React Hooks:
`useEffect`

useState covers the basics of adding a piece of state and providing a means of updating it.

We need a way to listen to state value changes and invoke code as a side-effect...

Enter `useEffect`

`useEffect` for side effects

We could do this if we had a method that behaved like this:

Setup a callback function that will run whenever a list of variables changes value.

Returning to our Counter

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

function Counter() {
  const [count, setCount] = useState(0)

  function handleClickButton() {
    setCount(count + 1)
  }

  return (
    <div>
      <p>
        Count: {count} <button onClick={handleClickButton}>Click Me</button>
      </p>
    </div>
  )
}
```

Let us setup some code that runs whenever the count changes!

```
function theCountChanged() {
  console.log(`Wow, the count changed and is now ${count}`)
}
```

- But we want this called when the count changes!

Enter useEffect!

- Accepts *TWO* arguments.
  - The first is the function to call
  - The second is an *array* of values to watch for changes

```
function theCountChanged() {
  console.log(`Wow, the count changed and is now ${count}`)
}

const listOfDataToWatchForChanges = [count]

useEffect(theCountChanged, listOfDataToWatchForChanges)
```

Simplify

- Using our rule:

Whenever we define a variable and use it once, we can replace the variable usage with the value

```
useEffect(
  function theCountChanged() {
    console.log(`Wow, the count changed and is now ${count}`)
  },
  [counter]
)
```

Simplify

- Since `theCountChanged` is now an anonymous inline function we can remove the name.

```
useEffect(
  function () {
    console.log(`Wow, the count changed and is now ${count}`)
  },
  [counter]
)
```

Notice the *effect* function runs the first time too!

- This is because `useEffect` will always run *at least* once!
- The first time, it acts like a function that runs when a component mounts.
- Future calls detect when state or props change. NOTE, the "watch" array could also contain props!

Use the `useEffect` to run code *once* at mount

If we need to do something just once when the component first mounts we can **provide an empty change list** and `useEffect` will only run once.

```
useEffect(function () {
  console.log(`This runs once when the component first mounts`)
}, [])
```

Since there is no watch array, there is nothing for `useEffect` to ever see.

NOTE: We can have multiple `useEffect`*!*

Using `useEffect` and `useState`

Build an app to use the `OneList` API

One List Refresher

- API Documentation

Base URL: `https://one-list-api.herokuapp.com`

Action	URL	Body
Get all items	`/items?access_token=illustriousvoyage`	None
Create item	`/items?access_token=illustriousvoyage`	`{ "item": { "text": "Learn about Regular Expressions" } }`
Get one item	`/items/42?access_token=illustriousvoyage`	None
Update an item	`/items/42?access_token=illustriousvoyage`	{ "item": { "text": "Learn about useEffect" } }`
Delete an item	`/items/42?access_token=illustriousvoyage`	None

As always, we start with static HTML and CSS

```
import React from 'react'
import logo from './images/sdg-logo.png'

export function App() {
  return (
    <div className="app">
      <header>
        <h1>One List</h1>
      </header>
      <main>
        <ul>
          <li>Do a thing</li>
          <li>Do something else</li>
          <li>Do a third thing</li>
          <li>Remind me about the important thing</li>
          <li>The important things are the important things</li>
        </ul>
        <form>
          <input type="text" placeholder="Whats up?" />
        </form>
      </main>
      <footer>
        <p>
          <img src={logo} height="42" alt="logo" />
        </p>
        <p>&copy; 2020 Suncoast Developers Guild</p>
      </footer>
    </div>
  )
}
```

```css
@import url('https://fonts.googleapis.com/css2?family=Comfortaa:wght@300&family=Neucha&display=swap');

:root {
  font: 24px / 1 sans-serif;
}

html {
  height: 100%;
}

body {
  margin: 0;
  padding: 0;
  color: #fff;
  font-family: 'Comfortaa';
  background-color: #309869;
}

.app {
  align-items: center;
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}

h1 {
  font-weight: 300;
  text-transform: uppercase;
}

main {
  flex: 1;
}

input {
  border: none;
  padding: 0.3em 0.2em 0;
  width: 100%;
  color: #074863;
  font-size: 1em;
  line-height: 1.5em;
  background-color: #fff;
  opacity: 0.5;
}

input:focus {
  outline: 0;
  background-color: #fff;
  opacity: 1;
}

a:link,
a:visited {
  line-height: 1rem;
  text-decoration: none;
  color: #fff;
  font-size: 0.6rem;
}

ul {
  padding: 0;
  list-style: none;
  cursor: pointer;

  li {
    display: flex;
    justify-content: space-between;

    margin: 0.7em;

    &.completed {
      text-decoration: line-through;
      opacity: 0.5;
    }
  }
}

ul,
input {
  font-family: 'Neucha', cursive;
}

ul li,
form {
  align-items: flex-start;
  display: flex;
}

header,
main,
footer {
  max-width: 30em;
}

footer {
  margin-bottom: 0.5em;
  font-size: 0.8em;
  text-align: center;

  img {
    vertical-align: middle;
    // animation: spin infinite 10s linear;
  }
}

@keyframes spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

ONE LIST

Do a thing

Do something else

Do a third thing

Remind me about the important thing

The important things are the important things

Whats up?

Step 2: Convert static JSX to JSX derived from state

```
const [todoItems, setTodoItems] = useState([
  { id: 1, text: 'Do a thing', complete: false },
  { id: 2, text: 'Do something else', complete: false },
  { id: 3, text: 'Do a third thing', complete: false },
  { id: 4, text: 'Remind me about the important thing', complete: false },
  {
    id: 5,
    text: 'The important things are the important things',
    complete: false,
  },
])
```

```
<ul>
  {todoItems.map(function (todoItem) {
    return <li key={todoItem.id}>{todoItem.text}</li>
  })}
</ul>
```

Step 3 - Try changing some data

Change text, add items, remove items, change completed state

Notice we need a little logic on our `li` *so we can change the class based on complete*

```
<ul>
  {todoItems.map(function (todoItem) {
    return (
      <li key={todoItem.id} className={todoItem.complete ? 'completed' : ''}>
        {todoItem.text}
      </li>
    )
  })}
</ul>
```

Step 4 - Actions

- Hook this up to the API and load data when the component first mounts
- Time for `useEffect`

```
useEffect(function () {
  console.log('this runs when the component first mounts')
}, [])
```

Update to load from the API

- This time, we will use `axios` to see how it improves on `fetch`

NOTE: We need to add the library!

Adding a library

- From the terminal and the same place we run `npm start` we can add a library to our project with:

```
npm install axios
```

This is very similar to our C# equivalent of `dotnet package add`

Update our `useEffect`

```
useEffect(async function () {
  const response = await axios.get(
    'https://one-list-api.herokuapp.com/items?access_token=cohort42'
  )

  if (response.status === 200) {
    console.log(response.data)
  }
}, [])
```

NICE!

Thanks axios

- No `headers`
- No `await response.json`
- Axios knows we are getting back JSON data and provides `response.data`

TypsScript issues

We cannot use an async function directly in `useEffect`

The solution is to define the async function *INSIDE* and then call it.

```
useEffect(function () {
  async function loadItems() {
    const response = await axios.get(
      'https://one-list-api.herokuapp.com/items?access_token=cohort42'
    )

    if (response.status === 200) {
      console.log(response.data)
    }
  }

  loadItems()
}, [])
```

Step 5 - Update state

- Make our default state an empty array again
- Inside the `useEffect`, call `setTodoItems`

```
const [todoItems, setTodoItems] = useState([])

useEffect(async function () {
  const response = await axios.get(
    'https://one-list-api.herokuapp.com/items?access_token=cohort42'
  )

  if (response.status === 200) {
    console.log(response.data)

    setTodoItems(response.data)
  }
}, [])
```

TypeScript ... again

Now our `todoItems` is of type `never[]` because our default state is `[]`.

We need to teach `TypeScript` the shape of our API.

```
type TodoItemType = {
  id: number
  text: string
  complete: boolean
  updated_at: Date
  created_at: Date
}

const [todoItems, setTodoItems] = useState<TodoItemType[]>([])
```

Beyond `useEffect` - more practice with React

Make the input field function correctly!

- Let the user type and track their item in state
- Add a state: `newTodoText`
- Add a `value` and `onChange` to the input

```
const [newTodoText, setNewTodoText] = useState('')

<input
  type="text"
  placeholder="Whats up?"
  value={newTodoText}
  onChange={function (event) {
    setNewTodoText(event.target.value)
  }}
/>
```

When the user presses enter!

- Since this `input` is inside a `form` we will get an `onSubmit` for the form!

```
function handleCreateNewTodoItem() {
  console.log(`Time to create a todo: ${newTodoText}`)
}
```

```
<form onSubmit={function(event) {
  // Don't do the normal form submit (which would cause the page to refresh)
  // since we are going to do our own thing
  event.preventDefault()

  handleCreateNewTodoItem()
}}>
```

Update `handleCreateNewTodoItem` to submit

```
async function handleCreateNewTodoItem() {
  const response = await axios.post(
    'https://one-list-api.herokuapp.com/items?access_token=cohort42',
    { item: { text: newTodoText } }
  )

  if (response.status === 201) {
    console.log(response.data)
  }
}
```

We get back an updated item, but how do we update the state?

- We have two choices:
  - Append the item
  - Reload the entire list and replace it!

We can look at both options

Append the item

Once we have the item we can build a new list of todos with that one appended.

```
const newTodo = response.data
// Create a new array by *spreading* the old list and putting our new item at the end. Use [newTodo, ...todoItems] to *prepend* the new item
const newTodoItems = [...todoItems, newTodo]

setTodoItems(newTodoItems)
```

Replace the list

Replacing the list can be straightforward when we are not sure how to manipulate the state.

```
const refreshTodoResponse = await axios.get(
  'https://one-list-api.herokuapp.com/items?access_token=cohort42'
)

setTodoItems(refreshTodoResponse.data)
```

One bug: Our new todo item input should clear

Add this to the end of `handleCreateNewTodoItem`

```
setNewTodoText('')
```

Click to mark an item as completed

Now we are giving the items some behavior, so it might be time to refactor those into a component!
We can reuse our TodoItemType for our `props`

```
type TodoItemProps = {
  todoItem: TodoItemType,
}

export function TodoItem(props: TodoItemProps) {
  return (
    <li className={props.todoItem.complete ? 'completed' : ''}>
      {props.todoItem.text}
    </li>
  )
}

<ul>
  {todoItems.map(function (todoItem) {
    return <TodoItem key={todoItem.id} todoItem={todoItem} />
  })}
</ul>
```

We can now add a click handler on the `li` inside the `TodoItem` component

```
export function TodoItem(props: TodoItemProps) {
  function toggleCompleteStatus() {
    console.log('Clicked!')
  }

  return (
    <li
      className={props.todoItem.complete ? 'completed' : ''}
      onClick={toggleCompleteStatus}
    >
      {props.todoItem.text}
    </li>
  )
}
```

What logic do we need in that code?

We want to send a `PUT` request to the API that looks like this if the item is *NOT* complete:

```
{
  "item": {
    "complete": true
  }
}
```

We want to send a `PUT` request to the API that looks like this if the item *is* complete:

```
{
  "item": {
    "complete": false
  }
}
```

Try an `if/else`

```
async function toggleCompleteStatus() {
  if (props.todoItem.complete) {
    const response = await axios.put(
      `https://one-list-api.herokuapp.com/items/${props.todoItem.id}?access_token=cohort42`,
      { item: { complete: false } }
    )

    if (response.status === 200) {
      console.log(response.data)
    }
  } else {
    const response = await axios.put(
      `https://one-list-api.herokuapp.com/items/${props.todoItem.id}?access_token=cohort42`,
      { item: { complete: true } }
    )

    if (response.status === 200) {
      console.log(response.data)
    }
  }
}
```

Look at the similarity in the branches!

The only difference is that if the `props.complete` is true, we send `false` -- and if it is `false` we send `true`

Simplify:

```
async function toggleCompleteStatus() {
  const response = await axios.put(
    `https://one-list-api.herokuapp.com/items/${props.todoItem.id}?access_token=cohort42`,
    { item: { complete: !props.todoItem.complete } }
  )

  if (response.status === 200) {
    console.log(response.data)
  }
}
```

Except we do not have a way to reload the list

STATE: DOWN, EVENTS: UP

Refactor the App a little

Extract the code used in the `useEffect` to make a function we can share.

```
function loadAllItems() {
  async function loadItems() {
    const response = await axios.get(
      'https://one-list-api.herokuapp.com/items?access_token=cohort42'
    )

    if (response.status === 200) {
      setTodoItems(response.data)
    }
  }
  loadItems()
}

useEffect(loadAllItems, [])
```

Now we have a function we can share with the `TodoItem` via props

Add it to the props supplied to the `TodoItem`

```
<TodoItem key={todoItem.id} todoItem={todoItem} reloadItems={loadAllItems} />
```

And add `reloadItems` to `TodoItemProps`

This type definition says "reloadItems is a function that takes no arguments and returns nothing"

```
type TodoItemProps = {
  todoItem: TodoItemType
  reloadItems: () => void
}
```

Use it after we finish with the API

```
async function toggleCompleteStatus() {
  await axios.put(
    `https://one-list-api.herokuapp.com/items/${props.todoItem.id}?access_token=cohort42`,
    { item: { complete: !props.todoItem.complete } }
  )

  props.reloadItems()
}
```

We can use this to cleanup `handleCreateNewTodoItem`

```
async function handleCreateNewTodoItem() {
  await axios.post(
    'https://one-list-api.herokuapp.com/items?access_token=cohort42',
    { item: { text: newTodoText } }
  )

  loadAllItems()

  setNewTodoText('')
}
```

props, props, props, props

We keep seeing `props.` in our `TodoItem` component

Refactor using destructuring!

```
export function TodoItem(props: TodoItemProps) {
  const { todoItem, reloadItems } = props

  async function toggleCompleteStatus() {
    await axios.put(
      `https://one-list-api.herokuapp.com/items/${todoItem.id}?access_token=cohort42`,
      { item: { complete: !todoItem.complete } }
    )

    reloadItems()
  }

  return (
    <li
      className={todoItem.complete ? 'completed' : ''}
      onClick={toggleCompleteStatus}
    >
      {todoItem.text}
    </li>
  )
}
```

We can improve this code

- We can also *destructure* right inside the function definition.
- So:

```
export function TodoItem(props) {
  const { id, text, complete, reloadItems } = props
```

can become:

```
export function TodoItem({ todoItem, reloadItems }: TodoItemProps) {
```

Furthermore, this looks like our function takes *arguments* -- but it is really *destructuring* props!

```
export function TodoItem({ todoItem, reloadItems }: TodoItemProps) {
  async function toggleCompleteStatus() {
    await axios.put(
      `https://one-list-api.herokuapp.com/items/${todoItem.id}?access_token=cohort42`,
      { item: { complete: !todoItem.complete } }
    )

    reloadItems()
  }

  return (
    <li
      className={todoItem.complete ? 'completed' : ''}
      onClick={toggleCompleteStatus}
    >
      {todoItem.text}
    </li>
  )
}
```

Destructure the destructured

```
export function TodoItem({
  todoItem: { id, text, complete },
  reloadItems,
}: TodoItemProps) {
  async function toggleCompleteStatus() {
    await axios.put(
      `https://one-list-api.herokuapp.com/items/${id}?access_token=cohort42`,
      { item: { complete: !complete } }
    )

    reloadItems()
  }

  return (
    <li className={complete ? 'completed' : ''} onClick={toggleCompleteStatus}>
      {text}
    </li>
  )
}
```

Extract it to a file in the `components/` directory