STATE

Lect5

1

Objectives

- Review how to use state using Hooks
- Demonstrate how to perform initialisation
- Introduce default props
- Passing information between components

2

Hooks

- Special functions linked with React's state and lifecycle features.
- Replaces the use for **class components** simpler development.
- Handle side effects like data fetching.
- Share logic across components without altering their structure (via custom hooks).

3

JavaScript Destructuring

- The destructuring assignment is a unique syntax that helps "to unpack" objects or arrays into a group of variables.
- Arrays

```
let arr = [ 1,2, 3, 4];

let a = arr[0];
let b = arr[1];
let c = arr[2];
let d = arr[3];
```

- `// destructuring arrays`
- `let [a,b,c,d] = arr;`

4

- Object destructuring
- let student = {

```
    name: "Fred",
    CAO: "L0002344",
    age: 23
}
let name = student.name;
let CAO = student.CAO;
let age = student.age;
```

```
// destructuring objects
let {name, CAO, age } = student;
console.log(name);
```

5

```
function render(props) {
    var name = props.name;
    var age = props.age;
}

// destructuring functions
function render({name, age}) { }
```

- Destructuring makes the assignment of variables even easier

6

- ReactJS is a front-end JavaScript library for building user interfaces.
- Everything in ReactJS is a component and to pass in data to these components we used props.
- All our components examples have only use static data.
- React State an object allows us to hold data that may change over the lifetime of the component.

7

- Components are the core building blocks of React applications.
- React Components are built in one of two ways **functions** or **classes**.
- Function components are simple. They accept props and can return JSX
- Using state hooks simplifies the handling of data when using component functions.
- Using hooks components can manage the state without writing a class.

8

Props Vs State

- Props are immutable i.e. once set the props cannot be changed.
- State is an observable object that is to be used to hold data that may change over.
- States can be used in Class Components and Functional components with the use of React Hooks (`useState` and other methods).
- While Props are set by the parent component, State is generally updated by event handlers.

9

Setting useState

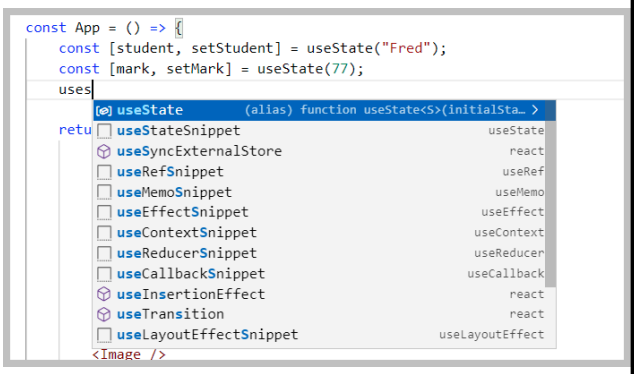
- The most popular hook is `useState`
- First import `useState` from React
- `import { useState} from 'react';`
- The state isn't updated explicitly.
- A state is a variable which exists inside a component, that cannot be accessed and modified outside the component and can only be used inside the component.


```
const App = () => {
    console.log("React test msg");
    let [student, setStudent] = useState("Fred");
    const [mark, setMark] = useState(77);
    //const [bioList, setBio]= useState({Athletes})

    student = "Mary";

    return (<>
        <Heading />
        <input value={student}
            onChange={(evt) => setStudent(evt.target.value)}
        />
```

Won't work for updates

10

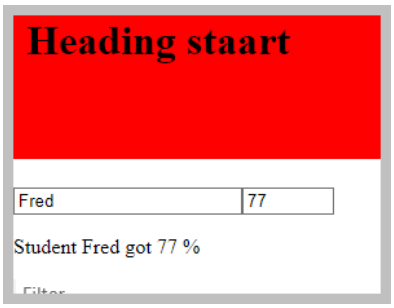
- React provides its own `setState()` hook to assign data values.
- The App component is a functional React component, a function that returns JSX markup.
- It's also a stateful component using the useState() Hook.
- The useState() hook returns an array.



11

- Setting State

```
App.jsx > App
const App = () => {
    const [student, setStudent] = useState("Fred");
    const [mark, setMark] = useState(77);

    return (<>
        <Heading />
        <input value={student} />
        <input type="number" name="mark" min="0"
               max="100" step="1" value={mark}
               />
        <p>Student {student} got {mark} %</p>
```



12

Updating state values

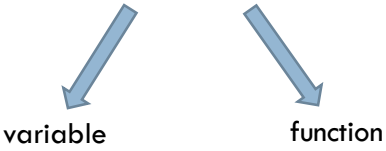
- Users may now change data state.
- **`useState`** function returns a variable with the current state value and another function to update that value.
- The initial state values used may change in response to some event user events.

```
const App = () => {
    const [student, setStudent] = useState("Fred");
    const [mark, setMark] = useState(77);


    return (<>
        <Heading />
        <input value={student}
            onChange={(evt) => setStudent(evt.target.value)}
        />
        <input type="number" name="mark" min="0"
                max="100" step="1" value={mark}
                onChange={(evt) => setMark(evt.target.value)}
                />
        <p>Student {student} got {mark} %</p>
```

13

- The useState() Hook returns an array.
- The first item is the state value.
- The second is the function used to update the value

```
const [student, setStudent] = useState("Fred");
const [mark, setMark] = useState(77);
```



14

- You can have as many pieces of state in your component as you need.
- The best practice is to have one call `to useState()` per state value.

```
const [mark, setMark] = useState(77);
```

15

- Whenever the user changes the text in the `<input>` field, the `onChange` event is triggered.
- The handler for this event calls `setName()`, passing it `e.target.value` as an argument.
- The argument passed to `setName()` is the new state value of name.

```
const [student, setStudent] = useState("Fred");
const [mark, setMark] = useState(77);


return (<>
    <Heading />
    {/* <AthletesList bioList={bioList} /> */}
    <input value={student}
        onChange={(evt) => setStudent(evt.target.value)}
    />
    <input type="number" name="mark" min="0"
            max="100" step="1" value={mark}
            onChange={(evt) => setMark(evt.target.value)}
            />
    <p>Student {student} got {mark} %</p>
```

16

```
src > components > EmailInput.jsx > EmailInput
1   function EmailInput() {
2
3       let errorMessage = 'no issues';
4
5       function evaluateEmail(event) {
6           console.log('Hello');
7           const enteredEmail = event.target.value;
8
9           if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
10              errorMessage = 'The entered email address is invalid.';
11          } else {
12              errorMessage = '';
13          }
14          console.log(`Error message ${errorMessage}`);
15      };
16
17      return (
18          <div>
19              <input placeholder="Your email" type="email" onBlur={evaluateEmail} />
20              <p>{errorMessage}</p>
21          </div>
22      );
23  };
```

17

Exercise: update component with message

- When a user enters the value and clicks outside of the email input field, the `evaluateEmail` method called with `evt.target.value` sets the entered value to the `errorMsg` variable.
- The `evt.target` returns the element that triggered the event, which is the input field for the error message.
- The evt.target.value thus gives us the value, which is entered in nput box

18

- The **`useState`** is probably the most used and useful hook, but react has over twenty others.
- Another hook use with rendering components is **`useEffect`**. Used for tasks such as fetching data or cleaning up code.
- `useEffect(didUpdate)` expects a function as an argument
- Mutations, timers, logging, and other side effects are not allowed inside the main body of a function component (referred to as React's render phase). Doing so will lead to confusing bugs and inconsistencies in the UI.
- Instead, use `useEffect`.
  - The function passed to `useEffect` will run after the render is committed to the screen.

19

`useEffect` on loading example

- Waits for three second before rendering the student data.

src > App.jsx > ...

```
1  import { useState, useEffect } from "react";
2  import Heading from "./components/Heading";
3  import Image from "./components/Image";
4  import EmailInput from "./components/EmailInput"
5
6  const App = () => {
7      const [student, setStudent] = useState("Loading ...");
8      const [mark, setMark] = useState(77);
9
10
11     function fetchStudent() {
12         return new Promise((resolve) => {
13           setTimeout(() => {
14             resolve({ name: "Fred", mark: 33 });
15           }, 3000);
16         });
17       }
18
19     useEffect(() => {
20         fetchStudent().then((student) => {
21             setStudent(student.name);
22             setMark(student.mark);
23         });
24     });
25
26     return (<>
```

20

- Sometimes navigation of an app will cause components to unmount before responses to their API requests arrive.
- This will cause an error because the component will attempt to update the state values of a component that has been removed.
- The `useEffect()` Hook has a mechanism to clean up things such as pending API requests when the component is removed.

21

```
function Student() {
    const [name, setName] = useState("loading...");
    const [mark, setMark] = useState("loading...");

    Promise.config({ cancellation: true });

    function fetchStudent() {···
    }

    useEffect(() => {
        const promise = fetchStudent().then((user) => {
            setId(user.id);
            setName(user.name);
        });

        return () => {
            promise.cancel();
        };
    });

    return (
        <>
            <p>Name: {name}</p>
            <p>Mark: {mark}</p>
        </>
    );
}
```

```
function fetchStudent() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve({ name: "fred", mark: 40 });
        }, 3000);
    });
}
```

22

- The **useEffect** creates a promise by calling the `fetchStudent`.
- It also returns a function, which React runs when the component is removed.
- Here the promise that is created by calling `fetchUser()` is cancelled by calling `promise.cancel()`.
- This prevents the component from trying to update its state after it has been removed.
- Example used the bluebird library that supports promises

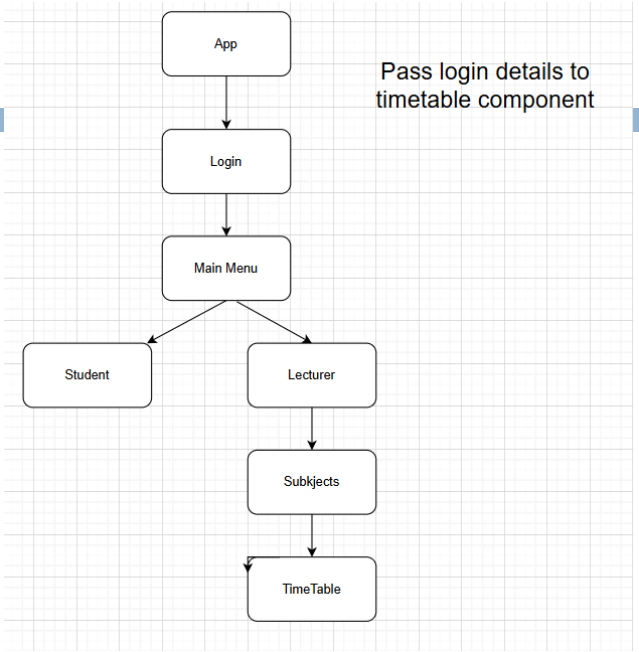
23

- If you want to run **useEffect** only on the first render of a component (also called only on mount), then you can pass in a second argument to **useEffect**.
- The second argument is called dependency array.
- If the dependency array is empty, the side-effect function used in React's useEffect Hook has no dependencies, meaning it runs only the first time a component renders.

```
const AthleteList = () => {
  // runs after the component first renders
  useEffect(() => {
    console.log("useEffect called - loading athlete data...");

    setTimeout(() => {
      setAthletes(athletesData);
    }, 1000); // delay just to simulate loading
  }, []); // empty dependency array “run once” on mount
```

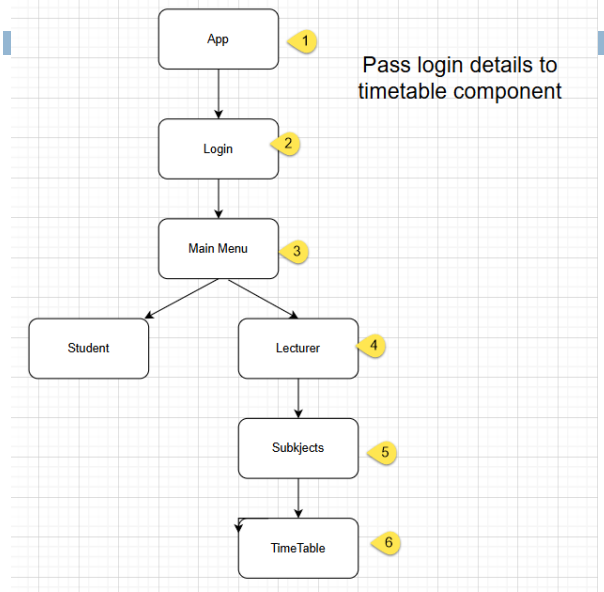
24

- managing state across nested components can become very difficult.
- The **useContext** hook offers a simple and efficient solution to share state between components without the need for prop drilling



25

prop drilling

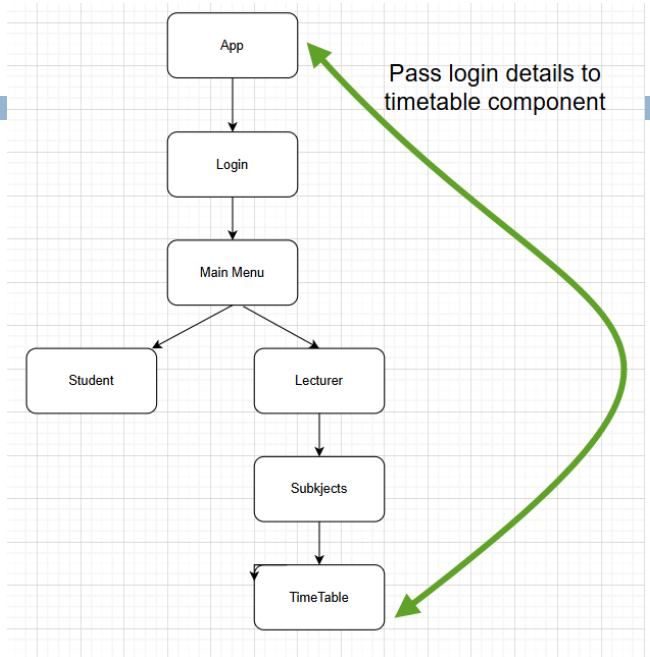


Using props this is prop drilling

26

- Cluttering every component.
  - Login to Timetable
- Rather than passing down the props through each component, React Context provides a way bypassing components.
  - Due to the fact that a top-level component supplies the context with the information, a component can consume it whenever it needs to.

27



28

Create the context

hook-demo > src > components > DemoContext.jsx > default

```
1   import React from 'react';
2
3   // Create a Context with a default value of null
4   const DemoContext = React.createContext({});
5
6   export default DemoContext;
```

29

React Context

- The top-level react component (Login) provides the context
- React component (TimeTable) as a child components consumes the context.
- Components Menu, Lecturer and Subjects don't use this info, so they don't consume the context.

30

- The App component doesn't need to pass down anything via components Lecturer in the props so that it reaches component TimeTable

```
import LoginExample from './components/LoginExample';
import DemoContext from './components/DemoContext';
import { Lecturer } from './components/Lecturer';

function App() {
  return (
    <DemoContext.Provider value={{person: { fname: 'Gerard', lname: 'mCCloskey'}}}>
    <div className="App">
      <LoginExample />
      <Lecturer />
    </div>
    </DemoContext.Provider>
  );
}

export default App;
```

31

```
<DemoContext.Consumer>
  {(value) => (
    <div>
      {value.company.fname} TimeTable
    </div>
  )}
</DemoContext.Consumer>
```

32

```jsx
import React, { useContext } from 'react';
import DemoContext from './DemoContext';

export const TimeTable = () => {
  const value = useContext(DemoContext);
  return <div>{value.person.lname}, {value.person.fname} TimeTable</div>;
};

export default TimeTable;
```

The Consumer must use a function as its child (render prop pattern)

33

- When should you use React Context
  - Pass data (like user info) **without prop drilling**
  - Keep global state available to multiple components
  - Combine with hooks

34

Summary

- `useState` is a JavaScript object used by React to represent information about a component.
- Hooks used for functional components,
- React Hooks available since version 16.
- Key hooks now **useState, useEffect** and **useContext.**

35