Лекіція 2

1) Рендер компонентів
2) Стейт
3) Події

Multiple-page application

Це підхід який включає декілька окремих HTML-сторінок.

- Архітектура клієнт-сервер
- Вся логіка живе на сервері
- На кожен запит сервер надсилає готовий HTML-документ
- Перезавантаження сторінки при кожному запиті
- Погана інтерактивність
- Відмінне SEO

Single-page application

Сучасний підхід – сайт, на якому користувач ніколи не переходить на інші HTML-сторінки. Інтерфейс, замість запиту HTML-документів з сервера, перемальовується на клієнті, на одній і тій самій сторінці, без перезавантаження.

- **Архітектура клієнт-сервер**
- **При завантаженні сайту сервер завжди віддає стартову HTML-сторінку index.html**
- **Кожен наступний запит на сервер отримує лише дані у JSON-форматі**
- **Оновлення інтерфейсу відбувається динамічно на клієнті**
- **Завантаження першої сторінки може бути досить повільним (лікується)**
- **Логіка, не пов'язана із безпекою, живе на клієнті**
- **Слабке SEO (лікується)**
- **Складність коду та його підтримки масштабується з кількістю функціоналу застосунку**

Рендер компонентів
Virtual DOM

- Що таке DOM (Document Object Model)

Об’єктне представлення вмісту HTML документу та інтерфейс для управління цим об’єктом

- DOM API (Application programming interface)

Якщо ми хочемо змінити вміст першого елементу “list”, будемо використовувати DOM API

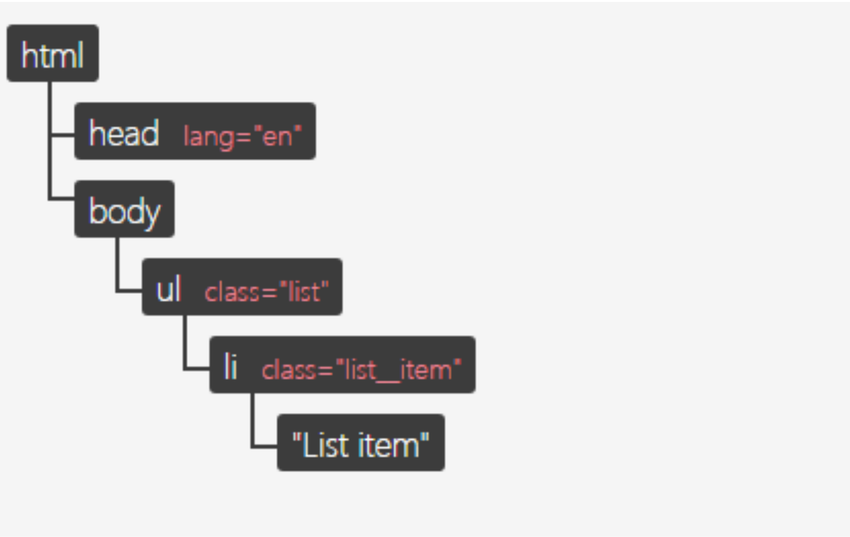
```
const listItemOne = document.getElementsByClassName("list__item")[0];
listItemOne.textContent = "List item one";

const list = document.getElementsByClassName("list")[0];
const listItemTwo = document.createElement("li");
listItemTwo.classList.add("list__item");
listItemTwo.textContent = "List item two";
list.appendChild(listItemTwo);
```

Простий HTML документ

```
<!doctype html>
<html lang="en">
 <head></head>
 <body>
    <ul class="list">
        <li class="list__item">List item</li>
    </ul>
  </body>
</html>
```

DOM дерево



Рендер компонентів

Virtual DOM

З кожною зміною DOM браузер виконує кілька трудомістких операцій. Часті операції оновлення такого дерева негативно впливають на продуктивність та реакцію інтерфейсу. Тому він повільний, та оновлювати його необхідно ефективно.

Virtual DOM можна розглядати, як копію звичайного DOM, яку можна часто оновлювати. Після внесення усіх змін до Virtual DOM, ми бачимо, які зміни слід внести у DOM та здійснити їх більш направлено та ефективно.

Об’єкт Virtual DOM

```
const vdom = {
    tagName: "html",
    children: [
        { tagName: "head" },
        {
            tagName: "body",
            children: [
                {
                    tagName: "ul",
                    attributes: { "class": "list" },
                    children: [
                        {
                            tagName: "li",
                            attributes: { "class": "list__item" },
                            textContent: "List item"
                        } // end li
                    ]
                } // end ul
            ]
        } // end body
    ]
} // end html
```

Рендер компонентів
Virtual DOM

- Як працює VDOM під капотом.

У React кожен елемент інтерфейсу – це компонент (кастомний або вбудований), який залежить від пропсів або стану, і представлений вузлами віртуального DOM-дерева. Взаємодія користувача з інтерфейсом змінює стан застосунку.

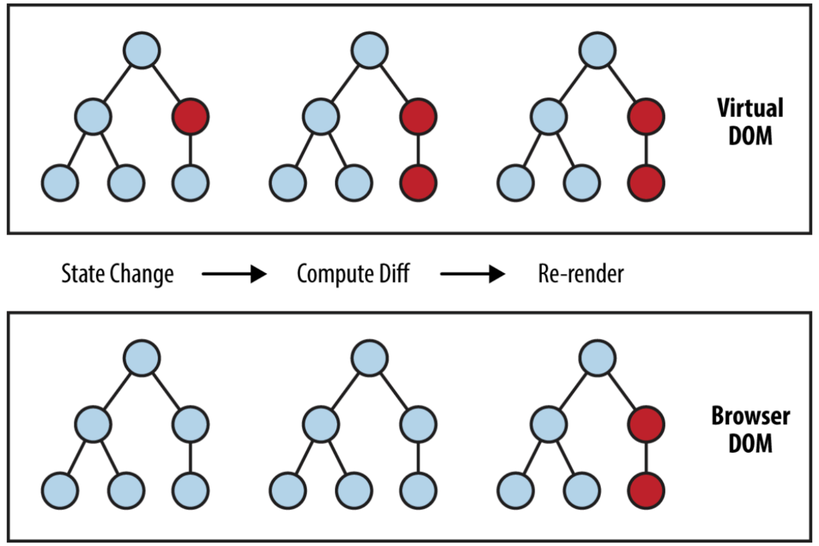
Порівняння та зміни

```
const diffs = [
    {
        newNode: { /* нова версія першого елементу списку */ },
        oldNode: { /* початкова версія першого елементу списку */ },
        index: /* індекс елементу у батьківському списку дочірніх вузлів */
    },
    {
        newNode: { /* другий елемент списку*/ },
        index: { /* */ }
    }
]
```

Копія VDOM

```
const copy = {
    tagName: "ul",
    attributes: { "class": "list" },
    children: [
        {
            tagName: "li",
            attributes: { "class": "list__item" },
            textContent: "List item one"
        },
        {
            tagName: "li",
            attributes: { "class": "list__item" },
            textContent: "List item two"
        }
    ]
};
```

Після того як зібрали всі відмінності, робиться заміна лише необхідних елементів у DOM дереві.



Рендер компонентів

Virtual DOM

- Virtual DOM в React

Рендер list з використанням React

```
import React from 'react';
import ReactDOM from 'react-dom';

const list = React.createElement("ul", { className: "list" },
    React.createElement("li", { className: "list__item" }, "List item")
);

ReactDOM.render(list, document.body);
```

Підсумок

Отже, Virtual DOM — інструмент, що дозволяє взаємодіяти з елементами DOM простіше та ефективніше. Virtual DOM представлено у вигляді об'єкта Javascript, який ми можемо змінювати так часто, як нам потрібно. Зміни, здійснені над об'єктом накопичуються, а фактичний DOM оновлюється направлено та рідше.

Оновлення list за допомогою React

```
const newList = React.createElement("ul", { className: "list" },
    React.createElement("li", { className: "list__item" }, "List item one"),
    React.createElement("li", { className: "list__item" }, "List item two");
);

setTimeout(() => ReactDOM.render(newList, document.body), 5000);
```

State

State – це концепція в React в якій міститься інформація, що впливає на результат рендерингу

Стан (state) є одним з основних понять в React і використовується для збереження та управління даними в компонентах React. Стан представляє собою об'єкт, який містить дані, які можуть змінюватись протягом життєвого циклу компонента. Коли стан змінюється, React автоматично оновлює відображення компонента, що призводить до оновлення користувацького інтерфейсу.

Основні принципи роботи зі станом в React:

Ініціалізація стану: Стан можна ініціалізувати в конструкторі класового компонента або за допомогою хука `useState` в функціональному компоненті. Наприклад:

```
// Класовий компонент
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      counter: 0,
    };
  }
  // ...
}
```

```
// Функціональний компонент
function MyComponent() {
  const [counter, setCounter] = useState(0);
  // ...
}
```

State

Отримання значення стану: Для отримання значення стану використовуйте `this.state` у класових компонентах або просту змінну, отриману в результаті виклику хука `useState`, у функціональних компонентах. Наприклад:

```
// Класовий компонент
render() {
  return <div>{this.state.counter}</div>;
}

// Функціональний компонент
function MyComponent() {
  const [counter, setCounter] = useState(0);
  return <div>{counter}</div>;
}
```

State

Зміна стану: Для зміни стану використовуйте метод `setState` у класових компонентах або функцію, отриману в результаті виклику хука `useState`, у функціональних компонентах. Важливо пам'ятати, що стан у React є незмінним, тому ви не повинні змінювати його напряму. Замість цього, ви повинні створювати новий об'єкт, який містить оновлені дані стану. Наприклад:

```
// Класовий компонент
handleClick() {
  this.setState({ counter: this.state.counter + 1 });
}

// Функціональний комплнент
handleClick() {
  setCounter(counter + 1);
}
```

Нове значення стану можна обчислювати на основі поточного значення стану або інших даних. В класових компонентах метод `setState` приймає об'єкт з оновленими значеннями властивостей стану або функцію, яка приймає поточний стан і возвращает об'єкт з оновленими значеннями. У функціональних компонентах функція, отримана від `useState`, приймає нове значення стану або функцію, яка приймає поточне значення стану і повертає нове значення.

```
// Класовий компонент
handleClick() {
  this.setState((prevState) => {
    return { counter: prevState.counter + 1 };
  });
}

// Функціональний компонент
function handleClick() {
  setCounter((prevCounter) => prevCounter + 1);
}
```

Sate

Асинхронність оновлення стану: Оновлення стану в React може бути асинхронним, тому не можна покладатися на негайну зміну значення стану після виклику `setState` або функції стану. Якщо вам потрібно виконати певну дію після оновлення стану, ви можете використовувати колбек-функцію, передану у `setState` (у класових компонентах) або використовувати ефекти (у функціональних компонентах).

Розширений стан: Стан в React може бути об'єктом, який містить більше однієї властивості. Ви можете додати додаткові властивості до стану і оновлювати їх незалежно один від одного. Наприклад:

```
// Класовий компонент
constructor(props) {
  super(props);
  this.state = {
    counter: 0,
    message: 'Привіт, React!'
  };
}

handleButtonClick() {
  this.setState({ counter: this.state.counter + 1 });
}

handleMessageChange(event) {
  this.setState({ message: event.target.value });
}

// Функціональний компонент
function MyComponent() {
  const [counter, setCounter] = useState(0);
  const [message, setMessage] = useState('Привіт, React!');

  function handleButtonClick() {
    setCounter(counter + 1);
  }

  function handleMessageChange(event) {
    setMessage(event.target.value);
  }

  // ...
}
```

State

Так як працювати доведеться далі в більшості з хуками, то більш детально про useState:

```
const [counter, setCounter] = useState(0);
```

Виклик хука “useState” створює стан і метод, який змінюватиме його значення. У якості параметра хук приймає початковий стан, в нашому випадку число 0. У стані може зберігатися будь-який тип даних.

counter = наше значення стейту

setCounter = функція за допомогою якої ми можемо змінювати стейт

useState() = hook який нам надає React для обробки стану компонента

0 = початковий стан компонента

Ми не можемо змінювати стан компонента напряму

```
const incrementHandler = () => {
  counter = 2
};
```

- **Інтерфейс залежить від стану компонента.**
- **Стан може змінитися як реакція на дії користувача.**
- **Під час зміни стану дані передаються вниз по дереву компонентів.**
- **Компоненти повертають оновлену розмітку і змінюється інтерфейс.**

Події

Для нативної події браузера в React створюється об'єкт-обгортка “SyntheticEvent Object” з ідентичним інтерфейсом. Це необхідно, щоб забезпечити крос-браузерність та оптимізувати продуктивність.

У React події (events) використовуються для обробки дій користувача, таких як натискання кнопок, введення тексту, наведення курсору тощо. React надає спеціальний синтаксис для роботи з подіями, який дозволяє додавати обробники подій до елементів інтерфейсу.

Основні принципи роботи з подіями в React:

Події назначаються в JSX: Ви можете назначити обробник подій безпосередньо в JSX, використовуючи синтаксис подібний до HTML. Наприклад, для назначення обробника події натискання кнопки ви можете використати атрибут `onClick`:

```
return <button onClick={event => console.log(event)}>Click me!</button>
```

Обробники подій: Обробники подій - це функції, які викликаються при виникненні певної події. Вони приймають об'єкт події як параметр і можуть містити логіку для обробки події. Наприклад:

```
function handleClick() {
  console.log('Кнопка була натиснута');
}
```

Події

Стандартні події: В React ви можете використовувати стандартні події, такі як `onClick`, `onChange`, `onSubmit` тощо, аналогічно до того, як це робиться в звичайному JavaScript. Ви також можете використовувати інші події, які підтримуються браузерами.

Обробка подій в класових компонентах: У класових компонентах React обробники подій оголошуються як методи класу. Наприклад:

```
class MyComponent extends React.Component {
  handleClick() {
    console.log('Кнопка була натиснута');
  }

  render() {
    return <button onClick={this.handleClick}>Натисни мене</button>;
  }
}

export default MyComponent;
```

Обробка подій в функціональних компонентах: У функціональних компонентах React обробники подій можуть бути оголошені як звичайні функції.

```
const MyComponent = () => {

  const handleClick = () => {
    console.log('Кнопка була натиснута');
  };

  return <button onClick={handleClick}>Натисни мене</button>;
};

export default MyComponent;
```

Події

Передача додаткових параметрів: Іноді вам може знадобитись передати додаткові параметри до обробника події. В такому випадку використовується стрілочна функція або функція з замиканням. Наприклад:

```
<button onClick={() => handleClick(param1, param2)}>Натисни мене</button>
```

Отримання інформації про подію: Об'єкт події, переданий в обробник, містить корисну інформацію про подію. Наприклад, ви можете отримати значення введеного тексту в поле вводу або отримати координати курсору миші. Наприклад:

```
function handleInputChange(event) {
  console.log(event.target.value); // Отримання значення поля вводу
}
```

```
function handleMouseMove(event) {
  console.log(event.clientX, event.clientY); // Отримання координат миші
}
```

Події

Відміна події: У React ви можете відмінити стандартну поведінку події, якщо ви хочете заборонити певні дії. Для цього викликайте метод `preventDefault()` на об'єкті події. Наприклад:

```
function handleLinkClick(event) {
  event.preventDefault(); // Відміна переходу за посиланням
  // Додаткова логіка
}
```

- Додавання обробника подій з EventTarget.addEventListener() майже не використовується, за рідкісним винятком.
- Пропси подій – не виняток та іменуються за допомогою camelCase. Наприклад onClick, onChange, onSubmit, onMouseEnter.
- У проп події передається посилання на callback-функцію, яка буде викликана під час настання події.
- Обробники подій отримують екземпляр SyntheticEvent Object.

В React "під капотом" реалізовано делегування подій. Слухачі не додаються безпосередньо до DOM-елементів. Передача колбека – це просто реєстрація функції, яка буде викликана внутрішніми механізмами реакта під час настання події.

Події

Анонімні колбеки

Інлайн колбеки вважаються антипатерном. Щоразу, коли компонент ререндериться, буде створена нова callback-функція. У багатьох випадках це нормально. Але, якщо callback передається як проп компонентам нижче у дереві, вони будуть знову відрендерені, оскільки прийдуть нові пропи посилального типу (функція). До того ж великі інлайн функції в JSX заважають читабельності розмітки компонента.

```
return (
  <button
    type='button'
    onClick={(event) => {
      console.log('button was clicked!', event); // працює
    }}
  />
);
```

Події

Кастомні методи

Найчастіше обробники подій оголошуються як методи класу чи функції, після чого jsx-атрибуту передається посилання на метод.

```
const handleClick = (event) => {
  console.log("button was clicked!", event);
}

return (
  <button
    type='button'
    onClick={handleClick}
  />
);
```

Домшнє завдання

1) Зробити дуже простий ToDo List

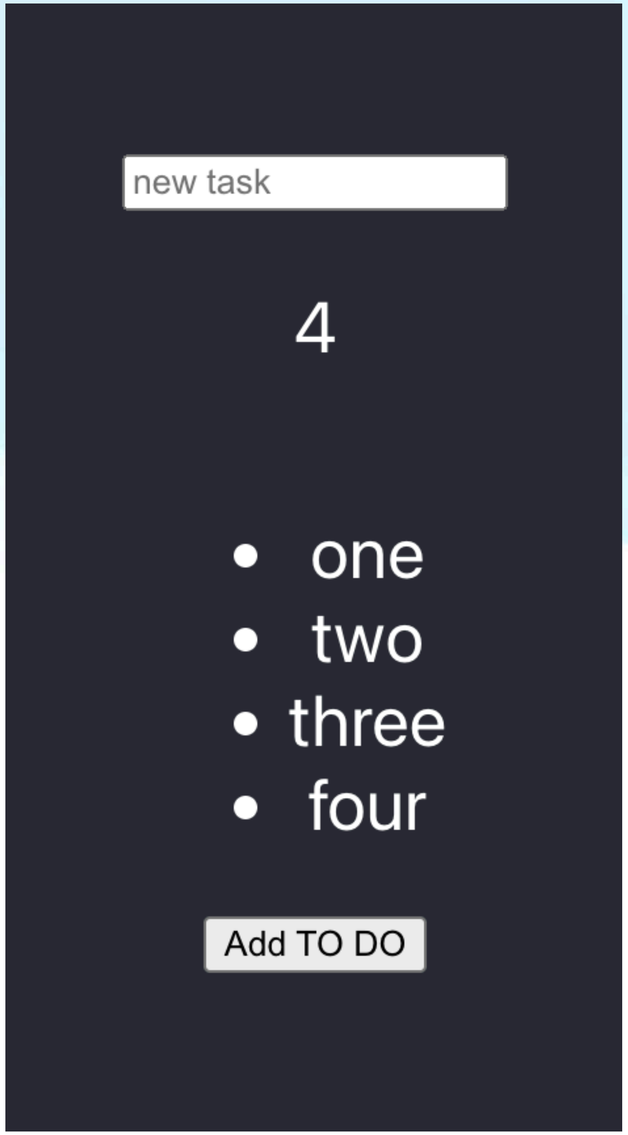
2) додати input і кнопку по якій ми будемо записувати наші to do.

3) Після вводу нашого to do в інпут і після натискання на кнопку має додатись до To Do List

4) Маємо побачити відмальовану To Do в інтерфейсі

5) Показати кількість to do у списку

6) Додати евент щоб можна будо додавати to do по натисканню кнопки “enter”



Корисні посилання

Virtual DOM

Як працює стейт

Події

Reconciliation React