17.1-Compound Component Pattern

19 March 2024 00:39

LECTURE
THE COMPOUND COMPONENT PATTERN

what the Compound Component pattern is all about. So, the idea of a Compound Component is that we can create a set of related components that together achieve a common and useful task,
for example, implementing a counter, such as we are going to do in this lecture. But of course, this pattern can also be used in all kinds of components that are actually more useful, for example modal windows, pagination, tables, and so on.
So basically, the way we implement this is that we create a parent component, and then a few different child components that really belong to the parent, and that really only make sense when used together with the parent component.

```
<select name="choice">
  <option value="first">First Value</option>
  <option value="second" selected>Second Value</option>
  <option value="third">Third Value</option>
</select>
```

And a good example of this is the HTML select and option elements. So the select element implements a select box and the option element implements each of the select options inside the select box. Right? So we have used this kind of HTML all the time. And so these option elements can really only be used inside a select element. So they only make sense within that element.
And so this is essentially exactly the same principle as we are going to use in compound components. And this will then allow us to implement highly flexible and highly reusable components with a very, very expressive API. And all without basically using no props at all.

```
<div>
  <h1>Compound Component Pattern</h1>
  <Counter
    iconIncrease="+"
    iconDecrease="-"
    label="My NOT so flexible counter"
    hideLabel={false}
    hideIncrease={false}
    hideDecrease={false}
    positionCount="top"
  />

  <Counter>
    <Counter.Decrease icon="-" />
    <Counter.Count />
    <Counter.Increase icon="+" />
    <Counter.Label>My super flexible counter</Counter.Label>
  </Counter>
</div>
```

This is the example for Compound - Component Pattern
1st Comp is our usual way of create a component which need several custom prop inOrder to Customize the Component. This will Lead to Problem called " Prop exposion".

2nd Comp is the example for the Compound - component See How we can Customize the Component, just by Changing the tag, according to our needs.

Counter.js | App.js | index.js

```
src > Counter.js > ...
1  import { createContext } from "react/cjs/react.development";
2  import { useContext } from "react/cjs/react.development";
3  import { useState } from "react/cjs/react.development";
4  // 1.create a context
5  const CounterContext = createContext();
6  // 2. create parent component
7  function Counter({ children }) {
8    const [count, setCount] = useState(0);
9    const increase = () => setCount((c) => c + 1);
10   const decrease = () => setCount((c) => c - 1);
11   return (
12     <CounterContext.Provider value={{ count, increase, decrease }}>
13       <span>{children}</span>
14     </CounterContext.Provider>
15   );
16 }
17 // 3. create child components to help implementing the common tasks
18 function Count() {
19   const { count } = useContext(CounterContext);
20   return <span>{count}</span>;
21 }
22 function Label({ children }) {
23   return <span>{children}</span>;
24 }
25 function Increase({ icon }) {
26   const { increase } = useContext(CounterContext);
27   return <button onClick={increase}>{icon}</button>;
28 }
29 function Decrease({ icon }) {
30   const { decrease } = useContext(CounterContext);
31   return <button onClick={decrease}>{icon}</button>;
32 }
33 // 4. Add child components as properties  to the parent component
34 Counter.Count = Count;
35 Counter.Label = Label;
36 Counter.Increase = Increase;
37 Counter.Decrease = Decrease;
38 export default Counter;
39
```

This is the Blue Print for creating a Compound - Component Pattern.

1st → Create a new Comp called <Counter /> in a separate file
→ Inside this Comp we are using some Usestate & derived fn() to set the useState. So now the Question is how we are going pass this useState value to Counter.Count, Counter.Label, Counter.Increa & Counter.Decre

And so again, how will this component here know what the current state is? Or also, how will this component right here know how to increase the state and this one how to decrease it? Well, as we just mentioned, we cannot use props for that. But thankfully for us, we have another solution, which is to actually use context. And yeah, you heard that right. We are going to use context to implement the compound component pattern. And so this is a really great use case for the context API.

2nd → Create a Context.
→ And enclose the {children} Comp by CounterContext.Provider & setting its value prop as {} obj count, increa, decrea. So that all children element can acces them values.

3rd → Creating the Child Comp function() & making child Comp as flexible as can, by providing custom prop, which can passed during the use of this <Comp />.

```jsx
export default function App() {
  return (
    <div>
      <h1>Compound Component Pattern</h1>
      <Counter>
        <Counter.Label>My super flexible counter</Counter.Label>
        <Counter.Count />
        <Counter.Increase icon="+" />
        <Counter.Decrease icon="-" />
      </Counter>
    </div>
  );
}
```

Compound Component Pattern

My super flexible counter3 + -

note :- using this child Propatie Ouside the parent Comp will not work, if the child comp uses any State values (0)) derived state values. if not it may work like counter.Label

```jsx
<Counter.Label>outside lable </Counter.Label>

<div>
  <Counter>
    <span>
      <Counter.Count />
    </span>
```

My super flexible counter3 + -

outside lable

0 + Age −

```jsx
<div>
  <Counter>
    <span>
      <Counter.Count />
    </span>
    <Counter.Increase icon="➕" />
    <Counter.Label>Age</Counter.Label>
    <Counter.Decrease icon="➖" />
  </Counter>
</div>
```

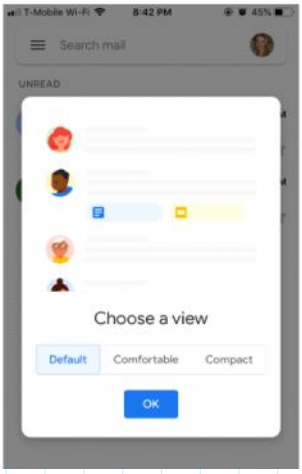
Compound Component Patt

My super flexible counter3 + -

0 ➕ Age ➖

passing emoji as a prop.

This is how we implement a counter as a compound component. Now, this might look pretty silly for just implementing a counter. And the truth is that this does require us to write a lot more code and it might be a bit harder to implement as well. But as we saw earlier, there are also many advantages to this.



What is a mobile modal?

Mobile modals (also known as modal windows, overlays, or pop-up messages) are a type of in-app messaging. They are large UI elements that sit on top of an application's main window—often with a layer of transparency behind them to give users a peek into the main app.

Up Next we are going to create a Modal Form component.

```jsx
import ...

function Cabins() {
  const [showForm, setShowForm] = useState(false);

  return (
    <>
      <Row type="horizontal">
        <Heading as="h1">All cabins</Heading>
        <p>Filter / Sort</p>
      </Row>
      <Row>
        <CabinTable />

        <Button onClick={() => setShowForm((show) => !show)}>
          Add new cabin
        </Button>
        {showForm && <CreateCabinForm />}
      </Row>
    </>
  );
}

export default Cabins;
```

→ So inside our Cabins.jsx Comp, has one useState Hook, to display the Form called <CreateCabinForm /> when button is pressed.

Now we want display this <CreateCabinForm /> Comp inside our <Modal /> Comp.

So we are copying this useState(), <Button /> & <CreateCabinForm />

```jsx
function AddCabin() {
  const [isOpenModal, setIsOpenModal] = useState(false);
  return (
    <div>
      <Button onClick={() => setIsOpenModal((show) => !show)}>
        Add new cabin
      </Button>
      {isOpenModal && (
        <Modal>
          <CreateCabinForm />
        </Modal>
      )}
```

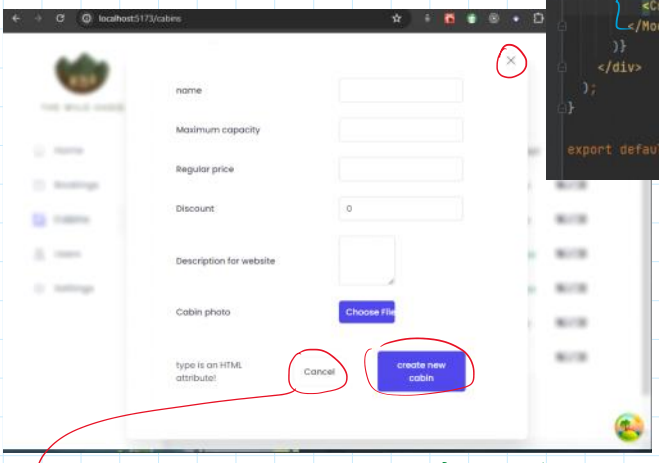
→ creating a new cabin Comp inside Cabin folder.

State name is changed

→ if isOpenModal is "true" then display modal Comp, which inturn <CreateCabinForm />

& pasting the State hook & Button & CreateCabin Form Comp is pasted here.

```
export default Cabins;
```




```
        <Modal>
          <CreateCabinForm />
        </Modal>
      )}
    </div>
  );
}

export default AddCabin;
```

then display modal comp, which inturn ↙ CreateCabinForm />

now.

```
function Cabins() {
  return (
    <>
      <Row type="horizontal">
        <Heading as="h1">All cabins</Heading>
        <p>Filter / Sort</p>
      </Row>
      <Row>
        <CabinTable />
        <AddCabin />
      </Row>
    </>
  );
}
```

→ inside Cabin Comp, we are calling ↙ AddCabin /> comp. then AddCabin return a ↙ btn/> & modal comp.

So now need to updat isOpenModal State & close it comp. when then buttons are clicked.

```
function AddCabin() {
  const [isOpenModal, setIsOpenModal] = useState( initialState: false);
  return (
    <div>
      <Button onClick={() => setIsOpenModal( value: (show : boolean ) => !show)}>
        Add new cabin
      </Button>
      {isOpenModal && (
        <Modal onClose={() => setIsOpenModal( value: false)}>
          <CreateCabinForm onCloseModel={() => setIsOpenModal( value: false)} />
        </Modal>
      )}
    </div>
  );
}
```

So this State is present in the AddCabin Comp. & Our "close" btn present in ↙model comp/> & "Cancel" & "Create new cabin" btn is in ↙CreateCabinForm /> Comp.

So we are paasing the setIsOpenModal() as a Prop

```
function Modal({ children, onClose }) {
  return (
    <Overlay>
      <StyledModal>
        <Button onClick={onClose}>
          <HiXMark />
        </Button>

        <div>{children}</div>
      </StyledModal>
    </Overlay>
  );
}

export default Modal;
```

```
<FormRow>
  type is an HTML attribute!
  <Button
    onClick={() => onCloseModel?.()}
    variation="secondary"
    type="reset"
  >
    Cancel
  </Button>
  <Button disabled={isWorking}>
    {isEditSession ? "Edite cabin" : "create
  </Button>
</FormRow>
</Form>
```

→ inside Create Cabin Form

→ during Form Submission

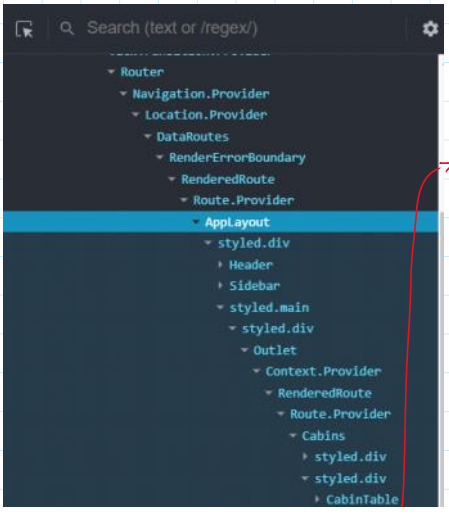
```
function onSubmit(data) {
  console.log(data);
  const image = typeof data.image === "string" ? data.

  if (isEditSession)
    editCabin(
      args: { newCabinData: { ...data, image }, id: edit
      {
        onSuccess: (data) => {
          console.log(data);
          reset();
          onCloseModel?.();
        },
      },
    );
  else
    createCabin(
      args: { ...data, image: image },
      options: {
        onSuccess: (data) => {
          console.log(data);
          reset();
          onCloseModel?.();
        },
```

→ we are using optional chaining because, this ↙CreateCabinForm/> comp can be used in multiple places where it may not get accss to onCloseModel fns to update the state.

```
return (
  <Form
    onSubmit={handleSubmit(onSubmit, onError)}
    type={onCloseModel ? "modal" : "regular"}
  >
    <FormRow label="name" error={errors?.name?.messa
```

→ changing style of the Form according the "type prop".

```
const Form = styled.form`
  ${(props
    props.type === "regular" &&
    css
      padding: 2.4rem 4rem;

      /* Box */
      background-color: var(--color-grey-0);
      border: 1px solid var(--color-grey-100);
      border-radius: var(--border-radius-md);
    }

  ${(props) =>
    props.type === "modal" &&
    css
      width: 80rem;
    }

  overflow: hidden;
  font-size: 1.4rem;
`;

export default Form;
```



React Portal :-

→ So when we inspect our DOM Tree we can see that our model comp inside the AddCabin under all these other Comp.

But now with a React portal, as I was just saying we can basically render this component completely outside of this DOM structure and really render it anywhere that we want.

↳ document.body

→ document.body

→ `const el = document.querySelector(".myclass");`

createPortal

`createPortal` lets you render some children into a different part of the DOM.

```
<div>
  <SomeComponent />
  {createPortal(children, domNode, key?)}
</div>
```

`createPortal(children, domNode, key?)`

JSX

To create a portal, call `createPortal`, passing some JSX, and the DOM node where it should be rendered:

```
function Modal({ children, onClose }) {
  return createPortal(
    <Overlay>
      <StyledModal>
        <Button onClick={onClose}>
          <HiXMark />
        </Button>

        <div>{children}</div>
      </StyledModal>
    </Overlay>,
    document.body, domNode
  );
}

export default Modal;
```

JSX

directly under body

Results

```
<html lang="en">
▶ <head> ⋯ </head>
▼ <body>
  ▼ <div id="root">
    ▶ <div class="tsqd-parent-container" style="--tsqd-panel
      31px; --tsqd-panel-width: 500px; --tsqd-font-size: 16p
      </div>
    ▶ <div class="sc-lnsjTu eYvTpm">⋯</div> grid
      <div style="position: fixed; z-index: 9999; inset: 16p
      -events: none; margin: 8px;"></div>
    </div>
    <script type="module" src="/src/main.jsx?t=1710851558064
    </script>
  ▼ <div class="sc-dJGMql gOIPrl"> == $0
    ▶ <div class="sc-ifyrAs sMZac">⋯</div>
    </div>
  </body>
▶ <quillbot-extension-portal style="display: initial; visibi
```

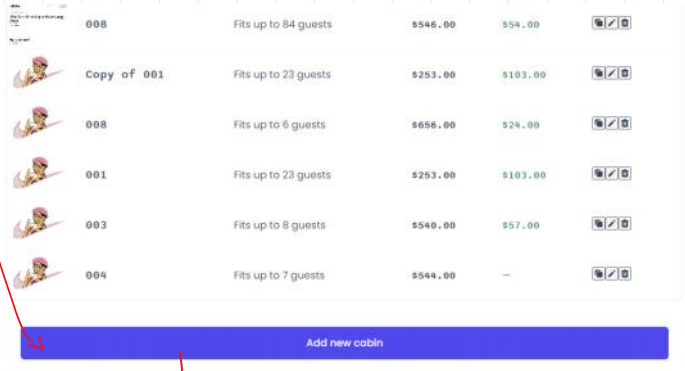
model Comp

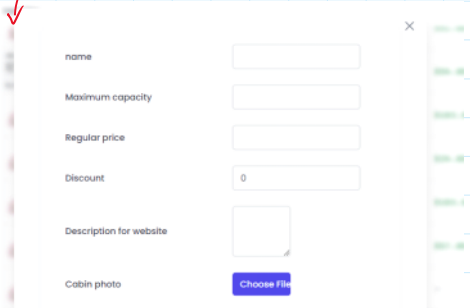
And so why do we even need to use this portal? Well, the main reason why a portal becomes necessary is in order to avoid conflicts with the CSS property overflow set to hidden. So many times we build a component like a modal and it works just fine, but then some other developer will reuse it somewhere else and that somewhere else might be a place where the modal will get cut off by a overflow hidden set on the parent. So this is basically all about reusability and making sure that the component will never be cutoff
by an overflow property set to hidden on some parent element. So in order to avoid this kind of situation we simply render the modal completely outside of the rest of the DOM.

LECTURE
CONVERTING THE MODAL TO A COMPOUND COMPONENT

And the reason why that is necessary is that this modal that we have built is really not ideal when it comes to the state management and to the way in which we actually render this modal. So remember how we render the modal right here based on this isOpenModal state. Now the problem with this is that we really do not want the parent component of <modal /> to be responsible for creating this piece of state and to keep track of whether the modal is open or not. So again, it shouldn't be the task of the AddCabin component here to track whether right now the modal should be displayed or not. So instead, the modal component itself should actually know whether it is currently open or not, and so it should keep this state internally. this basically encapsulate the elements or tags inside the Model component. And then the component should give us simply a way to open the modal and also a way to pass in the content that we actually want to display inside the modal. So basically we want some button to open the modal, and we want the window itself. So these two components together should form the modal component. And if this sounds a lot like the compound component pattern,

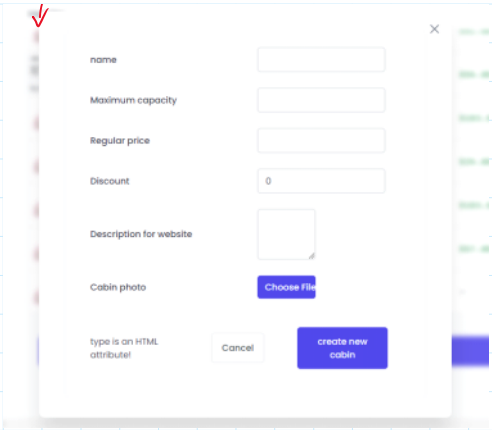
```
function AddCabin() {
  const [isOpenModal, setIsOpenModal] = useState( initialState: false);
  return (
    <div>
      <Button onClick={() => setIsOpenModal( value: (show :boolean ) => !show)}>
        Add new cabin
      </Button>
      {isOpenModal && (
        <Modal onClose={() => setIsOpenModal( value: false)}>
          <CreateCabinForm onCloseModel={() => setIsOpenModal( value: false)} />
        </Modal>
      )}
    </div>
  );
}
```





In our pervious code this is the Button used to show the <Modal /> comp

Now we want these <Button /> & <Modal /> Comp to be a Compound-Comp & work independently without the of parent Comp, which is <AddCabin /> & stat used inside AddCabin Comp.



In Our perivous code This is the Button used to show the <Modal /> comp

Now we want them <Button /> & <Modal /> Comp to be a Compound-Comp & work independenty without the of parent Comp, which is <AddCabin /> & stat used inside AddCabin Comp.

```jsx
function AddCabin() {
  return (
    <Modal>
      <Modal.Open opens="cabin-form">
        <Button>Add new cabin</Button>
      </Modal.Open>

      <Modal.Window name="cabin-form">
        <CreateCabinForm />
      </Modal.Window>
    </Modal>
  );
}
```

→ So Our AddCabin should look Some thing like this.

→ Compound - Comp

Compound - propertie

Compound - propertie

```jsx
// Procedure for creating a COMPOUND COMPONENT
// 1. Creating the Context
const ModalContext = createContext();

// 2. Create parent component
function Modal({ children }) {
  const [openName, setOpenName] = useState( initialState: "");

  const close = () => setOpenName( value: "");
  const open = setOpenName;

  return (
    <ModalContext.Provider value={{ openName, close, open }}>
      {children}
    </ModalContext.Provider>
  );
}
```

→ Inside Our <Modal /> comp file.

→ This is the name which provide while calling <Modal.Open> & <Modal.window> to identify them & to open appropriat <window/> of its <button/>

→ enclosing the childruns of <Modal /> comp within the Context.Provider.

```jsx
// 3. create child component to help implementing the common tasks
function Open({ children, opens: opensWindowName }) {
  const { open } = useContext(ModalContext);

  return children;
}
```

before

```jsx
// 3. create child component to help implementing the common tasks
function Open({ children, opens: opensWindowName }) {
  const { open } = useContext(ModalContext);
  After
  return cloneElement(children, props: { onClick: () => open(opensWindowName) });
}
```

```jsx
function AddCabin() {
  return (
    <Modal>
      <Modal.Open opens="cabin-form">
        <Button>Add new cabin</Button>
      </Modal.Open>

      <Modal.Window name="cabin-form">
```

→ Modal.Open passes <Button> comp as its child which gets return from the Open Comp fn()
But how to add the onclick fn() to this <Button/> Comp. to Open the <window /> comp.

cloneElement → This is where we use Advanc React funtion called clone Element.

Pitfall

Using `cloneElement` is uncommon and can lead to fragile code. See common alternatives.

`cloneElement` lets you create a new React element using another element as a starting point.

```
const clonedElement = cloneElement(element, props, ...children)
```

```jsx
// 3. create child component to help implementing the common tasks
function Open({ children, opens: opensWindowName }) {
  const { open } = useContext(ModalContext);

  return cloneElement(children, props: { onClick: () => open(opensWindowName) });
}
```

→ accessing "Open" fun() from context
→ By the help of "CloneElement" we are returning the cloned Button tag along with onClick proop with update state fun() in it.

Which will be Similar to <Button OnClick= () => Open(opens windowNam)>"Add new cabin" </Button>

onClick prop with update state fn() init.

Which will be Similar to <Button OnClick= () => Open(opensWindowNam) >"Add new cabin" </Button>

```jsx
function AddCabin() {
  return (
    <Modal>
      <Modal.Open opens="cabin-form">
        <Button>Add new cabin</Button>
      </Modal.Open>

      <Modal.Window name="cabin-form">
```

```jsx
// 3. create child component to help implementing the common tasks
function Open({ children, opens: opensWindowName }) {
  const { open } = useContext(ModalContext);

  return cloneElement(children, { onClick: () => open(opensWindowName) });
}
```

which will look like

```jsx
<Button onClick={() => open("cabin-form")}>Add new cabin</Button>
```

Great!, Now we have passed the useState update fn() to the <Button /> comp, which is not even in the AddCabin.jsx file. without moving up the State logic to the parent element. Now we can when button is click, it should update OpenName state.

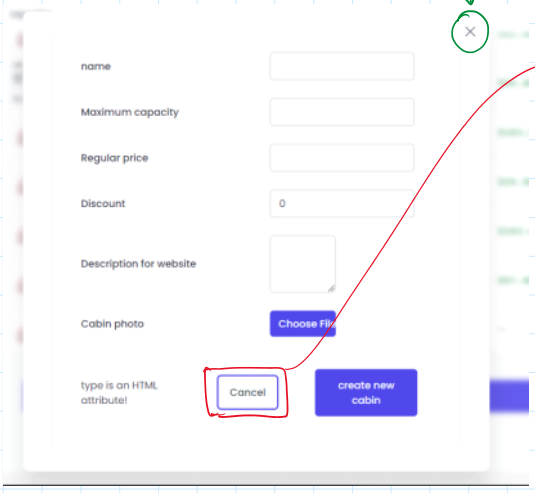
```jsx
function AddCabin() {
  return (
    <Modal>
      <Modal.Open opens="cabin-form">
        <Button>Add new cabin</Button>
      </Modal.Open>

      <Modal.Window name="cabin-form">
        <CreateCabinForm />
      </Modal.Window>
    </Modal>
  );
}
```

```jsx
function Window({ children, name }) {
  const { openName, close } = useContext(ModalContext);

  if (name !== openName) return null;

  return createPortal(
    <Overlay>
      <StyledModal>
        <Button onClick={close}>
          <HiXMark />
        </Button>

        <div>{children}</div>
      </StyledModal>
    </Overlay>,
    document.body,
  );
}
```

→ if button name doesn't match then it not render the <window /> Comp.

→ accessing "openName" value & close fn() from Context.

calling close fn() during onClick.



But Our Cancel button, should also closes the window Comp. Again the State value is present in <AddCabin/> comp & we need pass the setOpenName() or close() function to update the State value.

Now again we are going to use the CloneElement.

```jsx
function Window({ children, name }) {
  const { openName, close } = useContext(ModalContext);

  if (name !== openName) return null;

  return createPortal(
    <Overlay>
      <StyledModal>
        <Button onClick={close}>
          <HiXMark />
        </Button>

        <div>{cloneElement(children, { onCloseModel: close })}</div>
      </StyledModal>
    </Overlay>,
    document.body,
  );
}
```

whenever onCloseModel prop is called inside the <CreateCabinForm/> comp it calls the close fn() here.

child Component

```jsx
function CreateCabinForm({ cabinToEdit = {}, onCloseModel }) {
  const { id: editId, ...editValues } = cabinToEdit;
  const isEditSession = Boolean(editId);

  const { register, handleSubmit, reset, getValues, formState } = useForm({
    defaultValues: isEditSession ? editValues : {},
  });

  const { errors } = formState;
```

Last step.

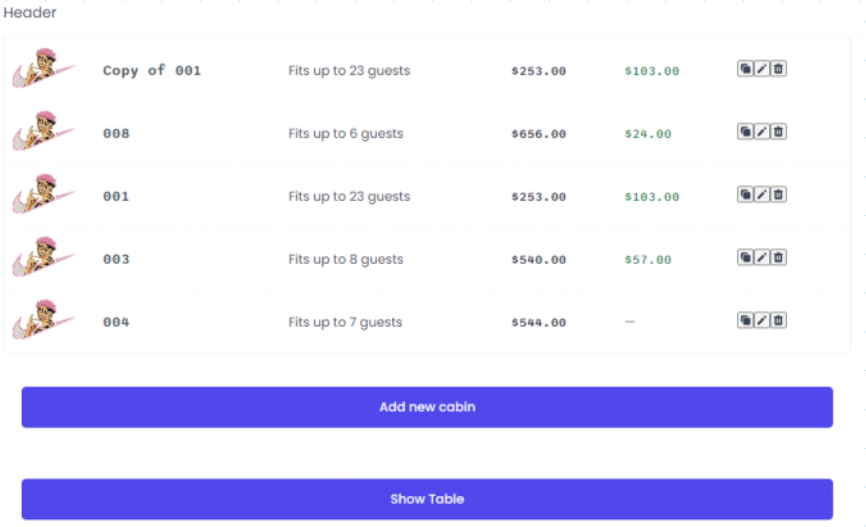
```jsx
// 4. Add child components as properties to the parent component
Modal.Open = Open;
Modal.Window = Window;

export default Modal;
```

```
if (isEditSession)
  editCabin(
    args: { newCabinData: { ...data, image }, id: editId },
    {
      onSuccess: (data) => {
        console.log(data);
        reset();
        onCloseModel?.();
      },
    },
  );
else
  createCabin(
    args: { ...data, image: image },
    options: {
      onSuccess: (data) => {
        console.log(data);
        reset();
        onCloseModel?.();
      },
    },
  );
}
```

→ calls onCloseModel After Edit, which update "OpenName State" inside the Model.

calls After the Cabin creation.

```
function AddCabin() {
  return (
    <Modal>
      <Modal.Open opens="cabin-form">
        <Button>Add new cabin</Button>
      </Modal.Open>

      <Modal.Window name="cabin-form">
        <CreateCabinForm />
      </Modal.Window>

      {/*  Displaying one more button and window  */}
      <Modal.Open opens="table">
        <Button>Show Table</Button>
      </Modal.Open>

      <Modal.Window name="table">
        <CabinTable />
      </Modal.Window>
    </Modal>
  );
}
```



→ now are showing ShowTable button, which opens the <CabinTable />

Now we pass any <Button /> and any Component to be displayed inside Our Model Comp, which is a Compound-Comp.

So don't focus too much on the specific implementation details of this specific modal component. So when you use this compound component pattern in the future to maybe build your own components, then those won't be maybe modal components. And so really what I want you to retain the most from this lecture is really how we can use these different basically child components to together achieve a goal. So in this case, displaying a modal window. And it's always in the same way. So we have some state and some state updating functions here in the parent component. Then we pass that with a context, and then in the child components, we use those to achieve the goal of displaying a modal window in this case.

Certainly! The **Compound Component Pattern** in React is a powerful technique that allows you to create reusable components by composing them together to build more complex user interfaces. Here are some use cases where you can leverage this pattern:

1. **Navigation Bar**:
   - Create a reusable navigation bar component that encapsulates navigation links, dropdown menus, and other related elements.
   - Users can customize the appearance and behavior of individual navigation items while maintaining a consistent overall design.
2. **Tabs and Tab Panels**:
   - Implement a tabbed interface where each tab is a compound component.
   - Users can easily switch between different content panels while keeping the logic and styling encapsulated within the tab components.
3. **Accordion or Collapsible Sections**:
   - Build an accordion component that allows users to expand or collapse sections of content.
   - Each section header and its associated content can be separate compound components.
4. **Modal Dialogs**:
   - Design a modal dialog with customizable content (title, body, buttons, etc.).

    - Users can compose their own modals by combining predefined modal components.
5. **Form Fields and Validation**:
    - Create form input components (e.g., text input, checkbox, select) that work together seamlessly.
    - Users can easily build complex forms by composing these components, and validation logic can be shared among them.
6. **Product Carousel or Slideshow**:
    - Develop a carousel component that displays product images or other content.
    - Each slide can be a compound component, allowing customization of content, navigation, and animation.
7. **Dropdown Menus**:
    - Construct dropdown menus with various options.
    - Users can define their own menu items and customize their behavior.
8. **Date Pickers and Time Selectors**:
    - Implement date and time input components.
    - Users can combine these components to create flexible date pickers or time selectors.
9. **Accordion Tabs for FAQs or Documentation**:
    - Build an FAQ section or documentation page with collapsible sections.
    - Each FAQ item or documentation section can be a compound component.
10. **Customizable Alerts and Notifications**:
    - Design alert or notification components with different styles (success, warning, error, etc.).
    - Users can compose alerts with specific messages and appearance.

Remember that the Compound Component Pattern promotes separation of concerns, making it easier to reason about and test your components. By defining a set of child components, you provide a flexible API for customization and control within your application[1234].

From <https://www.bing.com/chat?form=NTPCHB>

From <https://www.bing.com/chat?form=NTPCHB>