# Web Front-End Development
# Week 12: React Data Flow

Lists and Keys

When you render a list of items, React asks you to assign a unique string to the *key* property on each element in a list, differentiate each component from its siblings.
Keys tell React about the identity of each component, so that it can maintain the state across rerenders.
Keys help React identify which items have changed, are added, or are removed.

- use a string that uniquely identifies a list item
- use IDs from your data as keys
- use the item index as a key as a last resort
- don't use indexes for keys if the items can reorder

If you don't you will receive the following warning:
Warning: Each child in an array or iterator should have a unique "key" prop.

Keys are used by React but they don't get passed to your components and there is no way for a component to access its own key. Even though it may look like it is part of props, it cannot be referenced with `this.props.key`.
If you need the same value in your component, pass it explicitly as a prop with a different name.

Example:
Favorite #3 (App3.js, Favorite3.js, and FavoriteList3.js)

- Restructured to add the FavoriteList component to handle the list of foods. The App component now just renders FavoriteList.
    - Note that you must import all components you render
    - Also updated App.css
- FavoriteList
    - Instead of having App pass in the names of the food to the component, I created an array of foods in FavoriteList.
        - foods is an array of objects so each object has a key and a name.
        - In the future if I want to take this as input I can move this into state
- The FavoriteList component now returns an unordered list of food
    - We use the map() function to call the createFood(item) function for every item in the food array
    - We create a <li> element with the key property using the key value from the array so each item has a unique id. Then we generate a <Favorite> component passing the item's name as the prop name.
- The Favorite component is mostly unchanged, only the div for each item's style has moved into FavoriteList

## Data Flow

Along with the ability for a component class to pass down its state as a value to be displayed, it can also pass down the ability to *change* its state through a function.
These are distinct abilities:

- A stateless child component that receives the state can access and displays its value
- A different stateless child component has a way to *change* its parent's state

A *stateful* component is a component which has state. That component defines a function that calls `this.setState()`.
The stateful (parent) component passes that function down to a stateless (child) component.

A *stateless* (child) component defines a function that calls the passed-down function by accessing it through props.
The function in the stateless (child) component can take an *event object* as an argument.
The stateless (child) component class uses this new function as an event handler.
When the event fires, the event handler is called and the parent's state updates.

A stateful, parent component passes down an *event handler* to a stateless, child component. The child component then uses that *event handler* to update its parent's `state`.

Parent component:

```
constructor(props) {
  super(props);
  this.state = { name: 'Aileen' };
  this.changeName = this.changeName.bind(this);
}

changeName(newName) {
  this.setState({name: newName});
}

render() {
 return <Child name={this.state.name} onChange={this.changeName}
/>
}
```

Due to the way that event handlers are bound in JavaScript, `this.changeName()` loses its `this` when it is used in render(). Therefore we have to bind `this.changeName()` to `this` in the constructor method.
In React, whenever you define an event handler that uses `this`, you need to add `this.methodName = this.methodName.bind(this)` to your constructor function.

Child component:

```
constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
}

handleChange(e) {
  const name = e.target.value;
  this.props.onChange(name);
}

<select onChange={this.handleChange}>
```

Here is how the Parent component's state would be set:

1. A user triggers an onChange event by selecting a value in the select list.
2. The onChange event from is being listened for on the select element.
3. When the event fires it calls the *event handler* function assigned to it -- `this.handleChange`
4. `handleChange(e)` takes an *event object* as an argument. Using the event object you can access the target's value. The target is the element that the event was fired on.
5. That name is then passed to `this.props.onChange(name)`. `props.onChange` was assigned the value of `changeName` in the parent component.
6. Inside of the body of `changeName`, **`this.setState()`** is called and the component's state is changed.

Any time that you call this.setState(), this.setState() AUTOMATICALLY calls .render() as soon as the state has changed.

Think of `this.setState()` as actually being two things: `this.setState()`, immediately followed by `.render()`.

*That* is why you can't call `this.setState()` from inside of the `.render()` method. `this.setState()` *automatically* calls `.render()`. If `.render()` calls `this.setState()`, then an infinite loop is created.

Example:
Favorite #4 (Favorite4.js, and FavoriteList4.js)

- FavoriteList
  - Added totalVotes as state to FavoriteList and the method updateTotalVotes to setState. In the constructor I added the line to bind this for updateTotalVotes.
  - Votes don't take place in this component so I need to pass this function as a prop to the Favorite component called voteChange.
  - Added the ability to see totalVotes in the last line of return()
- Favorite
  - Every time a vote is made we use the prop voteChange to call the method updateTotalVotes in the FavoriteList component. We could also pass data as an argument if the function needed any.

## Lifting State Up

To determine where state should live for each piece of state in your application:

- Identify every component that renders something based on that state.
- Find a common owner component (a single component above all the components that need the state in the hierarchy).
- Either the common owner or another component higher up in the hierarchy should own the state.
- If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.
- Pass state down to other components as props.

When several components need to reflect the same changing data. In React, sharing state is accomplished by moving it up to the closest common ancestor of the components that need it. This is called "lifting state up". https://reactjs.org/docs/lifting-state-up.html

- When you want to aggregate data from multiple children or to have two child components communicate with each other, move the state upwards so that it lives in the parent component. The parent can then pass the state back down to the children via props, so that the child components are always in sync with each other and with the parent.
- State lives in one component and only that component can change it. Don't try to sync state between different components, let the data flow top-down.
- If something can be derived from either props or state, it probably shouldn't be in the state, just calculate it in render().