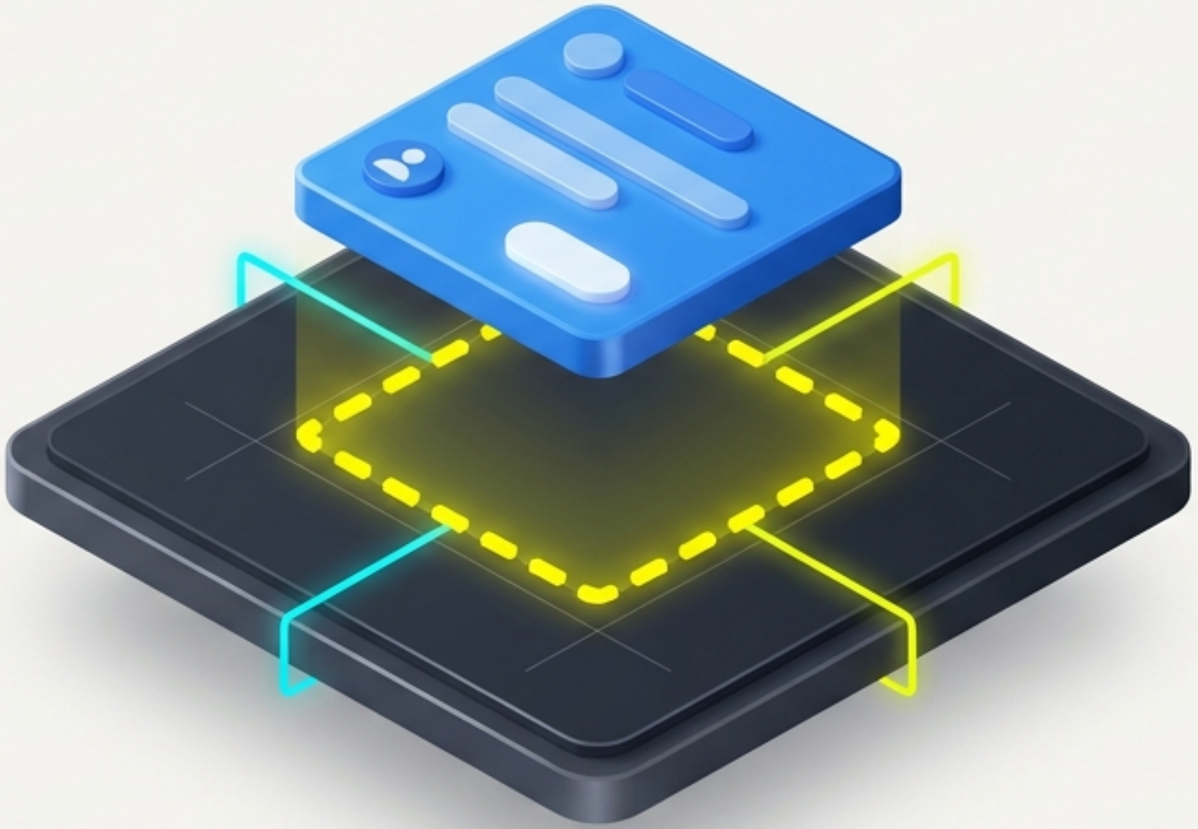
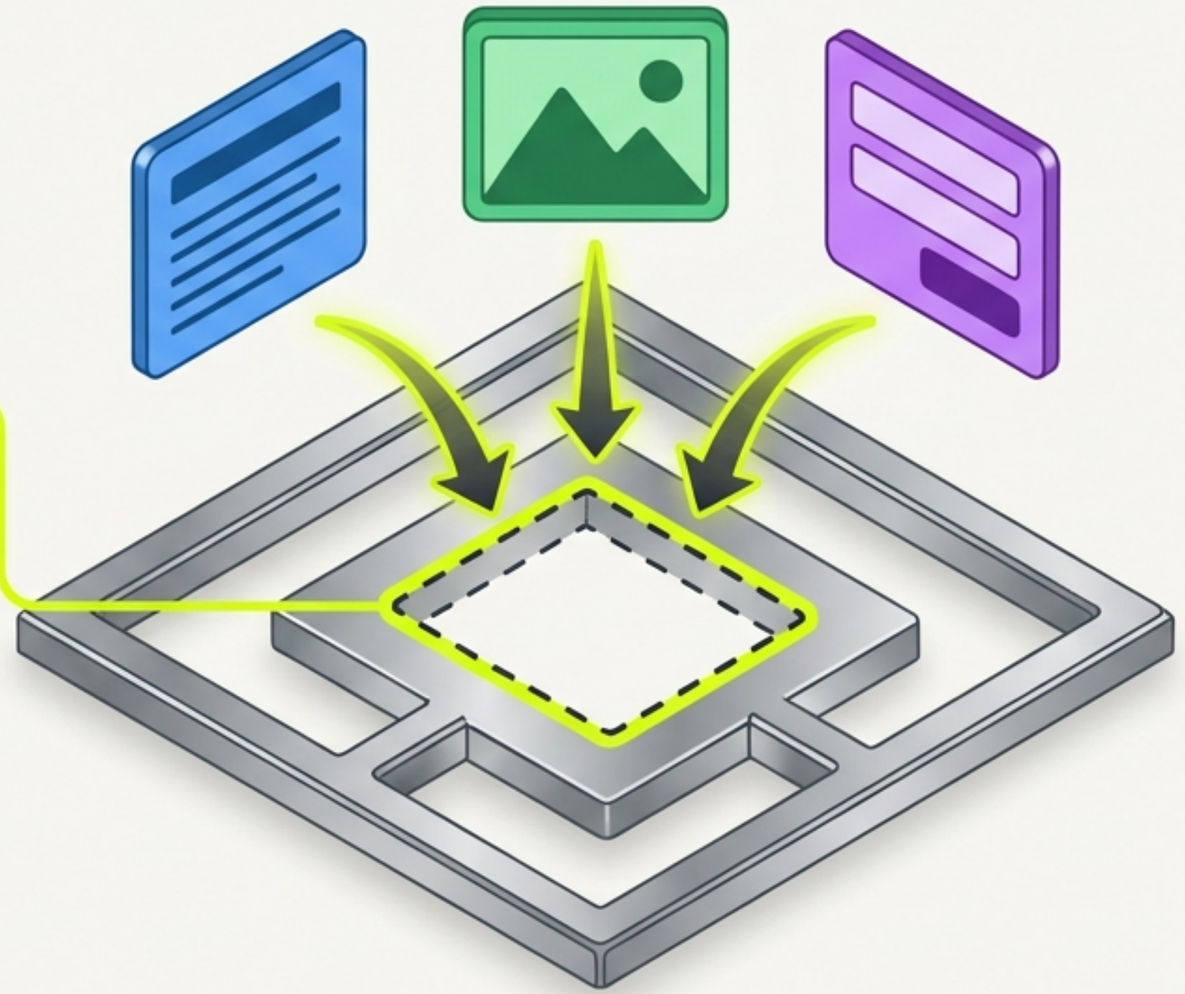
Mastering React Composition



The architecture of composition relies on an agnostic structural container

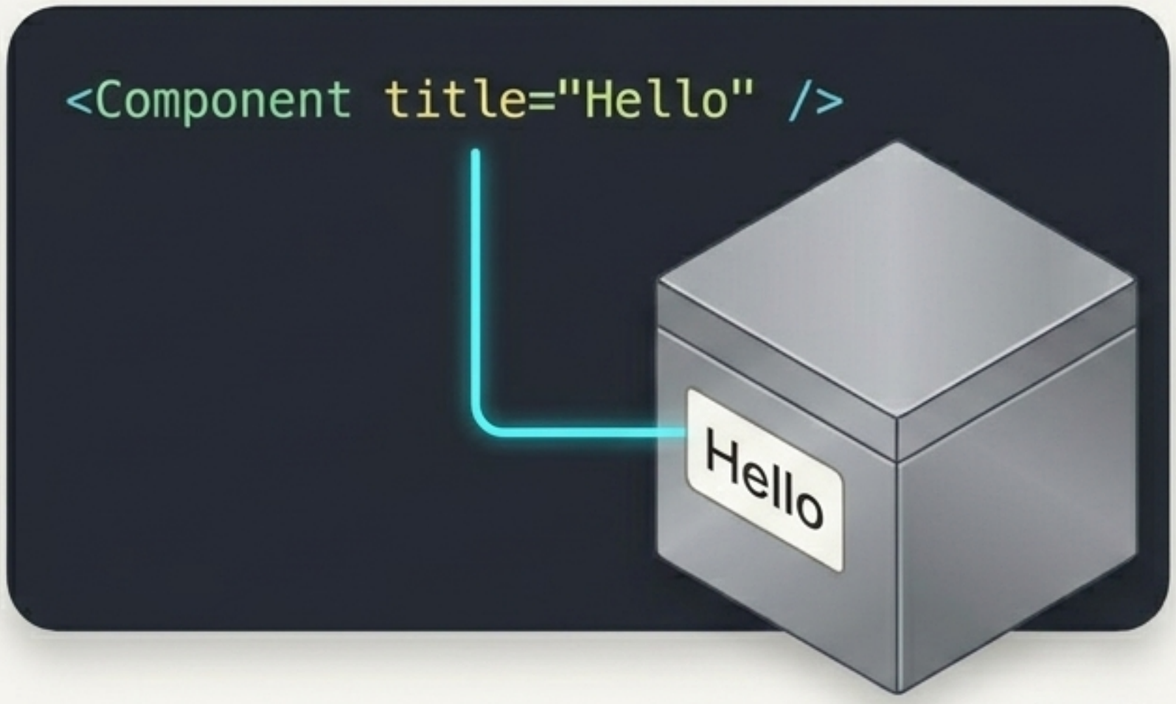
```
function Section({ title, children }) {
  return (
    <section>
      <h2>{title}</h2>
      <div>{children}</div>
    </section>
  );
}
```



The parent container handles the layout logic but remains entirely ignorant of its contents. It provides the space for whatever the application needs to render next.

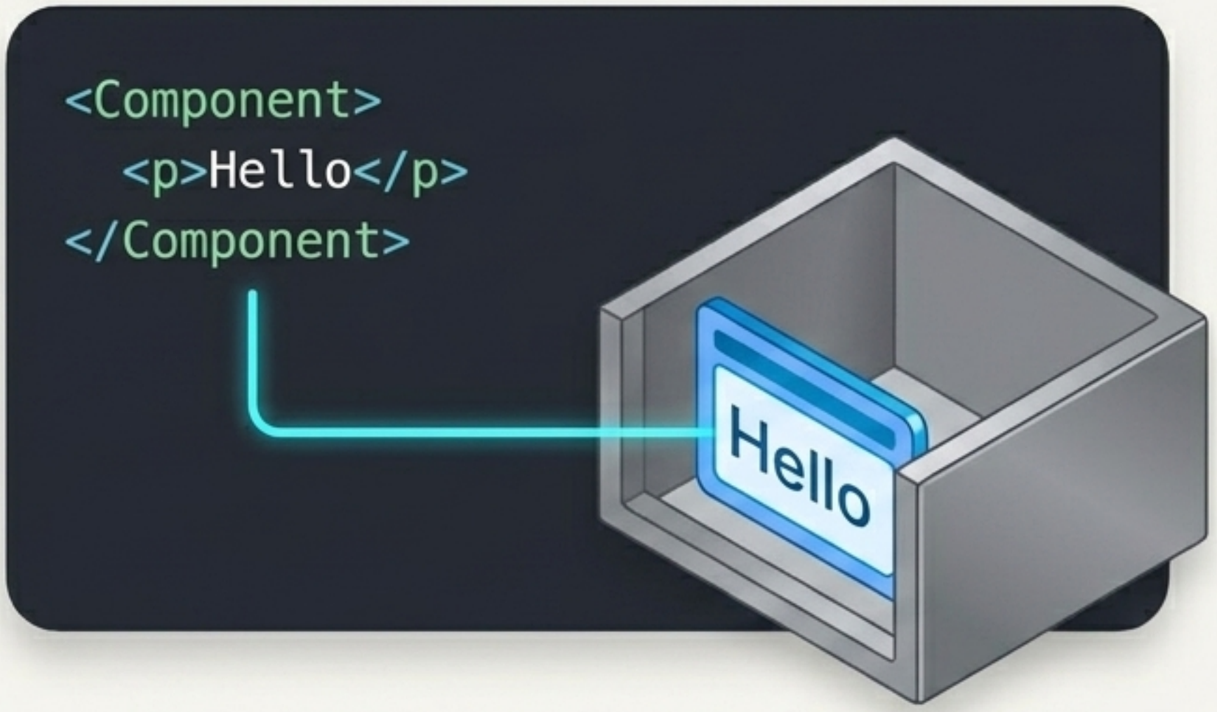
Nested composition replaces rigid attribute props

Passed as Attributes



Best for primitive data (strings, booleans) and configuration.

Passed as Nested Content



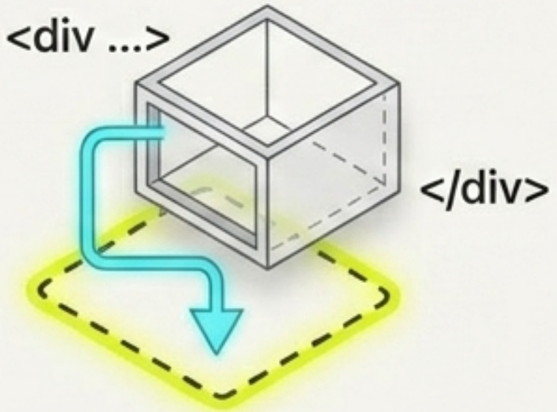
Best for UI layout, complex element structures, and dynamic rendering.

Children props make composition more natural and readable by mirroring actual HTML DOM nesting.

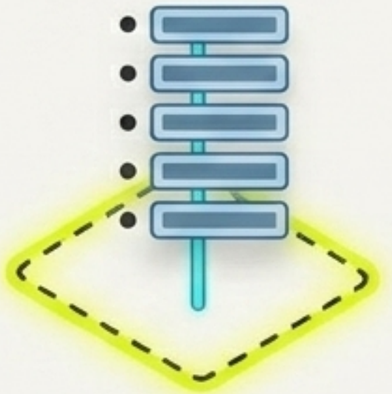
The slot accepts the entire renderable React universe

Strings & Numbers

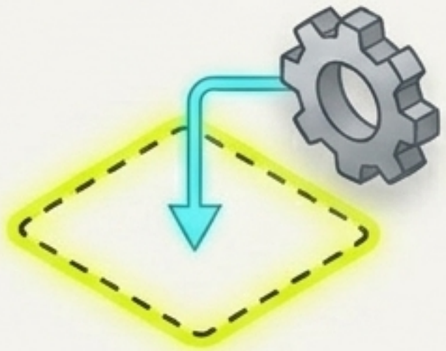
JSX Elements



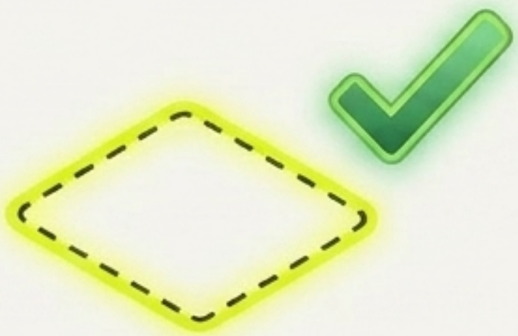
Arrays



Functions

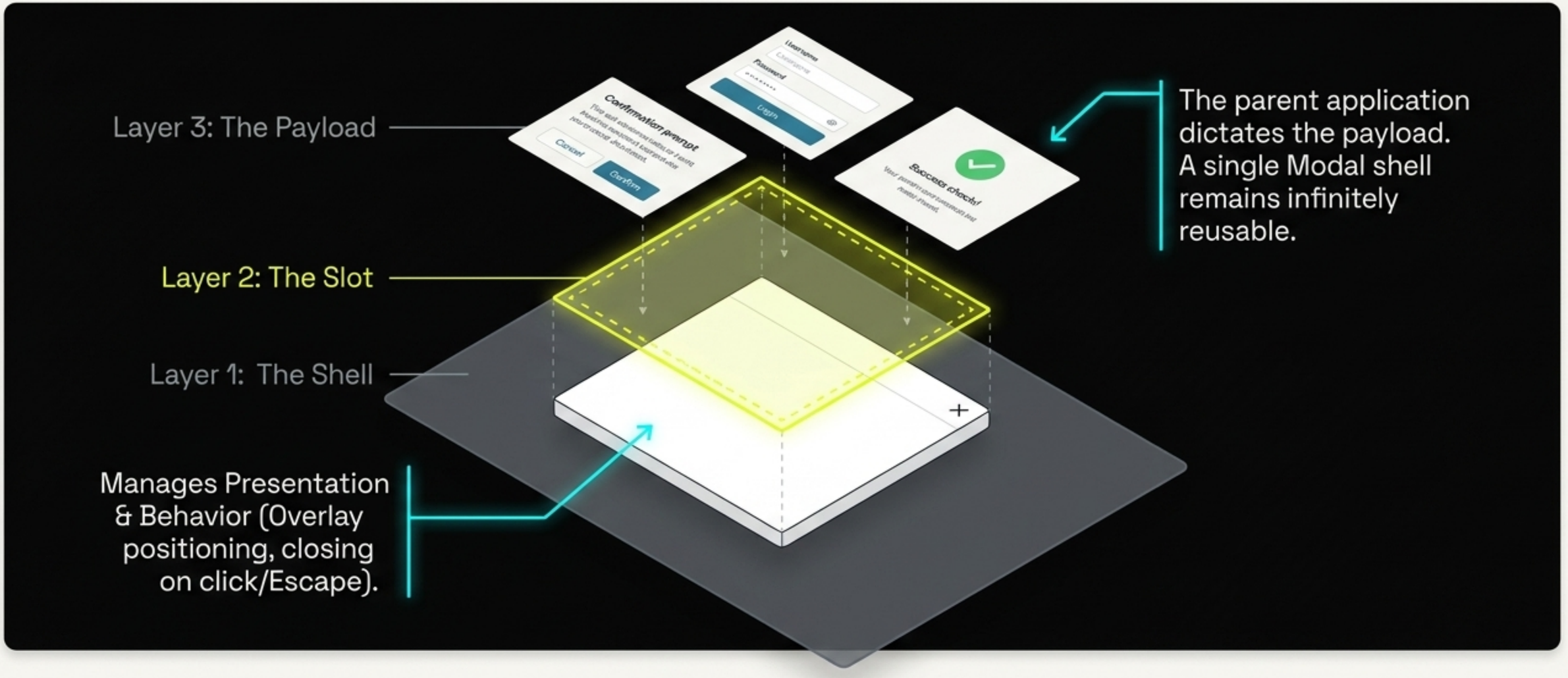


Null / Undefined



Anything React can render is a valid child. The name “children” is reserved by React, though additional slots can be created using regular named props.

Modals perfectly isolate presentation behavior from **dynamic** content



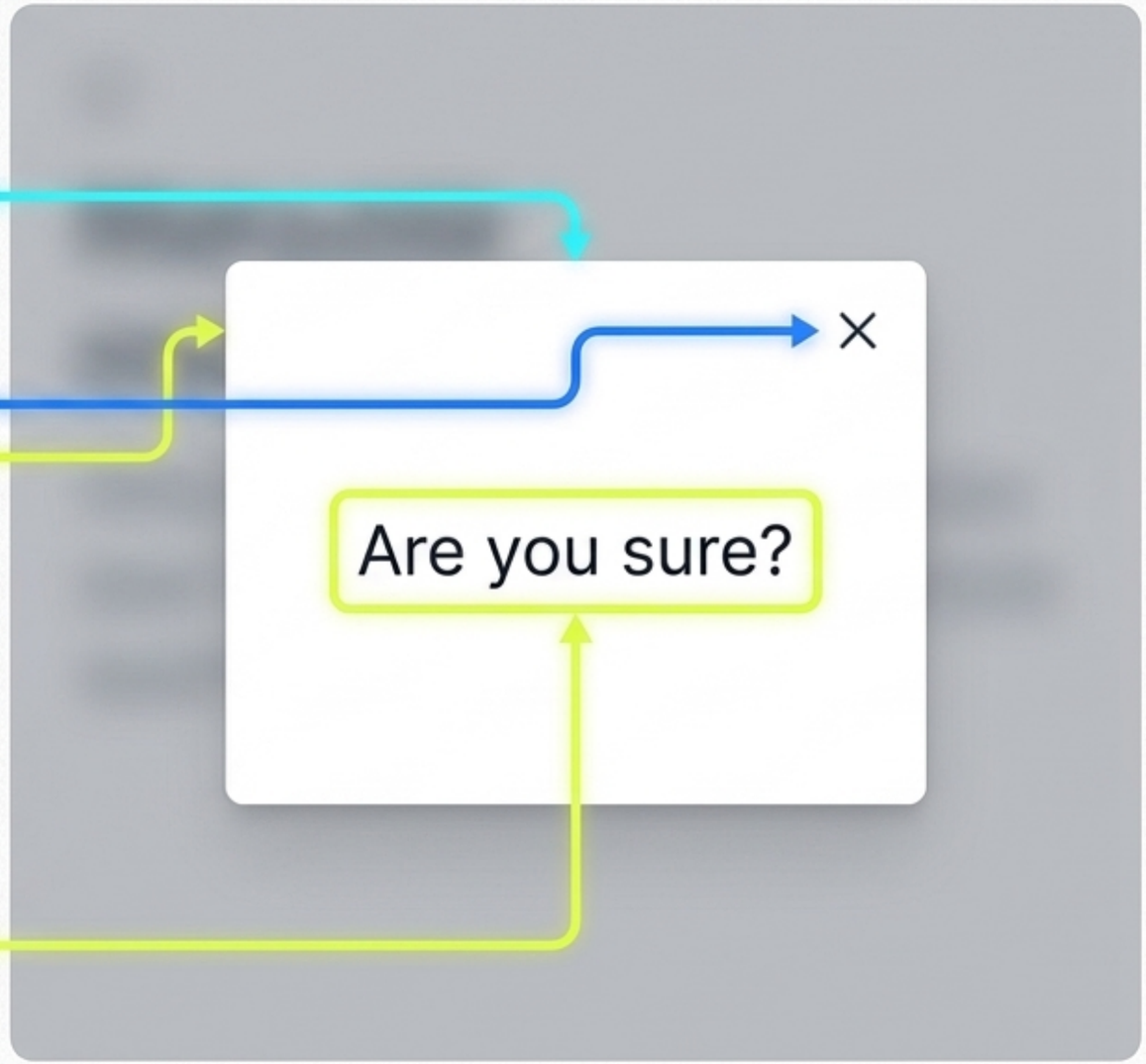
Wiring the modal component to the application state

Modal Component

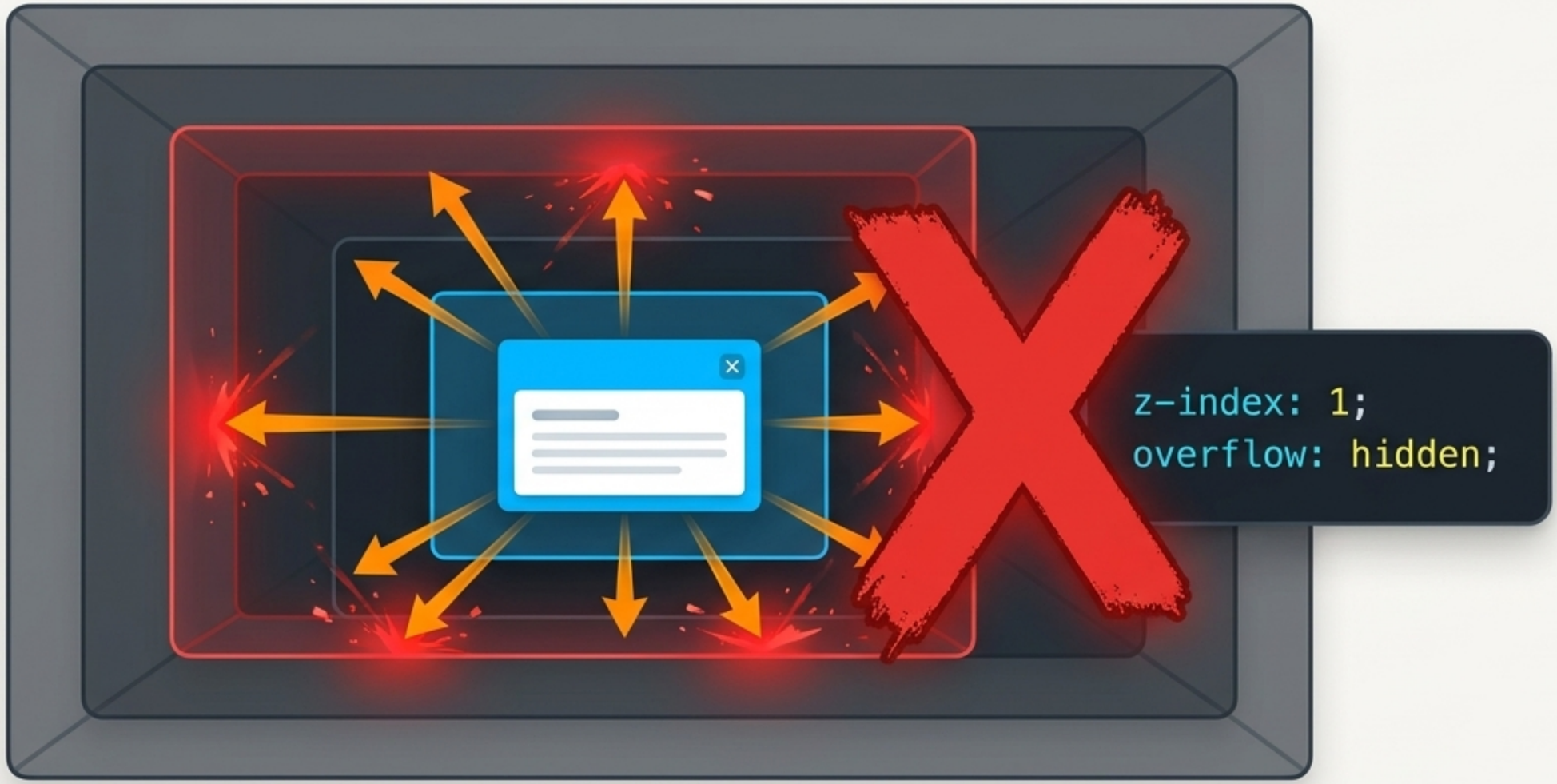
```
function Modal({ isOpen, children }) {
  if (!isOpen) return null;
  return (
    <div className="modal-overlay">
      <div className="modal-content">
        <button onClick={onClose}>✕</button>
        {children}
      </div>
    </div>
  );
}
```

Application Usage

```
<Modal isOpen={open}>
  <p>Are you sure?</p>
</Modal>
```

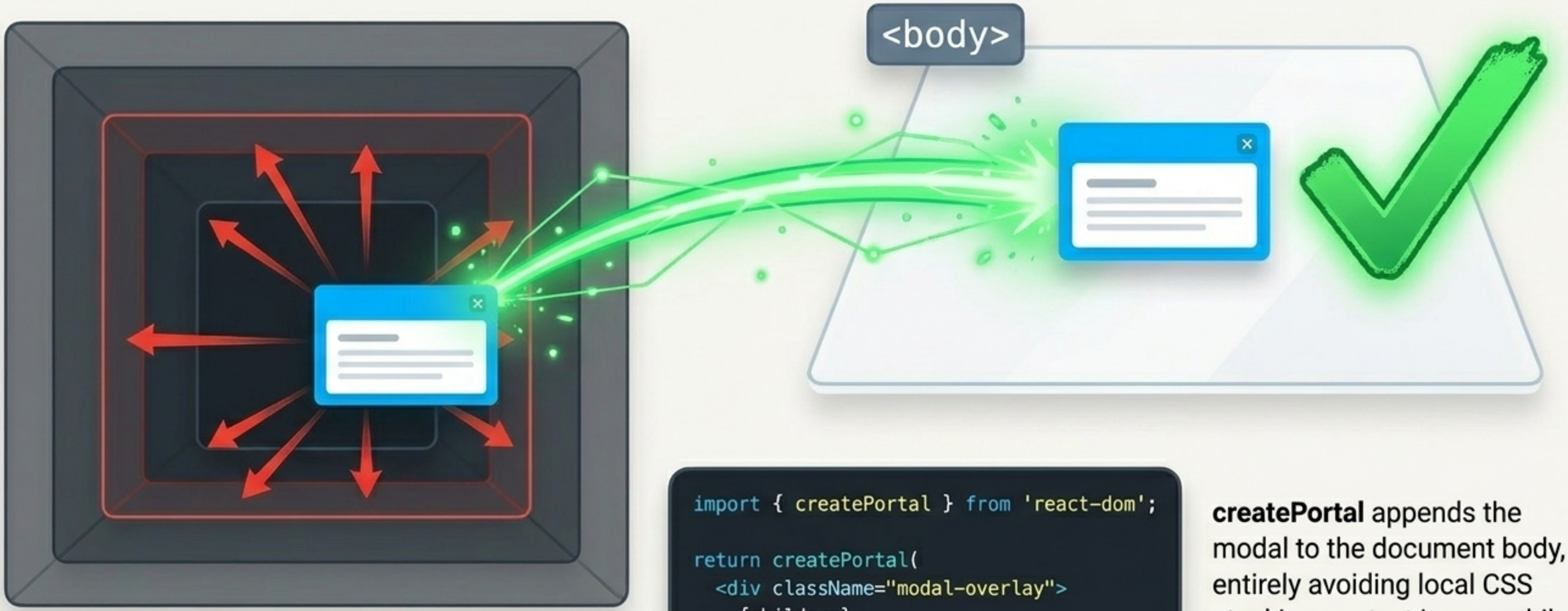


The DOM hierarchy traps deeply nested modal components



Because the modal is rendered inside the component tree, inherited CSS rules (like stacking contexts and overflow limitations) can clip or bury the overlay, ruining the user interface.

React Portals teleport the payload safely out of the nesting trap



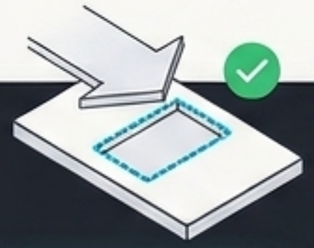

```
import { createPortal } from 'react-dom';

return createPortal(
  <div className="modal-overlay">
    {children}
  </div>,
  document.body
);
```

createPortal appends the modal to the document body, entirely avoiding local CSS stacking context issues while maintaining React's internal component state tree.

Locking down the slot with production-grade type validation

PropTypes (Vanilla JS)



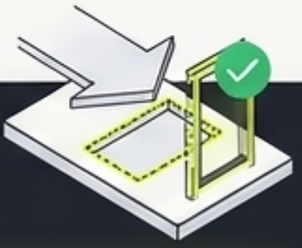
The Slot

```
import PropTypes from 'prop-types';

children: PropTypes.node.isRequired
```

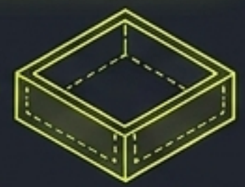
- Triggers runtime warnings in the browser console.
- 'node' perfectly covers the entire renderable universe (JSX, strings, arrays).

TypeScript



The Slot

```
interface Props {
  children: React.ReactNode;
}
```

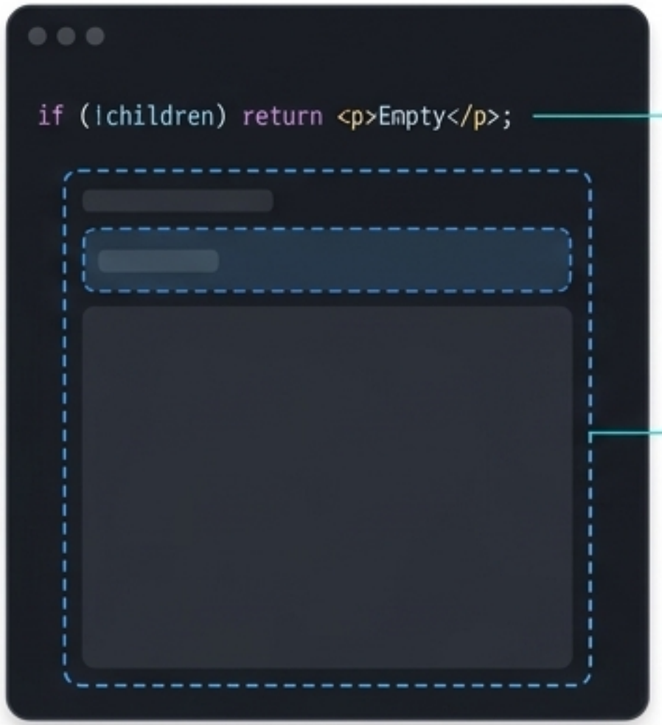


- Enforces strict compile-time checks before the app runs.
- TypeScript does not add children automatically; typing must be perfectly explicit in the component interface.

The Developer's Composition Cheatsheet

Checking Existence
How to know if the slot is full

```
if (!children) return <p>Empty</p>;
```

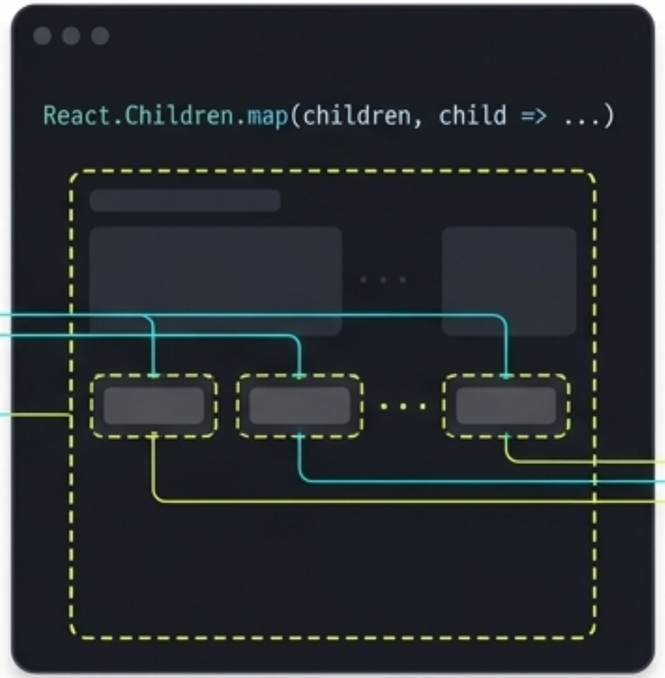


Alternate: React.Children.count(children)

Multiple Children
Dealing with arrays of elements

You can map over elements safely using React utilities.

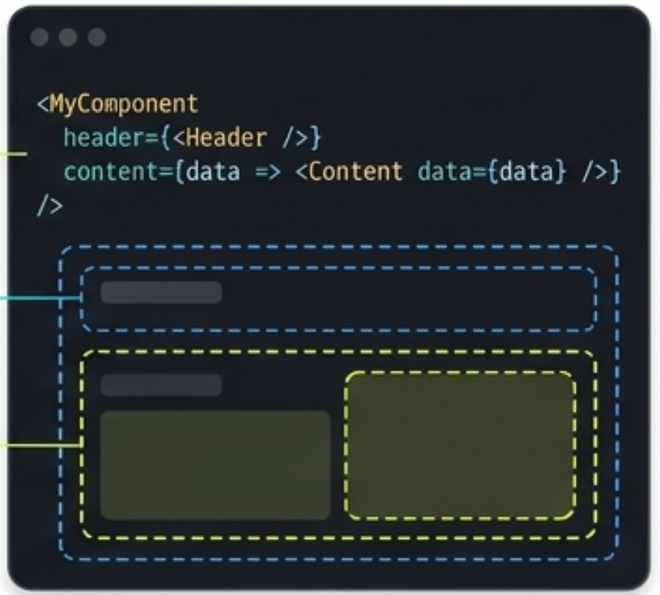
```
React.Children.map(children, child => ...)
```



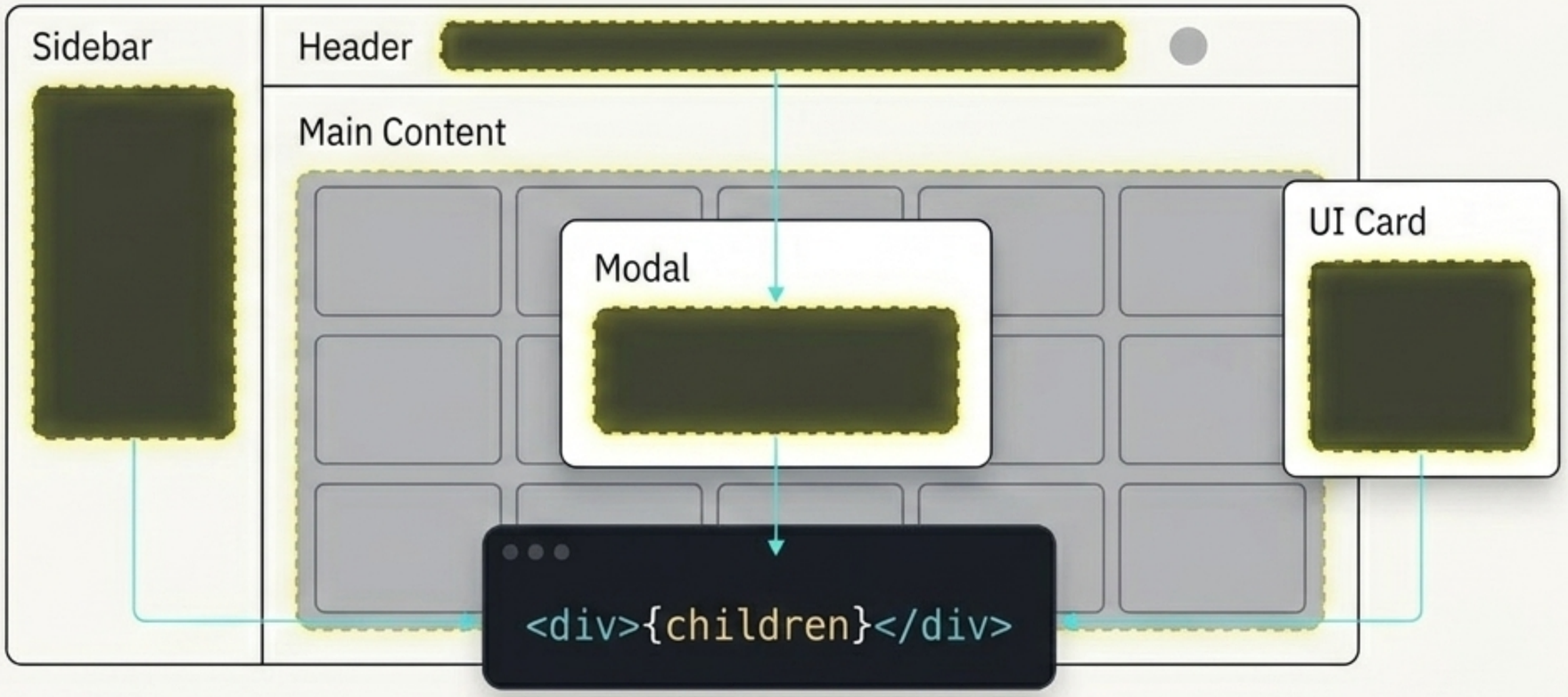
Advanced Patterns
Going beyond the basics

When {children} isn't enough, professional codebases utilize the "render props" pattern or pass multiple named components as standard attributes.

```
<MyComponent
  header={<Header />}
  content={data => <Content data={data} />}
/>
```



The Composition Engine builds predictable, scalable interfaces



The `children` prop is not just a syntax trick; it is the definitive React design pattern. By mastering it, you write less repetitive code, eliminate prop drilling, and build applications fundamentally designed for change.