




Modal Dialog IV Solution

React ▾



Yangshun Tay 
Ex-Meta Staff Engineer



Hard



30mins



121 done

Solution

We'll build on top of **Modal Dialog III's solution**. Like in Modal Dialog III, only interactions need to be added; the styling and structure can remain the same.

The following behaviors need to be implemented:

1. Upon dialog open, focus on the first tabbable element
2. Focus trapping – the focus is trapped within the dialog and cannot escape the dialog
3. Upon dialog close, focus returns to the element that invoked the dialog

1. Upon dialog open, focus on the first tabbable element

This functionality is quite straightforward to implement. Upon mounting of the component, query all the elements and focus on the first tabbable element. The tabbable elements within the dialog can be selected with the following selector:

```
dialogEl.querySelectorAll(
  'button, [href], input, select, textarea, [tabindex]:not([tabindex="-1"])',
);
```

This is implemented as the `useFocusOnFirstTabbableElement` hook.

2. Focus trapping

Focus trapping is a technique used to control and manage keyboard focus within a specific area or component of a webpage or application (e.g. modal dialogs, dropdown menus).

Practically, the following behavior is implemented:

- When focus reaches the last element, **Tab** moves focus to the first tabbable element
- When focus is on the first element, **Shift** + **Tab** moves focus to the last tabbable element
- All interactive elements within the trapped area remain focusable

- Elements outside the trapped area are temporarily made unfocusable

Tabbing between the non-boundary elements can remain as per normal. Only the first and last elements need special handling. We need to hijack the tabbing events on these boundary elements and programmatically focus on the required elements instead of allowing the browser to decide (which relies on DOM order).

We can add a `keydown` event listener to the dialog and filter for `Tab` event presses. In the callback:

1. If the currently focused element (via `document.activeElement`) is the last tabbable element and the `Tab` key is pressed, focus on the first tabbable element instead.
2. If the currently focused element (via `document.activeElement`) is the first tabbable element and the `Shift` + `Tab` keys are pressed (by checking the `event.shiftKey` property), focus on the last tabbable element instead.

To find all the tabbable elements, the same selector as per part 1 can be used. Note that it is advisable to find all the tabbable elements on-demand, during the `Tab` key event, rather than at the start when the dialog is mounted and retaining a reference to the list. This is because the contents of the dialog can be modified anytime; the list of tabbable elements on mount might not be the same as when the user actually tabs through them.

Before changing the focus programmatically, remember to call `event.preventDefault()`, otherwise the browser will proceed with the default behavior – focus on the next element determined by the DOM order.

This is implemented as the `useFocusTrap` hook.

3. Upon dialog close, focus returns to the element that invoked the dialog

The currently focused element on the page can be obtained using `document.activeElement`. When the dialog is mounted, save a reference to that element (if it exists). On unmounting, programmatically focus on that element.

This is implemented as the `useReturnFocusToTrigger` hook. An important thing to note is that this hook has to be **called before** `useFocusOnFirstTabbableElement`, otherwise the focus would have been set on the first tabbable element, even before the hook gets a chance to save the reference to the focus trigger.

Test cases

1. Dialog opening
 - Verify that focus moves to the first focusable element in the dialog when it opens
 - Ensure that focus is trapped within the dialog
 - Check that elements outside the dialog are not focusable

2. Focus order

- Confirm that **Tab** key moves focus through elements in a logical order
- Verify that **Shift** + **Tab** reverses the focus order
- Ensure that all interactive elements within the dialog are focusable

3. Focus trapping

- Test that focus wraps from the last to the first element when tabbing
- Verify that focus wraps from the first to the last element when using **Shift** + **Tab**

4. Dialog closing

- Confirm that pressing **Esc** key closes the dialog
- Verify that clicking outside the dialog closes it
- Ensure that focus returns to the element that opened the dialog

Accessibility

Congratulations, we have implement almost everything on **Dialog (Modal) | ARIA Authoring Practices Guide**. Do read the page, there are some nuances related to the usage of `aria-describedby` attribute and automatic focus placements.

Resources

- [Dialog \(Modal\) | ARIA Authoring Practices Guide](#)
- [Dialog – Radix Primitives](#)
- [Dialog | Reach UI](#)
- [Dialog - Headless UI](#)

Follow up

A possible problematic scenario is having multiple dialogs open at once / stacked because the open dialog's contents allow opening of another dialog.

A limitation of our current implementation is surfaced – since most of the event listeners are added on the global level, all the open dialogs might respond to a **Escape** event and close at the same time, when the expected behavior is that only the topmost dialog closes.

Fixing this will require code to track the stack of open dialogs and only close the topmost dialog. However, implementing this is non-trivial and usually beyond the scope of interviews. That said, mentioning this scenario to the interviewer will probably give you brownie points!

public

- index.html

src

- App.tsx
- ModalDialog.tsx
- index.tsx
- styles.css

package.json

tsconfig.json

App.tsx [ModalDialog.tsx](#) styles.css

```
import {
  ComponentProps,
  RefObject,
  useEffect,
  useId,
  useRef,
} from 'react';
import { createPortal } from 'react-dom';

export default function ModalDialog({
  open = false,
  ...props
}: Readonly<{
  open?: boolean;
}> &
  ComponentProps<typeof ModalDialogImpl>) {
  if (!open) {
    return null;
  }

  return <ModalDialogImpl {...props} />;
}

/**
 * Invokes a function when a key is pressed.
 */
function useOnKeyDown(
  key: string,
  fn: (event: KeyboardEvent) => void,
) {
  useEffect(() => {
    function onKeyDown(event: KeyboardEvent) {
      if (event.key === key) {
        fn(event);
      }
    }

    document.addEventListener('keydown', onKeyDown);

    return () => {
      document.removeEventListener('keydown', onKeyDown);
    };
  }, [fn]);
}

/**
```