific context while providing clear guidance on proven approaches.

The Paradigm Shift: From Imperative to Declarative

Before we dive into React's technical details, let's discuss a fundamental shift that applies to modern application development more broadly. This mental transition affects not just how you write code, but how you think about building complex systems.

Most of us come to modern frameworks from a world where we tell computers exactly what to do, step by step. "Get this element, change its text, add a class, remove another element, show this thing, hide that thing." It's very procedural, very explicit, and it feels natural because that's how we approach most tasks in life.

Modern application frameworks—React included—ask you to flip that thinking. Instead of saying "here's how to change the interface," these tools want you to say "here's what the interface should look like right now." It's like the difference between giving someone turn-by-turn directions versus showing them the destination on a map and letting GPS figure out the route.

Traditional Approaches to Interfaces

Example

```
// Traditional imperative approach - step-by-step instructions
function incrementCounter() {
  const counter = document.getElementById('counter');
  const currentValue = parseInt(counter.textContent);
  counter.textContent = currentValue + 1;

  if (currentValue + 1 > 10) {
    counter.classList.add('warning');
  }
}
```

This is imperative programming—you're giving explicit instructions for what needs to happen, when it needs to happen, and how it should happen.

The Declarative Alternative

Example

```
// Declarative approach - describe the desired outcome
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div className={count > 10 ? 'warning' : ''}>
      {count}
      <button onClick={() => setCount(count + 1)}>+</button>
    </div>
  );
}
```

See the difference? Instead of telling the system how to update things, you describe what the end result should look like. The framework handles the transformation details.

Why This Mental Shift Matters

This declarative approach pays dividends as applications grow in complexity:

- **Predictability**: When you can look at code and immediately understand what it will produce for any given input, debugging becomes much easier
- **Maintainability**: Changes become localized and safer when you're describing outcomes rather than procedures
- **Composability**: Declarative pieces naturally combine in predictable ways
- **Testability**: You can verify outcomes directly rather than simulating complex sequences of actions

I'll be honest—this shift doesn't happen overnight. For the first few weeks with any declarative framework, you might find yourself fighting against this approach, trying to control every detail imperatively. That's completely normal. But once this pattern clicks, you'll wonder how you ever built complex systems any other way.

Rationale

Let me give you some context about where React came from, because understanding its origins helps explain why it works the way it does. React wasn't born in a vacuum—it was Facebook's answer to very real, very painful problems they were facing with their user interfaces.

Picture this: it's 2011, and Facebook's interface is getting more complex by the day. Users are posting, commenting, liking, sharing, messaging—and all of these actions need to update multiple parts of the interface simultaneously. The old approach of manually manipulating the DOM for each change was becoming a nightmare.

The specific problem that sparked React's creation was deceptively simple: the notification counter. You know, that little red badge that shows you have new messages? It seemed straightforward enough, but in a complex application, that counter might need to update when you receive a message, read a message, delete a message, or when someone comments on your post. Keeping track of all the places that counter needed to update, and ensuring they all stayed in sync, was driving engineers crazy.

React emerged as their solution to this coordination chaos. Instead of manually tracking every possible update, what if the interface could automatically reflect the current state of the data? What if you could describe what the interface should look like, and React would figure out what needed to change?

The Core Problems React Solved

React wasn't created by academics in a lab—it was born from the frustration of trying to build complex, interactive interfaces with traditional DOM manipulation. Every React pattern and principle exists because it solved a real problem that developers were actually facing:

Coordination chaos: When multiple parts of an interface needed to update in response to one change, keeping everything in sync manually was error-prone and exhausting.

Performance bottlenecks: Frequent DOM manipulations were slow, and optimizing them manually required extensive effort and expertise.

Code complexity: As applications grew, the imperative code required to manage interface updates became an unmaintainable mess.

Reusability struggles: Creating truly reusable interface components with traditional approaches was like trying to build LEGO sets that only worked in one specific configuration.

React as Library, Not Framework

React is often called a "library" rather than a "framework," and this distinction matters for how you approach building applications. React focuses specifically on building user interfaces and managing component state. Unlike frameworks that provide opinions about routing, data fetching, project structure, and build tools, React leaves these decisions to you and the broader ecosystem.

What this means in practice: - React provides the core tools for building components and managing their behavior - You choose your own routing solution (React Router, Next.js routing, etc.) - You decide how to handle data fetching (fetch API, Axios, React Query, etc.) - You select your preferred styling approach (CSS modules, styled-components, Tailwind, etc.) - You configure your own build tools (Vite, Create React App, custom Webpack, etc.)

This library approach offers flexibility to choose the best tools for your specific needs, but it also means more decisions to make and a need to understand how different pieces work together.

React in the Component Ecosystem

React popularized component-based architecture for web applications, but it's part of a broader movement toward this approach. Understanding React's place in this ecosystem helps contextualize its patterns and principles.

Other component-based approaches include Vue.js (more framework-like with built-in solutions), Angular (full framework with strong opinions), Svelte (compiles to optimized JavaScript), and Web Components (browser-native standards). React's influence on this ecosystem has been significant—many patterns that originated in React have been adopted by other libraries, and React itself has evolved by incorporating ideas from the broader community.

Note

Why this context matters

Understanding that React is one approach among many helps you make informed decisions about when to use it and how to combine it with other tools. The patterns we'll explore in this book aren't exclusively React-specific—many translate to other component-based approaches and general application architecture principles.

The Thinking Framework for React Development

Building with React isn't just about learning syntax and APIs—it's about developing a way of thinking that leads to maintainable, scalable applications. The learning approach we discussed earlier applies directly to mastering React's patterns and principles.

Important

A note on "best practices"

Throughout this book, you'll encounter various approaches and patterns often called "best practices." It's important to understand that React itself is deliberately unopinionated—it provides tools but doesn't dictate how to use them. What

works best depends heavily on your specific context: team size, project requirements, timeline, and experience level.

At its core, effective React applications revolve around several key principles:

Architecture first, implementation second: The most successful applications start with thoughtful planning, not rushed coding. Taking time to think through component relationships, data flow, and user interactions before writing code saves countless hours of refactoring later.

Visual planning: Before writing code, successful 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 both a learning strategy and 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.

How This Book Supports Your Learning Journey

This book is designed around the learning principles we discussed—starting with genuine interest by showing you what becomes possible when you master React's patterns, providing clear scope for each concept and how it connects to the bigger picture, encouraging active engagement through examples and exercises, building in reflection opportunities to consider alternative approaches and trade-offs, and allowing repetition as concepts reappear in different contexts to build deeper understanding.

Important

Your learning journey is unique

While this framework has proven effective for many developers, adapt it to your own learning style and context. The goal isn't to follow a rigid formula, but to approach React learning with intention and strategy.

React: From DOM Manipulation to Component Composition

Remember that mental shift from imperative to declarative thinking? Well, now we're going to see it in action. This is where React starts to feel different from any JavaScript you've written before, and honestly, this is where a lot of developers either fall in love with React or get really frustrated with it.

I want to be upfront with you: this section is going to change how you think about building user interfaces. We're not just learning new syntax or API calls—we're developing a completely different approach to organizing and structuring interactive applications. It's the difference between thinking like a micromanager who controls every detail and thinking like an architect who designs systems that work elegantly on their own.

The good news? Once you get this mindset, building complex interfaces becomes way more enjoyable. The not-so-good news? It might feel uncomfortable at first if you're used to having direct control over every DOM element and every interaction.

Understanding the Mental Shift

Let me show you what I mean with a real example. Most of us come to React from a world where building interactive UIs meant a lot of `getElementById`↩ ↪ , `addEventListener`, and manually updating element properties. It's very hands-on, very explicit, and it gives you the illusion of total control.

But here's the thing—that control comes at a massive cost when your application grows beyond a few simple interactions.

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.

Let me give you a concrete example that illustrates this shift. Say you're building a simple modal dialog—you know, one of those popup windows that appears over your main content.

In traditional JavaScript, your brain thinks like this: "When someone clicks the open button, I need to find the modal element, add a class to make it visible, probably add an overlay, maybe animate it in, add an event listener to close it when they click outside…" You're thinking in terms of a sequence of actions.

Example

The old way: imperative thinking

```
// Traditional imperative approach - lots of manual steps
function openModal() {
  const modal = document.getElementById('modal');
  const overlay = document.getElementById('overlay');

  modal.style.display = 'block';
  overlay.style.display = 'block';
  modal.classList.add('modal-open');

  // Add event listeners
```

```
  overlay.addEventListener('click', closeModal);
  document.addEventListener('keydown', handleEscape);

  // Prevent body scroll
  document.body.style.overflow = 'hidden';
}

function closeModal() {
  // Reverse all the steps above...
  const modal = document.getElementById('modal');
  const overlay = document.getElementById('overlay');

  modal.classList.remove('modal-open');
  modal.style.display = 'none';
  overlay.style.display = 'none';

  // Clean up event listeners
  overlay.removeEventListener('click', closeModal);
  document.removeEventListener('keydown', handleEscape);

  // Restore body scroll
  document.body.style.overflow = '';
}
```

Look at all those steps! And that's just for a simple modal. Imagine if you have multiple modals, or nested modals, or modals with different behaviors. The complexity explodes quickly.

React asks you to think differently:

Example

The React way: declarative thinking

```
// React's approach - describe WHAT it 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)} />
      {isModalOpen && <Overlay onClick={() => setIsModalOpen(false)} />}
    </div>
  );
}

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>
  );
}
```

See the difference? In the React version, I'm not telling the browser how to open the modal step by step. Instead, I'm saying "here's what the UI should look like when the modal is open, and here's what it should look like when it's closed." React figures out all the DOM manipulation details.

This felt really weird to me at first. My initial reaction was "but I want control over exactly how things happen!" But here's what I discovered: you don't actually want that control. What you want is predictable, maintainable code. And the declarative approach gives you that in spades.

Tip

Why this matters

Once you start building anything more complex than a simple modal, the imperative approach becomes a nightmare to manage. You end up with state scattered everywhere, complex interdependencies, and bugs that are incredibly hard to

track down. The declarative approach scales beautifully because each component just describes what it should look like, period.

The Compounding Benefits

I know this mental shift feels strange if you're used to having direct control over the DOM. But stick with me here, because the benefits compound quickly:

Your code becomes predictable: When you can look at a component and immediately understand what it will render for any given state, debugging becomes so much easier.

Testing gets simpler: Instead of simulating complex user interactions and DOM manipulations, you just pass props to a component and verify what it renders.

Reusability happens naturally: When components describe their appearance based on props, they automatically become more flexible and reusable.

Bugs become obvious: Most React bugs happen because your state doesn't match what you think it should be. With declarative components, the relationship between state and UI is explicit and easy to trace.

I remember the moment this clicked for me. I was building a complex form with conditional fields, validation states, and dynamic sections. In traditional JavaScript, it would have been a mess of event handlers and DOM manipulation. But with React's declarative approach, each part of the form just described what it should look like based on the current data. It was like magic.

Component Boundaries and Responsibilities

Identifying the right boundaries for your components is perhaps the most critical skill when building React applications. 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.

The Architecture-First Mindset

Effective React applications begin not with code, but with thoughtful planning. Before writing any component code, experienced developers engage in what we call "architectural thinking"—the practice of mapping out component relationships, data flow, and interaction patterns before implementation begins.

Important

Definition: Architectural thinking

Architectural thinking is the deliberate practice of designing your application's structure before writing code. It involves:

- **Component planning**: Identifying what components you need and how they relate to each other
- **Data flow design**: Determining where state lives and how information moves through your application

- **Responsibility mapping**: Deciding which components handle which concerns
- **Integration strategy**: Planning how different parts of your application will work together

This upfront planning ensures 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, and 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.

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

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

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.