Section 4: React State & Working with Events

Brian E. Nguyen

October 27, 2021

Contents

1 Module Introduction **2**

2 Listening to Events & Working With Event Handlers **2**

3 How Component Functions Are Executed **4**

4 Working with 'State' **6**

5 A Closer Look at the "useState" Hook **8**
5.1 Per-Component Basis 8
5.2 Using `const`? 10
5.3 Closing Mentions 10

6 State Can Be Updated in Many Ways **10**

7 Adding Form Inputs **10**

8 Listening to User Input **12**

9 Working With Multiple States **14**

10 Using One State Instead (and What's Better) **16**

11 Updating the State That Depends on the Previous State **17**

12 Handling Form Submission **19**

13 Adding Two-Way Binding **20**

14 Child-to-Parent Component Communication (Bottom-up) 22
14.1 Intro 22
14.2 Pass Data from `ExpenseForm` to `NewExpense` 22
14.3 Pass Data from `NewExpense` to `App` 24

15 Lifting the State Up 25

1 Module Introduction

In this module, we will take a closer look at the following

- user interaction
  - this includes events like click, inputs, etc.
- state management
  - so far we can only build static applications where the state never changes, and that's not what we want

2 Listening to Events & Working With Event Handlers

- we only have one state in our application, which is the inital state
- let's start with clicks on a button which you want something to happen
- in the `ExpenseItem` component, let's add a button tag
  - this will be a temporary button with no styles so that we can practice with React states

```
// Expense Item
import ExpenseDate from "./ExpenseDate";
import Card from "../UI/Card";
import "./ExpenseItem.css";

const ExpenseItem = (props) => {
  return (
    <Card className="expense-item">
      <ExpenseDate date={props.date} />
      <div className="expense-item__description">
```

```
        <h2>{props.title}</h2>
        <div className="expense-item__price">${props.amount}</div>
      </div>
      <button>Change Title</button>
    </Card>
  );
};

export default ExpenseItem;
```

- the goal of this button is to change the title when the button is clicked
- React has a simple way of detecting button clicks
- on all built-in HTML elements, we have full access to native DOM events which we can listen to
- we will add a special prop in JSX called `on`. This can be followed by `onClick`, etc.
  - from there, we can add JavaScript logic to it

```
import ExpenseDate from "./ExpenseDate";
import Card from "../UI/Card";
import "./ExpenseItem.css";

const ExpenseItem = (props) => {
  return (
    <Card className="expense-item">
      <ExpenseDate date={props.date} />
      <div className="expense-item__description">
        <h2>{props.title}</h2>
        <div className="expense-item__price">${props.amount}</div>
      </div>
      <button onClick={() => {console.log('Clicked!')}}>Change Title</button>
    </Card>
  );
};

export default ExpenseItem;
```

- we typically want to define a function before the `return` statement

```
import ExpenseDate from "./ExpenseDate";
import Card from "../UI/Card";
import "./ExpenseItem.css";

const ExpenseItem = (props) => {
  const clickHandler = () => {
    console.log("Clicked!");
  };
  return (
    <Card className="expense-item">
      <ExpenseDate date={props.date} />
      <div className="expense-item__description">
        <h2>{props.title}</h2>
        <div className="expense-item__price">${props.amount}</div>
      </div>
      <button onClick={clickHandler}>Change Title</button>
    </Card>
  );
};

export default ExpenseItem;
```

- when we call the function, we don't add parentheses to it. Why? Because JavaScript will execute the function when the entire JSX line is parsed
- it's a convention that these functions end with `Handler`. Not everyone does this, but take note of it

3 How Component Functions Are Executed

- reacting to events is an important first step. How can we now change what shows up on the screen?
- we can add a new variable called `title` and pass it into the JSX code.
- now that we created a variable, we can use the `clickHandler` function to let us change the title

```
// ExpenseItem.js
import ExpenseDate from "./ExpenseDate";
```

```
import Card from "../UI/Card";
import "./ExpenseItem.css";

const ExpenseItem = (props) => {
  let title = props.title;

  const clickHandler = () => {
    title = "Updated!";
  };
  return (
    <Card className="expense-item">
      <ExpenseDate date={props.date} />
      <div className="expense-item__description">
        <h2>{title}</h2>
        <div className="expense-item__price">${props.amount}</div>
      </div>
      <button onClick={clickHandler}>Change Title</button>
    </Card>
  );
};

export default ExpenseItem;
```

- but if we were to actually click on the button, the title doesn't change. Why is that? The click handler still executed though and the value of `title` is "Updated!"
- the reason is simply because React doesn't work like this. We'll dive deeper into the reason later in the course
- what you need to know right now is that your component is a function; the only special thing about this function is that it returns JSX code
- we never explicity call our component functions. We just use them like HTML elements
- **the main point:** by using components like HTML elements, we make React aware of these component functions. When React evaluates the JSX, it will call the component functions
  - then the component functions will call any component functions inside of them. This process will repeat itself until there is no

more JSX

- now, we need a way to tell React to reevaluate these component functions when we execute a handler

4 Working with 'State'

- *state* is not a React-specific concept
- we need to trigger a re-evaluation so that we can change our title
  - note that variables, like `title` are not triggered in the re-evaluation
  - React doesn't care about that. The component function doesn't get re-evaluated just because a variable changed
- to tell React that the component function should run again, we need to import something from the React library called `useState`
  - this is a function provided by the React library which allows us to define values as state, where changes to these values should reflect in the component function

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

- *inside* of our component function, we just call `useState` to use it
- *tip:* hook functions start with *use* and can only be called inside component functions, but not inside nested functions
- with `useState`, it wants a default-state value. We can assign an initial variable of `props.title` and pass it as an argument inside `useState`

```
// ExpenseItem.js
import React, { useState } from "react";

import ExpenseDate from "./ExpenseDate";
import Card from "../UI/Card";
import "./ExpenseItem.css";

const ExpenseItem = (props) => {
  useState(props.title);
```

```
  let title = props.title;

  const clickHandler = () => {
    title = "Updated!";
  };
  return
...
}
```

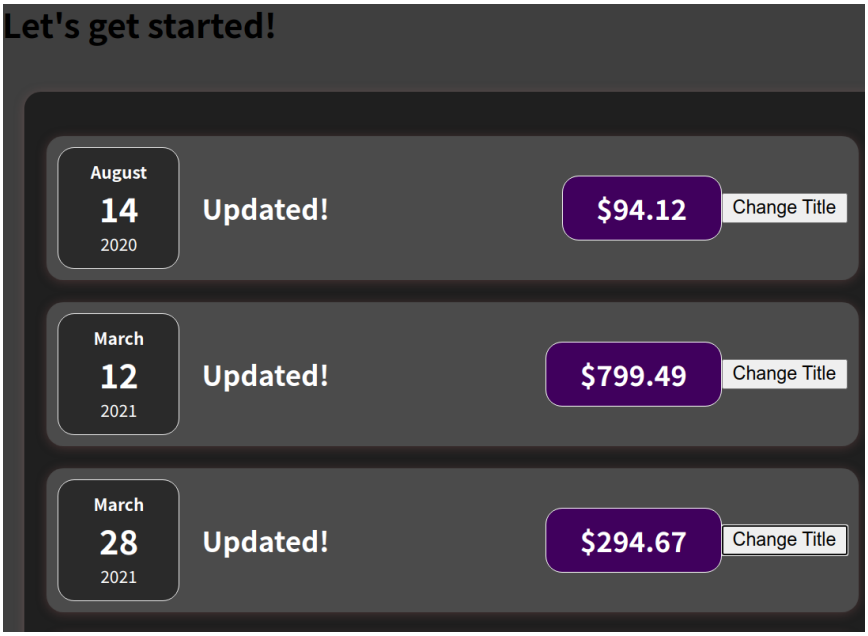
- `useState` not only lets us use the special variable in other places, but it also returns a function that we can call to assign a new value to that variable
  - `useState` returns an array where it has two values inside of it:
    1. the variable itself
    2. the updating function
  - you can use array-destructuring to retrieve these values

```
const [title, setTitle] = useState(props.title)
```

- this is now what the updated function looks like with `setTitle` in use

```
const ExpenseItem = (props) => {
  const [title, setTitle] = useState(props.title);

  const clickHandler = () => {
    setTitle("Updated!");
  };
  ...
}
```

- note that the `setTitle` function doesn't only assign the new value to a new variable, but it is also managed by React somewhere in memory because it's a special variable
- now when we click on a button, the title now updates it



5 A Closer Look at the "useState" Hook

5.1 Per-Component Basis

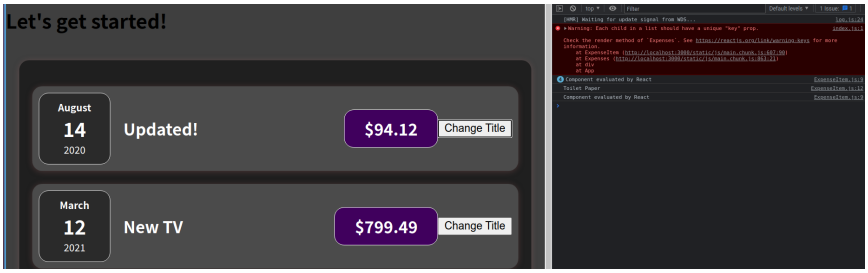
- `useState` registers some value as a state for the component in which it is being called
  - to be more precise, it registers it for a specific component instance
- in our `Expenses` component, our `ExpenseItem` component is used four times

```
// Expenses.js

// Note that the props passed in have 4 objects
const Expenses = (props) => {
  return (
    <Card className="expenses">
      {props.expenses.map(function (obj, i) {
        return (
          <ExpenseItem
            title={props.expenses[i].title}
```

```
            amount={props.expenses[i].amount}
            date={props.expenses[i].date}
          />
        );
      })}
    </Card>
  );
};
```

- the `useState` function is called 4 times when we create the 4 `ExpenseItem` components, and each are managed independently by React
  - that is why when we click on a button and title changes, the rest of the titles don't change, because they have their own state
- if we put a `console.log()` in our component, when we load up a page, this code will run 4 times; however, when we click on a button, it only runs once

```
// ExpenseItem.js
 const ExpenseItem = (props) => {
  const [title, setTitle] = useState(props.title);
  console.log("Component evaluated by React");
  const clickHandler = () => {
    setTitle("Updated!");
    console.log(title);
  };
  ...
```



- the takeaway is that *state is separated on a per-component basis*

5.2 Using `const`?

- why are we using `const` at the array-destructuring part when we eventually assign it a new value?
- note that we are not assigning a value with the = sign when we update the state
- instead, we call the state-updating function, and the value is managed somewhere else in React
- we never assign the value of `title` with the = operator; therefore, using `const` is fine

5.3 Closing Mentions

- knowing state is important, because in more complex apps, there could be times when a value doesn't update when it should
- using state is simple.
  1. You just register state with `useState`,
  2. you always get two values,
  3. you call the updating function, and
  4. you use the first value for outputs in JSX code

6 State Can Be Updated in Many Ways

Thus far, we update our state **upon user events**, (e.g. upon a click). That's very common but not required for state updates. *You can update states for whatever reason you many have.

Later in the course, we'll see HTTP requests where we want to update the state based on the HTTP response, but you could also be updating state because a timer (set with `setTimeout()`) expired for example

7 Adding Form Inputs

- on thing that is missing from our app is getting user input
- we will add new components and a new category of components.
  - We will call this folder `NewExpense`

- inside of the folder, we will add a new component called `NewExpense` where it will render a form for users to input information, as well as its styles
    - click here to access the CSS code for NewExpense

- we will also take the form and put it into a separate component so that the form logic is in it's own component. This will be called `ExpenseForm.js`
    - click here for the ExpenseForm CSS code

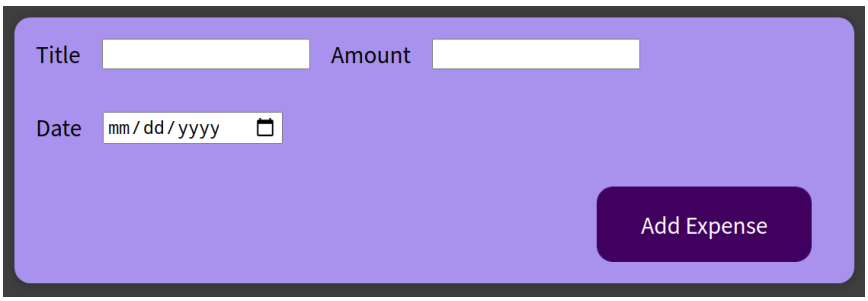
```
// ExpenseForm.js
import "./ExpenseForm.css";
const ExpenseForm = () => {
  return (
    <form>
      <div className="new-expense__controls">
        <div className="new-expense__controls">
          <label>Title</label>
          <input type="text" />
        </div>
        <div className="new-expense__controls">
          <label>Amount</label>
          <input type="number" min="0.01" step="0.01" />
        </div>
        <div className="new-expense__controls">
          <label>Date</label>
          <input type="date" min="2019-01-01" max="2021-12-31" />
        </div>
      </div>
      <div className="new-expense__actions">
        <button type="submit">Add Expense</button>
      </div>
    </form>
  );
};

export default ExpenseForm;
```

- we will add event listeners to this form soon

- now we will import this component to `NewExpense`

```
// NewExpense.js
import ExpenseForm from "./ExpenseForm";

import "./NewExpense.css";

const NewExpense = () => {
  return (
    <div className="new-expense">
      <ExpenseForm />
    </div>
  );
};

export default NewExpense;
```

- now we want to render the `NewExpense` component inside `App.js`



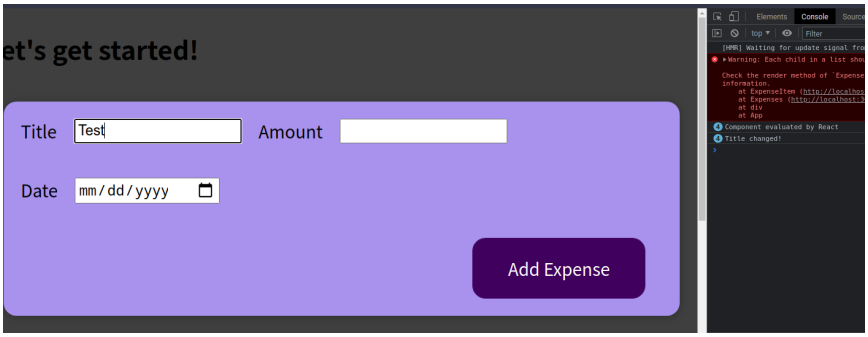
- in the next lesson, we will figure out how to make the form retrieve input, because it's not doing anything right now

8 Listening to User Input

- let's add an `onChange` event listener to the user input on `ExpenseForm.js`

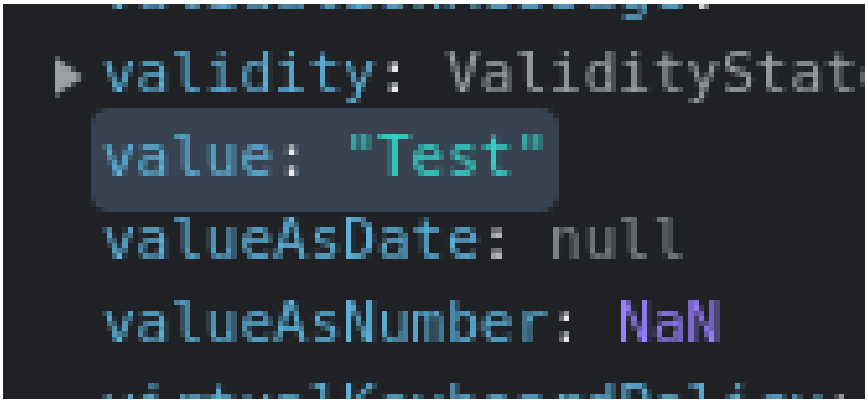
```
import "./ExpenseForm.css";
const ExpenseForm = () => {
  const titleChangeHandler = () => {
    console.log("Title changed!");
  };
```

```
  return (
    <form>
      <div className="new-expense__controls">
        <div className="new-expense__controls">
          <label>Title</label>
          <input type="text" onChange={titleChangeHandler} />
        </div>
        <div className="new-expense__controls">
          <label>Amount</label>
          <input type="number" min="0.01" step="0.01" />
        </div>
        <div className="new-expense__controls">
          <label>Date</label>
          <input type="date" min="2019-01-01" max="2021-12-31" />
        </div>
      </div>
      <div className="new-expense__actions">
        <button type="submit">Add Expense</button>
      </div>
    </form>
  );
};

export default ExpenseForm;
```



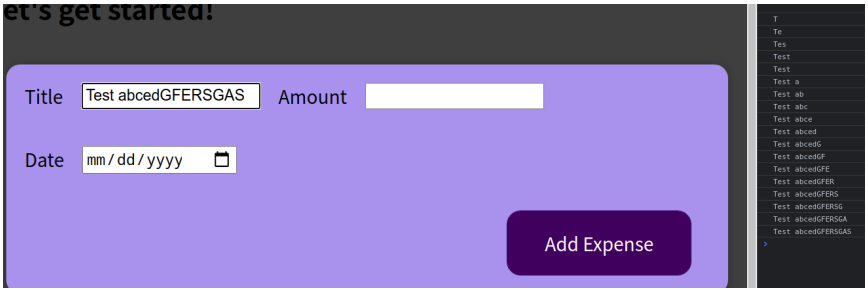
- the title changes for every keystroke. But now we actually want to get the user input
- we can get the event as an object. This has a lot of data, but the one

we're interested in is `target` that has a property called `value`. This holds the value of the user input



- we can now use that value for our purposes

```
// ExpenseForm.js
const ExpenseForm = () => {
  const titleChangeHandler = (event) => {
    console.log(event.target.value);
  };
```



9 Working With Multiple States

- the question is what do we want to do with this value?
  - store it somewhere so that when the form is submitted, we can use that value
- to make this value survive once the function is re-executed, we can use state

- we will create the `useState` function and pass in an empty string; the reason is initally when the component is rendered for the first time, nothing will be entered.
    - we will define the returned values as `enteredTitle` and `setEnteredTitle`, and we can also do this for the other states for amount and date

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

import "./ExpenseForm.css";
const ExpenseForm = () => {
  const [enteredTitle, setEnteredTitle] = useState("");
  const [enteredAmount, setEnteredAmount] = useState("");
  const [enteredDate, setEnteredDate] = useState("");

  const titleChangeHandler = (event) => {
    setEnteredTitle(event.target.value);
  };

  const amountChangeHandler = (event) => {
    setEnteredAmount(event.target.value);
  };

  const dateChangeHandler = (event) => {
    setEnteredDate(event.target.value);
  };
  return (
    <form>
      <div className="new-expense__controls">
        <div className="new-expense__controls">
          <label>Title</label>
          <input type="text" onChange={titleChangeHandler} />
        </div>
        <div className="new-expense__controls">
          <label>Amount</label>
          <input
            type="number"
            min="0.01"
            step="0.01"
            onChange={amountChangeHandler}
```

```
          />
        </div>
        <div className="new-expense__controls">
          <label>Date</label>
          <input
            type="date"
            min="2019-01-01"
            max="2021-12-31"
            onChange={dateChangeHandler}
          />
        </div>
      </div>
      <div className="new-expense__actions">
        <button type="submit">Add Expense</button>
      </div>
    </form>
  );
};

export default ExpenseForm;
```

- one problem you may have encountered is how can you manage more than one state?
  - you can call `useState` more than once inside of a component
- it's not unusual to have more than one state for a component. You can and will have multiple

10 Using One State Instead (and What's Better)

- there is an alternative from using multiple states, and it's up to you on what you choose
- what we have now is 3 states for one purpose
- what we can do instead is passing in an object into the `useState()` function

```
// const [enteredTitle, setEnteredTitle] = useState("");
// const [enteredAmount, setEnteredAmount] = useState("");
```

```
  // const [enteredDate, setEnteredDate] = useState("");

const [userInput, setUserInput] = useState({
    enteredTitle: '',
    enteredAmount: '',
    enteredDate: '',
  });

  const titleChangeHandler = (event) => {
    // setEnteredTitle(event.target.value);
    setUserInput({
      ...userInput,
      enteredTitle: event.target.value
    })
  };
```

- when you set the value for one field, you also need to ensure that the other two values don't get lost
- you would need to use the spread operator (...), which takes an object, pulls out all of the key-value pairs, and overides the variable
- it's more common to see separate states instead of them merged into one, which is fine
- what's not fine is how we are updating the state

11 Updating the State That Depends on the Previous State

```
// ExpenseForm.js
  const titleChangeHandler = (event) => {
    // setEnteredTitle(event.target.value);
    setUserInput({
      ...userInput,
      enteredTitle: event.target.value,
    });
  };

  const amountChangeHandler = (event) => {
```

```
    // setEnteredAmount(event.target.value);
    setUserInput({
      ...userInput,
      enteredAmount: event.target.value,
    });
  };

  const dateChangeHandler = (event) => {
    // setEnteredDate(event.target.value);
    setUserInput({
      ...userInput,
      enteredDate: event.target.value,
    });
  };
```

- what we did here with updating the state is not entirely correct
- it could fail, and it's not a good practice to update like this
- what's the problem? we depend on the previous state. We need to copy the other values so that we don't lose them
- **Important note:** when you update state and depend on the previous state, you should not use what we have currently. Instead, you should pass in another function into the `setUserInput()` function. This anonymous function receives the previous state, which is held inside of a variable `prevState`

```
// ExpenseForm.js
 const titleChangeHandler = (event) => {
    // setEnteredTitle(event.target.value);
    // setUserInput({
    //   ...userInput,
    //   enteredTitle: event.target.value,
    // });
    setUserInput((prevState) => {
      return { ...prevState, enteredTitle: event.target.value };
    });
  };
```

- why use this instead of the other method? They both work fine

- keep in mind that React schedules updates and doesn't perform them instantly
- if you schedule updates many times, you could be depending on an outdated or incorrect state snapshot if you use the old approach
- our new function will always hold the latest state snapshot
- for now, we will switch back to the multiple-state option

12 Handling Form Submission

- let's make sure that the form is submitted when the button is pressed
- we could add an `onClick` event to it, but that wouldn't be necessary. When a button, especially one that has `type='submit'` into it, is nested inside of a `<form>` tag, then the button will automatically emit an event
- we will add an `onSubmit` event to the form and a new function called `submitHandler()`
- the default behavior of when a form is submitted is that the page reloads, because it is sending a request, and that's not what we want
- just like all handler functions, we get an `event` object, and we can prevent page reloads

```
// ExpenseForm.js
  const submitHandler = (event) => {
    event.preventDefault();
  };
```

- now we can retrieve the entered data and add them into a new object called `expenseData`

```
// ExpenseForm.js
  const submitHandler = (event) => {
    event.preventDefault();

    const expenseData = {
      title: enteredTitle,
```

```
      amount: enteredAmount,
      date: new Date(enteredDate),
    };
  };
```

- now let's log it so that we can see the output

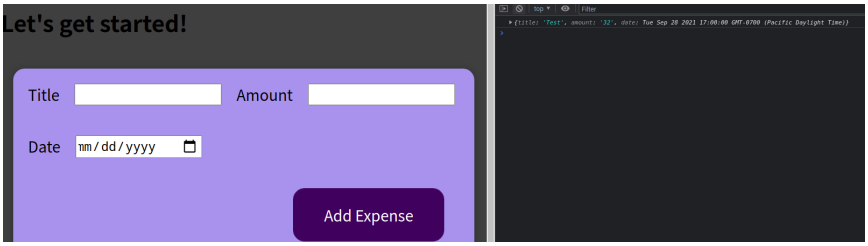
13 Adding Two-Way Binding

- how can we clear inputs after a form has been submitted?
- we can use a technique called **two-way binding**, where we don't just listen to changes, but we can also pass a new value back into the input
- this is very simple. On the `<input>` tag, we just need to add the `value` attribute and bind it to `enteredTitle`, etc.
  - when the form is submitted, then we can set it back to an empty string with `setEnteredTitle()`, etc.

```
const submitHandler = (event) => {
    event.preventDefault();

    const expenseData = {
      title: enteredTitle,
      amount: enteredAmount,
      date: new Date(enteredDate),
    };

    console.log(expenseData);
    setEnteredTitle('');
    setEnteredAmount('');
    setEnteredDate('');
  };

  return (
```

```
    <form onSubmit={submitHandler}>
      <div className='new-expense__controls'>
        <div className='new-expense__controls'>
          <label>Title</label>
          <input
            type='text'
            value={enteredTitle}
            onChange={titleChangeHandler}
          />
        </div>
        <div className='new-expense__controls'>
          <label>Amount</label>
          <input
            type='number'
            min='0.01'
            step='0.01'
            value={enteredAmount}
            onChange={amountChangeHandler}
          />
        </div>
        <div className='new-expense__controls'>
          <label>Date</label>
          <input
            type='date'
            min='2019-01-01'
            max='2021-12-31'
            value={enteredDate}
            onChange={dateChangeHandler}
          />
        </div>
      </div>
      <div className='new-expense__actions'>
        <button type='submit'>Add Expense</button>
      </div>
    </form>
  );
};
```

- now when we submit the form, we retrieve our data and the form will be cleared



14 Child-to-Parent Component Communication (Bottom-up)

14.1 Intro

- the problem is that this data is nice, but we don't need it inside of the `ExpenseForm` component. Instead, we need it inside `NewExpense` or `App`
- we need to pass data to the `App` component
- so far we only learned how to pass data down. Now we need to figure out passing it up
- we actually saw how it works, but you most likely missed it
  - in `ExpenseForm`, we are listening to user input. The change handlers are an example of this
  - you can think of the attributes in the `<input>` tag as an example

14.2 Pass Data from `ExpenseForm` to `NewExpense`

1. let's say we want to pass `expenseData` from `ExpenseForm` to `NewExpense`. Props can be only passed from parent to child. As a first step inside of `NewExpense`, we will add a new prop to `ExpenseForm`
   - note that you cannot skip components
2. We can create our own custom props. Let's call it `onSaveExpenseData`

```
// NewExpense.js
import ExpenseForm from "./ExpenseForm";

import "./NewExpense.css";
```

```
const NewExpense = () => {
  return (
    <div className="new-expense">
      <ExpenseForm onSaveExpenseData />
    </div>
  );
};

export default NewExpense;
```

1. Then we will create a new function that will expect submitted data from the form. From there, we can pass that function to `onSaveExpenseData`

```
const NewExpense = () => {
  const saveExpenseDataHandler = (enteredExpenseData) => {
    const expenseData = {
      ...enteredExpenseData,
      id: Math.random().toString(),
    };
  };
  return (
    <div className='new-expense'>
      <ExpenseForm onSaveExpenseData={saveExpenseDataHandler} />
    </div>
  );
};
```

1. Now we need to use this function inside of our `ExpenseForm` component. We would need to pass in props inside of our component
2. Inside of our `submitHandler()` function, we would need to call `props.onSaveExpenseData()` and pass in `expenseData`

```
// ExpenseForm.js
const ExpenseForm = (props) => {
  ...
  props.onSaveExpenseData(expenseData);
  setEnteredTitle('');
  setEnteredAmount('');
  setEnteredDate('');
}
```

- now when we submit data, the data is now being logged to `NewExpense`
- we can also tell that the data is new because it now has a randomly generated ID

14.3 Pass Data from `NewExpense` to `App`

1. create a new function called `addExpenseHandler` inside of ~App~A

```
// App.js
const addExpenseHandler = (expense) => {
    console.log(expense);
  };
```

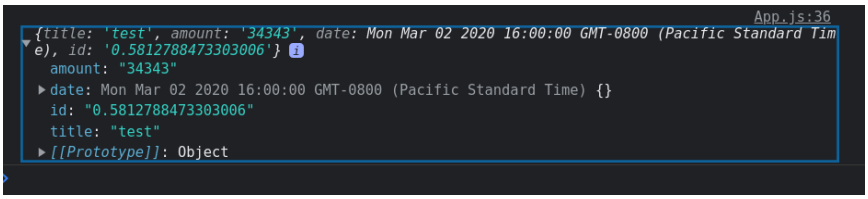
1. create a prop called `onAddExpense` to `NewExpense` tag and add the function to it

```
// App.js
return (
    <div>
      <h2>Let's get started!</h2>
      <NewExpense onAddExpense={addExpenseHandler} />
      <Expenses expenses={expenses} />
    </div>
  );
```

1. Allow `NewExpense` to accept props and call the function with `expenseData` passed into it

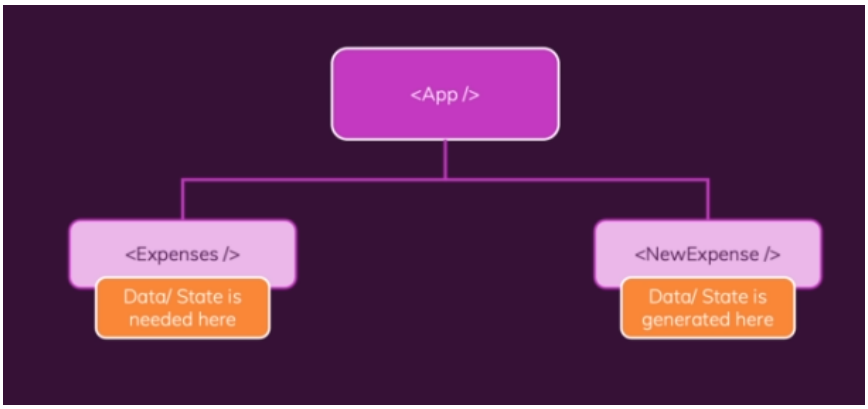
```
// NewExpense.js
const NewExpense = (props) => {
  const saveExpenseDataHandler = (enteredExpenseData) => {
    const expenseData = {
      ...enteredExpenseData,
```

```
    id: Math.random().toString(),
  };
  props.onAddExpense(expenseData);
};
```



15 Lifting the State Up

- in the last lecture, we learned a very important concept of moving props from child to parent
- we will now learn about a concept called **Lifting State Up**. What is that about?
- consider this component tree, which is similar to the app that we have
  - the `NewExpense` component generates data and state
  - it's common that you generate data and state in a component, but you don't need them specifically in that component. Rather, you would move them to another one. In this case, we need to send data from `NewExpense` to `Expenses`



- we would want to hand the data over, but it doesn't work like that between two sibling components