Conclusion

39

Chris Dobson

39

React Todo List

Chris Dobson

Jul 29, 2019 • 12 Min read • 13,406 Views

Jul 29, 2019 • 12 Min read • 13,406 Views

Web Development React

Introduction

This two part guide will use React to develop a simple todo list. We will start by learning how to display a list, add functions to mark an item as complete, and, finally, how to filter the list by complete and overdue items.

The second part, will make our list more dynamic as we explore how to use a mock API to retrieve and update the list, as well as add functions to add new items.

Conclusion

39

The guide requires a version of React that includes the hooks API and will assume that Bootstrap is available. Extra CSS will be ovided by this file.

39

The application will start with a header component at the top of the page. The header component starts by simply displaying the title of the application and looks like this:

javascript

```javascript
function Header() {
  return (
    <header>
      <nav className="navbar navbar-light bg-light">
        <span className="navbar-brand">
          Todo list
        </span>
      </nav>
    </header>);
}
```

Listing Items

The first things a todo list needs is a list of todo items in the view. To start with this list, will be defined in a json file like this:

Conclusion

39

39

json

```json
{
  "Items": [
        {
          "id": "1",
          "timestampDue": 1561881986756,
          "name": "Pay credit card bill",
          "complete": false
        }
        ...
      ]
    }
```

The data contains a unique id, a timestamp of the date/time the todo item becomes due, the name of the item, and a boolean to determine whether it has been completed or not. This file can be imported into the application like this:

javascript

```javascript
import todo from "./mockData.json";
```

To display the items in the view, this code can be used:

javascript

```javascript
<ul className="list-group">
  {todo.Items.map(item => (
    <li key={item.id} className="list-group-item">
      {`${item.name} - ${dateformat(new Date(item.timestampDue), "dd-mmm-y
        </li>))}
    </ul>
```

39

~~~~his creates a `ul` element and maps each item in the list to an `li` element that contains just the name of the todo item along with the date the item is due. The date is being formatted using the dateformat package. Note the `key` prop that is being set on the `li`; when rendering multiple components from a list React requires a unique key prop for each component in order to determine when the list, and any items in it, have changed.

While this code is fine, we should be using principles of component composition, especially given that it is known that extra functionality will shortly be added to an item in the list. To do this the `li` element can be extracted into a new component that accepts the item as a prop. This is done like this:

javascript

```javascript
function Item({ item }) {
  return (
    <li className="list-group-item">
      {`${item.name} - ${dateformat(new Date(item.timestampDue), "dd-mmm-y
    </li>);
}
```
~~~~

This functional component takes a prop of `item`. The syntax in the functions argument list `{ item }` is using object destructuring to tract each prop; the function could have just accepted an 39 gument like this `(props)` and referenced the item like this `props.item`.

This new component can be consumed like this:

javascript

```javascript
{todo.Items.map(item => <Item key={item.id} item={item} />)}
```

All that has been done here is that, rather than map each item directly to an `li` element, each item is mapped to the `Item` component. Note that the key attribute is now set on the `Item` component.

One feature that might be useful in this UI is to highlight any items that are past their due date. To calculate whether an item is overdue, simply take the `timestampDue` property of the item and check if it is before the current timestamp; it should also not return true for any completed items.

This function will calculate if an item is overdue:

javascript

```javascript
function isOverdue(item) {
  return !item.complete && item.timestampDue < new Date().getTime();
```

```
    }
```

39

s can then be used to set the class of the `li` element to `list-oup-item list-group-item-danger` when overdue and `list-group-item list-group-item-info` if not, like this:

javascript

```javascript
const itemClass = `list-group-item list-group-item-${isOverdue(item) ? "da
return (
  <li className={itemClass}>
    ...
  </li>);
```

The todo app now has a list of todo items and distinguishes between those that are overdue and those that aren't.

Complete Items

Once an item on the list has been performed, it needs to be set as completed so that it can be shown as complete or, later in the guide, not displayed at all.

Currently, the todo items are imported from a json file and accessed directly by the code; as we are going to start changing the items in the list, they will need to be stored in a component state. This is done like this:

39

javascript

```javascript
const [items, setItems] = React.useState(todo.Items);
```

This code uses the state hook; this hook returns an array and the code uses array destructuring to get a state that lists the current items and a function - `setItems` - that can be used to update that state. Instead of using `todo.Items` to access the items in the todo list, `items` should now be used.

In order to complete an item in the list, the code will need to create a list with the `complete` property of the specified item set to `true` and pass this list to `setItems` which will update the state. When the state has changed React will re-render the components to reflect this change. This function will set an item to be complete:

javascript

```javascript
function completeItem(id) {
  const updatedItems = items.map(item =>
    (item.id === id ? { ...item, complete: true } : item));
  setItems(updatedItems);
}
```

Conclusion

39

This function is using the map array function to project a new list item. If the ID of the item does not match the ID of the item that is ing set then `map` returns the same item; if it does match the ID 39 en the object spread operator is used to return a new item with all of the same properties except that `complete` is set to true. This new list is then passed to `setItems` and the relevant components are updated.

All that remains is to allow a user to set an item as complete and display an item as complete in the UI; both of these changes will be made in the `Item` component.

Firstly, the component will need to add a button for the user to click to complete an item, and then accept a new prop of `completeItem` that will be a function to be called when the button is clicked and will ultimately call the `completeItem` function with the ID of the item. Finally, a class can be added to the text of the item so that it can be shown as complete - in the example CSS used in this guide, a complete item will be shown crossed out and slightly opaque.

The final `Item` component looks like this:

javascript

```javascript
function Item({ item, completeItem }) {
  const itemClass = `list-group-item list-group-item-${isOverdue(item) ?
  return (
    <li className={itemClass}>
```

```javascript
        <div className="item">
          <span className={`item-title${item.complete ? " complete-item" :
            {`${item.name} - ${dateformat(new Date(item.timestampDue), "dd-
          </span>
          {!item.complete && (
            <button type="button" className="btn btn-link" onClick={complet
              Complete item
            </button>)}
        </div>
      </li>);
  }
```

The new `div` inside the `li` is just there to ensure that the button is displayed on the right hand side of the component, the `span` is used to display the text as complete or not, and the button is only rendered if the item is not already complete.

There will also need to be a change to the usage of this component - a `completeItem` prop is now required. The list of items is now built like this:

javascript

```javascript
{items.map(item => (
  <Item key={item.id} item={item} completeItem={() => completeItem(item.ic
```

The `completeItem` prop is set to an arrow function that passes the current `item.id` to the `completeItem` function.

39

ter the List

The final piece of functionality to be added to the application is filtering of the list. Currently, all of the items in the list are shown regardless of their completion status, meaning that, after a while, it may be difficult for a user to spot the incomplete items. The application should not show complete items unless the user specifically asks for them and should also allow the user the option to only show overdue items.

These filters need to be stored in component state and set via UI in the `Header` component. The filter state will be initialized like this:

javascript

```javascript
const [filter, setFilter] =
  React.useState({ overdueOnly: false, includeComplete: false});
```

The `filter` object and the `setFilter` function are passed to the `Header` component as props and the following components added:

javascript

```javascript
<Checkbox
  label="Overdue items only"
  selected={filter.overdueOnly}
  select={() => setFilter({...filter, overdueOnly: true})}
```

39

```
    unSelect={() => setFilter({...filter, overdueOnly: false})}
  />
  <Checkbox
    label="Include complete items"
    selected={filter.includeComplete}
    select={() => setFilter({...filter, includeComplete: true})}
    unSelect={() => setFilter({...filter, includeComplete: false})}
  />
```

Now that the filters are updated by selecting/deselecting these checkboxes, the list that is rendered needs filtering according to the `filters` object. This can be done by using the array filter function like this:

javascript

```javascript
const filteredItems = todo.filter(item =>
  (filter.includeComplete || !item.complete) &&
  (!filter.overdueOnly || isOverdue(item)),
);
```

Now `filteredItems` should be used to create the list:

javascript

```javascript
{filteredItems.map(item => (
  <Item key={item.id} item={item} completeItem={() => completeItem(item.ic
```

Sample

This todo application now uses an API to get the list, adds and completes items, and has options to filter the list. A sample todo application using this code can be found here.

Conclusion

Continue on to the next part of this guide, React Todo List with Functions, where we will learn about using a mock API to retrieve and update the list and add functions to add new items.