Лекція 3

1) Рендер за умовою
2) Рендер колекцій
3) Що таке key
4) Props
5) Передача пропсів між компонентами

Рендер за мовою

Для рендеру розмітки за умовою використовуються оператори розгалужень та умов. Умови можна перевіряти перед поверненням розмітки або прямо в JSX.

Якщо за умовою нічого не повинно бути відрендерено, можна повернути “null”, “undefined”, “false” - вони не рендеряться.

Якщо умова буде істиною буде виконано тендер компоненту.

Залежно від вашої потреби, ви можете змінити умову та замінити вміст рендерованих компонентів на власний.

Рендер за мовою

“if” за допомогою логічного оператора “&&”

```
const HelloWorldComponent = () => {
  const value = 1;

  return (
    <div>
      {value === 1 && <div>I'm value equals one</div>}
      {value === 2 && <div>I'm value equals two</div>}
    </div>
  );
};

export default HelloWorldComponent;
```

“if…else” за допомогою тернарного оператора

```
const HelloWorldComponent = () => {
  const value = 1;

  return (
    <div>
      {value === 1 ? (
        <div>I'm value equals one</div>
      ) : (
        <div>I'm value equals two</div>
      )}
    </div>
  );
};

export default HelloWorldComponent;
```

За допомогою оператора “if”

```
const HelloWorldComponent = () => {
  const value = 1;

  if (value === 1) {
    return <div>I'm value equals one</div>;
  }

  if (value === 2) {
    return <div>I'm value equals two</div>;
  }

  return null;
};

export default HelloWorldComponent;
```

Рендер колекцій

У React, рендер колекцій (наприклад, масивів або списків) можна здійснити за допомогою методу ”map()”. Ось приклад, як це можна зробити:

```
function MyComponent() {
  const items = ['Item 1', 'Item 2', 'Item 3']; // Колекція, яку потрібно рендерити

  return (
    <div>
      {items.map((item, index) => (
        <p key={index}>{item}</p>
      ))}
    </div>
  );
}

export default MyComponent;
```

У цьому прикладі ми маємо масив items, який містить елементи, які ми хочемо відобразити. Використовуючи метод map(), ми перебираємо кожен елемент масиву і повертаємо JSX-код для кожного елементу.

Рендер колекцій

У нашому випадку, для кожного елементу створюється `<p>` елемент з вмістом елемента масиву (`{item}`). Також, ми встановлюємо атрибут `key` для кожного створеного елементу, що допомагає React-у ефективно оновлювати та рендерити компоненти при зміні даних.

```jsx
function MyComponent() {
  const items = ['Item 1', 'Item 2', 'Item 3']; // Колекція, яку потрібно рендерити

  return (
    <div>
      {items.map((item, index) => (
        <p key={index}>{item}</p>
      ))}
    </div>
  );
}

export default MyComponent;
```

У результаті, коли компонент `MyComponent` рендериться, кожен елемент масиву буде відображено у вигляді `<p>` елементу.

Key

Ключ (key) у React використовується для ідентифікації унікальності елементів масиву або списку компонентів під час їх рендерингу. Кожен елемент масиву або списку повинен мати унікальний ключ, щоб React міг ефективно визначати зміни та оновлювати компоненти.

Ось деякі основні причини використання ключа:

Ідентифікація елементів: Ключ допомагає React-у відслідковувати, які елементи змінились, були додані або видалені. Кожен ключ повинен бути унікальним у межах колекції елементів.

Підвищення ефективності: Ключі допомагають React-у оптимізувати процес оновлення компонентів. При оновленні колекції React порівнює ключі попередніх елементів з новими і визначає, які елементи потрібно оновити, які додати або видалити. Це дозволяє зменшити кількість операцій оновлення та зберегти ресурси.

Підтримка стану компонентів: Ключі допомагають зберігати стан компонентів при оновленні. Коли елемент масиву або списку має ключ, React може зберегти стан цього елементу навіть після оновлення колекції.

```
{items.map((item, index) => (
  <p key={index}>{item}</p>
))}
```

Key

Як правило, використовуються унікальні ідентифікатори або значення, які вже присутні в даних, як ключі для елементів.
Зазвичай це числа або рядки, які унікально ідентифікують кожен елемент.

Індекси масиву унікальні, проте вони не стабільні – при перетасовуванні колекції ключі змінюються. Дата і час – унікальні, але не стабільні, оскільки постійно збільшуються. Таким чином, під час кожного рендеру створюються нові ключі. Використання випадкового числа рівносильно тому, що ключі взагалі не використовуються, оскільки випадкові числа не є унікальними або стабільними.

Приклад використання ключа у рендерингу списку може виглядати так:

```
function MyComponent() {

  const items = [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' },
    { id: 3, name: 'Item 3' },
  ];

  return (
    <div>
      {items.map((item) => (
        <p key={item.id}>{item.name}</p>
      ))}
    </div>
  );
}

export default MyComponent
```

Key

Безпосередньо після рендерингу списку з ключами, React може використовувати ці ключі для визначення змін в колекції. Якщо ключі змінюються між оновлен
React розпізнає, які елементи були додані або видалені.

Наприклад, якщо у нашому компоненті `MyComponent` ми оновимо список елементів, зберігаючи лише перші два елементи,
React зрозуміє, що елемент з ключем “3” було видалено:

```jsx
function MyComponent() {
  const items = [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' }
  ];

  return (
    <div>
      {items.map(item => (
        <p key={item.id}>{item.name}</p>
      ))}
    </div>
  );
}

export default MyComponent;
```

Key

React знає, який елемент видалений, оскільки він порівнює ключі попередньої колекції з новим списком елементів.

Важливо пам'ятати, що ключі повинні бути унікальними лише в межах колекції. Якщо ви маєте дві незалежні колекції, то ключі можуть повторюватись між ними.

Наприклад, у випадку, коли ми маємо два незалежних списки елементів у компоненті:

```
function MyComponent() {
  const items1 = [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' }
  ];

  const items2 = [
    { id: 1, value: 'Value 1' },
    { id: 2, value: 'Value 2' }
  ];

  return (
    <div>
      {items1.map(item => (
        <p key={item.id}>{item.name}</p>
      ))}
      {items2.map(item => (
        <p key={item.id}>{item.value}</p>
      ))}
    </div>
  );
}
```

У цьому випадку, хоча ключі в обох списках збігаються, React розуміє, що це різні колекції, і вони не перетинаються між собою.

Отже, ключ (key) у React є важливим атрибутом, який допомагає визначити унікальність елементів у колекціях, підвищує ефективні

Props

Props (властивості) в React - це спосіб передачі даних вниз по дереву компонентів в React. Props є незмінними та доступними для читання, і вони передаються в компоненти як атрибути.

При зміні Props як і при зміні State компонент буде перерендерено!

Основні характеристики Props:

Передача даних: Props дозволяють передавати дані від батьківського компонента до дочірніх компонентів. Батьківський компонент встановлює значення Props, а дочірній компонент отримує ці значення та використовує їх для відображення.

Незмінність: Props є незмінними, тобто вони не можуть бути змінені дочірніми компонентами. Вони можуть бути тільки читані. Це забезпечує безпеку та передбачуваність в передачі даних між компонентами.

Використання в JSX: Props використовуються в JSX, щоб передавати значення атрибутів компонентів. Вони передаються як пари "ім'я-значення" і можуть містити будь-який тип даних, включаючи примітивні значення, об'єкти, функції або навіть компоненти.

Передача через дочірні компоненти: Props можна передавати через кілька рівнів дочірніх компонентів. Кожен наступний компонент може отримати Props від свого безпосереднього батька та передати їх своїм дочірнім компонентам.

Використання в компонентній логіці: Компоненти можуть використовувати Props для прийняття рішень та змінення свого внутрішнього стану. Значення Props можуть впливати на роботу компонента, управляти його відображенням та взаємодією з користувачем.

Props

Приклад використання Props:

```
function MyComponent(props) {
  return <p>Hello, {props.name}!</p>;
}
```

У цьому прикладі, компонент `MyComponent` приймає `props` як свій параметр. `props` є об'єктом, який містить всі передані властивості (props) від батьківського компонента.

У внутрішньому коді компонента, ми можемо отримати значення певної властивості `name` з `props` і використати її для відображення повідомлення. У нашому випадку, ми використовуємо `props.name` для відображення привітання з іменем, яке було передано у властивостях.

Props

Приклад використання `MyComponent`:

```
import MyComponent from './MyComponent';

function App() {
  return <MyComponent name='John' />;
}

export default App;
```

У компоненті `App` ми використовуємо `MyComponent` і передаємо йому властивість `name` зі значенням `"John"`. При рендерингу `MyComponent`, властивість `name` стає доступною як `props.name`, і воно буде використано для відображення повідомлення `"Hello, John!"`.

Props

Також ви можете передавати не примітивні типи даних як Props. Ось декілька прикладів:

Об’єкти:

```
// Батьківський компонент
const user = {
  name: 'John',
  age: 30,
};

return <ChildComponent user={user} />;
```

```
// Дочірній компонент
function ChildComponent(props) {
  return (
    <div>
      <p>Name: {props.user.name}</p>
      <p>Age: {props.user.age}</p>
    </div>
  );
}
```

Props

Масиви:

```
// Батьківський компонент
const numbers = [1, 2, 3, 4, 5];

return <ChildComponent numbers={numbers} />;
```

```
// Дочірній компонент
function ChildComponent(props) {
  const numberList = props.numbers.map((number) => (
    <li key={number}>{number}</li>
  ));

  return <ul>{numberList}</ul>;
}
```

Функції:

```
// Батьківський компонент
function greet(name) {
  alert(`Hello, ${name}!`);
}

return <ChildComponent greet={greet} />;
```

```
// Дочірній компонент
function ChildComponent(props) {
  function handleClick() {
    props.greet('John');
  }

  return <button onClick={handleClick}>Greet</button>;
}
```

Props

У цих прикладах ми передаємо об'єкти, масиви та функції як Props. Дочірні компоненти можуть отримати ці значення і використовувати їх у своєму відображенні або взаємодії з користувачем. Важливо пам'ятати, що при передачі складних типів даних, таких як об'єкти чи масиви, їх необхідно правильно обробляти в дочірньому компоненті, враховуючи їх структуру та особливості використання.

Props Drilling:

Props drilling (також відомий як прокладання пропсів) відбувається, коли потрібно передати дані через багато проміжних компонентів від батька до дитини. Це відбувається шляхом передачі пропсів через кожен компонент у ланцюжку, навіть якщо деякі компоненти не потребують цих пропсів.

Props

Ось приклад простого компонентного дерева, де відбувається Props drilling:

```jsx
// Батьківський компонент
function ParentComponent() {
  const message = 'Hello, world!';

  return <ChildComponent message={message} />;
}

// Дочірній компонент
function ChildComponent(props) {
  return <GrandchildComponent message={props.message} />;
}

// Внуковий компонент
function GrandchildComponent(props) {
  return <p>{props.message}</p>;
}
```

У цьому прикладі `ParentComponent` передає `message` як пропс до `ChildComponent`, і потім `ChildComponent` передає його до `GrandchildComponent`. Цей процес передачі пропсів через кожний компонент у ланцюжку відбувається для того, щоб `GrandchildComponent` могло відобразити повідомлення `message`.

Props

Props drilling має свої переваги і недоліки:

Переваги:

Простота: Props drilling є простим і прямолінійним способом передачі даних між компонентами.

Передбачуваність: Ви чітко бачите, звідки походить кожен пропс і куди йде.

Гнучкість: Props drilling дозволяє зберігати дані на рівні вищих компонентів, використовувати їх в багатьох дочірніх компонентах або навіть модифікувати дані у проміжних компонентах.

Недоліки:

Зайва передача даних: Якщо вам потрібно передати дані через багато проміжних компонентів, це може спричинити зайву передачу даних в компонентах, які їх не використовують.

Збільшена складність: Із зростанням кількості компонентів у ланцюжку може збільшуватися склад

Таким чином, використання Props дозволяє нам передавати дані вниз по дереву компонентів і використовувати їх у відображенні та логіці компонентів. Це допомагає створювати повторно використовувані та динамічні компоненти, що реагують на зміни вхідних даних.

Корисні посилання

Умовний рендеринг

All the Conditional Renderings in React

Списки і ключі

Props

Props

Домашнє завдання

1) Продовжуємо модифікувати наш Туду Ліст з попереднього ДЗ

2) Створити початковий массив данних для модифікування яки має включати мін. 3 елементи

3) Один туду (об’єкт в масиві) має включати в себе “id” і “name”

4) Відрендерити цей массив в дочірньому компоненті

5) При додаватись має об’єкт з такими саме значеннями (id, name)

6) Реалізувати видалення туду (додати кнопку для видалення і при натисканні видаляти вказану туду)