How to React Slider

DECEMBER 02, 2019 BY ROBIN WIERUCH - EDIT THIS POST

In this React component tutorial by example, we will create a React Slider Component with React Hooks and a Function Component. You can see the final output of this implementation in this CodeSandbox or in this GitHub repository. If you want to implement it step by step, just follow the tutorial.

REACT SLIDER BY EXAMPLE

Let's start right away by giving our React Slider its style. This way, we can see our component right away in the browser. We will use Styled Components to style our Slider, but feel free to use something else like CSS Modules.

```
import React from 'react';
import styled from 'styled-components';

const StyledSlider = styled.div`
  position: relative;
  border-radius: 3px;
  background: #dddddd;
  height: 15px;
`;

const StyledThumb = styled.div`
  width: 10px;
  height: 25px;
  border-radius: 3px;
  position: relative;
  top: -5px;
  opacity: 0.5;
  background: #823eb7;
  cursor: pointer;
`;

const Slider = () => {
  return (
    <>
      <StyledSlider>
        <StyledThumb />
      </StyledSlider>
    </>
  );
};

const App = () => (
  <div>
    <Slider />
  </div>
);

export default App;
```

Now you should see the slider with its thumb already being rendered by React. We are using the Slider component in context of a React application by having an App component in place as well. Let's check how to implement its business logic in order to enable a user interacting with it.

REACT SLIDER: RANGE

Let's focus only on the Slider component now. We will give each part of the slider, the slider itself and its thumb, a React ref for reading from (and writing to) these DOM elements with direct DOM

manipulation later. Otherwise we couldn't access properties like the slider's width or thumb's position in our next steps.

```
const Slider = () => {
  const sliderRef = React.useRef();
  const thumbRef = React.useRef();

  const handleMouseDown = event => {};

  return (
    <>
      <StyledSlider ref={sliderRef}>
        <StyledThumb ref={thumbRef} onMouseDown={handleMouseDown} />
      </StyledSlider>
    </>
  );
};
```

Also we added a `onMouseDown` handler to our slider's thumb. This one is actually needed to capture a user's interaction with the slider. In the next step, we will add two more event handlers, which will be only active after the mouse down event has been triggered. One of these new events -- the mouse up event -- will make sure to de-register these new events.

```
const Slider = () => {
  const sliderRef = React.useRef();
  const thumbRef = React.useRef();

  const handleMouseMove = event => {
    // TODO:

    // set new thumb position while moving
    // by using the saved horizontal start position
  };

  const handleMouseUp = () => {
    document.removeEventListener('mouseup', handleMouseUp);
    document.removeEventListener('mousemove', handleMouseMove);
  };

  const handleMouseDown = event => {
    document.addEventListener('mousemove', handleMouseMove);
    document.addEventListener('mouseup', handleMouseUp);
  };

  return (
    <>
      <StyledSlider ref={sliderRef}>
        <StyledThumb ref={thumbRef} onMouseDown={handleMouseDown} />
      </StyledSlider>
    </>
  );
```

```
};
```

The onMouseDown handler's function does two things:

First, it registers two more handlers for the thumb, which only happens after the mouse down event has been triggered. This ensures that the thumb only moves while the mouse is down. If the mouse up event gets triggered eventually -- which has just been registered -- all newly registered handlers will be removed again. The mouse move event is the place where the actual logic of the range slider happens, but again, only if the mouse down event is active.

Second, it stores the difference of the thumb position and the actual click on the x-axis -- just to be more accurate here. We store it only *once* in order to reuse it later for *every* mouse move event. We will be using a React ref again, which makes sure that the value doesn't get lost in between of component re-renders. Also we are not using React state here, because we don't want to trigger a re-render of the component.

```
const Slider = () => {
  const sliderRef = React.useRef();
  const thumbRef = React.useRef();

  const diff = React.useRef();

  const handleMouseMove = event => {
    let newX =
      event.clientX -
      diff.current -
      sliderRef.current.getBoundingClientRect().left;
  };

  const handleMouseUp = () => {
    document.removeEventListener('mouseup', handleMouseUp);
    document.removeEventListener('mousemove', handleMouseMove);
  };

  const handleMouseDown = event => {
    diff.current =
      event.clientX - thumbRef.current.getBoundingClientRect().left;

    document.addEventListener('mousemove', handleMouseMove);
    document.addEventListener('mouseup', handleMouseUp);
  };

  ...
};
```

Note: We are only calculating the values along the x-axis, because we are not dealing with a vertical slider here. You can try on your own to convert this Slider component to a vertical Slider later as an exercise.

After we calculated the new position in the mouse move event, we can check whether the new position will be outside of our slider's range. If that's the case, we are using the boundaries of the slider's range instead of the new x-position.

```
const Slider = () => {
  ...

  const handleMouseMove = event => {
    let newX =
      event.clientX -
      diff.current -
      sliderRef.current.getBoundingClientRect().left;

    const end = sliderRef.current.offsetWidth - thumbRef.current.offsetWidth

    const start = 0;

    if (newX < start) {
      newX = 0;
    }

    if (newX > end) {
      newX = end;
    }
  };

  ...
};
```

Next, we will use the two values, the new position and the end of the range, to calculate the percentage of how far to move our thumb away from the left. Since the thumb itself has a width of 10px, we need to center it by removing half of its size, in order to not overflow the thumb to the right or left.

```
const getPercentage = (current, max) => (100 * current) / max;

const getLeft = percentage => `calc(${percentage}% - 5px)`;

const Slider = () => {
  ...

  const handleMouseMove = event => {
    let newX =
      event.clientX -
      diff.current -
      sliderRef.current.getBoundingClientRect().left;

    const end = sliderRef.current.offsetWidth - thumbRef.current.offsetWidth
```

```
    const start = 0;

    if (newX < start) {
      newX = 0;
    }

    if (newX > end) {
      newX = end;
    }

    const newPercentage = getPercentage(newX, end);

    thumbRef.current.style.left = getLeft(newPercentage);
  };

  ...
};
```

The React slider example should work now. We have used direct DOM manipulation to set the new `left` position of the slider's thumb. You could have also used React state here, but it would trigger React's internal state management very often when moving the thumb of the slider and lead to a re-render of the component with every mouse move. Doing it our way, we use direct DOM manipulation and avoid the actual re-rendering of React and do the manipulation of the DOM ourselves.

Exercise:

- Try the example with React's useState Hook instead of the `thumbRef.current.style.left` assignment
- Try the example with a vertical instead of the horizontal slider example

---

REACT SLIDER: COMPONENT

In the end, we would like to have a real React Slider Component with a slim API to the outside. At the moment, we cannot pass any props to the Slider Component and we don't get any current values from it with callback functions. Let's change this.

First, we will pass some initial values to our Slider Component. Let's say we want to have an initial position for the thumb and a max value for the range. We could pass and use them the following way for the initial render:

```
...

const Slider = ({ initial, max }) => {
  const initialPercentage = getPercentage(initial, max);
```

```
  const sliderRef = React.useRef();
  const thumbRef = React.useRef();

  ...

  return (
    <>
      <StyledSlider ref={sliderRef}>
        <StyledThumb
          style={{ left: getLeft(initialPercentage) }}
          ref={thumbRef}
          onMouseDown={handleMouseDown}
        />
      </StyledSlider>
    </>
  );
};

const App = () => (
  <div>
    <Slider initial={10} max={25} />
  </div>
);
```

Second, we will provide a callback function for the Slider Component which passes the recent set value to the outside. Otherwise, a React component using our Slider component wouldn't be able to receive any updates from it.

```
...

const getPercentage = (current, max) => (100 * current) / max;

const getValue = (percentage, max) => (max / 100) * percentage;

const getLeft = percentage => `calc(${percentage}% - 5px)`;

const Slider = ({ initial, max, onChange }) => {
  ...

  const handleMouseMove = event => {
    let newX = ...

    ...

    const newPercentage = getPercentage(newX, end);
    const newValue = getValue(newPercentage, max);

    thumbRef.current.style.left = getLeft(newPercentage);

    onChange(newValue);
  };
```

```
  return (
    <>
      <StyledSlider ref={sliderRef}>
        <StyledThumb
          style={{ left: getLeft(initialPercentage) }}
          ref={thumbRef}
          onMouseDown={handleMouseDown}
        />
      </StyledSlider>
    </>
  );
};

const App = () => (
  <div>
    <Slider
      initial={10}
      max={25}
      onChange={value => console.log(value)}
    />
  </div>
);
```

Third, we will show the Slider's *initial* and maximum range:

```
...

const SliderHeader = styled.div`
  display: flex;
  justify-content: flex-end;
`;

...

const Slider = ({ initial, max, onChange }) => {
  ...

  return (
    <>
      <SliderHeader>
        <strong>{initial}</strong>
         / 
        {max}
      </SliderHeader>
      <StyledSlider ref={sliderRef}>
        <StyledThumb
          style={{ left: getLeft(initialPercentage) }}
          ref={thumbRef}
          onMouseDown={handleMouseDown}
        />
      </StyledSlider>
    </>
  );
};
```

And will replace the shown initial range with the current range by using direct DOM manipulation again -- in order to get around React's re-rerendering mechanism when using its state management:

```
const Slider = ({ initial, max, onChange }) => {
  ...

  const currentRef = React.useRef();

  ...

  const handleMouseMove = event => {
    ...

    thumbRef.current.style.left = getLeft(newPercentage);
    currentRef.current.textContent = newValue;

    onChange(newValue);
  };

  return (
    <>
      <SliderHeader>
        <strong ref={currentRef}>{initial}</strong>
         / 
        {max}
      </SliderHeader>
      <StyledSlider ref={sliderRef}>
        <StyledThumb
          style={{ left: getLeft(initialPercentage) }}
          ref={thumbRef}
          onMouseDown={handleMouseDown}
        />
      </StyledSlider>
    </>
  );
};
```

If you try your Slider component, you should see its initial, current (after a mouse move) and maximum value for its range. Again, we have used React's direct DOM manipulation via ref instead of state to prevent re-rendering the whole component after each mouse move event. Doing it this way, we keep the component highly performant for being reused within our actual React application.

And last but not least, we will show an opinionated formatted value by default for our slider's range -- which can be specified from the outside via the Slider's component API though:

```
const Slider = ({
  initial,
  max,
```

```
  formatFn = number => number.toFixed(0),
  onChange,
}) => {
  ...

  const handleMouseMove = event => {
    ...

    thumbRef.current.style.left = getLeft(newPercentage);
    currentRef.current.textContent = formatFn(newValue);

    onChange(newValue);
  };

  return (
    <>
      <SliderHeader>
        <strong ref={currentRef}>{formatFn(initial)}</strong>
         / 
        {formatFn(max)}
      </SliderHeader>
      <StyledSlider ref={sliderRef}>
        <StyledThumb
          style={{ left: getLeft(initialPercentage) }}
          ref={thumbRef}
          onMouseDown={handleMouseDown}
        />
      </StyledSlider>
    </>
  );
};

const App = () => (
  <div>
    <Slider
      initial={10}
      max={25}
      formatFn={number => number.toFixed(2)}
      onChange={value => console.log(value)}
    />
  </div>
);
```

That's it. You have styled a slider component in React, made its interaction possible, and gave it an API to interact with it from the outside. You are good to go from here to use or to improve the component.

Exercises:

- The Slider only works when moving the thumb around. Extend the Slider's functionality so that it moves the thumb around when clicking on the Slider's track instead of using the thumb directly.

- Pass a different `formatFn` to the slider. For instance, you could use a formatter function to translate the number to a time format (e.g. 135000 to 00:02:15:000 for milliseconds to hh:mm:ss:ms).

. . .

The React Slider Component was inspired by this pure JavaScript implementation. Let me know in the comments how you improved your component and how you liked the tutorial.

This tutorial is part 1 of 2 in the series.

Part 2: How to React Range

Show Comments

KEEP READING ABOUT REACT›

HOW TO REACT RANGE

In this React component tutorial by example, we will create a React Range Component with React Hooks and a Function Component . You can see the final output of this implementation in this...

REACT COMPONENT COMPOSITION

There are various advanced concepts and patterns in React to master on becoming a React developer. In this tutorial, I want to go through React's Component Composition step by step by evolving one...

Your email address SUBSCRIBE

View our Privacy Policy.

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms