Київський університет імені Бориса Грінченка

JavaScript-бібліотека React

Навчальний посібник для студентів спеціальності
Комп'ютерні науки

Рекомендовано Вченою радою Факультету інформаційних технологій та математики Київського університету імені Бориса Грінченка як навчальний посібник для лабораторних та самостійних робіт студентів галузі знань 12 Інформаційні технології

(протокол №6 від 29.08.2023 р.)

Рецензенти:

Бондарчук Андрій Петрович, доктор технічних наук, професор, директор науково-навчального Інституту інформаційних технологій Державного університету інформаційно-комунікаційних технологій;

Оніщенко Вікторія Валеріївна, доктор технічних наук, професор, професор кафедри інформаційних систем і технологій Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського»

Яскевич В.О.

Навчальний посібник розроблений для студентів галузі знань 12 Інформаційні технології, метою якого є підвищення рівня практичної підготовки з технологій розробки програмних продуктів. Посібник містить теоретичний матеріал з бібліотеки React, приклади коду, завдання для самостійної роботи. Виконання завдань, вправ розміщених у посібнику сприятиме засвоєнню набутих знань та практичних навичок з технологій розробки програмних продуктів. Посібник може бути корисним для будь-кого, хто хоче ознайомитися з сучасними тенденціями створення Вебдодатків.

Зміст

Вступ

React (також відомий як React.js або ReactJS) — декларативна, ефективна та гнучка бібліотека JavaScript для створення інтерфейсів користувача, що дозволяє створювати складні інтерфейси користувача з невеликих ізольованих фрагментів коду, які називаються «компонентами». Він підтримується Meta (раніше Facebook) та спільнотою окремих розробників і компаній. React можна використовувати як основу для розробки односторінкових, мобільних або серверних додатків із такими фреймворками, як Next.js. Однак React займається лише керуванням станом і відтворенням цього стану в DOM, тому створення додатків React зазвичай потребує використання додаткових бібліотек для маршрутизації, а також певної функціональності на стороні клієнта.

Історія React.JS

- Поточна версія React.JS – V18.2.0 (червень 2022 р.);
- Початковий випуск для громадськості (V0.3.0) травень 2013 року;
- React.JS вперше був використаний у 2011 році для функції стрічки новин Facebook;
- Створений Jordan Walke інженером-програмістом Facebook.

Спробувати React

Онлайн-майданчики

Якщо вам цікаво спробувати **React**, ви можете скористатися онлайн-майданчиками для коду:

- CodePen;
- CodeSandbox;
- Stackblitz.

Приклад можна побачити за посиланням.

Додавання React на вебсайт

React можна додати до HTML-сторінки. Почніть із додавання трьох скриптів: перші два дозволяють писати код React у JavaScript, а третій, Babel, дозволяє писати синтаксис JSX і ES6 у старих браузерах.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Home</title>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width" />
    <!--
      Need a visual blank slate?
      Remove all code in `styles.css`!
    -->
    <link rel="stylesheet" href="styles.css" />
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
    <script type="text/babel" src="script.js"></script>
</head>
<body>
    <div id="root"></div>
</body>
</html>
```

Повний приклад за посиланням.

Створення додатку React

Є два поширених способи налаштувати нову програму React:

- Утиліта CLI, create-react-app;
- Пакувальник JavaScript, наприклад Webpack.

Обидва варіанти використовують npm, менеджер пакетів node.js.

create-react-app

Facebook надає утиліту командного рядка під назвою **create-react-app**, яка автоматично налаштовує новий проект React із розумною типовою структурою проекту та набором функцій. Це найкращий спосіб почати новий проект новачку.

Щоб ініціалізувати нову програму, виконайте:

```
npx create-react-app my-app
cd my-app
npm start
```

а потім перейдіть до *http://localhost:3000/* у вашому браузері.

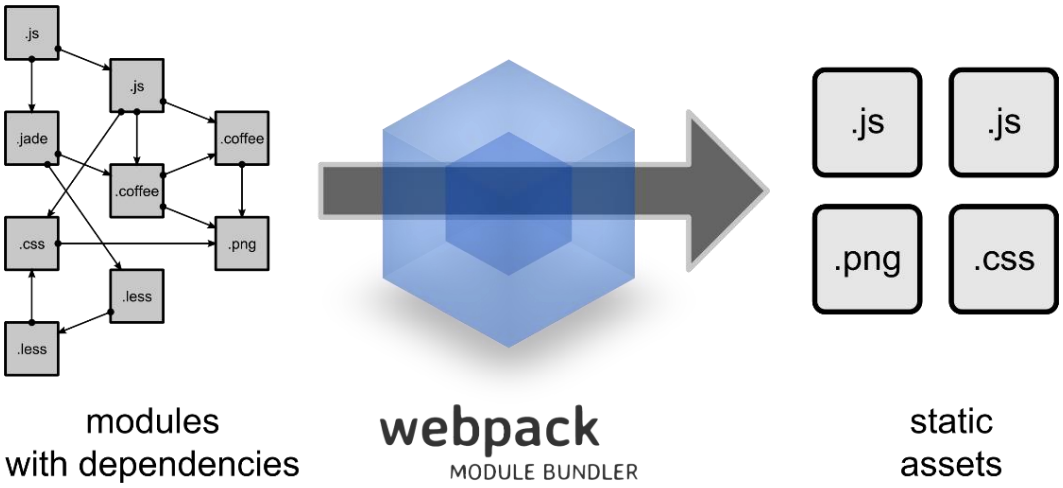
Повний посібник із початку роботи з додатком **create-react-app** за посиланням.

JavaScript пакувальник (Bundler)

Більшість додатків React створено за допомогою пакувальників JavaScript. Пакувальник об'єднує всі вихідні файли JavaScript в один файл, який потім можна включити в тег <script> сторінки HTML.

Найпоширенішим пакувальником є Webpack. Webpack легко налаштовується завдяки розгалуженій екосистемі плагінів. Якщо вам потрібно більше контролю та гнучкості, ніж пропонує програма create-react-app, подумайте про вивчення Webpack.

Це інструмент, який використовує create-react-app!



JSX

JSX — це спосіб опису візуального коду за допомогою поєднання коду **JavaScript** і розмітки **XML**. **JSX** є рекомендованим способом створення інтерфейсу.

Вирази JSX — це стислий синтаксис для використання API, *React.createElement (type, props, ...children)*.

Babel компілює JSX до викликів React.createElement().

Ці два приклади ідентичні:

```
const element = (
    <h1 className="greeting">
        Hello, React!
    </h1>
);
```

```
const element = React.createElement(
    'h1',
    { className: 'greeting' },
    'Hello, React!'
);
```

Вирази JSX можна використовувати будь-де, де можна використовувати будь-який інший вираз, наприклад, у операторі return або присвоєння значення змінній.

Різниця JSX та HTML

Усі атрибути HTML пишуться у стилі *camelCase*. Наприклад:

HTML

```
<input class="name"
       tabindex="2"
       onchange="console.log('changed!');">
```

JSX

```
<input
    class="name"
    tabindex="2"
    onchange="console.log('changed!');"
/>
```

Елементи повинні бути закриті. Наприклад:

HTML

```
<div>
    <span>Enter your name:</span><br>
    <input type="text" />
</div>
```

JSX

```
<div>
    <span>Enter your name:</span><br />
    <input type="text" />
</div>
```

Назви компонентів мають починатися з великої літери.

```
<Article />
React.createElement(Article, null);
```

Може бути лише один кореневий елемент.

```
<p>First paragraph</p >
<p>Second paragraph</p>
SyntaxError: Adjacent JSX elements must be wrapped in an enclosing tag
```

```
<div>
    <p>First paragraph</p>
    <p>Second paragraph</p>
</div>
```

Вирази та атрибути

Вирази у JSX

У JSX можна використовувати будь-який вираз JavaScript. Вирази JavaScript мають бути укладені у фігурні дужки **{ }**

```
const user = { name: 'John' };

const element = <div>Hello, {user.name}</div>;
```

Визначення атрибутів за допомогою JSX

Будь-які атрибути JSX стають властивостями (параметрами) елемента React. Значенням атрибута може бути рядок, як-от **foo="hello"**, або це може бути будь-який вираз JavaScript, укладений у фігурні дужки, як у **bar={baz}** (що встановлює значення параметра **bar** на **baz**).

```
const element = <div foo="hello" bar={baz} />
```

Запитання для самоконтролю

1. Що таке JSX і як він пов’язаний з React?
2. Які основні відмінності між JSX і HTML?
3. Як можна використовувати JavaScript в JSX?
4. Які основні принципи роботи з JSX в React компонентах?
5. Чим JSX відрізняється від шаблонів розмітки в інших фреймворках?
6. Чому використання JSX рекомендується в React?
7. Як повертати кілька елементів з компонента в JSX?
8. Чим JSX відрізняється від шаблонних мов, таких як Pug або Handlebars?

Основні поняття

Бібліотека React дозволяє визначити весь інтерфейс додатка як дерево об'єктів JavaScript. Ці об'єкти JavaScript називаються елементами React. React надає утиліти для створення та керування цим деревом.

Бібліотека використовується з прив'язками до певної платформи, наприклад, React DOM для Інтернету. Для рендерингу дерева елементів React використовуються власні елементи інтерфейсу користувача (тобто елементи DOM). Щоразу, коли дерево елементів React змінюється, React DOM оновлює елементи інтерфейсу користувача відповідно до цих змін.

Елементи

Елементи — це найменші будівельні блоки додатків React.

Елементи React створюються за допомогою API *React.createElement*. Як правило, цей API викликається за допомогою елементів *JSX*, але його також можна викликати безпосередньо. Сигнатура функції: *createElement(type, props, children)*.

Давайте ближче розглянемо внутрішню структуру дерева елементів React. Зараз не важливо розуміти точні деталі дерева елементів, але корисно зрозуміти суть того, як працює React.

У цьому прикладі кілька компонентів React створюються за допомогою JSX, а потім друкуються на консолі.

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import './style.css';

const myElement = (
    <div foo="bar">
        <button>Test</button>
        <span>Hello</span>
    </div>
);

// Simplify myElement and print it to the console
console.log(JSON.parse(JSON.stringify(myElement)));

const root = createRoot(document.getElementById('root'));

root.render(myElement);
```

«Живий» приклад за посиланням.

Якщо ви клацнете рядок тексту в консолі праворуч, ви зможете переміщатися по дереву об'єктів JavaScript.

```
{type: "div", key: null, ref: null, props: {…}, …}
      key: null
      props: Object
            children: Array[2]
            foo: "bar"
            <prototype>: Object
      ref: null
      type: "div"
```

```
      _owner: null
      _store: Object
      <prototype>: Object
```

Елемент React — це, по суті, об'єкт JavaScript, що містить **type** і об'єкт **props**.

Об'єкт **props** може містити довільні властивості, наприклад, **foo**, плюс дочірню властивість, що містить вкладені елементи React. Властивість **children** є дещо особливою, оскільки вона створюється автоматично з вкладених елементів JSX у коді.

Рендерінг (візуалізація)

Щоб побачити елементи React на екрані, потрібно відрендерити (візуалізувати) їх за допомогою спеціальної бібліотеки рендерингу для конкретної платформи.

Вузол DOM має бути обраний у програмі для відтворення елементів React. Вузол слід вибирати без будь-яких існуючих дочірніх вузлів або з припущенням, що всі існуючі дочірні вузли будуть замінені. React не може рендерити у вузол **Body**, але будь-який вузол можна вибрати всередині **Body**. Наприклад, якщо у вас є вузол з **id="app"**, ви можете отримати посилання на нього за допомогою **document.querySelector('#app')**.

Програми, створені лише за допомогою React, зазвичай мають один кореневий вузол DOM. Якщо ви інтегруєте React в існуючу програму, ви можете мати скільки завгодно ізольованих кореневих вузлів DOM.

Щоб відобразити елемент React, спочатку передайте елемент DOM у **ReactDOM.createRoot()**, а потім передайте елемент React у **root.render()**:

```
import React from 'react';
import ReactDom from 'react-dom/client';

const root = ReactDom.createRoot(document.getElementById('app'));
const element = <h1>Hello React!</h1>;
root.render(element);
```

Як правило, вам потрібно імпортувати React з **«react»**, навіть якщо ви явно не посилаєтеся на React у своєму коді. Це тому, що наш **JSX <h1/>** перетворюється за лаштунками на **React.createElement('h1')**, на який посилається React.

Компоненти та властивості (props)

Компоненти React дозволяють розділити інтерфейс користувача на незалежні частини, які можна багаторазово використовувати, і думати про кожну частину окремо.

Компоненти React схожі на шаблон, який приймає параметри як вхідні дані та виводить дерево елементів React.

Існує 2 способи визначення компонента React:

- **Функціональні компоненти** — функція, яка приймає параметри (так звані *props*) як вхідні дані та повертає елемент React;
- **Компоненти класу** — клас, який розширює *React.Component* і реалізує метод рендерингу.

Давайте розглянемо кожен спосіб визначення компонента React.

Функціональні компоненти

Найпоширенішим видом компонента є функція, яка повертає елемент React.

У цьому прикладі компонент *MyComponent* створюється за допомогою функції, що приймає на вхід один параметр.

Функція створює компонент *div* як елемент React. Елемент *div* містить єдиний дочірній елемент — елемент *button* (кнопка). Кнопка, у свою чергу, містить один дочірний елемент. Цей дочірній елемент є рядком і передається як текстовий атрибут.

Потім викликається метод '*render*', щоб відобразити цю ієрархію інтерфейсу користувача в DOM браузера.

```
import React from 'react';
import { createRoot } from 'react-dom/client';

function MyComponent(props) {
    return (
        <div style={{ padding: '30px', backgroundColor: 'lightblue' }}>
            <button>{props.text}</button>
        </div>
    );
}

const element = <MyComponent text="Hello, world!" />;

const root = createRoot(document.getElementById('app'));
root.render(element);
```

«Живий» приклад за посиланням.

Компоненти класу

Компоненти можна створити за допомогою класу **React.Component**, наприклад клас **MyComponent** розширює клас **React.Component** і перевизначає метод **render()**, що повертає елемент *React*. Це був оригінальний API компонентів до того, як функціональні компоненти були додані до React.

```
import React from 'react';
import { createRoot } from 'react-dom/client';

class MyComponent extends React.Component {
    render() {
        return (
            <div style={{ padding: '30px', backgroundColor: 'lightblue' }}>
                <button>{this.props.text}</button>
            </div>
        );
    }
}

const element = <MyComponent text="Hello, world!" />;

const root = createRoot(document.getElementById('app'));
root.render(element);
```

«Живий» приклад за посиланням.

Props доступні лише для читання

Незалежно від того, оголошується компонент як функція чи клас, він ніколи не має змінювати власні властивості (**props**).

Стилізація

Існує багато способів стилізації компонентів React. Не існує одного «найкращого» способу стилізації компонентів, і нові бібліотеки стилів створюються часто. Вибір методу часто залежить від особистих уподобань, оскільки функції кожного методу загалом подібні.

Розглянемо три поширені підходи:

- Вбудовані стилі;
- CSS і імена класів;
- css-in-js.

Вбудовані стилі

Атрибут **style** приймає об'єкт JavaScript із стильовими властивостями, а не рядок CSS. CSS стилі записані в *CamelCase* стилі. Наприклад, властивість **font-size**, буде записана як **fontSize**.

Це поширений і потужний спосіб стилізації компонентів. У цьому випадку не потрібні жодні бібліотеки, і забезпечується повна сумісність з будь-якими платформами (веб, мобільні, тощо). Стилізація може бути динамічною (обчислюється на основі атрибутів компонентів).

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function getRandomHexColor() {
    return `#${Math.floor(Math.random() * 16777215)
        .toString(16)
        .padStart(6, 0)}`;
}

function App() {
    const [color, setColor] = useState('tomato');

    return (
        <div
            style={{ padding: '30px', backgroundColor: color, textAlign: 'center' }}
        >
            <button onClick={() => setColor(getRandomHexColor())}>
                Change color
            </button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Переваги:

- Вбудованість (без бібліотек/залежностей);
- Динамічність (змінні, теми, злиття тощо);

- Стилі компонентів містяться в одному файлі з кодом і поведінкою;
- Незалежність від платформи, працює на будь-якому рендерері React (веб, Native,...).

Недоліки:

- Неможливо використовувати функції CSS із «коробки», такі як: псевдокласи, медіа-запити, анімація тощо;
- Неможливо використовувати вендорні префікси із «коробки»;
- Нова схема іменування/синтаксис для вивчення (стилі в camelCase, числа або рядки як значення).

CSS і класи

Один з найпростіших способів стилізації компонентів — за допомогою CSS. Ви можете використовувати CSS без необхідності використання будь-якого препроцесора або постпроцесора, просто використовуючи атрибут **className** у ваших компонентів для застосування стилів.

Ви можете отримати динамічні стилі, вибираючи іншу назву класу на основі властивостей компонента, наприклад:

```
<button className="btn">Click</button>
```

Також можна поєднувати вбудовані стилі з CSS та імена класів, хоча комбінування цих двох підходів слід зводити до мінімуму, інакше код швидко стає складним для розуміння.

Переваги:

- Повне використання всіх можливостей CSS;
- Немає потреби у вивченні нового синтаксису та додаткових технологій;
- Добра інтеграція з не-React бібліотеками, які використовують імена класів для стилізації.

Недоліки:

- Всі проблеми CSS (зазвичай великий та важкий для підтримки код);
- Компоненти не є самодостатніми в одному файлі JavaScript;
- Тільки DOM-рендерер (немає підтримки React Native).

CSS-in-JS

Styled components — це інструмент *CSS-in-JS*, який поєднує компоненти та стилізацію, надаючи безліч функціональних можливостей для стилізації компонентів у функціональний та повторно використовуваний спосіб.

Важливо зрозуміти щодо *Styled Components:* їх назва має бути прийнята буквально. Більше не потрібно стилізувати HTML-елементи на основі їх класів:

```
<button className="btn">Click</button>
```

```
button.btn {
    font-size: 2em;
    background-color: black;
    color: white;
}
```

Замість цього, вам потрібно визначати стилізовані компоненти, які мають свої вбудовані стилі:

```
const Button = styled.button`
  font-size: 2em;
  background-color: black;
  color: white;
`;
```

Styled Components дозволяють писати CSS у компонентах, не хвилюючись про конфлікти імен класів.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';
import styled from 'styled-components';

function getRandomHexColor() {
    return `#${Math.floor(Math.random() * 16777215)
        .toString(16)
        .padStart(6, 0)}`;
}
const Wrapper = styled.section`
  padding: 30px;
  text-align: center;
  background-color: ${(props) => props.color};
`;
const Btn = styled.button`
  padding: 8px;
  background-color: lightblue;
  border-radius: 5px;
`;

function App() {
    const [color, setColor] = useState('tomato');

    return (
        <Wrapper color={color}>
            <Btn onClick={() => setColor(getRandomHexColor())}>
                Change color
            </Btn>
        </Wrapper>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Переваги:

- Динамічність (залежить від властивостей компонента);
- Можливості CSS (псевдокласи, медіа-запити, анімації на ключових кадрах);
- Автоматичне додавання вендорних префіксів;
- Знайомий синтаксис CSS;
- Стилі компонентів знаходяться в тому самому файлі, що й код і поведінка;
- Працює на будь-якому рендерері React (веб, нативний тощо).

Недоліки:

- Дуже догматичний стиль коду (створення компонента на кожен стиль);
- Збільшений розмір коду (повільніше завантаження).

Обробка подій

Обробка подій забезпечує можливість користувачу взаємодіяти з вебсторінкою та виконувати певні дії, коли настає певна подія, така як клік або наведення. Коли користувач взаємодіє з додатком, виникають події, наприклад: наведення курсору, натискання клавіші, подія зміни тощо.

Обробка подій у React дуже схожа на обробку подій у DOM-елементах, хоча є деякі синтаксичні відмінності, наприклад:

У HTML:

```
<script>
  function handleClick(e) {
    console.log(e);
  }
</script>
<div id="app"></div>
<button onClick="handleClick(event)">HTML</button>
```

У React:

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function handleClick(e) {
    console.log(e);
}

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

    return (
        <div>
            <button onClick={handleClick}>React</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Схема назв подій

Назви подій використовують стиль *camelCase*, тому потрібно використовувати **onClick** замість нативного еквіваленту **onclick**. React нормалізує ці назви у всіх браузерах, тому вам не потрібно брати до уваги неузгодженості між браузерами.

Повний список подій, які підтримує React, можна знайти тут за посиланням.

Компоненти користувача та події

Допустимо, необхідно створити власний компонент **CounterBtn** з подією **onClick**. Створення кнопки за допомогою **<CounterBtn onClick={() => ...} />** повідомить React створити екземпляр класу **CounterBtn** з властивістю **onClick**, значенням якої є функція. Однак саме це не змусить **CounterBtn** реагувати на кліки.

Тільки компоненти DOM можуть обробляти події DOM, такі як **onClick**, тому **CounterBtn** має відображати компонент DOM і передавати властивість **onClick**. Цей **CounterBtn**, насправді, є наскрізним для події кліка.

Назва переданої властивості **onClick** компоненту **CounterBtn** є довільною — можна називати її, як завгодно, головне, щоб десь всередині **CounterBtn** властивість передавалася в обробник події **onClick** DOM-компонента. Наприклад, ми можемо вирішити назвати властивість **onPress** і створити нашу кнопку **CounterBtn** як **<CounterBtn onPress={() => ...} />**. Усередині **CounterBtn** ми потім захочемо відобразити **<button onClick={props.onPress} />**.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

const CounterBtn = ({ title, onPress }) => (
    <button onClick={onPress}>{title}</button>
);

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

    return (
        <div>
            <p>Counter: {count}</p>
            <CounterBtn title="Add 1" onPress={() => setCount(count + 1)} />
            <CounterBtn title="Subtract 1" onPress={() => setCount(count - 1)} />
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Тут до **CounterButton** передається властивість **onPress**, яка потім передається в **onClick** елемента **button**.

Форми

Елементи форм HTML в React працюють трохи інакше, ніж інші DOM-елементи, оскільки елементи форм зберігають деякий внутрішній стан.

Традиційно, у веброзробці введені користувачем дані зберігаються в DOM, ці дані (наприклад, те, що користувач ввів у поле введення) витягуються з DOM для подальшого використання в логіці додатка. React значно спрощує обробку введення, розглядаючи компоненти форм, як такі, що не мають стану. Наприклад елемент **input** має властивість **value** та властивість **onChange**, і разом це дає повний контроль над введенням без необхідності торкатися DOM.

<Input>

Розглянемо текстове поле введення (**input**) та відправлення даних форми:

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function App() {
```

```
    const [text, setText] = useState('');

    function handleSubmit(event) {
        alert('A text was submitted: ' + text);
        event.preventDefault();
    }

    return (
        <form onSubmit={handleSubmit}>
            <label htmlFor="inputFld">Enter text: </label>
            <input
                id="inputFld"
                type="text"
                value={text}
                placeholder="type here ..."
                onChange={(e) => setText(e.target.value)}
            />

            <p>You entered: {text} </p>
            <input type="submit" />
        </form>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Кожного разу, коли компонент **input** відображається, йому передається поточне значення зі стану компонента. Кожного разу, коли користувач вводить текст у поле введення, стан оновлюється та зберігає нове значення, і компонент **input** знову відображається.

Це автоматично підтримує синхронізацію DOM із змінною стану (*text*). За допомогою цього підходу можна маніпулювати значенням поля введення, як звичайною змінною, тим часом як React виконує за лаштунками операції з DOM.

Можна керувати подією *submit*, додавши обробник події до атрибуту *onSubmit* форми.

<textarea>

У HTML — вміст елемента **<textarea>** вводиться між відкриваючим та закриваючим тегами. Тег **<textarea>** не підтримує атрибут *value*:

```
<textarea>
  Content of the textarea
</textarea>
```

У React елемент **<textarea>** використовує атрибут *value* замість цього. Таким чином форма, яка використовує **<textarea>**, може бути написана дуже подібно до форми, яка використовує **<input>**:

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function App() {
    const [text, setText] = useState('');

    function handleSubmit(event) {
        alert('A text was submitted: ' + text);
```

```
        event.preventDefault();
    }

    return (
        <form onSubmit={handleSubmit}>
            <label>
                <textarea
                    value={text}
                    placeholder="type here ..."
                    onChange={(e) => setText(e.target.value)}
                />
            </label>
            <br />
            <input type="submit" />
        </form>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Умовний рендеринг

Умовний рендеринг в React — це підхід, який дозволяє відображати компоненти на основі умов або стану додатку. За допомогою умовного рендерингу, можна контролювати, які компоненти будуть відображені на екрані залежно від поточного стану додатку.

Умовний рендеринг у React використовує знайомі концепції JavaScript, такі як оператор **if**, тернарний оператор **(?:)**, або логічний оператор **&&**. Ці оператори використовуються для створення умов, які визначають, коли той чи інший компонент має бути відображений.

Наприклад, можна використовувати умовний рендеринг, для відображення повідомлення про успішне завершення дії, або про помилку, залежно від результату виконання певної операції. Також можна відображати компоненти на основі значень змінних стану, які змінюються в процесі роботи додатка.

Умовний рендеринг є потужним інструментом у React, який дозволяє створювати динамічні інтерфейси, які змінюються відповідно до умов або стану вашого додатка. Це дозволяє створювати гнучкі, адаптивні та інтерактивні вебдодатки з більшим контролем над тим, що відображається на екрані.

Оператор логічне І (&&)

Коли потрібно відобразити щось або нічого, можна використати оператор **&&**. На відміну від оператора **&**, оператор **&&** не обчислює праву частину виразу, якщо ліва частина вже може визначити кінцевий результат.

Наприклад, якщо перший вираз оцінюється як *false*, у виразі **false && ...**, немає потреби обчислювати наступний вираз, оскільки результат завжди буде *false*.

У React ви можете використовувати такі вирази, як у прикладі нижче:

```
    return <div>{showHeader && <Menu />}</div>;
```

Якщо **showHeader** оцінюється як *true*, то компонент **<Menu/>** буде повернуто виразом. Якщо **showHeader** оцінюється як *false*, компонент **<Menu/>** буде проігнорований, і буде повернуто порожній ** **.

У наступному прикладі відображаємо компонент картку, який приймає пропси **title** і, можливо, **subtitle**. Елемент **h2** буде відображений тільки для підзаголовків, якщо вони існують.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';
import './style.css';

function Card({ title, subtitle }) {
    return (
        <div className="card">
            <h1>{title}</h1>
            {subtitle && <h2>{subtitle}</h2>}
        </div>
    );
}

function App() {
    return (
        <div>
            <Card title={'Title'} />
            <Card title={'Title'} subtitle={'Subtitle'} />
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

У цьому прикладі, якщо підзаголовок (**subtitle**) не передається явно, значення **props** буде *undefined*, тому вираз **&&** оцінюється як ліва частина (*undefined*). Коли React бачить значення *undefined*, він не відображає нічого. Коли ми передаємо значення для **props subtitle**, ліва частина виразу вважається істинною, тому вираз оцінюється як права частина — у нашому випадку, компонент **h2**.

Тернарний оператор (?:)

Ще один спосіб умовного рендерингу елементів у React — використовувати тернарний оператор JavaScript (**умова? true: false**).

За допомогою тернарного оператора можна відображати різні React елементи залежно від того, чи передано значення **props**, або від його значення. Це зазвичай використовується для відображення значень за замовчуванням.

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import './style.css';

function Card({ title, subtitle }) {
    return (
        <div className="card">
            <h1>{title}</h1>
            {subtitle ? <h2>{subtitle}</h2> : <h3>No subtitle</h3>}
```

```
        </div>
    );
}

function App() {
    return (
        <div>
            <Card title={'Title'} />
            <Card title={'Title'} subtitle={'Subtitle'} />
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Оператор if else

Для більш складного рендерингу можна використовувати змінні для збереження React елементів, що дозволяє комбінувати їх по-різному.

У прикладі рендеринг значно відрізняється залежно від того, чи існують **props loading** та **error**.

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import './style.css';

function Card({ loading, error, title, subtitle }) {
    let content;

    if (error) {
        content = 'Error';
    } else if (loading) {
        content = <h3>Loading...</h3>;
    } else {
        content = (
            <div>
                <h1>{title}</h1>
                {subtitle ? <h2>{subtitle}</h2> : <h3>No subtitle</h3>}
            </div>
        );
    }

    return <div className="card">{content}</div>;
}

function App() {
    return (
        <div>
            <Card error={true} />
            <Card loading={true} />
            <Card loading={false} title={'Title'} subtitle={'Subtitle'} />
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Списки та ключі

Кожному компоненту React можна передати спеціальний **prop**, який називається ключем (**key**). React використовує цей ключ для визначення ідентичності елемента, що відображається. Розуміння ідентичності елемента є критичним для покращення продуктивності та мінімізації маніпуляцій з DOM. Наприклад, якщо перший елемент у списку з тисяч елементів не повинен бути відображений, React потребує способу виявити це. Замість рендеру тисяч елементів, React може видалити один DOM-вузол першого елемента, що є значно ефективнішим.

Під час рендерингу окремих компонентів (у порівнянні зі списками компонентів), React автоматично призначає ключі елементам на основі їх порядку у рендерингу.

Таким чином, якщо ви бажаєте, ви можете надати ключ для кожного елемента. Але, в більшості випадків, вам не потрібно думати про ключі, оскільки React автоматично про них піклується. Головний випадок, коли потрібно використовувати ключі — під час рендерингу списків елементів.

Списки

Давайте розглянемо випадок рендерингу списку компонентів, використовуючи мапування масиву даних.

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import './style.css';

const data = [
    { id: 1, name: 'Coronis' },
    { id: 2, name: 'Zephyrus' },
    { id: 3, name: 'Urania' },
    { id: 4, name: 'Troilos' },
    { id: 5, name: 'Aeolus' },
];

function App() {
    return (
        <div>
            {data.map((item) => (
                <div className="card">{item.name}</div>
            ))}
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Під час запуску цього коду, він буде працювати, але ви отримаєте попередження, що не надано **«key»** для елементів списку.

Ключі дозволяють React відстежувати елементи. Таким чином, якщо елемент оновлюється або видаляється, буде перерендерений лише цей елемент, а не весь список. Ключі мають

бути унікальними для кожного сусіднього елемента. Але вони можуть бути дубльованими глобально.

Загалом, ключем має бути унікальний ідентифікатор, призначений кожному елементу. Як один із варіантів, можна використовувати індекс масиву як ключ.

Давайте переробимо попередній приклад, щоб додати ключі:

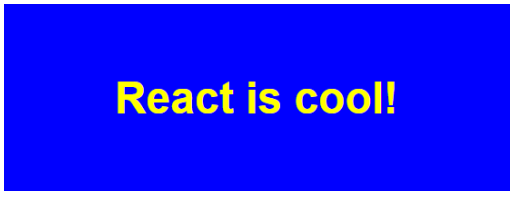
```
import React from 'react';
import { createRoot } from 'react-dom/client';
import './style.css';

const data = [
    { id: 1, name: 'Coronis' },
    { id: 2, name: 'Zephyrus' },
    { id: 3, name: 'Urania' },
    { id: 4, name: 'Troilos' },
    { id: 5, name: 'Aeolus' },
];

function App() {
    return (
        <div>
            {data.map((item) => (
                <div className="card" key={item.id}>{item.name}</div>
            ))}
        </div>
    );
}
const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Завдання

Завдання 1. Виведення тексту

Вивести текст «React Native is cool!» в центрі екрану, як показано на картинці.



Завдання зробити двома способами:

- Вбудовані стилі;
- CSS і імена класів.

Завдання 2. Вивід елементів масиву

Вивести всі імена з масиву users на екран.

```
const users = [
    { name: 'John Doe', id: 1 },
    { name: 'Jane Doe', id: 2 },
    { name: 'Billy Doe', id: 3 },
];
```

Додаткове завдання

Створити окремий компонент для відображення елементу списку. Компонент може бути оформлений як за допомогою класу так і функції, на Ваш розсуд. Допомога за посиланням.

Завдання 3. Показати/сховати елемент на екрані

Необхідно при натисканні на кнопку показати/сховати текст на екрані. Також змінити напис на кнопці з *«Hide Element Below»* на *«Show Element Below»*. Допомога за посиланням.

```
function App() {
    return (
        <div style={styles.container}>
            <button>Hide Element Below</button>
            <p>Toggle Challenge</p>
        </div>
    );
}
```

Завдання 4. Стан та властивості компоненту

Вивести, скільки разів користувач натиснув на кнопку.

```
function App() {
    return (
        <div style={styles.container}>
            <button>Click me</button>
            <p>Button has been clicked: </p>
        </div>
    );
}
```

Завдання 5. Доступність кнопки

Зробити кнопку недоступною, якщо в поле не введено жодного символу.

```
function App() {
    return (
        <div style={styles.container}>
            <p style={styles.title}>Task: accessibility of the button</p>
            <input type="text" style={styles.input} placeholder="" />
            <br />
            <br />
            <input type="submit" />
        </div>
    );
}
```

Запитання для самоконтролю

1. Що таке елемент у React і як він відрізняється від компонента?
2. Як створити елемент у React і які основні властивості в нього можна передати?
3. Як відрізнити простий елемент від компонента в React?
4. Як відрізнити функціональний компонент від класового компонента в React?
5. Як використовувати елемент або компонент у JSX коді?
6. Що таке властивості (props) в React і як вони використовуються в компонентах?
7. Як передати властивості з батьківського компонента до дочірнього компонента?
8. Які типи властивостей можна передавати в компоненти та як їх валідувати?

9. Як використовувати значення властивостей у компоненті?
10. Чи можна змінювати значення властивостей у компоненті та чому?
11. Як можна стилізувати компоненти в React?
12. Як використовувати CSS класи для стилізації компонентів?
13. Як використовувати внутрішні стилі (inline styles) для стилізації компонентів?
14. Як використовувати зовнішні бібліотеки для стилізації компонентів?
15. Як можна динамічно змінювати стилі компонентів у залежності від умов?
16. Як обробляти події в компонентах React?
17. Як прив'язати обробники подій до елементів у JSX коді?
18. Як передати дані або аргументи до обробника подій у React компонентах?
19. Як відключити або зупинити обробку події в React?
20. Що таке умовний рендеринг і коли він використовується в React компонентах?

Хуки (Hooks)

Хуки (hooks) — це новий механізм, який був введений у React версії 16.8. Вони дозволяють використовувати стан (**state**) та інші функціональні можливості React у компонентах, які були написані в функціональному стилі, без необхідності використання класових компонентів.

Хуки дають можливість компонентам React зберігати та маніпулювати станом, виконувати ефекти (side effects) та взаємодіяти з життєвим циклом компонентів, все це за допомогою простих функцій.

Екосистема React включає вбудовані хуки, такі як, **useState**, **useEffect**, **useContext** та багато інших. Також є можливість створювати власні хуки, які дозволяють повторно використовувати логіку між різними компонентами.

Хуки дозволяють писати більш компактний і зрозумілий код, спрощують тестування компонентів та сприяють використанню кращих практик у розробці React додатків. Вони стали одним з найпопулярніших інструментів у React екосистемі і значно покращили розробку функціональних компонентів..

Хук стану (useState)

Функціональні компоненти були використані в попередніх прикладах.

Приклади функціональних компонентів:

```
function Example(props) {
    // You can use hooks here!
    return <View />;
}
```

```
const Example = (props) => {
    // You can use hooks here!
    return <View />;
};
```

У своїй чистій формі такий компонент не має стану. Щоб додати стан, ви повинні використовувати класи або додати хуки.

Хук **useState** дозволяє «запам'ятовувати» значення в межах функції компонента. Оскільки функція компонента може бути викликана багато разів протягом життєвого циклу компонента, будь-яка змінна, яка оголошується звичайним способом (тобто за допомогою **let myVar = ...**), буде скинута. За допомогою **useState**, React може запам'ятовувати змінну стану, щоб вона правильно передавалася екземпляру компонента.

API

Хук useState приймає один аргумент — початковий стан і повертає масив, що містить два елементи:

- **state** — поточний стан;
- **setState** — функція, яка оновлює стан.

Наприклад:

```
const [state, setState] = useState(InitialValue)
```

Давайте використаємо useState для оновлення значення лічильника. Загалом, useState може зберігати будь-який тип значення: число, рядок, масив, об'єкт і т.д.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

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

    return (
        <div>
            <p>Counter: {count}</p>
            <button onClick={() => setCount((prevCount) => prevCount + 1)}>
                Increment
            </button>
            <button onClick={() => setCount(count - 1)}>Decrement</button>
            <button onClick={() => setCount(0)}>Reset</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Виклик функції **useState** зі значенням **0** повертає масив, який містить це початкове значення разом з функцією для його оновлення. За рахунок операції деструктирізації присвоюємо значення елементів цього масиву — змінним з назвами **count** і **setCount**.

Функції **setCount** можна передавати безпосередньо нове значення:

```
setCount(count - 1)}
```

або функцію, що приймає попереднє значення стану та повертає нове значення стану:

```
setCount((prevCount) => prevCount + 1)
```

У прикладі аргумент називається **prevCount**, але ви можете назвати його так, як вам подобається.

Приклад: використання useState з масивом

Пам'ятайте, стан може містити будь-який тип значення! Ось приклад використання **useState** з масивом.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

let numb = 1;

function App() {
    const [todos, setTodo] = useState([]);
    function AddTodo() {
        setTodo([...todos, `task ${numb++}`]);
    }
```

```
    return (
        <div>
            <button onClick={AddTodo}>Add ToDo</button>
            <ul>
                {todos.map((todo, i) => (
                    <li key={i}>{todo}</li>
                ))}
            </ul>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Починаємо з початкового стану **[]**, і після натискання **«Add ToDo»** нові значення додаються до списку.

Зверніть увагу, що коли ми викликаємо **setTodo**, ми не додаємо нове завдання до масиву **todos**, наприклад за допомогою команди *push*. Замість цього, передається новий масив, який містить деструктурований масив **todos** і нове завдання. *Чому?*

Хуки можуть повідомити React, про необхідність повторно запустити функцію компонента та оновити його інтерфейс. Хук **useState** повідомляє React про необхідність повторно запускати функції компонента, коли ми викликаємо **setTodo** з іншим значенням. Внутрішньо **useState** порівнює поточне значення **todos** зі значенням, яке ми передали в **setTodo** за допомогою оператора строгої рівності **«===»**. Якщо ми використовуємо змінну посилального типу, таку як масив, і змінюємо лише вміст цього масиву, то **useState** не виявить цю зміну і не сповістить React про необхідність повторного запуску функції компонента. Це призведе до того, що наш інтерфейс користувача відображатиме старі дані. Щоб уникнути цієї проблеми, нам потрібно створити новий масив, щоб **useState** виявив, що наші дані змінилися, і відобразив найновіші дані.

Якщо вам потрібно використовувати кілька змінних стану в функціональному компоненті, у вас є кілька варіантів:

- викликати useState більше одного разу;
- помістити все в об'єкт.

Немає нічого поганого в тому, щоб викликати useState кілька разів. Коли у вас вже є 4 або 5 викликів useState, це може бути трохи незручно.

Зовсім не обов'язково використовувати кілька змінних стану. Змінні стану можуть нормально зберігати об'єкти та масиви, тому ви можете групувати пов'язані дані разом. Однак, на відміну від *this.setState* в класі, оновлення змінної стану завжди замінює її.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function App() {
    const [counts, setCounts] = useState({
        countA: 0,
        countB: 0,
    });
```

```
    const incA = () =>
        setCounts({
            ...counts,
            countA: counts.countA + 1,
        });
    const incB = () =>
        setCounts({
            ...counts,
            countB: counts.countB + 1,
        });
    const badIncA = () =>
        setCounts({
            countA: counts.countA + 1,
        });

    return (
        <>
            <div>A: {counts.countA}</div>
            <div>B: {counts.countB}</div>
            <button onClick={incA}>Increment A</button>
            <button onClick={incB}>Increment B</button>
            <button onClick={badIncA}>Increment A Badly</button>
        </>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Хук useReducer

Якщо використовується складна логіка зміни стану, або стан має багато значень в якості альтернативи **useState**, можна використати хук **useReducer.**

API

Хук useReducer приймає два аргумента та третій необов'язковий:

- **reducer** — чиста функція, яка приймає стан і дію, та повертає нове значення стану на основі дії;
- **InitialState** — значення початкового стану, як і в useState;
- **init** (необов'язково) — функція, що викликається для лінивого створення екземпляра *InitialState*

і повертає масив, що містить два елементи: поточний стан та функцію *dispatch*, яка використовується для оновлення стану.

Наприклад:

```
const [state, dispatch] = useState(reducer, initialState, init)
```

Перепишемо попередній приклад з допомогою **useReducer** для оновлення значення лічильника. Під час натискання на кнопку **«Increment»** викликається функція **dispatch** з параметром **{type: 'ADD'}**. Цей параметр передається функції **reducer**, як **action**. Функція

reducer обробляє відповідний **case 'ADD'** та повертає оновлений **state**, який і буде відображений.

```
import React, { useReducer } from 'react';
import { createRoot } from 'react-dom/client';

const initialState = { counter: 0 };

function reducer(state, action) {
    switch (action.type) {
        case 'ADD':
            return { ...state, counter: state.counter + 1 };
        case 'SUB':
            return { ...state, counter: state.counter - 1 };
        case 'RESET':
            return initialState;
    }
    return state;
}

function App() {
    const [state, dispatch] = useReducer(reducer, initialState);

    return (
        <div>
            <p>Counter: {state.counter}</p>
            <button onClick={() => dispatch({ type: 'ADD' })}>Increment</button>
            <button onClick={() => dispatch({ type: 'SUB' })}>Decrement</button>
            <button onClick={() => dispatch({ type: 'RESET' })}>Reset</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Шаблони reducer

Хоча **useReducer** не застосовує жодних конкретних форм/шаблонів для станів (*state*) та дій (*action*), є кілька, які поширені при управлінні середніми/великими об'ємами даних:

- **Дії (actions)**, які передаються редуктору (*reducer*) — це об'єкти, що містять *type* і *payload*;
- Часто визначають константи для *type*, перераховуючи кожну дію, яку функція *reducer* вміє обробляти;
- **actionCreator** — функції, які абстрагують деталі об'єктів *actions* від решти програми;
- **Стан (state)** — це об'єкт, тому легко додавати до нього поля в міру зростання додатка.

Ці шаблони допомагають зберегти код *reducer* самостійним для кожного виду сутності в додатку (наприклад, завдання, публікації, фотографії).

Ці шаблони допомагають зберегти автономний код редуктора для кожного типу сутностей у нашій програмі (наприклад, завдань, публікацій, фотографій).

Приклад:

Додаємо до попереднього прикладу кнопки, що змінюють значення лічильника на задану величину. Цю величину можна передати через додаткове поле об'єкту **action**. Зазвичай це поле має ім'я **payload**, хоча можна використати будь-яку іншу назву. В обробнику нових кнопок викликаємо функцію **dispatch** та передаємо їй об'єкт з полем **type**, та полем **payload** з бажаним значенням зміни лічильника. В **reducer** додаємо обробник для нового **type**.

```
case 'ADD_NUMBER':
    return { ...state, counter: state.counter + action.payload };
```

```
import React, { useReducer } from 'react';
import { createRoot } from 'react-dom/client';

const initialState = { counter: 0 };

function reducer(state, action) {
    switch (action.type) {
        case 'ADD':
            return { ...state, counter: state.counter + 1 };
        case 'SUB':
            return { ...state, counter: state.counter - 1 };
        case 'ADD_NUMBER':
            return { ...state, counter: state.counter + action.payload };
        case 'RESET':
            return initialState;
    }
    return state;
}

function App() {
    const [state, dispatch] = useReducer(reducer, initialState);

    return (
        <div>
            <p>Counter: {state.counter}</p>
            <button onClick={() => dispatch({ type: 'ADD' })}>Increment</button>
            <button onClick={() => dispatch({ type: 'SUB' })}>Decrement</button>
            <button onClick={() => dispatch({ type: 'ADD_NUMBER', payload: 15 })}>
                Add 15
            </button>
            <button onClick={() => dispatch({ type: 'ADD_NUMBER', payload: -15 })}>
                Subtraction 15
            </button>
            <button onClick={() => dispatch({ type: 'RESET' })}>Reset</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Створення actions

Визначимо константи для опису типів дій, які функція **reducer** буде обробляти:

```
const types = {
    ADD: 'ADD',
    SUB: 'SUB',
    ADD_NUMBER: 'ADD_NUMBER',
```

```
    RESET: 'RESET',
};
```

Не обов'язково об'єднати константи в одному об'єкті, можна створити кожну окремо.

Перепишемо функцію **reducer** використовуючи створенні константи.

Додаємо об'єкт **actionCreators**, який буде містити функції для створення відповідних об'єктів **actions**. Що дозволить абстрагувати деталі об'єктів **actions** від решти програми. Це корисно тому що зазвичай в проектах відокремлюють в окремі файли презентаційні компоненти та файли логіки роботи, обробки даних, тощо.

```
const actionCreators = {
    add: () => ({ type: types.ADD }),
    sub: () => ({ type: types.SUB }),
    addNumber: (number) => ({ type: types.ADD_NUMBER, payload: number }),
    reset: () => ({ type: types.RESET })
};
```

У виклик функції **dispatch** можна вкласти виклик відповідної функції з **actionCreators**, не думаючи про деталі створення відповідного **actions**.

```
import React, { useReducer } from 'react';
import { createRoot } from 'react-dom/client';

const initialState = { counter: 0 };

const types = {
    ADD: 'ADD',
    SUB: 'SUB',
    ADD_NUMBER: 'ADD_NUMBER',
    RESET: 'RESET',
};

const actionCreators = {
    add: () => ({ type: types.ADD }),
    sub: () => ({ type: types.SUB }),
    addNumber: (number) => ({ type: types.ADD_NUMBER, payload: number }),
    reset: () => ({ type: types.RESET })
};

function reducer(state, action) {
    switch (action.type) {
        case 'ADD':
            return { ...state, counter: state.counter + 1 };
        case 'SUB':
            return { ...state, counter: state.counter - 1 };
        case 'ADD_NUMBER':
            return { ...state, counter: state.counter + action.payload };
        case 'RESET':
            return initialState;
    }
    return state;
}

function App() {
    const [state, dispatch] = useReducer(reducer, initialState);

    return (
        <div>
            <p>Counter: {state.counter}</p>
            <button onClick={()=> dispatch(actionCreators.add())}>Increment</button>
            <button onClick={()=> dispatch(actionCreators.sub())}>Decrement</button>
```

```
            <button onClick={()=> dispatch(actionCreators.addNumber(15))}>
                Add 15
            </button>
            <button onClick={() => dispatch(actionCreators.addNumber(-15))}>
                Subtraction 15
            </button>
            <button onClick={() => dispatch(actionCreators.reset())}>Reset</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Хук useEffect

Хук **useEffect** — це універсальний інструмент серед усіх хуків. Він є вирішенням багатьох проблем: як отримати дані під час монтування компонента, як запустити код при зміні стану або властивості, як налаштувати таймери або інтервали — можна перераховувати без кінця.

Хук **useEffect** використовується для виклику функцій з побічними ефектами всередині компонента.

API

Хук useEffect приймає 2 аргументи:

- **Callback (функція зворотного виклику)** — функція з побічними ефектами;
- **Залежності** — необов'язковий масив, що містить значення залежностей.

Коли наша функція компонента виконується, *callback* буде викликано, якщо які-небудь залежності змінилися з часу останнього виконання функції компонента.

Приклад:

Тут ми використовуємо **useEffect**, щоб змінювати колір фону при зміні **count1**. Кожного разу, коли змінюється **count1**, викликається **callback**, оскільки **count1** вказано як залежність.

```
import React, { useState, useEffect } from 'react';
import { createRoot } from 'react-dom/client';

function getRandomHexColor() {
    return `#${Math.floor(Math.random() * 16777215)
        .toString(16)
        .padStart(6, 0)}`;
}

function App() {
    const [count1, setCount1] = useState(0);
    const [count2, setCount2] = useState(0);

    useEffect(() => {
        document.body.style.backgroundColor = getRandomHexColor();
    }, [count1]);

    return (
        <div>
            <p>Counter: {count1}</p>
```

```
            <button onClick={() => setCount1(count1 + 1)}>Increment count 1</button>
            <p>Counter: {count2}</p>
            <button onClick={() => setCount2(count2 + 1)}>Increment count 2</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Undefined або порожній масив залежностей

Якщо масив залежностей *порожній* або *невизначений (undefined)*, useEffect поводитиметься інакше.

Порожній масив [] — функція зворотного виклику викликається лише один раз, відразу після першого рендерингу компонента.

undefined — функція зворотного виклику викликається під час кожного рендеру компонента (щоразу, коли виконується функція компонента).

undefined

Якщо масив залежностей не визначений, функція зворотного виклику запускатиметься щоразу, коли запускатиметься функція компонента, наприклад, кожного разу, коли ми натискаємо кнопку, а **useState** повідомляє нашому компоненту про повторний запуск.

```
import React, { useState, useEffect } from 'react';
import { createRoot } from 'react-dom/client';

function getRandomHexColor() {
    return `#${Math.floor(Math.random() * 16777215)
        .toString(16)
        .padStart(6, 0)}`;
}

function App() {
    const [count1, setCount1] = useState(0);
    const [count2, setCount2] = useState(0);

    useEffect(() => {
        document.body.style.backgroundColor = getRandomHexColor();
    });

    return (
        <div>
            <p>Counter: {count1}</p>
            <button onClick={() => setCount1(count1 + 1)}>Increment count 1</button>
            <p>Counter: {count2}</p>
            <button onClick={() => setCount2(count2 + 1)}>Increment count 2</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Порожній масив []

Якщо масив залежностей є порожнім, функція зворотного виклику викликається лише один раз (і постійно встановлює колір фону сторінки).

```
import React, { useState, useEffect } from 'react';
import { createRoot } from 'react-dom/client';

function getRandomHexColor() {
    return `#${Math.floor(Math.random() * 16777215)
        .toString(16)
        .padStart(6, 0)}`;
}

function App() {
    const [count1, setCount1] = useState(0);
    const [count2, setCount2] = useState(0);

    useEffect(() => {
        document.body.style.backgroundColor = getRandomHexColor();
    }, []);

    return (
        <div>
            <p>Counter: {count1}</p>
            <button onClick={() => setCount1(count1 + 1)}>Increment count 1</button>
            <p>Counter: {count2}</p>
            <button onClick={() => setCount2(count2 + 1)}>Increment count 2</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Хук useMemo

Хук **useMemo** використовується для оптимізації обчислень у React компонентах. Час від часу, компонентам потрібно виконувати складні обчислення. Наприклад, при наявності великого списку працівників та запиту на пошук, компонент повинен фільтрувати імена працівників за запитом.

У таких випадках, з використанням обережності, ви можете спробувати покращити продуктивність своїх компонентів за допомогою техніки *мемоізації*.

API

Хук useMemo приймає 2 аргументи:

- **compute** — функція, яка обчислює результат;
- **dependencies** — масив, що містить значення залежностей.

```
const memoizedResult = useMemo(compute, dependencies);
```

Під час початкового рендерингу, **useMemo(compute, dependencies)** викликає функцію **compute**, мемоізує результат обчислення і повертає його компоненту.

Якщо під час наступних рендерів залежності не змінилися, то **useMemo()** не викликає функцію **compute**, а повертає запам'ятоване значення.

Але якщо залежності змінюються під час повторного рендерингу, то **useMemo()** викликає функцію **compute**, мемоізує нове значення і повертає його.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function fib(n) {
    console.log('fib(n) called!');
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

function App() {
    const [count, setCount] = useState(0);
    const [numb, setNumb] = useState(0);

    const fibNumb = fib(numb);

    return (
        <div>
            Fibonacci number of
            <input
                type="number"
                min="0"
                max="10"
                value={numb}
                onChange={(e) => setNumb(e.target.valueAsNumber)}
            />
            is {fibNumb}
            <hr />
            <p>Counter: {count}</p>
            <button onClick={() => setCount(count + 1)}>Increment count</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Кожного разу, коли змінюється значення в полі введення, обчислюється число Фібоначчі *fib(n)*, і в консоль виводиться повідомлення *'fib(n) called!'*.

З іншого боку, кожного разу, коли натискається кнопка **«Increment count»**, значення стану **count** оновлюється. Оновлення значення стану **count** спричиняє повторний рендеринг компонента. Але як побічний ефект, під час повторного рендерингу обчислюється число Фібоначчі знову — в консоль виводиться повідомлення *'fib(n) called!'*.

Можна мемоізувати обчислення числа Фібоначчі під час повторного рендерингу компонента, використовуючи хук **useMemo()**.

Замість простого виклику **fib(numb)**, використовуйте **useMemo(() => fib(numb), [numb])**. Таким чином, React мемоізує обчислення числа Фібоначчі.

```
import React, { useState, useMemo } from 'react';
import { createRoot } from 'react-dom/client';
```

```
function fib(n) {
    console.log('fib(n) called!');
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

function App() {
    const [count, setCount] = useState(0);
    const [numb, setNumb] = useState(0);

    const fibNumb = useMemo(() => fib(numb), [numb]);

    return (
        <div>
            Fibonacci number of
            <input
                type="number"
                min="0"
                max="10"
                value={numb}
                onChange={(e) => setNumb(e.target.valueAsNumber)}
            />
            is {fibNumb}
            <hr />
            <p>Counter: {count}</p>
            <button onClick={() => setCount(count + 1)}>Increment count</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Кожного разу, коли змінюється значення введення, в консоль виводиться повідомлення *'fib(n) called!'*, що є очікуваним.

Однак, якщо ви натискаєте кнопку **«Increment count»**, в консоль не виводиться повідомлення *'fib(n) called!'*, оскільки **useMemo(() => fib(numb), [numb])** мемоізує обчислення числа Фібоначчі.

Це стає можливим завдяки використанню хука **useMemo()**. Під час повторного рендерингу компонента, якщо значення **numb** (залежність) залишається незмінним, **useMemo()** повертає закешоване (збережене) значення обчислення числа Фібоначчі, не викликаючи функцію **fib()**. Таким чином, повторне обчислення числа Фібоначчі при оновленні значення **count** не відбувається, і повідомлення *'fib(n) called!'* не виводиться в консоль.

Хук useCallback

Хук **useCallback** в React використовується для мемоізації функцій. Він дозволяє повертати ту саму функцію протягом життєвого циклу компонента, якщо залежності не змінюються. Якщо залежності змінюються, хук **useCallback** повертає нову функцію.

Основна мета використання **useCallback** полягає у покращенні продуктивності шляхом запобігання непотрібним повторним рендерам функцій. При мемоізації функцій з допомогою **useCallback**, React може порівнювати функції за посиланням (**===**) замість глибокого порівняння їх значень. Це особливо корисно, коли функції передаються як

властивості (props) в дочірні компоненти, оскільки це дозволяє уникнути зайвих рендерів цих компонентів при зміні функцій-пропсів.

API

Хук useCallback приймає 2 аргументи:

- **callback** — це функція, яку бажано мемоізувати;
- **dependencies** — масив залежностей, які впливають на функцію. Якщо будь-яка залежність змінюється, буде повернена нова функція.

```
const memoizedCallback = useCallback(callback, dependencies);
```

Під час використання **useCallback** важливо зазначити правильні залежності, щоб забезпечити вірне функціонування мемоізації. Якщо залежності не вказані, функція буде мемоізована і буде повертатися та ж сама функція протягом всього життєвого циклу компонента.

Приклад:

У наступному прикладі обчислюємо кількість разів, коли наш компонент **Logger** запускається. Оскільки **Logger** обгорнутий **memo**, він буде запускатися тільки тоді, коли змінюються його властивості (props). У випадку з **normalFunction**, функція змінюється кожного разу, коли ми натискаємо кнопку. У випадку з **memoizedFunction**, повторно використовується та сама функція протягом 5 натисків кнопки, оскільки змінна **count5** повертає одне й те ж значення 5 разів поспіль.

```
import React, { useState, memo, useCallback } from 'react';
import { createRoot } from 'react-dom/client';

const Logger = memo((props) => {
    props.log();
    return null;
});

function App() {
    const [count, setCount] = useState(0);
    const count5 = Math.floor(count / 5);

    const memoizedFunction = useCallback(() => {
        console.log('useCallback');
    }, [count5]);

    const normalFunction = () => {
        console.log('normal');
    };

    return (
        <div>
            <p>Counter: {count}</p>
            <button onClick={() => setCount(count + 1)}>Increment count</button>
            <Logger log={memoizedFunction} />
            <Logger log={normalFunction} />
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Хук useRef

Хук **useRef** в React використовується для створення посилань на елементи DOM або інші значення, які зберігаються протягом життєвого циклу компонента.

Основна особливість хука **useRef** полягає в тому, що він дозволяє отримувати доступ до елементів DOM без потреби повторного рендерингу компонента. Коли ви створюєте посилання за допомогою **useRef**, воно залишається постійним навіть після зміни стану компонента або повторного рендерингу.

API

Синтаксис використання хука useRef наступний:

```
const reference = useRef(initialValue);
```

initialValue — необов'язковий параметр, який встановлює початкове значення посилання. Це може бути будь-яке значення, яке ви хочете зберегти.

Хук **useRef** повертає посилання (відоме як **ref**). Посилання — це об'єкт з особливим властивістю **current**.

reference.current дозволяє отримати значення посилання, а **reference.current = newValue** оновлює значення посилання.

Є дві правила, які варто запам'ятати про посилання:

- Значення посилання зберігається між перерендерингами компонента (залишається незмінним);
- Оновлення посилання не призводить до перерендерингу компонента.

Приклад:

Використовуйте посилання для збереження кількості натискань на кнопку.

```
import React, { useRef } from 'react';
import { createRoot } from 'react-dom/client';

function App() {
    const countRef = useRef(0);

    const handle = () => {
        countRef.current++;
        console.log(`Clicked ${countRef.current} times`);
    };
    console.log('I rendered!');
    return <button onClick={handle}>Click me</button>;
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

const countRef = useRef(0); Створюється посилання **countRef**, ініціалізоване значенням **0**.

Коли кнопка натискається, викликається функція **handle**, і значення посилання збільшується: **countRef.current++**. Значення посилання виводиться в консоль.

Оновлення значення посилання **countRef.current++** не викликає повторний рендеринг компонента. Повідомлення *«I rendered!»* виводиться в консоль лише один раз, при початковому рендерингу, і при оновленні значення посилання повторний рендеринг не відбувається.

Давайте перепишемо попередній приклад, використовуючи хук ***useState()*** для підрахунку кількості натискань на кнопку:

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function App() {
    const [count, setCount] = useState(0);
    const handle = () => {
        const updatedCount = count + 1;
        console.log(`Clicked ${updatedCount} times`);
        setCount(updatedCount);
    };
    console.log('I rendered!');
    return <button onClick={handle}>Click me</button>;
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Кожен раз, коли ви натискаєте, у консолі з'являється повідомлення *«I rendered!»* — це означає, що кожного разу, коли стан оновлюється, компонент рендериться.

Таким чином, основні відмінності між *посиланнями (references)* і *станами (states)* такі:

- Оновлення посилання не викликає повторний рендеринг, тоді як оновлення стану призводить до повторного рендерингу компонента;
- Оновлення посилання є синхронним (оновлене значення посилання доступне безпосередньо), тоді як оновлення стану є асинхронним (змінна стану оновлюється після повторного рендерингу).

Приклад:

Також можна зберігати інфраструктурні дані побічних ефектів в посиланні. Наприклад, можна зберігати в посиланні ідентифікатори таймерів, ідентифікатори сокетів, тощо.

У наступному прикладі використовується функція таймера **setInterval(callback, time)**, щоб збільшувати лічильник секундоміра кожну секунду. Ідентифікатор таймера зберігається в посиланні **intervalRef**.

```
import React, { useRef, useState, useEffect } from 'react';
import { createRoot } from 'react-dom/client';
function App() {
    const intervalRef = useRef();
    const [count, setCount] = useState(0);

    useEffect(() => {
```

```
        intervalRef.current = setInterval(
            () => setCount((count) => count + 1),
            1000
        );
    }, []);

    return (
        <div style={{ textAlign: 'center' }}>
            <h1>{count}</h1>
            <button
                onClick={() => {
                    clearInterval(intervalRef.current);
                }}
            >
                Stop
            </button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Ще одне корисне застосування хука **useRef()** — це отримання доступу до DOM-елементів. Це виконується в 3 кроки:

1. Визначається посилання для доступу до елемента: **const elementRef = useRef()**;
2. Посилання присвоюється атрибуту **ref** елемента: ** **;
3. Після монтування, **elementRef.current** вказує на відповідний DOM-елемент.

Приклад:

Можливо знадобитися доступ до DOM-елементів, наприклад, для фокусування на полі введення під час монтування компонента.

Для досягнення цього маємо створити посилання, присвоїти це посилання атрибуту **ref** тегу **input**, а після монтування викликати спеціальний метод **element.focus()** на елементі.

```
import React, { useRef, useEffect } from 'react';
import { createRoot } from 'react-dom/client';

function App() {
    const inputRef = useRef();

    useEffect(() => {
        inputRef.current.focus();
    }, []);

    return (
        <div>
            <input ref={inputRef} />
        </div>
    );
}
const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Хук useContext

Хук **useContext** використовується для отримання значення контексту у компоненті. Контекст в React дозволяє передавати дані компонентам незалежно від того, на якій глибині вони знаходяться в дереві компонентів. Контекст використовується для керування глобальними даними, такими як глобальний стан, тема, сервіси, налаштування користувача та багато іншого.

Хук **useContext** повертає значення контексту: **value = useContext(Context)**. Цей хук також забезпечує повторний рендеринг компонента, коли значення контексту змінюється.

Основна ідея використання контексту полягає в тому, щоб дозволити компонентам отримувати доступ до глобальних даних і робити повторний рендеринг, коли ці глобальні дані змінюються. Контекст допомагає вирішити проблему передачі props вглиб компонентів *(props drilling problem)*.

У контексті можна зберігати наступні дані:

- глобальний стан;
- тему;
- конфігурацію додатку;
- ім'я аутентифікованого користувача;
- налаштування користувача;
- бажану мову;
- колекцію сервісів.

Однак перед використанням контексту варто обдумати його застосування у додатку.

По-перше, використання контексту додає складності. Створення контексту, обгортання всього в провайдер, використання **useContext()** у кожному споживачі — збільшує складність коду.

По-друге, використання контексту ускладнює модульне тестування компонентів. Під час модульного тестування доведеться обгортати компоненти-споживачі в провайдер контексту. Це також включає компоненти, які не безпосередньо використовують контекст, але залежать від нього — предки компонентів-споживачів!

Приклад без використання контексту:

Найпростіший спосіб передати дані від батьківського до дочірнього компонента полягає у тому, що батько передає властивості (props) своєму дочірньому компоненту.

```
function MyComponent(props) {
    return (
        <div>
            <button>{props.text}</button>
        </div>
    );
}

function App() {
    return <MyComponent text="JSX" />;
}
```

Це звичайний спосіб передачі даних за допомогою властивостей (props) і можна використовувати цей підхід без проблем.

Ситуація змінюється, коли вкладений компонент не є безпосередньо дочірнім.

Тоді доведеться передавати дані як властивість (props) у кожен компонент, навіть у ті, які їх не використовують! Це не тільки складно, але й може призвести до непотрібного повторного рендерингу.

Наприклад:

```
import React from 'react';
import { createRoot } from 'react-dom/client';

function Title({ theme }) {
    const style = {
        background: theme.primary,
        color: theme.text,
        textAlign: theme.align,
    };

    return <h1 style={style}>Title</h1>;
}

function Nested({ theme }) {
    return <Title theme={theme} />;
}

function NestedTwice({ theme }) {
    return <Nested theme={theme} />;
}

function App() {
    const theme = {
        primary: 'blue',
        text: 'yellow',
        align: 'center',
    };
    return <NestedTwice theme={theme} />;
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Приклад з використанням контексту:

Використання контексту в React передбачає 3 кроки:

1. Створення контексту

Вбудована функція-фабрика **createContext(default)** створює екземпляр контексту:

```
import { createContext } from 'react';
const Context = createContext('Default Value');
```

Функція-фабрика приймає один необов'язковий аргумент: значення за замовчуванням.

2. Надання контексту

Компонент **Context.Provider**, доступний на екземплярі контексту, використовується для надання контексту своїм дочірнім компонентам, незалежно від того, на якій глибині вони знаходяться.

Для встановлення значення контексту використовуйте властивість **value** на **<Context.Provider value={value} />:**

```
function App() {
    const value = 'My Context Value';
    return (
        <Context.Provider value={value}>
            <MyComponent />
        </Context.Provider>
    );
}
```

Якщо необхідно змінити значення контексту, просто оновлюємо властивість **value**.

3. Використання контексту

Хук повертає значення контексту: **value = useContext(Context)**. Крім того, хук забезпечує повторний рендеринг компонента, коли значення контексту змінюється.

```
import { useContext } from 'react';
function MyComponent() {
    const value = useContext(Context);
    return <span>{value}</span>;
}
```

Наприклад, компонент, який потребує використання теми, може отримати доступ до неї безпосередньо з контексту.

```
import React, { useContext, createContext } from 'react';
import { createRoot } from 'react-dom/client';

const ThemeContext = createContext();

function Title() {
    const theme = useContext(ThemeContext);
    const style = {
        background: theme.primary,
        color: theme.text,
        textAlign: theme.align,
    };

    return <h1 style={style}>Title</h1>;
}

function Nested() {
    return <Title />;
}

function NestedTwice() {
    return <Nested />;
}

function App() {
    const theme = {
        primary: 'blue',
        text: 'yellow',
```

```
        align: 'center',
    };
    return (
        <ThemeContext.Provider value={theme}>
            <NestedTwice />
        </ThemeContext.Provider>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Приклад зміни контексту:

Коли значення контексту змінюється шляхом зміни властивості **value** провайдера контексту **<Context.Provider value={value} />**, всі його споживачі отримують сповіщення і повторно рендеряться.

```
import React, { useContext, createContext, useEffect, useState } from 'react';
import { createRoot } from 'react-dom/client';

const ThemeContext = createContext();

function Title() {
    const theme = useContext(ThemeContext);
    const style = {
        background: theme.primary,
        color: theme.text,
        textAlign: theme.align,
    };

    return <h1 style={style}>Title</h1>;
}

function Nested() {
    return <Title />;
}

function NestedTwice() {
    return <Nested />;
}

function App() {
    const [theme, setTheme] = useState({
        primary: 'blue',
        text: 'yellow',
        align: 'center',
    });

    useEffect(() => {
        setTimeout(() => {
            setTheme({ ...theme, text: 'blue', primary: 'yellow' });
        }, 3000);
    }, []);
    return (
        <ThemeContext.Provider value={theme}>
            <NestedTwice />
        </ThemeContext.Provider>
    );
}
```

```
const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

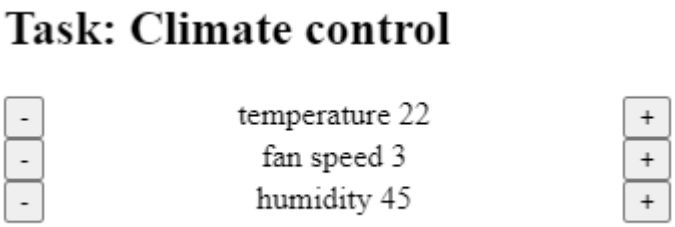
Завдання

Завдання 1. Кидання кубику

При натисканні на кнопку повинно відбуватися симуляція кидка кубика. Тобто потрібно викликати функцію, яка поверне випадкове число в діапазоні від 1 до 6. Результат має бути збережений в масиві та відображений на екрані. Використовуйте хук **useState** для розв'язання задачі.

Завдання 2. Клімат-контроль

Створіть додаток «Клімат-контроль» з трьома станами: «температура», «швидкість вентилятора» та «вологість».



Кожне значення відображається разом з міткою і парою кнопок **«+»** та **«-»**, для збільшення або зменшення значення.

При натисканні на кнопку **«+»** відповідне значення має збільшуватися, а при натисканні на кнопку **«-»** відповідне значення має зменшуватися.

Використовуйте хук **useState** для вирішення задачі.

Створіть дві окремі версії цього додатку:

- Значення мають бути збереженими в окремих змінних;
- Значення мають бути збереженими разом у одній змінній стану, у форматі об'єкта.

Завдання 3. Блокнот

Створіть програму «Блокнот», щоб створювати та зберігати нотатки. Додаток повинен:

- Відображати всі збережені нотатки;
- Дозволяти додавати нову нотатку;
- Видаляти всі наявні нотатки.

Використовуйте хук **useState** для управління станом. Для управління даними використовуйте хук **useReducer**.

Завдання 4. useMemo

Дано наступний код:

```jsx
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function factorial(n) {
    console.log(`factorial(${n}) called!`);
    return n <= 0 ? 1 : n * factorial(n - 1);
}

function App() {
    const [count, setCount] = useState(0);
    const [numb, setNumb] = useState(0);

    const factorialValue = factorial(numb);

    return (
        <div>
            <h2>Task: useMemo</h2>
            Factorial number of
            <input
                type="number"
                min="0"
                value={numb}
                onChange={(e) => setNumb(e.target.valueAsNumber)}
            />
            is {factorialValue}
            <hr />
            <p>Counter: {count}</p>
            <button onClick={() => setCount(count + 1)}>Increment count</button>
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

Змініть додаток так, щоб сторінка відображалася швидше, коли ви натискаєте кнопку **«Increment count»** (збільшення лічильника).

Кроки для перевірки правильності відповіді:

1. Введіть яке-небудь число;
2. Почекайте, поки зміниться вивід;
3. Клацніть кнопку **«Increment count»**;
4. У консолі не повинно виводитись нічого.

Завдання 5. useCallback

Дано наступний код:

```jsx
import React, { useState, useEffect } from 'react';
import { createRoot } from 'react-dom/client';

function List({ getItems }) {
    const [items, setItems] = useState([]);

    useEffect(() => {
        console.log('Rerendering this');
        setItems(getItems());
    }, [getItems]);

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

```
function App() {
    const [number, setNumber] = useState(0);
    const [dark, setDark] = useState(false);

    const getItems = () => {
        const defaultValueIfNumberIsNaN = Number.isNaN(number) ? 0 : number;
        return [
            defaultValueIfNumberIsNaN,
            defaultValueIfNumberIsNaN + 1,
            defaultValueIfNumberIsNaN + 2,
        ];
    };

    const theme = {
        backgroundColor: dark ? 'black' : 'white',
        color: dark ? 'white' : 'black',
    };

    return (
        <div style={theme}>
            <h2>Task: useCallback</h2>
            <input
                type="number"
                value={number}
                onChange={(e) => setNumber(parseInt(e.target.value, 10))}
            />
            <button onClick={() => setDark(!dark)}>Toggle</button>
            <List getItems={getItems} />
        </div>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

Необхідно змінити програму так, щоб натискання кнопки перемикача не призводило до виводу **«Rendering this»** у консоль.

Завдання 6. useRef

Дано наступний код (https://stackblitz.com/edit/react-eu6g6q?file=src%2Findex.js)

```
import React from 'react';
import { createRoot } from 'react-dom/client';

function App() {
    const handleClick1 = () => {
        // complete this
    };

    const handleClick2 = () => {
        // complete this
    };

    return (
        <>
            <h2>Task: useRef</h2>
            <input /> <button onClick={handleClick1}>Focus me</button>
            <br />
            <br />
            <input /> <button onClick={handleClick2}>Focus me</button>
        </>
```

```
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

Реалізуйте функції **«handleClick1»** і **«handleClick2»** таким чином, щоб під час натискання відповідної кнопки у фокусі був відповідний елемент введення.

Завдання 7. useContext

Дано наступний код:

```
import React from 'react';
import { createRoot } from 'react-dom/client';

function NumberDisplay() {
    return <p>The number in the context is 25</p>;
}

function App() {
    return (
        <>
            <h2>Task: useContext</h2>
            <NumberDisplay />
        </>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

Використайте хук **useContext** для передачі значення *25* в компонент *«NumberDisplay»*.

Завдання 8. Pomodoro таймер

Перед початком написання будь-якого коду вам потрібно зрозуміти обсяг майбутньої роботи, спробувати знайти спільну (загальну) логіку між компонентами і з'ясувати, як написати чистий і ефективний код.

Розбивши веб-сторінку на компоненти, ви готові розпочати написання коду.

Необхідно створити програму «Pomodoro-таймер», яка допомагатиме людям використовувати Техніку Pomodoro. Вона буде повідомляти користувача, коли робити перерви або продовжувати роботу, на основі заданих значень часу роботи і відпочинку.

Вимоги:

- Таймер повинен показувати хвилини і секунди;
- Таймер повинен відраховувати секунди, поки не досягне 00:00;
- Таймери повинні перемикатися між 25 хвилинами роботи і 5 хвилинами відпочинку;
- У таймера повинні бути кнопки запуску, зупинки та скидання;
- Додана можливість встановлювати будь-який час для таймерів (наприклад, 5 хвилин робочого часу і 5 хвилин перерви);
- Використовуйте хуки **useState, useRef, useEffect** та інші за необхідністю;
- Графічний дизайн інтерфейсу залишається на ваш розсуд.

Запитання для самоконтролю

1. Що таке хуки (hooks) в React і для чого вони використовуються?
2. Які основні вбудовані хуки є в React і які їх функції?
3. Як використовувати useState хук для управління станом у компонентах?
4. Як використовувати useEffect хук для виконання побічних ефектів в компонентах?
5. Як використовувати useContext хук для отримання значення контексту в компонентах?
6. Як використовувати useReducer хук для управління складним станом у компонентах?
7. Як використовувати useRef хук для отримання посилання на DOM елемент у компонентах?
8. Як використовувати useMemo хук для оптимізації обчислень в компонентах?
9. Як використовувати useCallback хук для кешування функцій в компонентах?

Патерни React

Патерни React — це практичні прийоми і методики, які використовуються під час розробки програм на основі бібліотеки React. Вони допомагають структурувати і організувати код, спрощують розробку, поліпшують читабельність і підтримку програми. Патерни React охоплюють широкий спектр аспектів, включаючи управління станом, управління компонентами, обробку подій, маршрутизацію, зв'язування даних і багато іншого.

Повернення кількох елементів

У React компоненти часто повертають кілька елементів. Зазвичай ці елементи обгортаються, наприклад, у **div**. У більшості випадків, обгортка **div** є *«непотрібною»* і додається лише через те, що React компоненти вимагають повертати лише один елемент. Така поведінка призводить до зайвого розмітки і, іноді, навіть до відображення недійсного HTML, що є неприпустимим.

Наприклад:

```
function Table() {
    return (
        <table>
            <tr>
                <Columns />
            </tr>
        </table>
    );
}

function Columns() {
    return (
        <div>
            <td>Hello</td>
            <td>World</td>
        </div>
    );
}
```

Це може призвести до відображення недійсного HTML.

```
<table>
    <tr>
        <div>
            <td>Hello</td>
            <td>World</td>
        </div>
    </tr>
</table>
```

React.Fragments

Фрагменти у React дозволяють групувати список дочірніх елементів без додавання додаткових вузлів до DOM, оскільки фрагменти не відображаються у DOM.

Ми можемо використовувати фрагменти за допомогою синтаксису **<React.Fragments>**. Таким чином, ми можемо записати компонент **Columns** наступним чином.

```
function Columns() {
    return (
        <React.Fragment>
            <td>Hello</td>
            <td>World</td>
        </React.Fragment>
    );
}
```

Тепер компонент **Table** відображатиме наступний HTML-код.

```
<table>
    <tr>
        <td>Hello</td>
        <td>World</td>
    </tr>
</table>
```

Приклад:

Часто потрібно повернути кілька елементів верхнього рівня з компонента, і ми можемо повернути **React.Fragment**. Фрагменти не додають додаткової розмітки в DOM (тобто не потрібно використовувати **div**, оскільки не потрібен додатковий елемент DOM).

Фрагменти можна використовувати за допомогою синтаксису на два способи. Перший використовує **React.Fragment**. Інший — новий скорочений синтаксис, використовуючи пусті дужки, який робить те саме.

Обидва синтаксиси працюють однаково. Однак, новіший синтаксис не підтримує ключі або атрибути.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';
import './style.css';

const DivComponent = ({ color }) => (
    <div>
        <p style={{ backgroundColor: color }}>with DIV</p>
        <hr />
    </div>
);
const FragmentComponent = ({ color }) => (
    <React.Fragment>
        <p style={{ backgroundColor: color }}>with React.Fragment</p>
        <hr />
    </React.Fragment>
);
const ShortSyntaxComponent = ({ color }) => (
    <>
        <p style={{ backgroundColor: color }}>Short Syntax</p>
        <hr />
    </>
);
function App() {
    return (
        <>
            <DivComponent color={'red'} />
            <FragmentComponent color={'yellow'} />
            <ShortSyntaxComponent color={'lightgreen'} />
        </>
    );
}
```

```
const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Довільні дочірні елементи

Іноді може знадобитись передавати довільні дочірні елементи в компонент, що дозволяє більшу гнучкість у його використанні. У React це можна зробити, використовуючи спеціальну властивість під назвою **«children»**.

Створення кнопки, що перевикористовується

Припустимо, потрібно створити перевикористовуваний компонент **Button**, якому можемо передавати вміст всередину. Один з підходів полягає у використанні властивості (props) **title** типу *string*:

```
import React from 'react';
import { createRoot } from 'react-dom/client';

function Button({ title, onClick }) {
    return <button onClick={onClick}>{title}</button>;
}

function App() {
    return (
        <>
            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
                title="Coffee Buy now"
            />

            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
                title="Hamburger Buy Later"
            />

            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
                title="Pizza Order"
            />
        </>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Проте це обмежує те, що можна передавати у компонент. Наприклад, не можливо передати *<svg>* іконку.

ReactNode

Ми можемо передавати довільні елементи React, використовуючи тип **ReactNode** як тип властивості **title**.

```
import React from 'react';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
import {
    faCoffee,
    faHamburger,
    faPizzaSlice,
} from '@fortawesome/free-solid-svg-icons';
import { createRoot } from 'react-dom/client';

function Button({ title, onClick }) {
    return <button onClick={onClick}>{title}</button>;
}

function App() {
    return (
        <>
            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
                title={
                    <>
                        <FontAwesomeIcon icon={faCoffee} />
                        <span> Buy now </span>
                    </>
                }
            />

            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
                title={
                    <>
                        <FontAwesomeIcon icon={faHamburger} />
                        <span> Buy Later </span>
                    </>
                }
            />

            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
                title={
                    <>
                        <FontAwesomeIcon icon={faPizzaSlice} />
                        <span> Order </span>
                    </>
                }
            />
        </>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Children

У React можна передавати дочірні елементи до компонента, вкладаючи їх усередину його **JSX**-тегу. Ці елементи (може бути нуль, один або більше) доступні всередині компонента як властивість з назвою **children**:

```
import React from 'react';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
import {
    faCoffee,
    faHamburger,
    faPizzaSlice,
} from '@fortawesome/free-solid-svg-icons';

import { createRoot } from 'react-dom/client';

function Button({ children, onClick }) {
    return <button onClick={onClick}>{children}</button>;
}

function App() {
    return (
        <>
            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
            >
                <FontAwesomeIcon icon={faCoffee} />
                <span> Buy now </span>
            </Button>
            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
            >
                <FontAwesomeIcon icon={faHamburger} />
                <span> Buy Later </span>
            </Button>
            <Button
                onClick={() => {
                    console.log('clicked!');
                }}
            >
                <FontAwesomeIcon icon={faPizzaSlice} />
                <span> Order </span>
            </Button>
        </>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Візуалізація властивостей (Render Props)

Деякі компоненти підтримують кілька дочірніх елементів певного типу, а не будь-якого типу. Наприклад, компонент вкладок акордеона приймає кілька дочірніх елементів, але відображає лише один елемент одночасно. Розглянемо кілька способів реалізації компонента Tabs.

Передача об'єктів

Можна створити об'єкт для кожної вкладки і передати масив цих об'єктів у компонент:

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function Tabs({ tabs }) {
    const [selectedId, setSelectedId] = useState(tabs[0].id);

    return (
        <>
            {tabs.map(({ id, title }) => (
                <button
                    key={id}
                    onClick={() => {
                        setSelectedId(id);
                    }}
                    style={{
                        border: 'none',
                        background: id === selectedId ? '#82BADE' : 'transparent',
                    }}
                >
                    {title}
                </button>
            ))}
            <hr />
            <div>{tabs.find((tab) => tab.id === selectedId).content}</div>
        </>
    );
}

function App() {
    return (
        <Tabs
            tabs={[
                { id: 'a', title: 'Tab A', content: 'Tab content A' },
                { id: 'b', title: 'Tab B', content: 'Tab content B' },
                { id: 'c', title: 'Tab C', content: 'Tab content C' },
            ]}
        />
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Перевагою цього підходу є відсутність додаткового рівня абстракції. Недолік полягає в тому, що потрібно створювати елементи React (властивість **content**) для кожного об'єкту вкладки, а не лише для видимої вкладки. Іншими словами, робиться певна кількість роботи, яка не використовується. В більшості випадків вплив на продуктивність буде незначним, але якщо

ви працюєте з великим набором вкладок (або, скоріше, з компонентом списку будь-якого виду), вам може знадобитись додаткова оптимізація.

Render props

Render Props (візуалізація властивостей) — це підхід у React, який дозволяє передавати функцію як властивість до компонента, що дозволяє компоненту викликати цю функцію для отримання необхідних даних або елементів UI.

За допомогою Render Props, компонент може створювати елементи React при потребі для різних частин інтерфейсу, замість передачі всіх даних заздалегідь через властивостей (props).

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

function Tabs({ tabIds, renderTitle, renderContent }) {
    const [selectedId, setSelectedId] = useState(tabIds[0]);

    return (
        <>
            {tabIds.map((id) => (
                <button
                    key={id}
                    onClick={() => {
                        setSelectedId(id);
                    }}
                    style={{
                        border: 'none',
                        background: id === selectedId ? '#E30613' : 'transparent',
                    }}
                >
                    {renderTitle(id)}
                </button>
            ))}
            <hr />
            <div>{renderContent(selectedId)}</div>
        </>
    );
}

function App() {
    return (
        <Tabs
            tabIds={['a', 'b', 'c']}
            renderTitle={(id) => `Tab ${id.toUpperCase()}`}
            renderContent={(id) => `Tab content ${id.toUpperCase()}`}
        />
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Контрольовані компоненти

Контрольовані компоненти очікують, що їхній стан буде переданий зовнішньо від їхнього батьківського компонента, у той час як неконтрольовані компоненти зберігають свій стан внутрішньо.

Це особливо стосується компонентів введення даних користувача. Більшість вбудованих компонентів форми (наприклад, input, select і т.д.) зазвичай використовуються як контрольовані компоненти, які очікують значення та властивість **onChange**. Однак, якщо значення не надано, ці компоненти працюють як неконтрольовані компоненти, зі своїм станом, що ініціалізується внутрішньо.

Неконтрольовані компоненти

Припустимо, що ми хочемо створити компонент **FormComponent**, який підтримує відправку даних через зворотний виклик **onSubmit**.

У цьому прикладі зберігається стан елементу введення внутрішньо в компоненті **FormComponent**. API компонента в цьому випадку складається лише з однієї властивості (prop) — **onSubmit**. Однак, ми не маємо можливості отримати доступ до поточного стану елементів введення за межами цього компонента.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

const FormComponent = ({ onSubmit }) => {
    const [input, setInput] = useState({});

    const handleInputChange = (e) =>
        setInput({
            ...input,
            [e.target.name]: e.target.value,
        });

    return (
        <form
            onSubmit={(e) => {
                onSubmit(input);
                setInput({});
                e.preventDefault();
            }}
        >
            <div>
                <label>Username:</label>
                <input
                    type="text"
                    name="username"
                    onChange={handleInputChange}
                    value={input.username ? input.username : ''}
                />
            </div>
            <div>
                <label>Password:</label>
                <input
                    type="text"
                    name="password"
                    onChange={handleInputChange}
                    value={input.password ? input.password : ''}
                />
            </div>
            <div>
                <input type="submit" />
            </div>
        </form>
    );
};
```

```
function App() {
    const [submitted, setSubmitted] = useState('');

    return (
        <>
            <FormComponent onSubmit={setSubmitted} />
            <p>submitted: {JSON.stringify(submitted)}</p>
        </>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

У цьому прикладі програмно встановлюється значення змінних стану, і це досягається шляхом додавання нового атрибуту **name** до полів введення (input). Це відбувається в рядку **[e.currentTarget.name]: e.currentTarget.value**, де це ім'я використовується, як властивість об'єкта стану і призначається йому значення з поля введення.

Цей підхід є масштабованим, оскільки кількість полів введення в формі не має значення, всі вони використовуватимуть ту саму функцію **handleInputChange**, і локальний стан буде відповідно оновлюватися.

Контрольовані компоненти

У наступному прикладі зберігається стан елементів введення зовнішньо в батьківському компоненті **App**, передаючи його через властивості **value** та слухач події зміни поля за допомогою властивості (prop) **onChange**. Тепер API компонента включає три властивості (props), тому воно трохи складніше, але часто вибирається цей підхід, щоб мати можливість використовувати поточний стан елементів введення навіть до їх відправки через **onSubmit**.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

const FormComponent = ({ onSubmit, onChange, value }) => {
    return (
        <form
            onSubmit={(e) => {
                onSubmit(value);
                e.preventDefault();
            }}
        >
            <div>
                <label>Username:</label>
                <input
                    type="text"
                    name="username"
                    onChange={onChange}
                    value={value.username ? value.username : ''}
                />
            </div>
            <div>
                <label>Password:</label>
                <input
                    type="text"
                    name="password"
```

```
                    onChange={onChange}
                    value={value.password ? value.password : ''}
                />
            </div>
            <div>
                <input type="submit" />
            </div>
        </form>
    );
};

function App() {
    const [input, setInput] = useState('');
    const [submitted, setSubmitted] = useState('');

    const handleInputChange = (e) =>
        setInput({
            ...input,
            [e.target.name]: e.target.value,
        });

    return (
        <>
            <FormComponent
                onSubmit={setSubmitted}
                onChange={handleInputChange}
                value={input}
            />
            <p>inputted: {JSON.stringify(input)}</p>
            <p>submitted: {JSON.stringify(submitted)}</p>
        </>
    );
}

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Модель даних

Часто компоненти представляють сутності схеми даних, такі як люди, статті, повідомлення, мультимедіа і т.д. У цьому випадку потрібно вирішити, як передавати дані у компоненти.

Це особливо часто трапляється, якщо використовується REST API, оскільки часто отримуються всі ресурси, а не лише дані, які потрібні (наприклад, при використанні GraphQL).

Передача ресурсу безпосередньо

Припустимо, у є масив об'єктів, де кожен об'єкт описує країну. Найпростіший спосіб передати його в компонент **Country** — це передати весь ресурс як властивість (props).

Недоліками цього підходу є те, що можуть бути передані дані, які не використовуються компонентом (наприклад, поле *languages*) і таким чином тісно зв'язуються компоненти і моделі даних.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';
import { countries } from './data';
```

```
function Country({ country }) {
    return (
        <article style={styles.article}>
            <h2>
                <img src={country.flags} alt={country.name} width="75" />
                  
                {country.name}
            </h2>
            <div style={styles.content}>
                <p>
                    <b>Capital: </b>
                    {country.capital}
                </p>
                <p>
                    <b>Population: </b>
                    {country.population}
                </p>
            </div>
        </article>
    );
}

function App() {
    return countries.map((country) => (
        <Country key={country.id} country={country} />
    ));
}
const styles = {
    article: {
        margin: '10px 10px 0 10px',
        borderRadius: '4px',
        padding: '10px',
        backgroundColor: '#C9CCD4',
    },
    content: {
        display: 'flex',
        justifyContent: 'space-around',
    },
};

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Дублювання властивостей

Більш поширеним є передача кожної окремої властивості, яка потрібно з ресурсу, у компонент. Хоча записати кожну властивість може бути трохи складно, явність допомагає легше вносити зміни у модель даних або компонент.

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';
import { countries } from './data';

function Country({ name, capital, population, flags }) {
    return (
        <article style={styles.article}>
            <h2>
                <img src={flags} alt={name} width="75" />
                  
                {name}
            </h2>
```

```
            <div style={styles.content}>
                <p>
                    <b>Capital: </b>
                    {capital}
                </p>
                <p>
                    <b>Population: </b>
                    {population}
                </p>
            </div>
        </article>
    );
}

function App() {
    return countries.map((country) => (
        <Country key={country.id} name={country.name} capital={country.capital}
population={country.population} flags={country.flags} />
    ));
}

const styles = {
    article: {
        margin: '10px 10px 0 10px',
        borderRadius: '4px',
        padding: '10px',
        backgroundColor: '#C9CCD4',
    },
    content: {
        display: 'flex',
        justifyContent: 'space-around',
    },
};

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Розповсюдження властивостей

Можна передати будь-яку кількість властивостей (props) з об'єкта до компонента, використовуючи синтаксис оператору **... spread**.

Це зручно, коли необхідно передати багато властивостей (props). Однак це також може призвести до передачі властивостей (props), які насправді не використовуються (наприклад, *languages* передаються до **<Country />**, хоча їх не використовують).

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';
import { countries } from './data';

function Country({ name, capital, population, flags }) {
    return (
        <article style={styles.article}>
            <h2>
                <img src={flags} alt={name} width="75" />
                  
                {name}
            </h2>
```

```
            <div style={styles.content}>
                <p>
                    <b>Capital: </b>
                    {capital}
                </p>
                <p>
                    <b>Population: </b>
                    {population}
                </p>
            </div>
        </article>
    );
}

function App() {
    return countries.map((country) => <Country key={country.id} {...country} />);
}

const styles = {
    article: {
        margin: '10px 10px 0 10px',
        borderRadius: '4px',
        padding: '10px',
        backgroundColor: '#C9CCD4',
    },
    content: {
        display: 'flex',
        justifyContent: 'space-around',
    },
};

const root = createRoot(document.getElementById('app'));
root.render(<App />);
```

«Живий» приклад за посиланням.

Запитання для самоконтролю

1. Як можна повернути кілька елементів з компонента в React?
2. Які можливості є для групування кількох елементів без додавання зайвих елементів у DOM?
3. Які переваги та недоліки використання React фрагментів для повернення кількох елементів?
4. Що таке довільні дочірні елементи в React і як вони використовуються?
5. Як передати довільні дочірні елементи в React компонент?
6. Що таке візуалізація властивостей (render props) в React і як вона використовується?
7. Як передати візуалізацію властивостей в компонент за допомогою render props?
8. Які переваги та недоліки використання візуалізації властивостей в React компонентах?
9. Що таке контрольовані елементи в React і як вони використовуються?
10. Як встановити і зчитати значення контрольованого елемента в React компоненті?
11. Як відслідковувати та реагувати на зміни значення контрольованого елемента?
12. Які переваги та недоліки використання контрольованих елементів в React?

13. Як використовувати контрольовані елементи для створення форм та взаємодії з користувачем?
14. Що таке пряма передача даних (Passing the resource directly) і як вона використовується?
15. Що таке дублювання властивостей (Duplicating the props) і як воно використовується?
16. Як використовувати розповсюдження властивостей (Spreading props) для передачі даних в компоненти React?

Рекомендовані джерела

1. Посібник: Quick start. [Електронний ресурс]. – Режим доступу : https://react.dev/learn
2. Tutorial: Tic-Tac-Toe. [Електронний ресурс]. – Режим доступу : https://react.dev/learn/tutorial-tic-tac-toe
3. Thinking in React. [Електронний ресурс]. – Режим доступу : https://react.dev/learn/thinking-in-react
4. Навчальний курс React Express [Електронний ресурс]. – Режим доступу : https://www.react.express/
5. React Tutorial. [Електронний ресурс]. – Режим доступу : https://www.w3schools.com/react/default.asp