

## forwardRef

- forwardRef is a utility function provided by React that enables you to pass a ref through a component to one of its child components.
- This is particularly useful when you want a parent component to directly interact with a DOM node or a child component's reference.
- React's default behavior does not allow passing ref props to functional components directly. forwardRef is used to overcome this limitation by forwarding the ref received by a parent to a child component.

### Key use cases

- **Accessing DOM nodes in child components:** Helps manage focus, animations, or other DOM manipulations in child components.
- **Custom components with external libraries:** Integrate custom components with libraries requiring access to native DOM elements.
- **Abstraction and Reusability:** Allows creating reusable components that still provide DOM access when required.

### Problems Solved by forwardRef

- Passing ref to functional components.
- Maintaining a clean component hierarchy while still providing DOM manipulation capabilities.
- Avoiding tightly coupling the parent to the child component's implementation details.

### Default ref Behavior in React

- In React, ref is used to access a DOM node or a class component instance. However, when you pass a ref to a functional component, React does not know how to forward it to the underlying DOM node because functional components do not have instances.
- In React, the default behavior of the ref attribute allows you to directly reference a DOM element or a class component instance. Refs provide a way to access and

interact with elements or components imperatively, bypassing the usual declarative approach of React.

### Key Points About Ref's Default Behavior:

- **Binding to a DOM Element:** When you attach a ref to a DOM element, it gives you access to the actual DOM node. For example, attaching a ref to an `<input>` element lets you access that input field in the DOM for operations like focusing or reading its value.
- **Usage with Class Components:** When used with class components, ref returns the instance of the component. This is useful for calling instance methods defined within the class component.
- **React Hooks and Functional Components:** In functional components, ref is commonly used in conjunction with the `useRef` hook to create and manage references.
- **Mutable and Persistent:** Refs are mutable and remain the same across re-renders of the component. They do not trigger a re-render when their value changes, which makes them suitable for tasks like managing focus or animations.
- **Null Initial Value:** By default, a ref starts with a null value until the referenced element or component is mounted.
- **Not for Data Flow:** Refs should not be used to manage state or data flow in the application. React encourages declarative state management, and refs are an escape hatch for specific use cases like accessing the DOM.

### Example of Default Ref Behavior: Attaching a ref to a DOM element

```
import React, { useRef } from 'react';
function InputFocus() {
  const inputRef = useRef(null);
  const focusInput = () => {
    inputRef.current.focus(); };    // Accessing the DOM input element
  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>    </div>
    );}
```

#### In the example:

- ref is used to directly access the DOM `<input>` element.
- The default behavior is to bind the ref to the actual DOM node.

### Example of ref not working with functional components

```
import React, { useRef } from "react";

const MyComponent = () => {
  return <input />;
};

const App = () => {

  const inputRef = React.useRef();
  return (
    <div>
      <MyComponent ref={inputRef} />    { /* It will not work */ }
    </div>
  );
};

export default App;
```

#### Problem with above code:

- ref is passed to MyComponent (a functional component).
- Functional components do not natively accept or forward ref.
- As a result, the ref does not attach to the <input> element inside MyComponent.

Hence, React will raise a warning because ref cannot be passed to a functional component directly.

#### How forwardRef fix this issue?

forwardRef solves this issue by allowing functional components to forward the ref to a child component or DOM node.

**Using React.forwardRef:** React.forwardRef allows a functional component to accept a ref from its parent and explicitly pass it to a child element. By wrapping MyComponent with forwardRef, we can pass the ref from App to the <input> inside MyComponent.

### Example

```
import React, { useRef, forwardRef } from "react";

// Wrapping MyComponent with forwardRef
const MyComponent = forwardRef((props, ref) => {
  return <input ref={ref} {...props} />;
});

const App = () => {
  const inputRef = useRef();

  // Function to focus the input field
  const focusInput = () => {
    inputRef.current.focus(); // Accessing the input element directly
  };

  return (
    <div>
      <MyComponent ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
};

export default App;
```

### Explanation

- A parent component (App) passes a ref to MyComponent.
- Inside MyComponent, forwardRef forwards that ref to the <input> element.
- The parent can now directly interact with the <input> element (e.g., focus it, get its value).
- The return statement provides the JSX to render. In this case, it renders an <input> element with the forwarded ref and any additional props.'

### Example: Using forwardRef with Additional Props

```
import React, { forwardRef, useRef } from 'react';

// Child component
const CustomInput = forwardRef((props, ref) => {
  return (
    <div>
      <label>{props.label}</label>
      <input ref={ref} placeholder={props.placeholder} />
    </div>
  );
});

// Parent component
const App = () => {
  const inputRef = useRef();

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <CustomInput ref={inputRef} label="Enter Text:" placeholder="Type here..." />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
};

export default App;
```

### Important Points

- forwardRef bridges the gap between functional components and refs, allowing parent components to access child DOM nodes or methods.
- When using forwardRef, the parent is decoupled from the child component's implementation details. This makes it easier to refactor child components without affecting the parent.

## useImperativeHandle Hook

- The useImperativeHandle hook in React is a specialized hook used to customize the value that is exposed when a component is accessed via a ref.
- Normally, when you use a ref on a component, it provides access to the DOM node or the entire child component.
- With useImperativeHandle, you can define and expose only specific functionalities or properties to the parent, keeping the component's internal details private and more controlled.
- In essence, it allows a parent component to perform imperative actions (e.g., triggering methods like focus() or reset()) on a child component while maintaining encapsulation and abstraction.
- The useImperativeHandle hook allows you to customize the ref exposed by a component. Normally, ref is used to access the DOM elements or child components. With useImperativeHandle, you can define what functionality or properties the parent can access on the child component.
- The primary purpose of useImperativeHandle is to control what the parent can "see" and "do" with the child component. It is particularly useful when you need to expose specific methods of a child component that are not directly tied to rendering, such as focus management, animations, or dynamic behaviors.

### How it Works

- It is used in combination with React.forwardRef because a child component must forward the ref to allow the parent to access it.
- Inside the child component, useImperativeHandle lets you define custom behavior for the ref.

```
useImperativeHandle(ref, () => ({  
  // Expose methods or properties here  
}));
```

### Use Cases

- **Custom Inputs:** For creating input components where the parent can programmatically call methods like focus() or clear().
- **Modals/Dialogs:** To allow the parent to open or close a modal using a method like openModal().
- **Animations:** To trigger animations on a component from the parent.

## Why is it Needed?

React is declarative, meaning you typically describe what the UI should look like, and React handles the updates. However, certain scenarios require imperative programming—direct commands to perform actions. For instance:

- Focusing an input element programmatically.
- Triggering custom behavior like resetting a form or closing a modal from outside the component.

Without `useImperativeHandle`, the parent would need access to the entire DOM node or component, leading to tightly coupled and less maintainable code. This hook ensures that only specific, controlled behavior is exposed.

- **Declarative Programming**
  - React is declarative by nature.
  - You describe what the UI should look like based on the current state, and React updates the DOM accordingly.
  - **Example:** Using state to manage the visibility of a modal.
- **Imperative Programming**
  - In some scenarios, you need to directly control how something happens, bypassing React's declarative flow.
  - **Example:** Manually triggering a focus on an input or starting an animation.

`useImperativeHandle` bridges the gap by allowing imperative actions (like focus or animations) while keeping the rest of your React app declarative.

## Key Features

- **Encapsulation and Abstraction:** It hides internal details of a component while allowing the parent to interact with it in a controlled way.
- **Custom Behavior:** Instead of exposing the entire DOM or component instance, you define what methods or properties should be accessible.
- **Combines with Refs:** Works in combination with `useRef` and `forwardRef` to allow this level of control.
- **Imperative Control:** Provides a mechanism to handle imperative tasks (like focus, scrolling, animations) that are otherwise challenging in a declarative framework.

## Comparison with Other React Features

- **Direct Refs:** `useImperativeHandle` is a more controlled alternative to directly accessing refs. Direct refs expose the entire component or DOM node, which may lead to misuse or unintended behavior.
- **State Props:** While state and props handle declarative data flow, `useImperativeHandle` complements them by enabling imperative interactions.

## When Should You Use It?

- **Reusable Components:** When creating reusable components that need to expose certain functionalities (e.g., `reset()`, `focus()`) while keeping their implementation hidden.
- **Complex UI Behavior:** When managing behavior like opening/closing modals or custom scrolling that isn't easily managed through props/state.
- **DOM-Heavy Interactions:** For components that involve extensive DOM manipulation where declarative approaches are inefficient or verbose.

## Syntax of `useImperativeHandle`

The `useImperativeHandle` hook is used in conjunction with `React.forwardRef` to customize what is exposed via a ref in a child component.

`useImperativeHandle(ref, createHandle, [dependencies]);`

where

- **ref:** The ref passed down from the parent (via `React.forwardRef`).
- **createHandle:** A function (set of functions) that returns an object containing the methods/properties to expose to the parent.
- **dependencies:** An optional array of dependencies. If specified, the `createHandle` is only re-executed when these dependencies change.

## Example Concept

```
import React, { useImperativeHandle, forwardRef, useRef } from 'react';
```

```
const ChildComponent = forwardRef((props, ref) => {  
  useImperativeHandle(ref, () => ({  
    myCustomMethod() {  
      console.log('This is a custom method exposed to the parent!');  
    }  
  }));  
});
```



```
    return <div>Child Component</div>;  
  });
```

```
export default ChildComponent;
```

**note :** In the parent, you can now access myCustomMethod through the ref.

### Example: Creating a Reusable Button Component

#### ReusableButton.js (Child Component)

```
import React, { useImperativeHandle, forwardRef, useRef, useState } from 'react';
```

```
const ReusableButton = forwardRef((props, ref) => {  
  const buttonRef = useRef(); // Internal ref for the button element  
  const [disabled, setDisabled] = useState(false); // Button state
```

```
  // Expose custom methods to the parent via the ref
```

```
  useImperativeHandle(ref, () => ({  
    clickButton() {  
      buttonRef.current.click(); // Programmatically click the button  
    },  
    disableButton() {  
      setDisabled(true); // Disable the button  
    },  
    enableButton() {  
      setDisabled(false); // Enable the button  
    }  
  }));
```

```
  return (  
    <button ref={buttonRef} disabled={disabled}>  
      {props.label || 'Click Me'}  
    </button>  
  );  
});  
export default ReusableButton;
```

### App.js (Parent Component)

```
import React, { useRef } from 'react';
import ReusableButton from './ReusableButton';

const App = () => {
  const buttonRef = useRef();

  const handleClick = () => {
    if (buttonRef.current) {
      buttonRef.current.clickButton(); // Programmatically trigger a click
    }
  };

  const handleDisable = () => {
    if (buttonRef.current) {
      buttonRef.current.disableButton(); // Disable the button
    }
  };

  const handleEnable = () => {
    if (buttonRef.current) {
      buttonRef.current.enableButton(); // Enable the button
    }
  };

  return (
    <div>
      <h1>Reusable Button Example</h1>
      <ReusableButton ref={buttonRef} label="Custom Button" />
      <div style={{ marginTop: '10px' }}>
        <button onClick={handleClick}>Programmatically Click</button>
        <button onClick={handleDisable}>Disable Button</button>
        <button onClick={handleEnable}>Enable Button</button>
      </div>
    </div>
  );
};

export default App;
```

To log some text to the console when the "Custom Button" in the child component is clicked, you can enhance the ReusableButton component by adding an onClick handler to the <button> element. This handler will log the desired message to the console whenever the button is clicked.

### ReusableButton.js (Child Component)

```
import React, { useImperativeHandle, forwardRef, useRef, useState } from 'react';
const ReusableButton = forwardRef((props, ref) => {
  const buttonRef = useRef(); // Internal ref for the button element
  const [disabled, setDisabled] = useState(false); // Button state
  // Expose custom methods to the parent via the ref
  useImperativeHandle(ref, () => ({
    clickButton() {
      buttonRef.current.click(); // Programmatically click the button
    },
    disableButton() {
      setDisabled(true); // Disable the button
    },
    enableButton() {
      setDisabled(false); // Enable the button
    }
  }));
  const handleClick = () => {
    console.log('Child button clicked!'); // Log message to the console
    if (props.onClick) {
      props.onClick(); // Call the parent's onClick handler if provided
    }
  };
  return (
    <button ref={buttonRef} disabled={disabled}
      onClick={handleClick} // Add the click handler here
    >
      {props.label || 'Click Me'}
    </button>
  );
});
export default ReusableButton;
```

To better understand the behavior of `useImperativeHandle`, you can experiment with the code by applying the following changes and observing the effects. These changes will make it easier to grasp how the hook works, step by step.

### Change 1: Add Logs to Visualize the Flow

Insert `console.log` statements inside the `useImperativeHandle` hook and the methods to observe when they are triggered.

#### Updated Child Component: ReusableButton.js

```
import React, { useImperativeHandle, forwardRef, useRef, useState } from 'react';
const ReusableButton = forwardRef((props, ref) => {
  const buttonRef = useRef();
  const [disabled, setDisabled] = useState(false);
  // Expose methods with logging
  useImperativeHandle(ref, () => {
    console.log('useImperativeHandle called');
    return {
      clickButton() {
        console.log('clickButton method called');
        buttonRef.current.click();
      },
      disableButton() {
        console.log('disableButton method called');
        setDisabled(true);
      },
      enableButton() {
        console.log('enableButton method called');
        setDisabled(false);
      },
    };
  });
  return (
    <button ref={buttonRef} disabled={disabled}>
      {props.label || 'Click Me'}
    </button>
  );
});
export default ReusableButton;
```

### Change 2: Expose Fewer or Different Methods

Experiment with exposing only one or two methods to understand how the parent reacts when limited functionality is available.

#### Modify `useImperativeHandle`:

```
useImperativeHandle(ref, () => ({
  clickButton() {
    console.log('Only clickButton is exposed');
    buttonRef.current.click();
  }
}));
```

**Observe:** Only `clickButton` is available in the parent. Methods like `disableButton` or `enableButton` are inaccessible.

### Change 3: Experiment with Dependencies

Add dependencies to the `useImperativeHandle` hook and modify state to observe how it affects the exposed methods.

#### Modify `useImperativeHandle`:

```
useImperativeHandle(ref, () => ({
  clickButton() {
    console.log('clickButton called - Current disabled state:', disabled);
    buttonRef.current.click();
  },
  toggleButton() {
    setDisabled((prev) => !prev);
  }
}), [disabled]); // Dependency added
```

#### Observe:

- When `disabled` changes, the behavior of `clickButton` and `toggleButton` updates accordingly.
- Check if the dependency causes `useImperativeHandle` to recompute the exposed methods.

### Change 4: Add a Visual Indicator for Button State

Include text or visual feedback to understand how `useImperativeHandle` interacts with the component state.

#### Updated Child Component

```
return (  
  <div>  
    <button ref={buttonRef} disabled={disabled}>  
      {props.label || 'Click Me'}  
    </button>  
    <p>{disabled ? 'Button is Disabled' : 'Button is Enabled'}</p>  
  </div>  
);
```

**Observe:** Visual feedback for disabled state changes when calling `disableButton` or `enableButton`.

### Change 5: Add Additional Stateful Behavior

Add a counter state to the child component, incrementing it every time `clickButton` is called. This helps demonstrate how internal states interact with exposed methods.

#### Updated Child Component

```
const [counter, setCounter] = useState(0);  
useImperativeHandle(ref, () => ({  
  clickButton() {  
    setCounter((prev) => prev + 1);  
    buttonRef.current.click(); }  
}));  
return (  
  <div>  
    <button ref={buttonRef}>`Clicked ${counter} times`</button>  
  </div>  
);
```

**Observe:** The button label updates with the number of times it has been clicked programmatically.

### Example : Exposing DOM Methods

Create a custom InputField component that allows the parent to programmatically focus or clear the input field.

#### InputField.js (Child Component)

```
import React, { useImperativeHandle, forwardRef, useRef } from 'react';
const InputField = forwardRef((props, ref) => {
  const inputRef = useRef();

  useImperativeHandle(ref, () => ({
    focusInput() {
      inputRef.current.focus();
    },
    clearInput() {
      inputRef.current.value = '';
    }
  }));

  return <input ref={inputRef} placeholder="Type something..." />;
});
export default InputField;
```

#### App.js (Parent Component)

```
import React, { useRef } from 'react';
import InputField from './InputField';

const App = () => {
  const inputRef = useRef();
  return (
    <div>
      <InputField ref={inputRef} />
      <button onClick={() => inputRef.current.focusInput()}>Focus Input</button>
      <button onClick={() => inputRef.current.clearInput()}>Clear Input</button>
    </div>
  );
};
export default App;
```

### Example: Form Validation

A form component exposes a method to check if all fields are valid.

### Form.js (Child Component)

```
import React, { useImperativeHandle, forwardRef, useState } from 'react';

const Form = forwardRef((props, ref) => {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

  useImperativeHandle(ref, () => ({
    isValid() {
      return name.trim() !== "" && email.includes('@');
    }
  }));

  return (
    <div>
      <input
        type="text"
        placeholder="Name"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <input
        type="email"
        placeholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
    </div>
  );
});

export default Form;
```



### App.js (Parent Component)

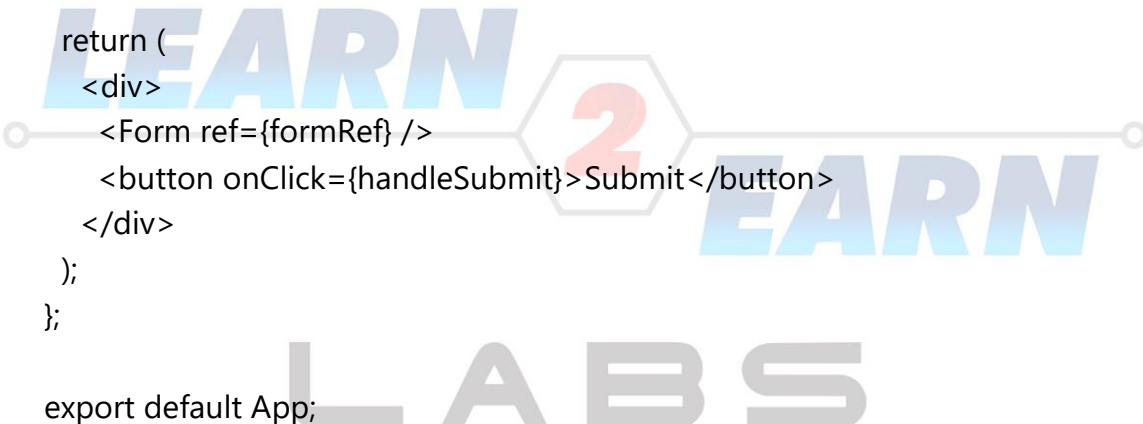
```
import React, { useRef } from 'react';
import Form from './Form';

const App = () => {
  const formRef = useRef();

  const handleSubmit = () => {
    if (formRef.current.isValid()) {
      alert('Form is valid!');
    } else {
      alert('Form is invalid. Please fill all fields correctly.');
    }
  };

  return (
    <div>
      <Form ref={formRef} />
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
};

export default App;
```

A large, semi-transparent watermark logo is centered over the code. It features the word "LEARN" in blue, a large red "2" inside a hexagon, the word "EARN" in blue, and the word "LABS" in grey below it.

## Homework

- Passing complex functionalities to the parent via useImperativeHandle.
- Combining useImperativeHandle with other hooks like useState and useEffect for advanced behavior.
- Sharing multiple methods or properties using a single reference.
- Integrating useImperativeHandle for Managing focus, blur, or other DOM-related actions on custom input components.

