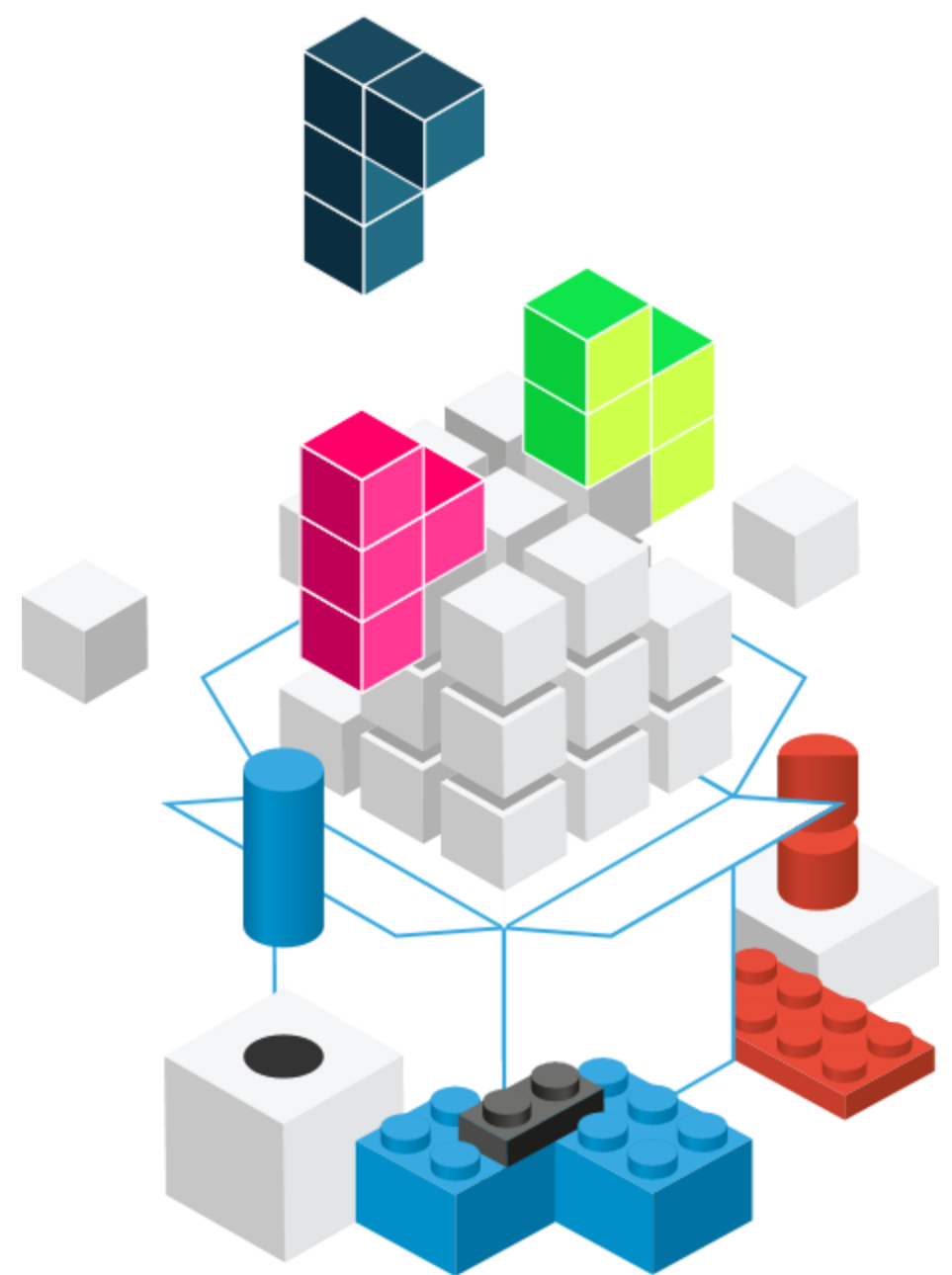


Forms

The Foundations of User Interaction

Fulvio Corno

Luigi De Russis





<https://react.dev/reference/react-dom/components#form-components>

Full Stack React, Chapter “Forms”

React Handbook, Chapter “JSX”

Forms, Events and Event Handlers

FORMS IN JSX

HTML Forms

- (Native) HTML Forms are *inconsistent*: different ways of handling values, events etc. depending on the type of input element
 - Consequence of backward compatibility
- For instance:
 - `onChange` on a radio button is not easy to handle
 - `value` in a `textarea` does not work, etc.
- React flattens this behavior exposing (via JSX) a more uniform interface
 - Synthetic Events

Value in JSX forms

- The **value** attribute always holds the current value of the field
- The **defaultValue** attribute holds the default value that was set when the field was created
- This also applies to
 - `textarea`: the content is in the `value` attribute; it is NOT to be taken from the actual content of the `<textarea>...</textarea>` tag
 - `select`: do not use the `<option selected>` syntax, but `<select value='id'>`

Change Events in JSX Forms

- React provides a more consistent **onChange** event
- By passing a function to the `onChange` attribute you can subscribe to events on form fields (every time `value` changes)
- `onChange` fires when typing a single character into an `input` or `textarea` field
- It works consistently across fields: even `radio`, `select` and `checkbox` input fields fire a `onChange` event

Event Handlers

- An Event Handler callback function is called with one parameter: an **event object**
- All event objects have a standard set of properties
 - **event.target**: *source* of the event
- Some events, depending on categories, have more specific properties

Synthetic Events

<https://react.dev/reference/react-dom/components/common#react-event-object>

- “High level events” wrap the corresponding DOM Events
- Same attributes as DOMEvent
- **target** points to the source of the event.
- In case of a *form element*
 - target.**value** = current input value
 - target.**name** = input element name

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

Synthetic Events

<https://reactjs.org/docs/events.html>

Category	Events
Clipboard	<code>onCopy</code> <code>onCut</code> <code>onPaste</code>
Composition	<code>onCompositionEnd</code> <code>onCompositionStart</code> <code>onCompositionUpdate</code>
Keyboard	<code>onKeyDown</code> <code>onKeyPress</code> <code>onKeyUp</code>
Focus	<code>onFocus</code> <code>onBlur</code>
Form	<code>onChange</code> <code>onInput</code> <code>onInvalid</code> <code>onReset</code> <code>onSubmit</code>
Generic	<code>onError</code> <code>onLoad</code>
Mouse	<code>onClick</code> <code>onContextMenu</code> <code>onDoubleClick</code> <code>onDrag</code> <code>onDragEnd</code> <code>onDragEnter</code> <code>onDragExit</code> <code>onDragLeave</code> <code>onDragOver</code> <code>onDragStart</code> <code>onDrop</code> <code>onMouseDown</code> <code>onMouseEnter</code> <code>onMouseLeave</code> <code>onMouseMove</code> <code>onMouseOut</code> <code>onMouseOver</code> <code>onMouseUp</code>
Pointer	<code>onPointerDown</code> <code>onPointerMove</code> <code>onPointerUp</code> <code>onPointerCancel</code> <code>onGotPointerCapture</code> <code>onLostPointerCapture</code> <code>onPointerEnter</code> <code>onPointerLeave</code> <code>onPointerOver</code> <code>onPointerOut</code>
Selection	<code>onSelect</code>
Touch	<code>onTouchCancel</code> <code>onTouchEnd</code> <code>onTouchMove</code> <code>onTouchStart</code>
UI	<code>onScroll</code>
Wheel	<code>onWheel</code>
Media	<code>onAbort</code> <code>onCanPlay</code> <code>onCanPlayThrough</code> <code>onDurationChange</code> <code>onEmptied</code> <code>onEncrypted</code> <code>onEnded</code> <code>onError</code> <code>onLoadedData</code> <code>onLoadedMetadata</code> <code>onLoadStart</code> <code>onPause</code> <code>onPlay</code> <code>onPlaying</code> <code>onProgress</code> <code>onRateChange</code> <code>onSeeked</code> <code>onSeeking</code> <code>onStalled</code> <code>onSuspend</code> <code>onTimeUpdate</code> <code>onVolumeChange</code> <code>onWaiting</code>
Image	<code>onLoad</code> <code>onError</code>
Animation	<code>onAnimationStart</code> <code>onAnimationEnd</code> <code>onAnimationIteration</code>
Transition	<code>onTransitionEnd</code>

Tip: Defining Event Handlers

- Define the function as...
 - an arrow function
 - a function expression

```
const handler = () => { ... }
```



```
handler = function() { ... }
```



Tip: Defining Event Handlers

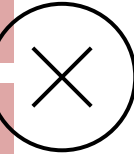
- Pass the *name* of the function as a prop
 - As a function object (not string)
 - Don't *call* the function

```
return <div handler={handler} />
```



```
return <div handler={handler()} />
```

```
return <div handler='handler' />
```



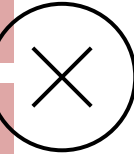
Tip: Defining Event Handlers

- Specify the *name* of the function prop in the event handler
- If you need to pass *parameters*, use an *arrow* function

```
return <button onClick=  
  {props.handler} />
```



```
return <button onClick=  
  {props.handler()} />
```



```
return <button onClick=  
  {props.handler(a, b)} />
```

```
return <button onClick=  
  {()=>props.handler()} />
```



```
return <button onClick=  
  {()=>props.handler(a, b)} />
```

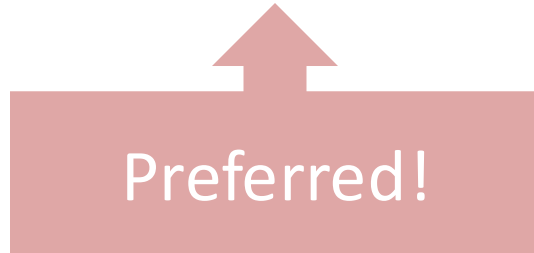
Who Owns The State?

- Form elements are **inherently stateful**: they hold a value
 - Input text form, selection, etc.
- React components are designed to handle the state
- The props and state are used to render the component
 - To correctly render the component from the virtual DOM, React needs to know which value must be set in the form element
 - Hence, on every change (onChange) React *must be notified* to get the new value and update the component state

Where Is The Source of Truth?

Controlled Form Components

- When the React component holds, in its state, the value to be shown in the form element, it is named a **controlled** form component



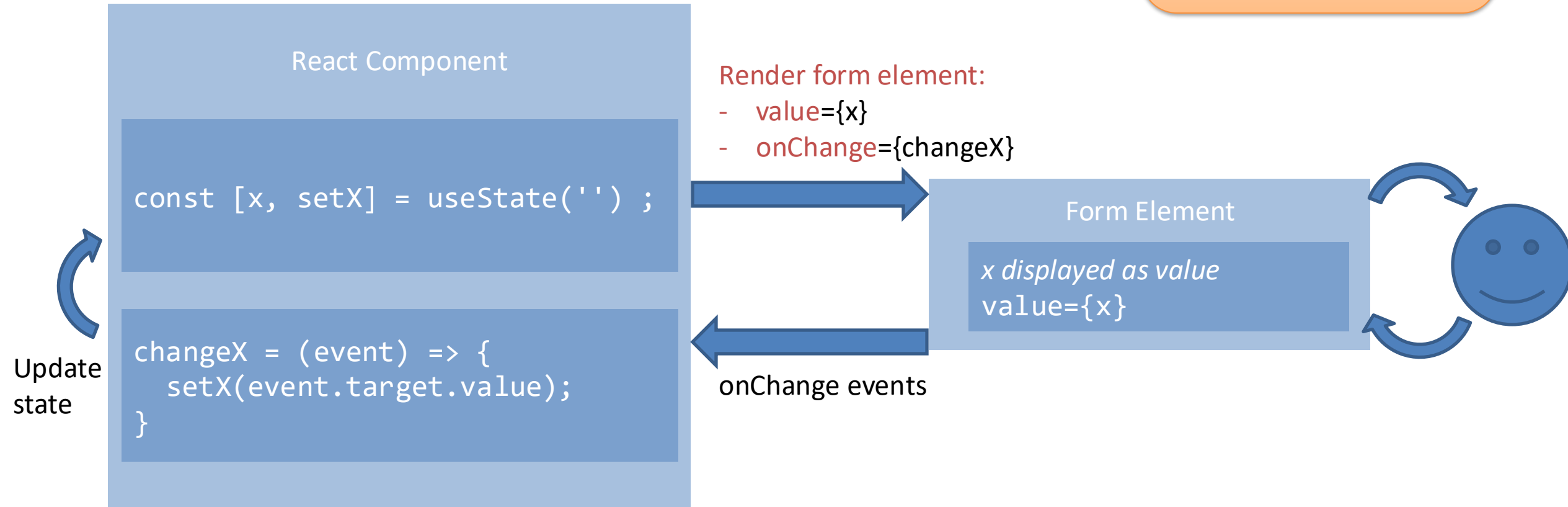
Uncontrolled Form Components

- In some occasions, it could be useful to keep the value directly in the HTML form element in the DOM: **uncontrolled** form component

Controlled Form Components



Setting value +
onChange makes the
form component fully
controlled



Controlled Form Component

- The event handler changes the state, `setXXX()` starts the update of the virtual DOM that then updates the actual DOM content

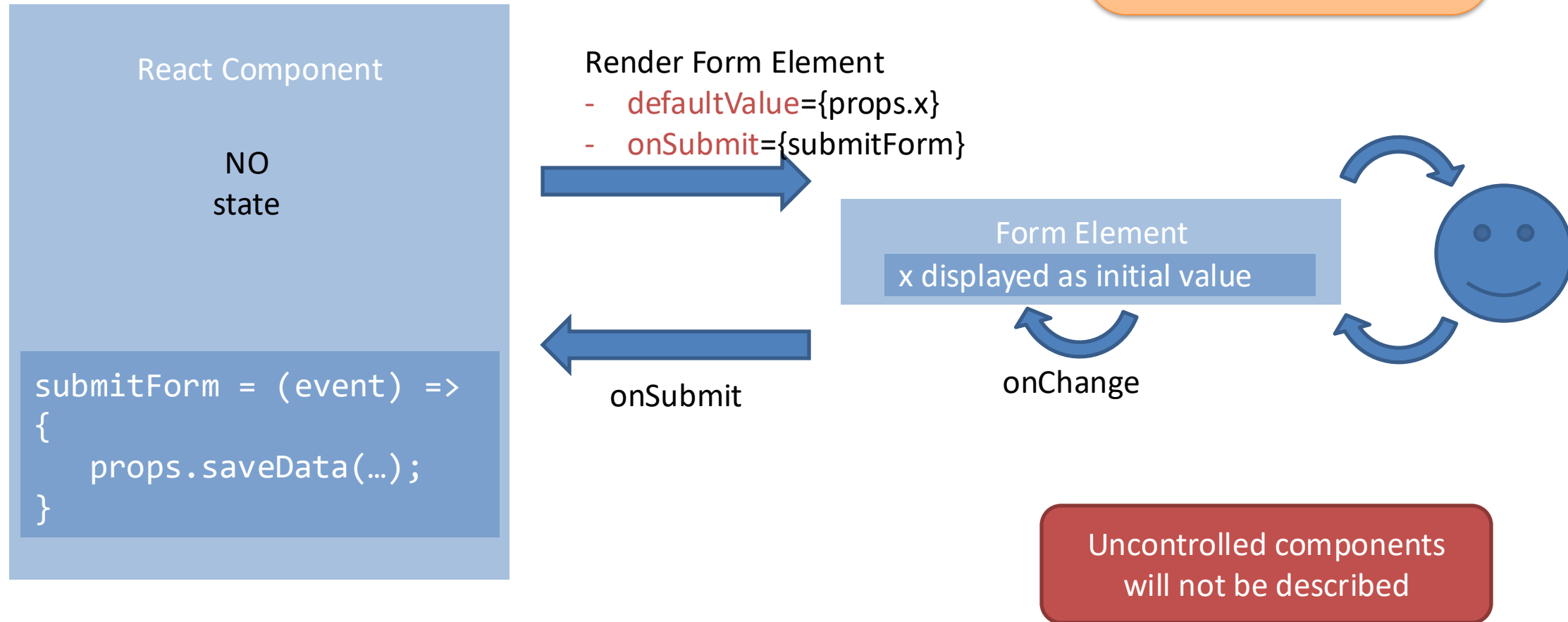
```
function MyForm (props) {  
  const [name, setName] = useState();  
  return <form onSubmit={handleSubmit}>  
    <label> Name:  
      <input type="text" value={name}  
        onChange={handleChange} />  
    </label>  
    <input type="submit" value="Submit" />  
  </form> ;  
}
```

```
handleSubmit = (event) => {  
  console.log('Name submitted: ' +  
    name);  
  event.preventDefault();  
}  
  
handleChange = (event) => {  
  setName(event.target.value) ;  
};
```

Uncontrolled Form Components



Not setting value +
onChange makes the
form component
uncontrolled



Tip: Form Submission

- The `onSubmit` event is generated by the `<form>` element
- Always call `event.preventDefault()` to avoid the submission (and reloading of the page)
- Perform *validation of all form data* before proceeding
 - Using checks on state variables (on a controlled component, they contain updated information)
 - May use validator <https://github.com/validatorjs/validator.js>

useActionState (React 19)

- Sometimes, it is tedious to use controlled form components
 - Need to write an event handler for every way data can change
 - Need to declare a state for each form component
 - Pipe all of the input state through a React component
- **useActionState** simplifies the process of handling forms
 - **New** hook in React 19
 - Remove the need for creating individual states and manually managing values, while providing a state to the form component
 - Built-in *loading state* available
 - Improve performance as there is no state updates/re-renders on every keystroke

<https://react.dev/reference/react/useActionState>

useActionState

- Create a component state that is updated when a *form action* is invoked
 - Get a **form action function** and an **initial state**
- It returns
 - A new **action** that you use in your form
 - The latest **form state**, initially set to provided initial state
 - An optional **loading state** that you can use while your action is processing

```
import { useActionState } from "react";

const increment = async (previousState, formData) => {
  return previousState + 1;
}

function SimpleForm() {
  const [formState, formAction, isPending] =
    useActionState(increment, 0);
  return (
    <form action={formAction}>
      {formState}
      <button type="submit">Increment</button>
    </form>
  )
}
```

Creating a `useActionState`

- `import{ useActionState } from "react";`
- `const [state, formAction, isPending] = useActionState(increment, 0);`
 - `state`: name of the form state
 - `formAction`: name of the function to use in the form's `action` attribute
 - `isPending`: a boolean state that says whether the form action is still pending
- `increment`: the action function, i.e., what happens when the form is submitted
- `0`: the initial state
- Array destructuring assignment to assign 3 values at once
- Setting an initial state is not mandatory, but **recommended**
 - Any serializable value that represents the entire initial state of the form
 - Ignored after the action is first invoked

The Action Function

- `const actionFunction => async(prevState, formData) {...}`
 - async function called when the form is submitted
- `prevState`, the latest available form state
 - The form state is the value *returned* by the action function when the form was last submitted
 - At the first call, it is the initial state passed to `useActionState`
- `formData`, the data submitted by the form
 - According to the standard `FormData` interface, <https://developer.mozilla.org/en-US/docs/Web/API/FormData>
- The function automatically calls `event.preventDefault()`
 - No need to explicitly write it!

Example

```
handleSubmit = async (prevState,
formData) => {
  const submittedName =
formData.get('name');

  console.log('Name submitted: ' +
submittedName);

  return {name: submittedName };
}
```

```
function MyForm (props) {
  const [state, formAction] =
useActionState(handleSubmit, {name:
props.name});

  return <form action={formAction}>
    <label> Name:
      <input name="name" type="text"
defaultValue={state.name} />
    </label>
    <input type="submit" value="Submit" />
  </form> ;
}
```

Advanced Usages

- A form can have multiple `useActionState` defined
 - e.g., what happens when the same form has a *delete* and *update* button
- In this case, instead of using `action` in the `<form>` component, you can use `formAction` inside any form components:
 - `<button formAction={updateAction}>Update</button>`
- `useActionState` works well with the other hooks and can be used along controlled form components, if needed

Alternatives to Built-in React Forms

- Formik
 - Keep things organized without hiding them too much
 - Form state is inherently ephemeral and local: does not use state management solutions (e.g., Redux/Flux) that would unnecessarily complicate things
 - Includes validation, keeping track of the visited fields, and handling form submission
 - <https://jaredpalmer.com/formik>
- React Hook Form
 - Abstract some of the form boilerplate code
 - Lightweight and extensible via plugins
 - Supports validation out of the box with error messages
 - <https://react-hook-form.com>

Tips: Handling Arrays in State

- React `setXXX()` with arrays requires that a new array is returned (cannot mutate the current state)
 - What is the correct way to handle arrays in React state?
- Use a new array as the value of the property
 - When referencing objects, use a *new object* every time a property changes
- Use a callback to ensure no modifications are missed
- Typical cases -- mostly triggered by form events
 - Add items
 - Update items
 - Remove items

<https://www.robinwieruch.de/react-state-array-add-update-remove>

Adding Items in array-valued state

```
// Append at the end: use .concat()
// NO .push(): it returns the number of
// elements, not the array
...

const [list, setList] = useState(['a',
  'b', 'c']);
...

setList(oldList =>
  return oldList.concat(newItem);
)
```

```
// Insert value(s) at the beginning
// use spread operator
...

const [list, setList] = useState(['a',
  'b', 'c']);
...

setList(oldList =>
  return [newItem, ...oldList];
)
```

<https://www.robinwieruch.de/react-state-array-add-update-remove>

Updating Items in array-valued state

```
// Update item: use map()
...
const [list, setList] = useState([11, 42, 32]);
...
// i is the index of the element to update
setList(oldList => {
  const list = oldList.map((item, j) => {
    if (j === i) {
      return item + 1; // update the item
    } else {
      return item;
    }
  });
  return list ;
});
```

<https://www.robinwieruch.de/react-state-array-add-update-remove>

Updating Items in array-of-objects state

```
// Update item: use map(); if items are objects, always return a new object if modified
...
const [list, setList] = useState([{id:3, val:'Foo'},{id:5, val:'Bar'}]);
...
// i is the id of the item to update
setList(oldList => {
  const list = oldList.map((item) => {
    if (item.id === i) {
      // item.val='NewVal'; return item; // WRONG: the old object must not be reused
      return {id:item.id, val:'NewVal'}; // return a new object: do not simply change content
    } else {
      return item;
    }
  });
  return list ;
});
```

Removing Items in array-valued state

```
// Remove item: use filter()
```

```
...  
const [list, setList] = useState([11, 42,  
32]);  
...
```

```
// i is the index of the element to remove  
setList(oldList=> {  
  return oldList.filter(  
    (item, j) => i !== j );  
});
```

```
// Remove first item(s): use destructuring
```

```
...  
const [list, setList] = useState([11, 42,  
32]);  
...
```

```
setList(oldList => {  
  const [first, ...list] = oldList;  
  return list ;  
});
```

<https://www.robinwieruch.de/react-state-array-add-update-remove>

Tip: Heuristics for State Lifting

- Presentational components
 - Forms, Tables, Lists, Widgets, ...
 - Should contain **local state** to represent their display property
 - Sort order, open/collapsed, active/paused, ...
 - Such state is *not interesting outside* the component
- Application components (or Container components)
 - Manage the information and the application logic
 - Usually don't directly generate markup, generate props or context
 - Most **application state** is “lifted up” to a **Container**
 - Centralizes the updates, single source of State truth

License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
 - **Share** — copy and redistribute the material in any medium or format
 - **Adapt** — remix, transform, and build upon the material
 - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
 - **Attribution** — You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - **NonCommercial** — You may not use the material for [commercial purposes](#).
 - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.
 - **No additional restrictions** — You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>

