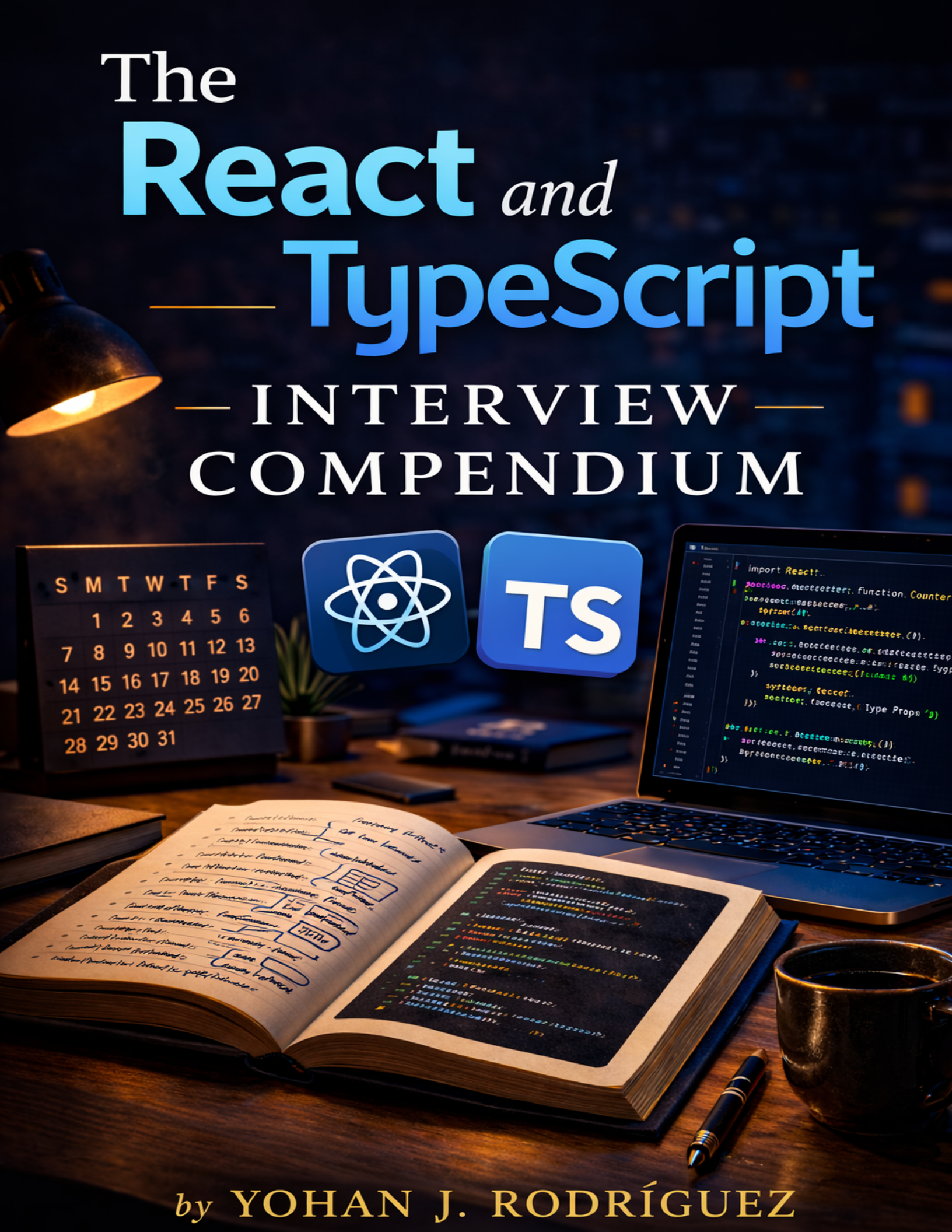
The React and TypeScript

INTERVIEW COMPENDIUM



by YOHAN J. RODRÍGUEZ

Preface

"First, solve the problem. Then, write the code."
John Johnson

React and TypeScript interviews reward precise mental models more than API trivia. A candidate can name every hook and still fail to explain why a form resets when a section is hidden, why a counter increments once instead of twice, or why an input typed into one row shows up under another after a sort. The same candidate may write a confident type for an API response and not notice that the compiler never sees the network. Interviewers use these gaps on purpose. They want to hear you describe what React does when a component runs, who owns a piece of state, where a type stops protecting you, and what you would measure before you optimized anything.

This book is for developers who already know the basics of React and TypeScript and want sharper interview answers. It is written for mid-level and senior interviews, and for engineers who need to explain decisions they already make at work. It is not a first tutorial. It contains more than a thousand questions across fifteen chapters, and each one is tied to the reasoning an interviewer is listening for: what can go wrong, which tradeoff matters, which type boundary prevents a bug, and how to move from a small code example to a production design decision.

The book has five parts. **Part I** lays the foundation: the React rendering model, including render versus commit, reconciliation, keys, and state identity, and then TypeScript for React developers, from `unknown` and discriminated unions to generics and `satisfies`. **Part II** covers React core: typing props and designing component APIs, state and events, effects and asynchronous flow, custom hooks, and forms with validation and accessibility. **Part III** moves to state, architecture, and data, with context and reducers, routing and application structure, server state and caching, and styling and design systems. **Part IV** covers quality, performance, and operations: measured optimization, testing strategy, and security, deployment, and observability. **Part V** closes with senior scenarios, including a method for turning a vague design prompt into a bounded answer, four worked capstone designs, and a structured way to troubleshoot production problems.

Most chapters share one shape. A short discussion of key concepts and of what a weak answer sounds like next to a strong one comes first. The questions follow. Each has an outline of a good answer, a note on what the interviewer is testing, and the mistake candidates most often make. Code listings come with a walkthrough that explains the design lesson and not only the syntax. Later sections turn to senior-level tradeoffs, debugging scenarios that move from symptom to likely cause to fix, and short exercises with solution sketches. Each chapter ends with a review checklist and a summary.

The book is deliberately about concepts rather than about one ecosystem. It does not teach a particular router, data library, test runner, or styling tool. Where a topic has many tools, it explains the underlying problem first, such as cache identity, cancellation, or scope, and treats any specific library as one way to solve it. That way the reasoning still holds when the tool on your team changes.

Use the book as a working reference. Read a question before its outline, answer it out loud, write the code yourself, and then compare your answer with the reasoning on the page. The goal is not to memorize phrasing. It is to build answers that are technically correct, specific to React and TypeScript, and defensible under follow-up questions.

Yohan J. Rodríguez
www.imabit.com

Contents

Preface . . . i

1 Typing Props, JSX, Components, and Composition **1**

1.1 Key Concepts . . . 1
1.2 Interview Perspective . . . 1
1.3 Why Component APIs Matter in Real React Apps . . . 2
1.4 Components as Typed Contracts . . . 3
1.5 Props as Public APIs . . . 4
1.6 Inline Props versus Named Prop Types . . . 4
1.7 Interface versus Type for Props . . . 5
1.8 Optional Props and Defaults . . . 6
1.9 Typing Children . . . 7
1.10 Event and Callback Props . . . 9
1.11 Composition over Configuration . . . 10
1.12 Discriminated Union Props . . . 11
1.13 Mutually Exclusive Prop Patterns . . . 13
1.14 Generic Components . . . 14
1.15 Render Props and Function-as-Child Patterns . . . 16
1.16 Polymorphic Component Concepts . . . 17
1.17 Component Extraction and API Design . . . 18
1.18 Presentational versus Container Boundaries . . . 19
1.19 Prop Spreading and Rest Props . . . 20
1.20 React.FC and Similar Footguns . . . 21
1.21 Common Bugs and Interview Pitfalls . . . 22
1.22 Debugging Prop and Type Errors . . . 23
1.23 Senior-Level Tradeoffs . . . 25
1.24 Interview Question Bank . . . 26
1.25 Mini Exercises . . . 28
1.26 Quick Review Checklist . . . 29
1.27 Chapter Summary . . . 30

1 | Typing Props, JSX, Components, and Composition

Component questions are usually API design questions in disguise. Interviewers are not only asking whether you can add a prop type to a component. They are asking whether you can design a contract that prevents invalid combinations, keeps common use cases ergonomic, composes cleanly with the rest of the UI, and stays understandable after six months of product growth.

1.1 Key Concepts

- Props are public API contracts, not just parameter lists.
- Reusable components need stricter prop design than page-only components.
- Optional props should represent intentional optional behavior, not hidden required assumptions.
- Children typing depends on the composition pattern: plain content, render functions, or named slots.
- Discriminated and mutually exclusive prop patterns prevent impossible UI states.
- Generic components should preserve useful type relationships without sacrificing inference.
- Prop spreading is flexible but can make contracts vague if used carelessly.
- Composition usually scales better than accumulating boolean flags and mega-config objects.

1.2 Interview Perspective

Weak component answers sound mechanical: "I make a props type and pass it to the component." Stronger answers focus on contract design: "This component is shared, so its props should reject invalid combinations, expose domain-level callbacks where possible, and remain easy to infer

at call sites. If the contract needs many booleans, I probably need variants, composition, or narrower components."

Component APIs multiply through a codebase. A weak internal page component might inconvenience one file. A weak reusable component can spread complexity into every team that uses it.

1.3 Why Component APIs Matter in Real React Apps

Components are one of the main boundaries in a React codebase. Their prop types decide:

- which states the UI can represent,
- which combinations are impossible,
- how much consumers must know about internal implementation,
- whether TypeScript inference helps or gets in the way,
- whether later refactors break callers safely or silently.

This matters more in reusable component libraries, feature modules, and shared design-system primitives than in a tiny one-off page component. A private component can tolerate some local looseness. A shared component becomes an interface other engineers depend on, so its prop model should be designed as carefully as a service contract.

Why do component typing questions matter so much in React interviews?

Answer outline: Because props are one of the most common contract boundaries in frontend systems. Interviewers use component questions to test whether you can design APIs that stay correct, ergonomic, and maintainable as reuse grows.

What the interviewer is testing: Whether you understand that component typing is API design, not decoration.

Common mistake: Treating prop typing as a small syntax task instead of a contract-design problem.

Why do reusable components need stricter prop design than page-only components?

Answer outline: Reusable components are consumed in many places by many engineers. Weak contracts multiply confusion and bugs. Page-only components can sometimes rely on local knowledge, but shared components need explicit valid combinations and predictable behavior.

What the interviewer is testing: Whether you adjust rigor to reuse scope.

Common mistake: Giving shared primitives the same loose contract quality as temporary page-level code.

1.4 Components as Typed Contracts

Props are the public API of a component. The implementation can change, but callers depend on the prop contract. Prop types should describe the component's intended usage rather than merely mirroring whatever destructuring happened in the initial implementation.

Listing 1.1: Named props contract for a small reusable component

```
type UserBadgeProps = {
  userId: string;
  displayName: string;
  isOnline: boolean;
};

function UserBadge({ userId, displayName, isOnline }: UserBadgeProps) {
  const status = isOnline ? "online" : "offline";

  return (
    <span data-user-id={userId} aria-label={`${displayName} is ${status}`}>
      {displayName}
    </span>
  );
}
```

Code walkthrough: `UserBadgeProps` is a small named contract with only the information the component needs. The prop names describe domain meaning rather than implementation quirks. TypeScript makes the API explicit, but the bigger lesson is that a component should expose a stable contract that callers can reason about without reading the component body.

What makes props a public API instead of just function parameters?

Answer outline: Props are the caller-facing contract for a component. Other engineers and features rely on their names, required fields, callback shapes, and valid combinations. That makes them public API even inside one codebase.

What the interviewer is testing: Whether you think about components as boundaries, not only implementation details.

Common mistake: Treating prop shape as an internal convenience and changing it casually without considering caller impact.

How do you know whether a component deserves a carefully designed prop type?

Answer outline: The more shared, reusable, and long-lived the component is, the more careful the API should be. A design-system input, modal, or table deserves more rigor than a one-off JSX wrapper used in one file.

What the interviewer is testing: Whether you can scale engineering discipline to component scope.

Common mistake: Applying either zero rigor everywhere or design-system rigor to every tiny local component.

1.5 Props as Public APIs

Thinking of props as API contracts changes how you name, group, and validate them. Good prop APIs:

- make common cases simple,
- make invalid cases impossible or at least very awkward,
- express intent through names that map to the product domain,
- hide DOM details when consumers do not need them,
- avoid forcing callers to know implementation trivia.

How do you keep a prop API focused?

Answer outline: Start from what the consumer needs to express, not from what the implementation happens to read today. Expose the smallest coherent set of inputs that lets callers describe intended behavior.

What the interviewer is testing: Whether you can separate consumer-facing contract design from internal code.

Common mistake: Exposing many implementation-shaped props because it feels easier than designing a cleaner API.

What is a smell that a component API is getting too large?

Answer outline: Too many booleans, many optional props with hidden dependencies, DOM event leakage through multiple layers, or a long prop list whose valid combinations are hard to explain. These usually signal the need for variants, composition, or smaller components.

What the interviewer is testing: Whether you can detect design drift early.

Common mistake: Adding another boolean or optional prop instead of reconsidering the shape of the contract.

1.6 Inline Props versus Named Prop Types

TypeScript lets you write props inline in a function signature or extract them into a named type or interface. The correct choice depends on scope and reuse. Local components can sometimes

tolerate inline shapes. Shared components usually benefit from named prop types because names improve readability, reuse, and discoverability.

When are inline prop types acceptable?

Answer outline: Inline prop types can be fine for tiny local components where the contract is short and not reused elsewhere. They keep context close to the implementation when the component is private and simple.

What the interviewer is testing: Whether you can balance brevity and maintainability.

Common mistake: Using large inline object types for shared or complex components.

When should you extract a named prop type?

Answer outline: Extract a named type when the component is shared, the contract is large, the prop model has meaningful business language, or the same shape appears in tests, stories, examples, or helpers.

What the interviewer is testing: Whether you understand when naming improves maintainability.

Common mistake: Leaving an important public API buried inside a large inline function signature.

1.7 Interface versus Type for Props

For many React components, `interface` and `type` are both reasonable. The practical distinction is more important than ideology. Interfaces are common for object-shaped contracts and support declaration merging. Type aliases also handle unions, intersections, tuples, and mapped types.

Should you use `interface` or `type` for component props?

Answer outline: Use whichever best fits the contract and your codebase conventions. Interfaces work well for plain object-shaped APIs. Type aliases become necessary for unions, intersections, conditional shapes, and some advanced patterns.

What the interviewer is testing: Whether you are pragmatic instead of dogmatic.

Common mistake: Claiming one is always the correct choice for all React code.

When does a props shape force you toward `type`?

Answer outline: When the contract is a union, intersection, mapped type, or other shape that an interface cannot express directly. Component variants and mutually exclusive props often push you toward `type`.

What the interviewer is testing: Whether you know when the language features actually differ.

Common mistake: Trying to force a variant-heavy API into a plain interface and losing clarity.

1.8 Optional Props and Defaults

Optional props are useful when the behavior is genuinely optional or when a sensible default exists. They become a smell when the component silently assumes the prop is really required, or when two optional props must appear together but the type does not enforce that relationship.

Listing 1.2: Optional props with deliberate defaults

```
type EmptyStateProps = {
  title: string;
  description?: string;
  actionLabel?: string;
  onAction?: () => void;
};

function EmptyState({
  title,
  description = "Nothing to show yet.",
  actionLabel,
  onAction
}: EmptyStateProps) {
  return (
    <section>
      <h2>{title}</h2>
      <p>{description}</p>
      {actionLabel && onAction ? (
        <button type="button" onClick={onAction}>
          {actionLabel}
        </button>
      ) : null}
    </section>
  );
}
```

Code walkthrough: `EmptyState` treats `description` as genuinely optional by providing an explicit default. It also pairs `actionLabel` with `onAction` in render logic, which hints at a later design question: should those really be modeled as a stronger relationship? The key lesson is that optional props are not permission to leave behavior vague.

When is an optional prop healthy?

Answer outline: When the feature is truly optional and the component has a clear behavior when the prop is omitted, or when a default value makes omission safe and intentional.

What the interviewer is testing: Whether you design omission behavior deliberately.

Common mistake: Marking props optional only to avoid compiler errors while still assuming they exist.

When does an optional prop become a smell?

Answer outline: When the component crashes or behaves ambiguously without it, when several optional props are secretly coupled, or when callers must read implementation details to know what omission means.

What the interviewer is testing: Whether you can identify weak contracts hidden behind question marks.

Common mistake: Turning a required invariant into an optional prop to make the call site easier temporarily.

How should default values influence prop design?

Answer outline: Defaults should make omitted props easy to reason about. They should simplify common cases, not hide complex behavior. If a default is surprising, the prop probably should not be optional.

What the interviewer is testing: Whether you understand defaults as API behavior, not just syntax.

Common mistake: Adding defaults that consumers cannot discover from the prop contract.

1.9 Typing Children

Children typing is often misunderstood because `children` is not always "just ReactNode." Many components do accept any renderable child content. Others need a function, or named slot props, or no children at all. The type should match the actual composition pattern.

Listing 1.3: Typing standard children and optional footer content

```
type PanelProps = {
  title: string;
  children: React.ReactNode;
  footer?: React.ReactNode;
};

function Panel({ title, children, footer }: PanelProps) {
  return (
    <section>
      <header>{title}</header>
      <div>{children}</div>
      {footer ? <footer>{footer}</footer> : null}
    </section>
  );
}
```

Listing 1.4: Typed slot-like composition with explicit regions

```
type ModalProps = {
  header: React.ReactNode;
  body: React.ReactNode;
  footer?: React.ReactNode;
};

function Modal({ header, body, footer }: ModalProps) {
  return (
    <section role="dialog" aria-modal="true">
      <header>{header}</header>
      <div>{body}</div>
      {footer ? <footer>{footer}</footer> : null}
    </section>
  );
}
```

Code walkthrough: The first example uses `React.ReactNode` because the panel accepts ordinary renderable content. The second example avoids an open-ended children tree by using explicit slot-like props: `header`, `body`, and `footer`. The TypeScript choice communicates the composition pattern directly: `children` should reflect intended composition, not habit.

How should ordinary children be typed?

Answer outline: Use `React.ReactNode` when the component accepts normal renderable content such as elements, strings, numbers, fragments, and conditional null values.

What the interviewer is testing: Whether you choose the ordinary children type for plain composition.

Common mistake: Using overly narrow element types when the component actually accepts broader content.

When should children not be typed as `ReactNode`?

Answer outline: When children is a function, when the component needs a more constrained relationship, or when named props communicate the structure better than a free-form tree.

What the interviewer is testing: Whether you can distinguish plain content from more structured composition.

Common mistake: Typing a render-function child as `ReactNode` and losing callable behavior.

Why is children typing often misunderstood?

Answer outline: Because developers overgeneralize from the most common case. Many components accept arbitrary content, but some patterns need stronger modeling to express contract intent and prevent misuse.

What the interviewer is testing: Whether you can choose the right abstraction instead of copying defaults.

Common mistake: Automatically adding `children` to every component, even when the component should not accept it.

1.10 Event and Callback Props

Reusable components should usually expose domain-level callbacks instead of leaking low-level DOM events through every layer. A search input might expose `onValueChange(value: string)` rather than `onChange(event: React.ChangeEvent<HTMLInputElement>)` unless the consumer truly needs event details.

Listing 1.5: Domain-level callback prop instead of event leakage

```
type SearchInputProps = {
  value: string;
  onValueChange: (nextValue: string) => void;
};

function SearchInput({ value, onValueChange }: SearchInputProps) {
  return (
    <input
      value={value}
      onChange={event => onValueChange(event.currentTarget.value)}
      placeholder="Search users"
    />
  );
}
```

Code walkthrough: `SearchInput` converts the DOM change event into a domain callback carrying the next string value. The implementation still types the actual event correctly, but the public API is simpler for consumers and tests. Event types belong at the DOM boundary unless callers need the event itself.

How should callback props usually be typed?

Answer outline: Prefer domain-level callback types such as `(value: string) => void` for reusable components. Use raw DOM event types when the event object itself is part of the consumer contract.

What the interviewer is testing: Whether you separate internal DOM wiring from external API design.

Common mistake: Passing DOM event objects through several layers when only a typed value is needed.

When is exposing the raw event actually reasonable?

Answer outline: When the consumer needs event-specific details such as selection range, modifier keys, or event prevention behavior, or when the component is intentionally a thin wrapper around

a DOM element.

What the interviewer is testing: Whether you can avoid rigid rules and discuss context.

Common mistake: Either banning event exposure completely or exposing events by default in all reusable APIs.

What is a common callback typing bug in reusable components?

Answer outline: Using the wrong event type, or exposing a callback shape that is too tied to a specific DOM element even though the component might later change implementation.

What the interviewer is testing: Whether you think about refactoring cost in public APIs.

Common mistake: Designing a domain component whose API hardcodes `HTMLInputElement` for no good reason.

1.11 Composition over Configuration

Large configuration objects and many boolean props look flexible at first, but they often turn into vague contracts. Composition can keep APIs smaller and clearer by letting callers provide UI structure directly.

Listing 1.6: Composition often scales better than config flags

```
type CardProps = {
  title: string;
  children: React.ReactNode;
  actions?: React.ReactNode;
};

function Card({ title, children, actions }: CardProps) {
  return (
    <section>
      <header>
        <h2>{title}</h2>
        {actions}
      </header>
      <div>{children}</div>
    </section>
  );
}

const card = (
  <Card title="Quarterly Revenue" actions={<button type="button">Export</button>}>
    <strong>$120,000</strong>
  </Card>
);
```

Code walkthrough: `Card` accepts `children` and optional `actions` rather than a long list of props like `showExportButton`, `exportLabel`, and `metricBody`. The TypeScript contract stays focused on structure rather than enumerating all future variants. Composition often scales better when the UI shape itself is what varies.

Why does composition often scale better than large configuration objects?

Answer outline: Composition lets callers supply the UI pieces that vary without forcing the component author to predict every future permutation as a prop. It keeps the API smaller and often more flexible.

What the interviewer is testing: Whether you understand composition as an API-scaling strategy.

Common mistake: Adding more booleans and strings every time a new layout variant appears.

When is configuration still better than composition?

Answer outline: When the component needs a small, closed set of simple variations and composition would make common use cases noisier. Small controlled variants can be easier to reason about than open-ended composition.

What the interviewer is testing: Whether you can discuss tradeoffs rather than repeating slogans.

Common mistake: Treating composition as automatically superior in all cases.

1.12 Discriminated Union Props

Discriminated unions are one of the strongest tools for component API design because they let the type system model variants explicitly. Instead of many optionals, the consumer chooses a mode, and the required props for that mode become clear.

Listing 1.7: Discriminated union props for UI variants

```
type LoadingCardProps = {
  status: "loading";
  title: string;
};

type ErrorCardProps = {
  status: "error";
  title: string;
  message: string;
  onRetry: () => void;
};

type SuccessCardProps = {
  status: "success";
  title: string;
  children: React.ReactNode;
};

type StatusCardProps = LoadingCardProps | ErrorCardProps | SuccessCardProps;

function StatusCard(props: StatusCardProps) {
```

```
  switch (props.status) {
    case "loading":
      return <section>{props.title}...</section>;
    case "error":
      return (
        <section>
          <p>{props.title}</p>
          <p>{props.message}</p>
          <button type="button" onClick={props.onRetry}>
            Retry
          </button>
        </section>
      );
    case "success":
      return (
        <section>
          <p>{props.title}</p>
          <div>{props.children}</div>
        </section>
      );
  }
}
```

Code walkthrough: `StatusCardProps` is a union of three variants keyed by `status`. Each variant carries only the fields it needs. TypeScript narrows inside the `switch`, and React rendering stays aligned with the business states. Variants become easier to read and harder to misuse when the type tells the truth.

When do discriminated unions simplify a component API?

Answer outline: When the component has a closed set of modes and each mode requires different fields. Unions are especially useful when optional props otherwise create illegal combinations.

What the interviewer is testing: Whether you can use TypeScript to remove impossible UI states.

Common mistake: Leaving every mode field optional instead of modeling the variants directly.

What makes a good discriminant field?

Answer outline: A field whose value clearly selects a variant and is stable enough for both humans and the compiler to reason about, such as `kind`, `variant`, `status`, or `mode`.

What the interviewer is testing: Whether you understand how narrowing becomes readable.

Common mistake: Choosing vague or overlapping discriminants that do not actually make branches clearer.

How do discriminated union props improve component maintenance?

Answer outline: They keep variant-specific fields aligned with variant-specific rendering. When a new variant is added, TypeScript helps identify the places that need to handle it.

What the interviewer is testing: Whether you connect type design to refactor safety.

Common mistake: Evaluating unions only by syntax cost instead of maintenance benefit.

1.13 Mutually Exclusive Prop Patterns

Some components should accept one of several paths, but never all at once. Navigation items often behave as either links or buttons. Dialog triggers might be controlled externally or manage local state internally. These are good candidates for mutually exclusive prop patterns using `never` or unions.

Listing 1.8: Mutually exclusive props with never fields

```
type LinkActionProps = {
  href: string;
  onClick?: never;
};

type ButtonActionProps = {
  onClick: () => void;
  href?: never;
};

type ActionChipProps = {
  label: string;
} & (LinkActionProps | ButtonActionProps);

function ActionChip(props: ActionChipProps) {
  if ("href" in props) {
    return <a href={props.href}>{props.label}</a>;
  }

  return (
    <button type="button" onClick={props.onClick}>
      {props.label}
    </button>
  );
}
```

Code walkthrough: `ActionChipProps` guarantees that callers either provide `href` or `onClick`, but not both. The `never` fields make invalid combinations noisy at compile time. The React lesson is that API correctness should be enforced before a component reaches ambiguous runtime behavior.

Why do mutually exclusive prop patterns matter?

Answer outline: They prevent impossible or ambiguous states from reaching the component. If a component cannot be both a link and a button at once, the type should reject that combination.

What the interviewer is testing: Whether you can make illegal states unrepresentable in component APIs.

Common mistake: Allowing incompatible props simultaneously and resolving the ambiguity with hidden precedence rules.

How does `never` help mutually exclusive props?

Answer outline: `never` marks fields that must not exist for a given variant, so TypeScript flags call sites that try to combine contradictory props.

What the interviewer is testing: Whether you know a practical use of `never` in component design.

Common mistake: Making the fields optional on every branch and relying on runtime checks later.

What runtime bug does this pattern prevent?

Answer outline: Inconsistent behavior when a component accepts multiple competing action props such as `href` and `onClick`. Without a stronger contract, callers may not know which behavior wins.

What the interviewer is testing: Whether you link type design to real user-facing ambiguity.

Common mistake: Treating the issue as merely stylistic rather than behaviorally dangerous.

1.14 Generic Components

Generic components can preserve caller types across items, values, callbacks, and render functions. They help when the relationship is real and inference remains good. They hurt when the generic layer exists only to look reusable while making every call site harder to understand.

Listing 1.9: Generic list component preserving caller item types

```
type ListProps<TItem> = {
  items: readonly TItem[];
  getKey: (item: TItem) => string;
  renderItem: (item: TItem) => React.ReactNode;
};

function List<TItem>({ items, getKey, renderItem }: ListProps<TItem>) {
  return (
    <ul>
      {items.map(item => (
        <li key={getKey(item)}>{renderItem(item)}</li>
```

```
      ))}
    </ul>
  );
}

type Product = {
  id: string;
  name: string;
};

const products: Product[] = [{ id: "p-1", name: "Monitor" }];

const productList = (
  <List
    items={products}
    getKey={product => product.id}
    renderItem={product => <span>{product.name}</span>}
  />
);
```

Code walkthrough: The component preserves the item type across `items`, `getKey`, and `renderItem`. Inference means callers do not have to spell out `Product` explicitly. This is the ideal generic situation: the abstraction keeps a real relationship and stays easy to consume.

When should a reusable component be generic?

Answer outline: When the caller's data type must flow through the component and back into callbacks or render functions. Lists, tables, selects, and data-driven render containers are common examples.

What the interviewer is testing: Whether you understand what generics preserve in UI code.

Common mistake: Introducing generics where a concrete domain type would be simpler and clearer.

How do you know a generic component has gone too far?

Answer outline: If inference breaks, callers need many manual annotations, or the component API becomes harder to explain than the duplicated concrete versions, the abstraction is probably too abstract.

What the interviewer is testing: Whether you can balance reuse with ergonomics.

Common mistake: Defending complexity because the component is "technically reusable."

Why is inference quality so important for generic components?

Answer outline: Because consumers interact with the generic API at call sites. If the common case does not infer naturally, the API becomes noisy and error-prone.

What the interviewer is testing: Whether you design for the consumer experience, not only the implementation.

Common mistake: Treating manual type arguments at every call site as normal.

1.15 Render Props and Function-as-Child Patterns

Render props let a component expose state or data to caller-supplied rendering logic. The pattern is useful when the shared behavior is real but the displayed UI varies a lot.

Listing 1.10: Render-prop component with typed data flow

```
type FetchBoundaryProps<TData> = {
  data: TData | null;
  isLoading: boolean;
  renderLoading: () => React.ReactNode;
  children: (data: TData) => React.ReactNode;
};

function FetchBoundary<TData>({
  data,
  isLoading,
  renderLoading,
  children
}: FetchBoundaryProps<TData>) {
  if (isLoading) {
    return <>{renderLoading()}</>;
  }

  if (!data) {
    return null;
  }

  return <>{children(data)}</>;
}
```

Code walkthrough: `FetchBoundary` owns loading gating and null handling while letting the caller render the successful state through a typed function. The important TypeScript detail is that the function child receives a real `TData`, not a vague `unknown`. Render props are about behavior reuse with caller-controlled UI.

When are render props useful?

Answer outline: When a component owns reusable behavior or state but consumers need substantial control over rendered UI. They are often useful for data boundaries, feature flags, or interaction wrappers.

What the interviewer is testing: Whether you can choose the pattern by problem shape.

Common mistake: Using render props for cases where plain composition or a simpler prop would be clearer.

How should a function-as-child pattern be typed?

Answer outline: Type the function explicitly with the values it receives and the renderable result it returns, such as `(data: TData) => React.ReactNode`.

What the interviewer is testing: Whether you can model nonstandard children contracts correctly.

Common mistake: Typing the function child as `ReactNode` and losing callable narrowing.

1.16 Polymorphic Component Concepts

Polymorphic components let consumers choose the rendered element through an `as` prop or a constrained set of supported element types. These APIs become complicated quickly, especially when refs and element-specific props are involved. In interviews, it is usually enough to explain the concept, the type-safety cost, and where a narrower API is safer.

Listing 1.11: Light polymorphic component with a constrained as prop

```
type InlineTextProps = {
  as?: "span" | "strong" | "label";
  children: React.ReactNode;
  htmlFor?: string;
};

function InlineText({ as = "span", children, htmlFor }: InlineTextProps) {
  if (as === "label") {
    return <label htmlFor={htmlFor}>{children}</label>;
  }

  if (as === "strong") {
    return <strong>{children}</strong>;
  }

  return <span>{children}</span>;
}
```

Code walkthrough: This example keeps the polymorphic idea deliberately narrow. `InlineText` allows only a small set of element choices and supports a limited prop difference for `label`. The lesson is not that every design-system primitive should be polymorphic. It is that polymorphism can be useful when the semantic element varies, but it should be introduced with discipline.

What is a polymorphic component in React terms?

Answer outline: It is a component whose rendered element type can vary, often through an `as` prop, while still preserving a coherent component-level API.

What the interviewer is testing: Whether you understand the pattern conceptually without overcomplicating it.

Common mistake: Jumping straight to a highly abstract implementation instead of first explaining the use case.

When is a light polymorphic API reasonable?

Answer outline: When the semantic tag may vary but the component's purpose stays the same, and the allowed element set is small enough to explain and type safely.

What the interviewer is testing: Whether you know how to constrain flexibility.

Common mistake: Making everything polymorphic because it feels more reusable.

What makes polymorphic components hard to type at a senior level?

Answer outline: Element-specific props, ref forwarding, event shapes, and attribute filtering all become more complex. That is why interview answers should emphasize tradeoffs, not only clever implementation.

What the interviewer is testing: Whether you can recognize abstraction cost.

Common mistake: Pretending polymorphism is free once the syntax compiles.

1.17 Component Extraction and API Design

Extracting a component is not automatically good architecture. The question is whether the extracted API represents a real reusable boundary or merely hides code without reducing complexity. Good extraction moves repeated behavior or UI contract to a place where the new component has a coherent purpose.

When should you extract a component instead of leaving JSX inline?

Answer outline: Extract when the UI block has its own meaning, contract, or reuse value, or when extraction improves readability by giving a repeated idea a name. Do not extract only to reduce line count mechanically.

What the interviewer is testing: Whether you think in terms of boundaries and meaning.

Common mistake: Extracting tiny markup fragments that create more navigation cost than clarity.

How do you know an extracted component has the wrong API?

Answer outline: When it requires many one-off props from each call site, leaks implementation details, or is harder to use than the inline JSX it replaced.

What the interviewer is testing: Whether you can evaluate extraction quality after the fact.

Common mistake: Assuming extraction is automatically a refactor improvement.

1.18 Presentational versus Container Boundaries

Presentational and container terminology is older, but the underlying idea still matters: some components mostly render UI from inputs, while others coordinate data fetching, state, or orchestration. The exact naming matters less than clear boundaries.

Listing 1.12: Presentational view separated from a coordinating container

```
type User = {
  id: string;
  displayName: string;
};

type UserListViewProps = {
  users: User[];
  onRefresh: () => void;
};

function UserListView({ users, onRefresh }: UserListViewProps) {
  return (
    <section>
      <button type="button" onClick={onRefresh}>
        Refresh
      </button>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.displayName}</li>
        ))}
      </ul>
    </section>
  );
}

function UserListContainer() {
  const [users] = useState<User[]>([{ id: "u-1", displayName: "Ada" }]);

  function refreshUsers(): void {
    console.log("refresh users");
  }

  return <UserListView users={users} onRefresh={refreshUsers} />;
}
```

Code walkthrough: `UserListView` stays focused on rendering, while `UserListContainer` coordinates state and refresh behavior. This is not a mandatory pattern for every feature. It is a useful example of separating view contracts from orchestration concerns when the complexity justifies it.

What is the value of a presentational/container split today?

Answer outline: It can improve testability, API clarity, and reuse by separating rendering contracts from data or orchestration logic. The split is optional, not mandatory dogma.

What the interviewer is testing: Whether you can discuss boundaries without repeating outdated slogans.

Common mistake: Treating container/presentational as a required folder pattern for all React code.

When is this split overkill?

Answer outline: When the feature is tiny, the logic is local, and splitting would add ceremony without making the code easier to read or test.

What the interviewer is testing: Whether you avoid unnecessary abstraction.

Common mistake: Splitting tiny components into artificial layers just because the pattern exists.

1.19 Prop Spreading and Rest Props

Prop spreading can make wrapper components flexible, especially when building thin DOM abstractions. It can also make a contract blurry by silently forwarding attributes you did not intend to support or document.

Listing 1.13: Prop spreading is flexible but broadens the contract

```
type InputBaseProps = {
  label: string;
} & React.InputHTMLAttributes<HTMLInputElement>;

function TextField({ label, ...inputProps }: InputBaseProps) {
  return (
    <label>
      {label}
      <input {...inputProps} />
    </label>
  );
}

// This is flexible, but broad prop spreading can also forward attributes you did not intend to support
    .
```

Code walkthrough: `TextField` combines a clear required prop, `label`, with the rest of the input attributes. That is useful for a thin wrapper. The tradeoff is that the public contract becomes broader and harder to narrate because many DOM attributes pass through automatically. Use this pattern intentionally, especially in shared primitives.

When is prop spreading helpful?

Answer outline: It is helpful for thin wrappers around DOM primitives or well-understood lower-level components where broad pass-through behavior is intentional and documented.

What the interviewer is testing: Whether you can defend the pattern in the right context.

Common mistake: Banning prop spreading entirely instead of distinguishing useful wrappers from vague APIs.

When is prop spreading dangerous?

Answer outline: When the component is higher-level, when you do not want all underlying attributes exposed, or when forwarded props make it unclear which attributes are actually supported.

What the interviewer is testing: Whether you understand contract broadening.

Common mistake: Forwarding everything and later being surprised that consumers depend on accidental support.

1.20 React.FC and Similar Footguns

`React.FC` is not inherently wrong, but it is not mandatory and can hide API questions. Many teams prefer explicit prop annotations because they make the accepted props and return intent more obvious at the component definition site.

Why is `React.FC` not always necessary?

Answer outline: Because a normal function with explicit props is usually enough and often clearer. You do not need `React.FC` to type a component correctly.

What the interviewer is testing: Whether you know the default without overcommitting to a style debate.

Common mistake: Assuming a component is not "properly typed" unless it uses `React.FC`.

What tradeoffs does `React.FC` have?

Answer outline: It can standardize component typing, but it may obscure the exact API shape and is unnecessary for many components. Teams often choose whichever convention makes props and return values clearest locally.

What the interviewer is testing: Whether you can discuss the tradeoff without cargo-cult rules.

Common mistake: Turning the topic into a dogmatic style argument instead of discussing readability and local convention.

1.21 Common Bugs and Interview Pitfalls

Many component typing bugs come from a mismatch between intended API and actual prop model. Some are TypeScript errors; others compile fine but allow ambiguous behavior that later becomes a product bug.

Listing 1.14: A component API footgun with too many conflicting options

```typescript
type ButtonProps = {
  href?: string;
  onClick?: () => void;
  isPrimary?: boolean;
  isSecondary?: boolean;
  isDanger?: boolean;
  isLoading?: boolean;
  iconLeft?: React.ReactNode;
  iconRight?: React.ReactNode;
  children: React.ReactNode;
};

function Button(props: ButtonProps) {
  if (props.href) {
    return <a href={props.href}>{props.children}</a>;
  }

  return <button type="button" onClick={props.onClick}>{props.children}</button>;
}

// The API permits conflicting states and forces consumers to guess valid combinations.
```

Code walkthrough: This example compiles, but the API is weak. Several boolean and optional props can combine in contradictory ways, and the contract does not clearly tell consumers what is valid. A type can be legal while the API is still poorly designed.

What is a classic component API footgun?

Answer outline: A component that accepts too many optional booleans and overlapping behavior props, allowing callers to create contradictory states the implementation resolves inconsistently.

What the interviewer is testing: Whether you can distinguish "typed" from "well designed."

Common mistake: Assuming the presence of a prop type means the API quality is already high.

Why can a valid TypeScript component API still be a bad API?

Answer outline: Because type-checking only proves consistency with the type, not whether the type expresses a good contract. A vague type can still compile while permitting awkward or impossible product states.

What the interviewer is testing: Whether you understand the limits of typing as a design tool.

Common mistake: Treating any compiling prop model as acceptable architecture.

1.22 Debugging Prop and Type Errors

Component API debugging is often about clarifying the contract. The right question is usually not "How do I silence TypeScript?" but "What API relationship did I fail to model clearly?"

A component accepts both `href` and `onClick` and behaves inconsistently. What is the likely cause?

Answer outline: The public API allows conflicting action modes. Model the contract as mutually exclusive link versus button props and make invalid combinations fail at compile time.

What the interviewer is testing: Whether you fix ambiguous APIs at the type boundary instead of patching runtime precedence rules.

Common mistake: Adding another runtime `if` without changing the contract.

A prop is optional but the component crashes when it is missing. How do you fix that?

Answer outline: Decide whether the prop is truly optional. If not, make it required. If it is optional, provide a safe default or branch explicitly on absence.

What the interviewer is testing: Whether you align runtime behavior with type intent.

Common mistake: Keeping the question mark and adding non-null assertions inside the component.

Children is typed too loosely and consumers pass unsupported structures. What do you inspect?

Answer outline: Check whether plain `ReactNode` is actually the right contract. If the component requires a specific structure or a function child, type that relationship explicitly.

What the interviewer is testing: Whether you can tighten composition contracts intentionally.

Common mistake: Treating all children use cases as interchangeable.

A generic list component loses type inference at call sites. What concern would you raise?

Answer outline: The generic API may be too abstract or shaped poorly for inference. Revisit prop order, callback positions, and whether the abstraction should stay generic at all.

What the interviewer is testing: Whether you can diagnose generic ergonomics issues.

Common mistake: Accepting manual type arguments everywhere as normal.

A discriminated union is not narrowing correctly inside a component. What do you check?

Answer outline: Confirm that the discriminant is explicit, that each branch has a unique literal, and that the component is branching on that field rather than on vague optional checks.

What the interviewer is testing: Whether you can debug union modeling instead of fighting the compiler.

Common mistake: Switching back to a broad optional bag because the first union attempt was poorly shaped.

A prop spread forwards invalid DOM attributes unexpectedly. What is the likely problem?

Answer outline: The component contract is broader than intended. Narrow the spread, whitelist supported attributes, or keep pass-through behavior only for lower-level wrappers.

What the interviewer is testing: Whether you understand how rest props broaden a contract.

Common mistake: Assuming all forwarded props are harmless because the browser will ignore some of them.

A reusable component has too many boolean props. What redesigns would you consider?

Answer outline: Consider discriminated unions, variant props, composition, or splitting the component into narrower components. The current API likely permits impossible states.

What the interviewer is testing: Whether you can redesign an API instead of merely typing it.

Common mistake: Adding yet another boolean and calling the API flexible.

A callback prop has the wrong event type. How do you investigate?

Answer outline: Check whether the component should expose a raw event at all. If yes, confirm the DOM element type matches the actual element. If not, change the public callback to a domain-level value.

What the interviewer is testing: Whether you can locate the right boundary for event details.

Common mistake: Threading a mismatched DOM event type through multiple wrapper layers.

A component API allows impossible states. What is the real fix?

Answer outline: Change the type model so the impossible states are not representable: required props, variants, mutually exclusive branches, or smaller components.

What the interviewer is testing: Whether you think in terms of contract redesign.

Common mistake: Leaving the weak contract in place and adding runtime warnings only.

A polymorphic component loses ref or prop type safety. What concern would you raise in an interview?

Answer outline: The abstraction cost may now outweigh the benefit. Constrain the supported element set, split the component, or simplify the API before adding more type machinery.

What the interviewer is testing: Whether you know when to push back on clever abstractions.

Common mistake: Treating more type gymnastics as automatically the right solution.

1.23 Senior-Level Tradeoffs

Why should component props be treated as API contracts?

Answer outline: Because shared components influence many callers over time. The prop model determines what is easy, what is safe, and what breaks when the component evolves.

What the interviewer is testing: Whether you treat component APIs with the same seriousness as other software interfaces.

Common mistake: Thinking frontend component contracts deserve less rigor than service or library APIs.

Why do discriminated unions often scale better than many optional props?

Answer outline: Because they express closed sets of valid states explicitly. Optional bags tend to drift into illegal combinations and unclear branch logic.

What the interviewer is testing: Whether you can choose modeling tools by maintenance cost.

Common mistake: Preferring many optional fields because they look simpler at first.

How do you balance flexibility and maintainability in component APIs?

Answer outline: Make common cases easy, keep invalid cases hard or impossible, constrain flexibility to real product needs, and avoid open-ended abstractions unless the reuse value is proven.

What the interviewer is testing: Whether you can design APIs for teams rather than for novelty.

Common mistake: Maximizing flexibility without pricing the long-term maintenance cost.

Why can generic components improve reuse but also become over-abstracted?

Answer outline: They improve reuse when they preserve meaningful caller types cleanly. They become over-abstracted when inference degrades and the consumer has to understand the generic machinery more than the UI problem.

What the interviewer is testing: Whether you can evaluate abstraction from the consumer side.

Common mistake: Celebrating abstraction even when call sites become harder to read.

How would you explain component boundaries in a senior interview?

Answer outline: Start with ownership and contract: what the component owns, what consumers must supply, what states are allowed, and whether the boundary is reusable enough to deserve a stronger API. Then explain why that boundary helps future change.

What the interviewer is testing: Whether you can discuss architecture through concrete UI boundaries.

Common mistake: Talking only about folder structure instead of API responsibility and ownership.

Why is `React.FC` discussion usually less important than API quality?

Answer outline: Because convention choice matters less than whether the contract is clear, safe, and readable. An excellent component API can exist with or without `React.FC`; a weak API stays weak either way.

What the interviewer is testing: Whether you keep style debates subordinate to real design quality.

Common mistake: Spending more energy on component typing syntax than on the public API itself.

1.24 Interview Question Bank

How do you type a basic component's props?

Answer outline: Define a named props type when the component is shared or nontrivial, keep the contract focused on consumer needs, and type the component parameters explicitly. Use `React.FC` only if it matches local conventions and still keeps the API clear.

What the interviewer is testing: Whether you can type a simple component without over-complicating it.

Common mistake: Exporting vague, over-broad prop types or leaning on `any`.

How should optional props be handled?

Answer outline: Optional props should have explicit omission behavior, usually a clear default or a real feature toggle. If omission makes the component unsafe, the prop should be required.

What the interviewer is testing: Whether you align the type contract with runtime behavior.

Common mistake: Marking props optional and then crashing when they are absent.

How can TypeScript prevent invalid prop combinations?

Answer outline: Use discriminated unions or mutually exclusive prop branches so the compiler rejects invalid combinations before runtime.

What the interviewer is testing: Whether you can design safer component contracts.

Common mistake: Allowing all combinations and relying on implementation precedence rules.

How should children be typed?

Answer outline: Use `ReactNode` for ordinary content, a function type for render props, and a more structured set of named props when the composition pattern is slot-like.

What the interviewer is testing: Whether you can match the type to the composition pattern.

Common mistake: Typing all children the same way regardless of behavior.

How do you type event handlers safely in reusable components?

Answer outline: Type internal event handlers to the correct DOM element, then consider exposing domain-level callbacks publicly unless event details are part of the component's contract.

What the interviewer is testing: Whether you understand boundary design around events.

Common mistake: Exposing low-level DOM events when the consumer only needs a typed value.

When should a component be generic?

Answer outline: When the caller's data shape must flow through the component and back into callbacks or render functions while preserving inference.

What the interviewer is testing: Whether you understand what generic reuse actually buys you.

Common mistake: Making a component generic when there is no important relationship to preserve.

When is composition better than adding more props?

Answer outline: When the varying part is UI structure or substantial behavior that the component author should not try to enumerate as dozens of configuration props.

What the interviewer is testing: Whether you know when to stop growing a prop list.

Common mistake: Adding another configuration prop instead of reconsidering the component boundary.

What is the difference between a page-only component and a reusable primitive in prop design?

Answer outline: A page-only component can sometimes rely on local assumptions. A reusable primitive needs a clearer, narrower, and more defensive contract because many callers depend on it.

What the interviewer is testing: Whether you scale API quality to reuse scope.

Common mistake: Giving shared primitives overly loose page-level APIs.

Why can prop spreading be both useful and risky?

Answer outline: It is useful for thin wrappers, but risky because it broadens the contract and may forward attributes or behaviors the component did not intend to support.

What the interviewer is testing: Whether you can discuss flexible wrapper design honestly.

Common mistake: Treating prop spreading as either universally bad or automatically free.

What is a good way to explain component API design in a senior interview?

Answer outline: Explain the component's purpose, its allowed states, its caller responsibilities, and how the type model prevents misuse while keeping the common path ergonomic.

What the interviewer is testing: Communication quality and design reasoning.

Common mistake: Listing TypeScript features without explaining the design outcome.

1.25 Mini Exercises

Exercise 1: Strengthen a button contract

Prompt: Redesign a button component that currently accepts both `href` and `onClick`, plus many booleans for variants.

Expected skill: Using mutually exclusive and discriminated prop patterns.

Hints: Split behavior mode from styling variant. Do not let incompatible action props coexist.

Solution sketch: Model link and action behavior as separate branches, then use a smaller variant field or composition pattern for appearance instead of many booleans.

Exercise 2: Repair an optional prop smell

Prompt: A reusable `Avatar` component has `src?` and `alt?`, but it crashes when `src` is missing and uses a confusing fallback when `alt` is omitted. Improve the API.

Expected skill: Aligning optionality with real behavior.

Hints: Ask which fields are truly optional and what defaults are safe.

Solution sketch: Make `alt` required if it is truly needed, or provide a safe explicit default. Either make `src` required or support a real fallback branch without crashing.

Exercise 3: Improve a generic list

Prompt: A generic list component forces callers to annotate type parameters manually and still loses item types in `renderItem`. Improve the design.

Expected skill: Preserving inference in generic component APIs.

Hints: Keep the relationship simple. Let `items` drive inference where possible.

Solution sketch: Accept `items: readonly TItem[]` and typed callbacks based on `TItem`. Avoid extra generic layers unless they preserve a real relationship.

Exercise 4: Replace configuration sprawl with composition

Prompt: A dashboard card has twelve props controlling icon placement, title layout, footer actions, and empty states. Propose a simpler API.

Expected skill: Recognizing when composition is a better boundary.

Hints: Ask which variations are structural rather than scalar.

Solution sketch: Keep a small core API and expose slots such as header actions, body content, or footer content instead of enumerating every structural variation as a prop.

Exercise 5: Tighten children typing

Prompt: A data-boundary component expects a function child but is typed with `children: React.ReactNode`. Fix the contract.

Expected skill: Typing function-as-child patterns correctly.

Hints: Decide what values the child function should receive.

Solution sketch: Change the type to something like `children: (data: TData) => React.ReactNode` and render it only when the required data is available.

1.26 Quick Review Checklist

- Can you explain why props are API contracts instead of just function parameters?
- Do you know when to use named prop types rather than large inline object types?
- Can you explain `interface` versus `type` pragmatically?
- Do you recognize when optional props are healthy versus when they hide a smell?
- Can you type ordinary children, render-function children, and slot-like composition differently?

- Do you know when to expose domain callbacks instead of raw DOM events?
- Can you use discriminated and mutually exclusive prop patterns to prevent impossible states?
- Can you explain when a component should be generic and when that generic abstraction is too much?
- Can you discuss polymorphic components at a tradeoff level without overengineering them?
- Do you understand the risks and benefits of prop spreading?
- Can you explain presentational versus container boundaries without dogma?
- Can you debug a weak prop contract by redesigning the API instead of silencing TypeScript?

1.27 Chapter Summary

Component typing in React is really contract design. Props describe what a component expects, what states it can represent, and what behaviors consumers can trigger. Optional props and defaults should be deliberate. Children and callbacks should be typed according to the composition pattern, not habit. Discriminated and mutually exclusive props prevent impossible UI states. Generic and polymorphic components are worth using only when inference and API clarity survive. Composition often scales better than boolean-heavy configuration. Senior React candidates explain component APIs the way good engineers explain any interface: in terms of correctness, ergonomics, change cost, and clear boundaries.