Laboratorul 08: React Hooks

1. React Hooks

Hooks [https://reactjs.org/docs/hooks-overview.html] sunt o noua adaugare in **React 16.8**. Va permit sa utilizati functiile de stare si alte functii React, fara a scrie o clasa. Avantajul este mai putin cod scris pentru a obtine aceeasi functionalitate.

1.1 useState()

useState [https://reactjs.org/docs/hooks-state.html] este un hook de React care va permite sa interactionati cu starea unei componente. Acesta va intoarce un vector format din 2 elemente:

- Variabila de stare
- Functia care va modifica variabila de stare

Functia **useState** primeste ca parametru valoarea initiala a starii

```
const [myState, myFunctionToChangeState] = useState(initialValue);
```

Atentie, valoarea initiala trebuie sa reflecte tipul de stare pe care vreti sa il retineti (string, numar, obiect, etc…)

O componenta poate apela **useState** de mai multe ori, pentru a retine stari diferite

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
```

Acest exemplu reda un contor. Cand faceti clic pe buton, creste valoarea:

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Observati cum valoarea initiala a starii este 0

1.2 useEffect()

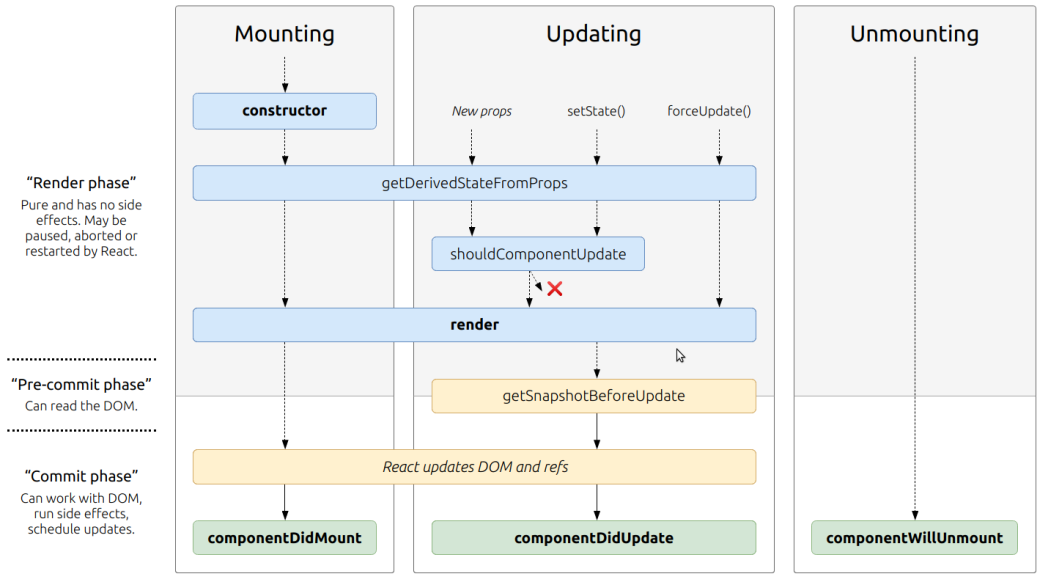
useEffect [https://reactjs.org/docs/hooks-effect.html] este al doilea hook important din React si va permite propagarea **efectelor colaterale** din React - sau pe scurt, **efecte** - (de exemplu, modificarea DOM-ului, afisarea unor alerte, preluarea unor date, etc…).

Orice actiune care nu se poate efectua in timpul randarii (render), pentru ca poate modifica structura componentei, este considerat efect.

Pe vremuri, cand componentele functionale nu erau atat de folosite si inca se foloseau foarte mult componentele clasa, aceste efecte erau tratate in **ciclurile de viata** ale componentelor. Aceste cicluri de viata nu sunt nimic mai mult decat metode ale claselor de React. Amintim cateva:

- **componentDidMount** - metoda care se executa dupa redarea initiala a componentei

- **componentDidUpdate** - metoda care se executa dupa ce state sau props au fost modificate
- **componentWillUnmount** - metoda care se executa inainte de stergerea componentei - adica este scoasa din DOM



In principal, **efectele** declarate folosind **useEffect** se realizeaza, prin analogie, in **componentDidMount**, **componentDidUpdate** si, daca este declarat un comportament specific si in **componentWillUnmount**.

Mai jos aveti un exemplu de efect care schimba titlul paginii:

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

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

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Prin analogie, asa s-ar fi scris folosind clase:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

Observati cum logica este duplicata in **componentDidMount** si **componentDidUpdate**. useEffect elimina aceasta duplicare

Exista cazuri cand doriti sa anulati un efect in momentul in care nu mai aveti nevoie de componenta sau cand proprietatile componentei se schimba. Acest proces se numeste **curatare**. In Reactul clasic, acest lucru era scris in metoda **componentWillUnmount**, sau in metoda **componentDidUpdate**. Pentru a face acest lucru in useEffect, este nevoie sa **returnati o functie**.

Asadar, ce se intampla in corpul functiei useEffect se intampla **de fiecare data** cand componenta se randeaza (render) si ce se returneaza se executa **de fiecare data** cand componenta a terminat de executat render.

De exemplu, sa presupunem ca avem un API de Chat care are doua metode: subscribe si unsubscribe. Aceste doua metode au doi parametri, un id si o functie de callback care seteaza statusul pe online sau offline. Ne dorim sa dam subscribe la inceputul componentei si sa dam unsubscribe, atunci cand parasim componenenta.

Folosind useEffect, acest lucru se poate realiza foarte usor:

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Pe de alta parte, folosind clasa, codul ar fi aratat astfel:

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

Observati cum ce este returnat in **useEffect** este scris in **componentWillUnmount** si ce este scris in corpul **useEffect** este scris in **componentDidMount**, asa cum am explicat anterior.

Codul din ultima coza este un cod ce are bug. Ganditi-va la urmatorul caz: cand **props** este modificat (adica atunci cand parintele ii trimite alta valoare pentru props), componenta **NU** va efectua unsubscribe, chiar daca id-ul s-a modificat. Acest lucru trebuie tratat tot in **componentDidMount**. useEffect elimina aceasta problema, executand codul la fiecare render.

Asa ar fi trebuit sa arate codul corect, folosind clase:

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentDidUpdate(prevProps) {
  // Unsubscribe from the previous friend.id
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // Subscribe to the next friend.id
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
```

1.3 useReducer()

useReducer [https://reactjs.org/docs/hooks-reference.html#usereducer], este o alternativa pentru useState(). Accepta un reducer de tip

```
(state, action) => newState
```

si returneaza state-ul curent si functia dispatch. Dispatch, care primeste o actiune ca argument, declanseaza actualizarea state-ului in functie de actiunea folosita.

Un reducer este o functie care primeste state-ul curent si o actiune si returneaza state-ul modificat, actiunea fiind modalitatea prin care se precizeaza cum se va modifica state-ul. O actiune este un obiect care trebuie obligatoriu sa aiba un tip (**type**).

Este recomandat sa fie folosit daca logica state-ului este mai complexa.

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

Exercitii

Vom extinde aplicatia creata la laboratorul precedent.

1. Folosind componentele App si Count, trimiteti valoarea counter-ului si functiile (incrementare, decrementare, reset) folosind props. Cand counter-ul ajunge 0, afisati o alerta.

Componenta **Count** si **App** trebuie sa fie componente functionale. Pentru afisarea alertei va trebui sa folositi **useEffect()**. Valoarea lui count va fi definita drept state in App, la fel si functiile de incrementare, decrementare si reset.

2. Modificati fisierele de css din extensia **.css** in **.scss**, daca nu ati folosit Sass in laboratorul trecut.

De asemenea, React permite modularizarea fisierelor SCSS prin denumirea acestora in **numeFisier.module.scss**. In acest caz, importarea si referentierea claselor css, se face in felul urmator:

```
numeFisier.module.scss

.container {
  display: flex;
  justify-content: space-between;
  align-items: center;
  flex-direction: row;
  padding: 1rem 0;
  font-size: 1em;
  font-weight: 700;
  [...]
}

import styles from './numeFisier.module.scss';

<div className={styles.container}>
[...]
</div>
```

In cazul in care doriti import-ul standard, fara module, acesta va arata in felul urmator:

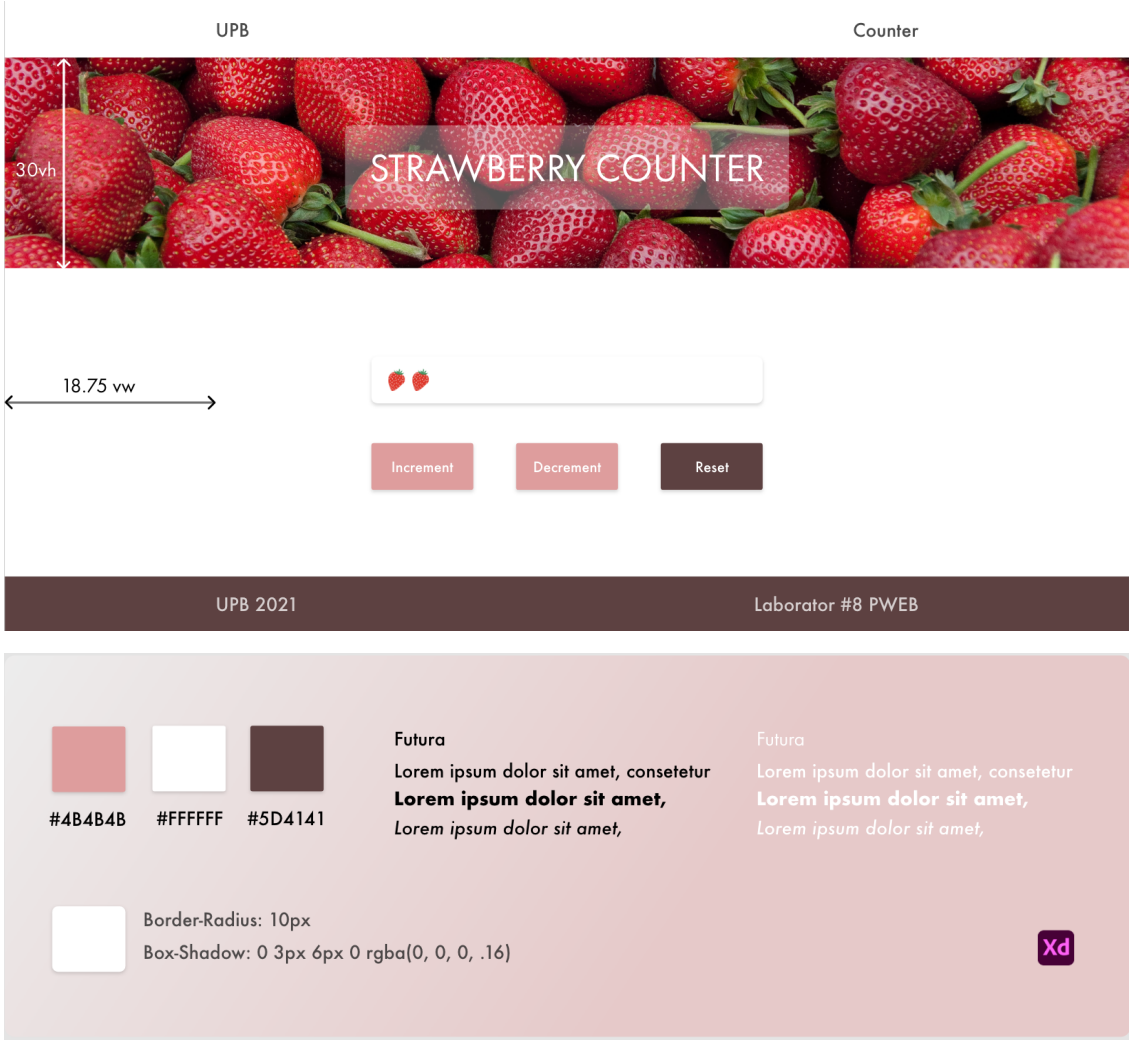
numeFisier.scss

```
.container {
  display: flex;
  justify-content: space-between;
  align-items: center;
  flex-direction: row;
  padding: 1rem 0;
  font-size: 1em;
  font-weight: 700;
  [...]
}

import './numeFisier.scss';

<div className="container">
[...]
</div>
```

3. Modificati counter-ul astfel incat sa nu mai afiseze un numar, ci sa afiseze o imagine de **counter** ori. Stilizarea interfetei poate urma design-ul de mai jos, folosind culorile alese si fontul, dar tematica poate fi schimbata dupa propriile idei (Puteti sa faceti o tema inspirata dintr-un film sau natura).



4. Componenta Layout va trebui sa alcatuiasca toata interfata utilizator, folosind **Header**, **props.children** si **Footer**.

pw/laboratoare/08.txt · Last modified: 2021/04/25 22:35 by alexandru.hogea