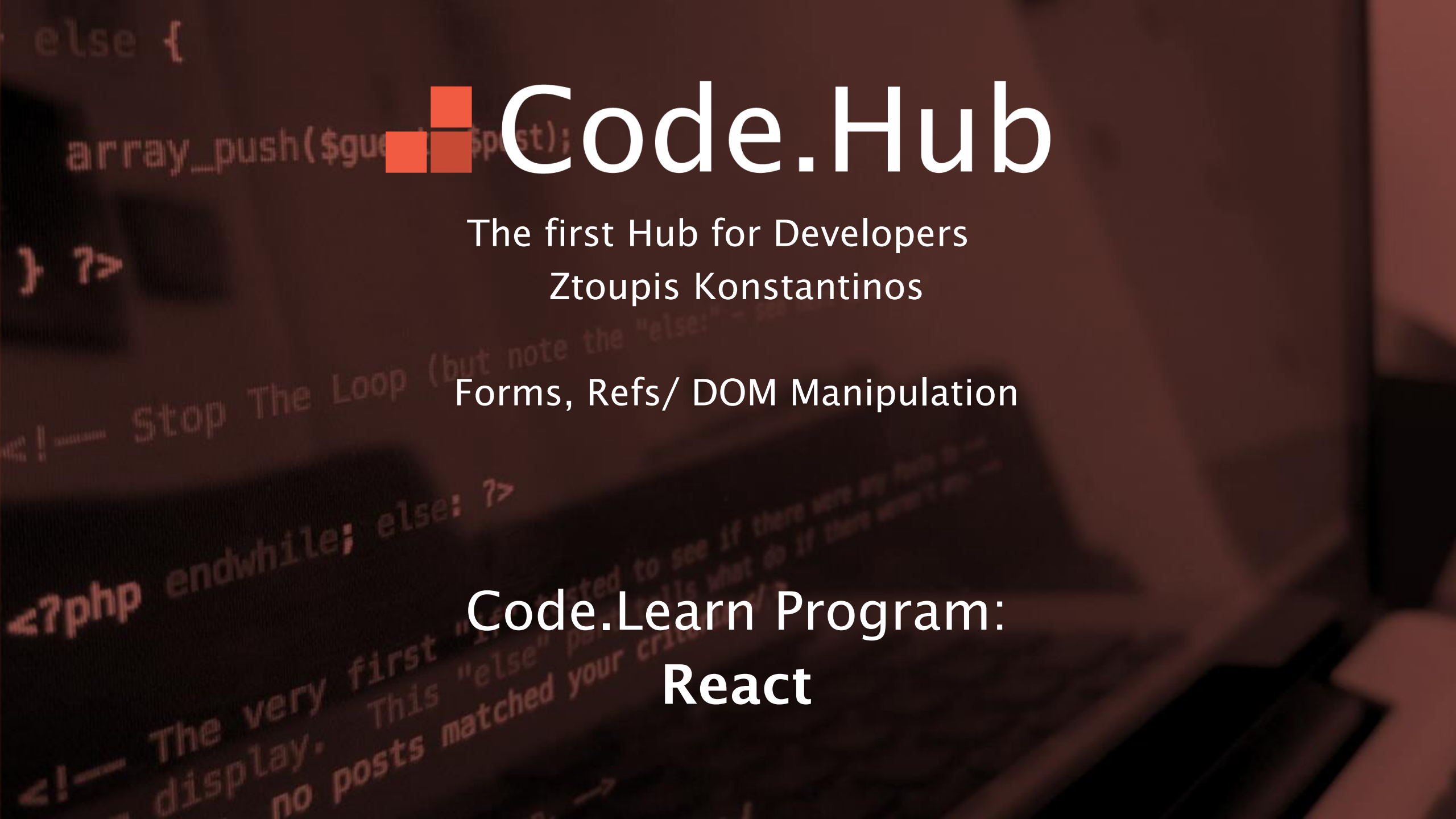


Code.Hub

The first Hub for Developers

Ztoupis Konstantinos

Forms, Refs/ DOM Manipulation

Code.Learn Program:
React

Forms

HTML form elements:

- work differently from other DOM elements in React
- naturally keep some internal state in React

```
<form>
  <label>
    Car: <input type="text" name="car" />
  </label>
  <input type="submit" value="submit" />
</form>
```

Form elements

In HTML, form elements such as <input>, <textarea>, <select>:

- typically maintain their own state
- update it based on user input

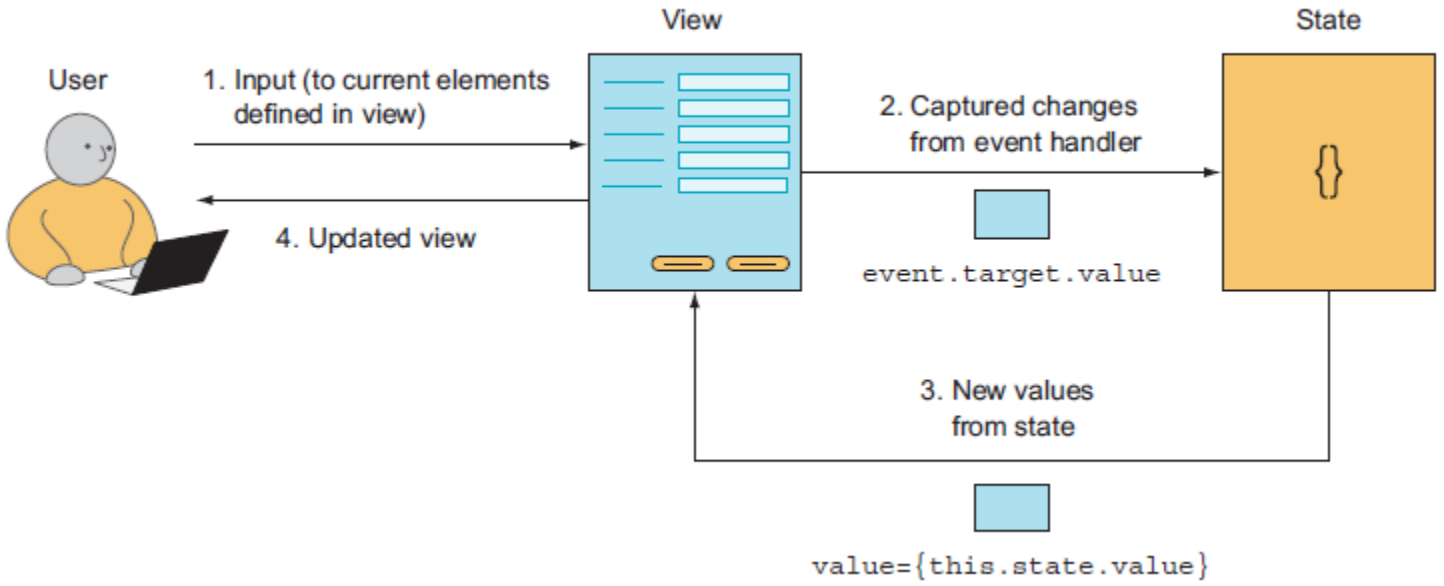
In React:

- mutable state is typically kept in the state property of components
- and only updated with setState() or useState(hooks)

Working with forms in React

- Defining forms and form elements
- Capturing data changes
- Using references to access data

The way to work with forms



The way to work with forms

- This approach is called one-way binding because the state changes views, and that's it.
- There's no trip back: only a one-way trip from state to view.
- With one-way binding, a library won't update the state (or the model) automatically.
- One of the main benefits of one-way binding is that it removes complexity when you're working with large apps where many views implicitly can update many states (data models) and vice versa

Controlled Components

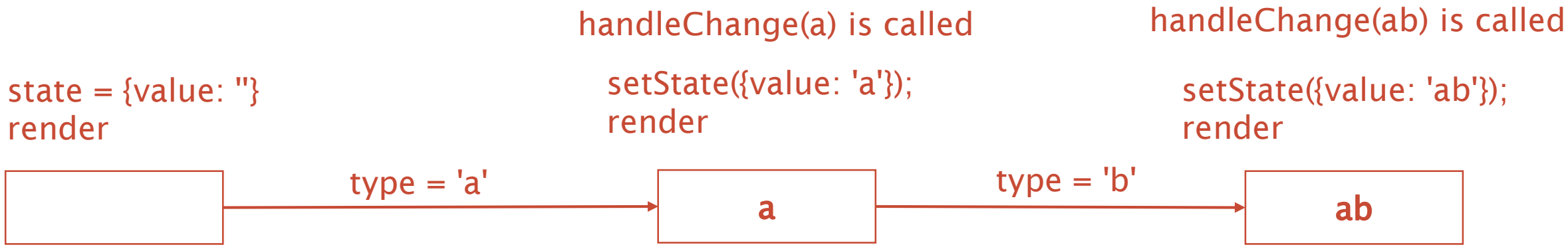
A tag form element whose value is controlled by React is called a “controlled component”

```
constructor() {
  super();
  this.state = {value: ''};
}
handleChange = (event) => {
  this.setState({value: event.target.value});
}
handleSubmit(event) {
  alert('Car: ' + this.state.value);
  event.preventDefault();
}
```

```
<form onSubmit={this.handleSubmit}>
  <label>
    Car: <input type="text"
value={this.state.value}
      onChange={this.handleChange} />
  </label>
  <input type="submit" value="submit" />
</form>
```

Controlled Components

Every time you type a new character, handleChange is called. It takes in the new value of the input and sets it in the state.



Controlled Components

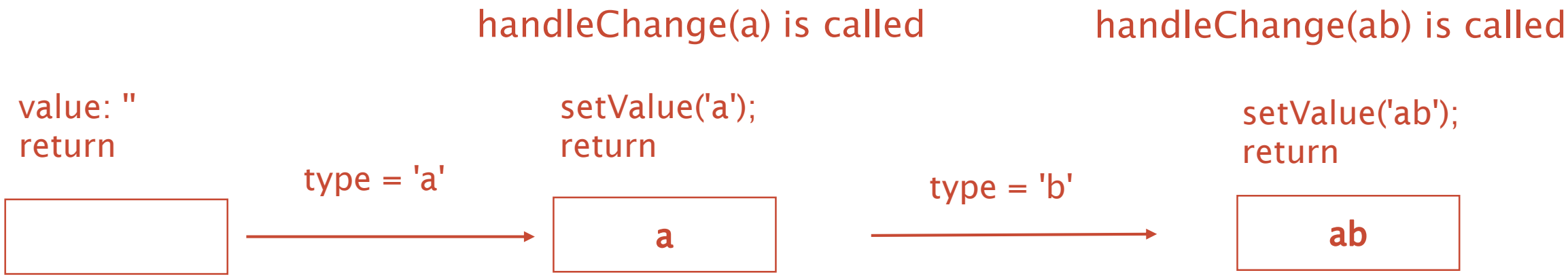
In functional components:

```
const [value, setValue] = useState('');

const handleSubmit = (event) => {
  alert('Car: ' + value);
  event.preventDefault();
}
const handleChange = event =>  setValue(event.target.value);
```

```
<form onSubmit={handleSubmit}>
  <label>
    Car: <input type="text" value={value}
      onChange={handleChange} />
  </label>
  <input type="submit" value="submit" />
</form>
```

Controlled Components



Controlled Components

- the form component always has the current value of the input, without needing to ask for it explicitly
- data (state) and UI (inputs) are always in sync. The state gives the value to the input, and the input asks the Form to change the current value.
- form component can respond to input changes immediately for example, by:
  - in-place feedback, like validations
  - disabling the button unless all fields have valid data
  - enforcing a specific input format, like credit card numbers

The textarea Tag

<textarea> element:

- defines its text by its children in HTML
- uses a value attribute in React

Textarea, in HTML

```
<textarea>
  The tiger (Panthera tigris) is the largest
  species among the Felidae and classified in
  the genus Panthera. It is most recognisable
  for its dark vertical stripes on reddish-orange
  fur with a lighter underside.
</textarea>
```

Textarea, in React with Class

```
<form onSubmit={this.handleSubmit}>
  <label>
    Car: <textarea value={this.state.value}
      onChange={this.handleChange} />
  </label>
  <input type="submit" value="submit" />
</form>
```

Textarea, in React with Hooks

```
<form onSubmit={handleSubmit}>
  <label>
    Car: <textarea value={value}
      onChange={handleChange} />
  </label>
  <input type="submit" value="submit" />
</form>
```

The select Tag

<select> element:

- creates a drop-down list in HTML
- uses a value attribute on the root select tag in React

Select, in HTML

```
<select>
  <option value="ferrari">Ferrari</option>
  <option selected
value="bmw">BMW</option>
  <option value="lada">Lada</option>
</select>
```

Select, in React with Class

```
<form onSubmit={this.handleSubmit}>
  <label> Car:
    <select value={this.state.value}
      onChange={this.handleChange}>
      <option
value="ferrari">Ferrari</option>
      <option value="bmw">BMW</option>
      <option value="lada">Lada</option>
</label>
  <input type="submit" value="submit" />
</form>
```

Select, in React with Hooks

```
<form onSubmit={handleSubmit}>
  <label> Car:
    <select value={value}
      onChange={handleChange}>
      <option value="ferrari">Ferrari</option>
      <option value="bmw">BMW</option>
      <option value="lada">Lada</option>
</label>
  <input type="submit" value="submit" />
</form>
```

Radios

- radios come in sets
- each element in a set has the same name attribute

Select, in HTML

```
<label>
  Tiger <input type="radio" value="tiger" name="pet" />
</label>
<label>
  Dog <input type="radio" value="dog" name="pet" />
</label>
<label>
  Cat <input type="radio" value="cat" name="pet" />
</label>
```

Radios

Select, in React with Class

```
constructor() {
  super();
  this.state = {pet: ''};
}

handleChange = (event) => {
  this.setState({pet: event.target.value});
}

handleSubmit = () => {
  alert('Pet: ' + this.state.pet);
}...
 <label>Tiger
   <input type="radio" value="tiger" name="pet“
      checked={pet === 'tiger'}
onChange={this.handleChange}
 />
 </label>
```

Select, in React with Hooks

```
const [pet, setPet] = useState('');
const handleChange = (event) => {
  setPet(event.target.value);
}
const handleSubmit = (event) => {
  alert('Pet: ' + pet);
  event.preventDefault();
}

...
<label>Tiger
   <input type="radio"
      value="tiger“
      name="pet"
      checked={pet === 'tiger'}
      onChange= {handleChange}
 />
 </label>
```

Code.Hub

Checkboxes

Checkboxes, in HTML

```
<label>
  Tiger <input type="checkbox" value="tiger" name="pet" />
</label>
<label>
  Dog <input type="checkbox" value="dog" name="pet" />
</label>
<label>
  Cat <input type="checkbox" value="cat" name="pet" />
</label>
```

Checkboxes, in React with Class

```
constructor() {
  super();
  this.state = {pet: []};
}
handleSubmit = (event) => {
  alert('Pet: ' + this.state.pet);
  event.preventDefault();
}
handleChange = (event) => {
  const value = event.target.value;
  const checked = event.target.checked;
  this.setState(({pet}) => {
    let newPet = [...pet];
    if (checked) {
      newPet = newPet.concat(value);
    } else {
      const index = newPet.indexOf(value);
      if (index > -1) {newPet.splice(index, 1);}
    }
    return {pet: newPet};
});}
```

```
<form onSubmit={this.handleSubmit} >
  <label>
   Tiger
   <input type="checkbox"
     value="tiger" name="pet"
     checked={pet.includes('tiger')}
     onChange={this.handleChange}
   />
  </label>
…
```

Code.Hub

Checkboxes with Hooks

```
const [pet, setPet] = useState([]);

const handleSubmit = (event) => {
  alert('Pet: ' + this.state.pet);
  event.preventDefault();
}

const handleChange = (event) => {
  const value = event.target.value;
  const checked = event.target.checked;
 setPet((prevState) => {
    let newPet = [...prevState];
    if (checked) {
      newPet = newPet.concat(value);
     } else {
       const index = newPet.indexOf(value);
       if (index > -1) {newPet.splice(index, 1);}
     }
     return newPet;
});}
```

```
<form onSubmit={handleSubmit} >
  <label>Tiger
    <input type="checkbox"
       value="tiger" name="pet"
       checked={pet.includes('tiger')}
       onChange={handleChange}
/>
</label>
…
```

Code.Hub

Multiple SELECT

select multiple options in a select tag by using array into the value

```
<select multiple={true} value={['ferrari', 'bmw']}>
```

Handling Multiple Inputs with Class

handle multiple controlled input elements

- add a name attribute to each element
- get value from event.target.name

```
handleInput ({target}) {
  const value = target.type === 'checkbox' ?
    target.checked : target.value;
    const name = target.name;
    this.setState({
      [name]: value
    });
  }
```

```
<input
  name="car" type="checkbox"
  checked={this.state.car}
  onChange={this.handleInput}
/>
<input
  name="tiger"
  type="number"
  value={this.state.tiger}
  onChange={this.handleInput}
/>
```

Handling Multiple Inputs with Hooks

```
const [car, setCar] = useState(true);
const [tiger, setTiger] = useState(5);

const handleCar = event => setCar(event.target.checked);
const handleTiger = event => setTiger(event.target.value);
```

Form elements in React

Elements	Value property	Change callback	New value in the callback
____	value="string"	onChange	event.target.value
<input type="checkbox" />	checked={boolean}	onChange	event.target.checked
<input type="radio" />	checked={boolean}	onChange	event.target.checked
<textarea />	value="string"	onChange	event.target.value
<select />	value="option value"	onChange	event.target.value

Formik

- a complete solution including
  - validation
  - keeping track of the visited fields
  - handling form submission
- one of the popular choices
- is built on the same principles of controlled components and managing state

Uncontrolled Components

- an alternative technique for implementing input forms
- form data is handled by the DOM itself
- no event handler for every state update
- use a ref to get form values from the DOM

Refs and the DOM

- refs provide a way to access DOM nodes or React elements created in the render method
- use cases for using refs:
  - Managing focus, text selection, or media playback
  - Triggering imperative animations
  - Integrating with third-party DOM libraries
- avoid using refs for anything that can be done declaratively

The ref attribute as a string

- the old way of creating a ref
- will likely be removed in a future release
- some issues associated with it

```
handleSubmit = (event) => {
  alert('Car: ' + this.refs.carInput.value);
  event.preventDefault();
}
…
<form onSubmit={this.handleSubmit}>
  <label>
    Car: <input type="text" ref="carInput" />
  </label>
  <input type="submit" value="submit" />
</form>
```

Creating Refs

- refs are created using React.createRef()
- attached to React elements via the ref attribute

```
class Component extends React.Component {
  constructor() {
    super();
    this.input = React.createRef();
  }

  render() {
    return <div ref={this.input} />;
  }
}
```

Accessing Refs

a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref:

```
this.input.current
```

The value of the ref differs depending on the type of the node:

- on an HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its current property
- on a custom class component, the ref object receives the mounted instance of the component as its current
- on function components because they don't have instances

Ref inside function

ref attribute inside a function component as long as you refer to a DOM element or a class component

```
function CustomTextInput(props) {
  const textInput = React.createRef();

  return (
    <div>
      <input
        type="text"
        ref={textInput}
      />
    </div>
  );
}
```

Ref Attribute

```
constructor() {
  this.input = React.createRef();
}
...
alert('Car: ' + this.input.current.value);
...
<label>
  Car:
  <input type="text" ref={this.input} />
  </label>
  <input type="submit" value="submit" />
```

Default Values

- the value attribute on form elements will override the value in the DOM
- use a defaultValue attribute instead of value
- <input type="checkbox"> and <input type="radio"> support defaultChecked
- <select> and <textarea> supports defaultValue

```
<form onSubmit={this.handleSubmit}>
  <label>
    Car:
    <input
      defaultValue= "Audi"
      type="text"
      ref={this.input}
    />
  </label>
  <input type="submit" value="submit" />
</form>
```

The file input Tag

<input type="file"> element:

- lets the user choose one or more files from their device storage
- its value is read-only
- an uncontrolled component in React
- its value can only be set by a user, and not programmatically

```
constructor() {
  super();
   this.file= React.createRef();
}

handleSubmit(event) {
  event.preventDefault();
  alert(`File:
${this.file.current.files[0].name}`);
}

render() {
  return (
    <form onSubmit={this.handleSubmit}>
    <label>Upload a file:
      <input type="file" ref={this.file} />
    </label>
    <button type="submit">Submit</button>
  </form>
)}
```

Code.Hub

Callback Refs

- not use ref attribute created by createRef()
- use a function

function receives the React component instance or HTML DOM element as its argument, which can be stored and accessed elsewhere

callback function to the ref attribute of a component

```
<input
  type="text"
  ref={element => this.textInput = element}
/>
```

the callback function can be accessed elsewhere

```
this.textInput.value
```

Callback Refs

```
constructor() {
  super();
  this.input = null;
  this.setTextInputRef = element => {this.input = element;};

  handleSubmit = (event) => {
    alert('Car: ' + this.input.value);
    event.preventDefault();
  };
}

render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>Car:
        <input type="text" ref={this.setTextInputRef} />
      </label>
      <input type="submit" value="submit" />
    </form>
  );
}
```

Radios

- radios come in sets
- each element in a set has the same name attribute
- setting a ref on each radio input is not ideal
- there's no DOM node that encapsulates a set of radios

Retrieving the value of the radio set can be obtained through three steps:

- set a ref on the <form> tag
- extract the set of radios from the form
  - a node list and a value is returned here
  - keep in mind that a node list looks like an array but is not, and lacks array methods
- grab the value of the set using dot

Radios

```
handleSubmit = (event) => {
  const { pet } = this.form;
  event.preventDefault();
}...
<form
  onSubmit={this.handleSubmit}
  ref={form => this.form = form}>
  <label>
    Tiger <input type="radio" value="tiger" name="pet" />
  </label>
....
</form>
```

Checkboxes

- checkbox may have multiple values selected
- more complicated than extracting the value of radios

Retrieving the selected values of the checkbox set can be done through these five steps:

- set a ref on the <form>
- extract the set of checkboxes from the form
  - a node list and a value is returned here
  - keep in mind that a node list looks like an array but is not, and lacks array methods

Checkboxes

- convert the node list to an array, so array methods are available
- use Array.filter() to grab only the checked checkboxes
- use Array.map() to keep only the values of the checked checkboxes

```
handleSubmit = (event) => {
  const { pet } = this.form;
  const checkboxArray = Array.prototype.slice.call(pet);
  const checkedCheckboxes = checkboxArray.filter(input => input.checked);
  const checkedCheckboxesValues =  checkedCheckboxes.map(input =>
input.value);
  event.preventDefault();
}

…
<form
  onSubmit={this.handleSubmit}
  ref={form => this.form = form}
>
  <label>Tiger
    <input type="checkbox" value="tiger" name="pet" />
  </label>
</form>
```

Forwarding a ref

Ref forwarding

- the technique of passing a ref from a component to a child component
- use of the React.forwardRef() method

```
const Input = React.forwardRef((props, ref) => (<input type="text" ref={ref} />));

const inputRef = React.createRef();

class TextInput extends React.Component {
  ...
  <form onSubmit={this.handleSubmit}>
    <Input ref={inputRef} />
    <input type="submit" value="submit" />
  </form>
```

React with useRef()

- introduced in React 16.7 and above version
- helps to get access the DOM node or element
- returns the ref object whose .current property initialized to the passed argument
- the returned object persist for the lifetime of the component.

```
const inputRef = useRef(initialValue);
```

```
function useRefExample() {
  const inputRef = useRef(null);

  const handleSubmit = (event) => {
    alert('Car: ' + inputRef.current.value);
    event.preventDefault();
  }

 return (
    <form onSubmit={handleSubmit}>
      <label> Car:
        <input type="text" ref={inputRef} />
      </label>
      <input type="submit" value="submit" />
    </form>
  );
}
```

Code.Hub

Conclusion

- controlled and uncontrolled form fields have their merit
- evaluate your specific situation and pick the approach
- form is incredibly simple in terms of UI feedback: uncontrolled with refs is entirely fine.

feature	uncontrolled	controlled
one-time value retrieval (e.g. on submit)	✓	✓
validating on submit	✓	✓
instant field validation	✗	✓
conditionally disabling submit button	✗	✓
enforcing input format	✗	✓
several inputs for one piece of data	✗	✓
dynamic inputs	✗	✓