Part 2

Communicating With Server

Topics

- How to render a data collection (array)
- How to submit user data to React via HTML forms
- How to fetch remote backend server with JS
- How to add CSS styles to React

a. Rendering a collection, modules

- It's useful to know funcitonal methods like `find`, `filter`, `reduce`, and `map`

Reduce

- Take an array and return just a number
- Use case: e.g. add all numbers in an array

```
const orders = [
  {id: 1, total: 11},
  {id: 2, total: 23},
]
// 0 is the initial value of acc (accumulator
let total = orders.reduce(
  (acc, currentOrder) => acc + currentOrder.total, 0
);
```

Rendering collections

- We can generate React element from an array using the **map** function
- Each element generated must have a **unique key value**

```
// notes is an array of objects
<ul>
  {notes.map(note =>
    <li key={note.id}>
      {note.content}
    </li>
  )}
</ul>
```

- The key attribute is needed so React can **rerender more efficiently** (only rerender the changed elements)

Using indexes as keys is **not recommended** for elements can change in the array

Refactoring modules

- common practice is to declare each component in their own file as an ES6-module
- the file will be named after the component and usually placed in the `src/component` directory
- We can export using **named export** and **default export**

b. Forms

```
// event handler for a form
const addNote = (event) => {
  event.preventDefault()
  console.log("button clicked", event.target)
}

<form onSubmit={addNote}>
  <button type="submit">submit</button>
</form>
```

- **event** is the event that triggers the call to event handler

- **event.preventDefault()** prevents the default action of submitting a form, because it will **cause the page to reload**
- **event.target** is the form itself

Access data contained in the form's input element

- We can use **controlled components (React components that controls the value of input elements)**

```
<input value={state} onChange={handleOnChange} />
```

Exercise:

- We can use string as value of the key property
- remember to prevent the default action of submitting HTML forms
- Use **template string** to form strings that contains **values from variables** `` `${newName} is already added to phonebook` ``
- Refactor code by extracting suitable parts **into new components.** Maintain the application's **state and all event handlers** in the **root component.**

c. Getting data from server

- We can use **JSON Server** to act as our server during development
  - Create a fake REST API quickly without coding
- `npx json-server --port 3001 --watch db.json`
- json-server enables the use of **server-side** functionality in development phase without the need to program it

The browser as a runtime environment

- JavaScript runtime environment follows the **asynchronouse model**, so the browser will not "freeze: when fetching data from a server

What the heck is the event loop anyway?

- JavaScript is **single threaded, it can only do one thing at a time**
- When a line of code is blocking (takes a long time to run such as requesting data), if it's executed synchronously, we'll have to wait for it to finish
- This is a problem in browser, because the browser will be stuck waiting
- The solution is **asynchronous callbacks**
- **Concurrency & the Event Loop**

Axios

- We will use **axios** for communicating between the browser and the server
- More pleasant to use than **fetch**
- `axios.get` returns a **promise**
- **promise:** an object that represents an asynchronous operation, it can have three distinct states
  - **pending:** the final value is not available yet
  - **fulfilled:** the operation has completed and the final value is available. This state is also called **resolved (successful)**
  - **rejected:** an error prevented the final value from being determined **(failed)**
- We can use `then` to access the result of the operation represented by the promise

```
axios
  .get('http://localhost:3001/notes')
  .then(response => {
    const notes = response.data
    console.log(notes)
  })
```

Effect-hooks

- **state-hook** provides state to React components defined as functions
- **effect-hook** lets you perform side effects in function components. (Data fetching, etc.)

```
useEffect(() => {
    console.log('effect')
    axios
      .get('http://localhost:3001/notes')
      .then(response => {
        console.log('promise fulfilled')
        setNotes(response.data)
      })
  }, [])
  console.log('render', notes.length, 'notes')

//render 0 notes
//effect
//promise fulfilled
//render 3 notes
```

- **useEffect** takes 2 parameters, the first is a **function (the effect),** the second parameter is to specify **how often the effect is run** (default to run after every completed render)
- setting the second parameter to `[]` will make the effect **only run with the first render** of the component

d. Altering data in server

Sending Data to the Server

- we send objects to server using the axios `post` method.

```
// noteObject is the payload
axios
  .post('http://localhost:3001/notes', noteObject)
  .then(res => {
    // update states
```

```
      setNotes(notes.concat(res.data));
      setNewNotes("");
    })
}
```

- To update individual data in server, we can either
  - **replace the entire data** with an **HTTP PUT** request
  - **only change specific data** with an **HTTP PATCH** request

Changing the importance of notes

- spread operator trick

```
const changedNote = { ...note, important: !note.important }
```

- map trick

```
notes.map(note => note.id !== id ? note : response.data)
```

- the `then` method of a promise also **returns a promise**

Cleaner syntax for defining object literals

- when the names of the keys and the assigned variables are the same, we can shorten the object definition

```
// from
{
  getAll: getAll,
  create: create,
  update: update,
}

// to
{
  getAll,
```

```
  create,
  update
}
```

Promises and errors

- the application should be able to handle different types of error situations **gracefully.**
- The errors (rejection) of a promise is handled using the `catch` method

```
axios
  .get('http://example.com/probably_will_fail')
  .then(response => {
    console.log('success!')
  })
  .catch(error => {
    console.log('fail')
  })
```

- **catch** method is used to define a handler function at the end of the promise chain, which is called **once any promise in the chain throws an error and the promise becomes rejected**

Exercise

- Send delete request with axios `delete`
- ask for user boolean input with **window.confirm**

e. Adding Styles to React app

- In React we have to use **className** attribute to add class to a tag

Improved error message

- we can implement a component to display errors & notifications
- `return null` inside a component to **render nothing** to the screen

- use `setTimeout` to get rid of the notification after a few seconds

Inline styles

- We can add **inline styles** in React through a **style** attribute
- The **hyphenated** CSS properties are written in **camelCase**

```
const Footer = () => {
  const footerStyle = {
    color: 'green',
    fontStyle: 'italic',
    fontSize: 16
  }
  return (
    <div style={footerStyle}>
      <br />
      <em>Note app, Department of Computer Science, University of Helsinki 2020</em>
    </div>
  )
}
```

- Inline styles make using **pseudo-classes** (::hover) more difficult
- React's philosophy is to have individual components as independent and resuable as possible. So inline style is good to use to do that.

```
takeaway:
1. handleFilter(index) vs. () => handleFilter(index)
2. let newShow = show vs let newShow = [...show] (show is an array, the first newShow points
to the same array while the second one is a deep copy)
```