DOI: https://doi.org/10.33216/2222-3428-2022-23-5
УДК 004.4'236

Щербаков Є.В., Щербакова М.Є.

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ БІБЛІОТЕКИ REACT

React - це популярна фронтенд-бібліотека з середовища JavaScript. Вона відома простотою використання та читабельністю коду веб-додатків та веб-сайтів, розроблюваних з її використанням. Бібліотека React - це ще одне рішення для розробки SPA-додатків на JavaScript, яка була випущена Facebook в 2013 році. React має акуратно розроблений API, стабільну, процвітаючу екосистему та велику спільноту користувачів, що дозволяє організаціям різних масштабів успішно впроваджувати цю бібліотеку. React базується на компонентній концепції. Компоненти React представляють собою багаторазові будівельні блоки для створення різного роду візуальних інтерфейсів користувача веб-додатків, в тому числі і SPA-додатків. Дані між елементами дерева React-компонентів додатка передаються як зверху вниз за допомогою об'єктів props (скорочення від properties), так і знизу вверх, використовуючи функції-обробники зворотного виклику. Стан – це ще одна із центральних концепцій React. Саме тут зберігаються дані додатка - тобто те, що може змінюватися з часом. Спочатку в екосистемі React використовувалися переважно компоненти, засновані на класах. Застосування таких компонентів, зазвичай, потребує додаткових зусиль у ході розробки, оскільки програмісту постійно треба переключатися між класами, компонентами вищого порядку та рендер-пропсами. Цих недоліків не мають компоненти React, базовані на звичайних функціях JavaScript. У зв'язку з переходом на використання функціональних компонентів в сучасну бібліотеку React був введений механізм запуску асинхронно виконуваних функцій, так званих хуків. Хуки — це функції, які дозволяють підключатися (hook into) до функцій стану та життєвого циклу React із функціональних компонентів. React оновлює браузерну DOM, дотримуючись стратегії внесення в дерево найменшої кількості змін, по можливості, без повторного рендерингу всієї DOM, що суттєво покращує ефективність візуального інтерфейсу веб-додатка.

Ключові слова: *фронтенд, SPA, компонент, DOM, віртуальна DOM, JSX, пропси, стан, потік даних, хуки, рендеринг.*

Вступ. Фронтенд-фреймворки та бібліотеки повністю змінили підхід розробників до веб-розробки, надавши їм гнучкість зосередитися на функціональності, а не на вмісті, та різко пришвидшивши робочі процеси по розробці веб-додатків та веб-сайтів [1]. Сьогодні найдомінантнішою бібліотекою інтерфейсу користувача у світі JS є React.js від Facebook, яка є невеликим набором програмних інструментів, але деякими великими ідеями: віртуальним DOM, розширенням синтаксису JavaScript (JSX) і компонентизацією. Не можна заперечувати серйозний вплив, який справила React на веб-розробку, особливо при створенні односторінкових веб-додатків. React має акуратно розроблений API, стабільну, процвітаючу екосистему та велику спільноту користувачів. У 2021 році 80% JS-розробників використовували бібліотеку React, а 70% розробників JS сказали, що будуть використовувати її знову.

Односторінкові веб-додатки. Односторінкові додатки (Single Page Application - SPA) стали популярними серед користувачів, завдячуючи не в останню чергу фреймворкам першого покоління, таким як Angular, Ember, Knockout та Backbone [2]. Використання цих фреймворків полегшувало створення веб-додатків, які виходили за рамки типових, створених з використанням тільки мови JavaScript та бібліотеки jQuery. Бібліотека React - це ще одне рішення для розробки SPA-додатків на JavaScript, який був випущений Facebook в 2013 році.

До появи SPA веб-сайти та веб-додатки рендерилися (формували розмітку HTML для відображення на екрані користувача) на сервері. Користувач задавав URL-адресу в браузері і запитував файл HTML і всі пов'язані з ним файли, в тому числі CSS і JavaScript, на веб-сервері. Після певної затримки в мережі користувач бачив відрендерений HTML у браузері (клієнті) і починав з ним взаємодіяти. Кожна додаткова заміна сторінки (тобто відвідування іншої URL-адреси) знову ініціювала цей ланцюжок подій. У цій версії з минулого, по суті, все важливе виконував сервер, тоді як клієнт відігравав мінімальну роль, просто відображаючи сторінку за сторінкою. Хоча для структурування додатка використовувалися базові HTML і CSS, лише трохи JavaScript додавалось до суміші, щоб зробити можливими взаємодії (наприклад, перемикання спадного списку) або розширений стиль (наприклад, розміщення підказки). Популярною бібліотекою JavaScript для такої роботи була jQuery.

На противагу цьому, сучасна мова JavaScript перемістила фокус із сервера на клієнта. Найекстремальніша версія цього: користувач відвідує URL-адресу і запитує один маленький файл HTML і один більший файл JavaScript. Після деякої мережевої затримки користувач бачить в браузері HTML-код, відрендерений за допомогою JavaScript, і починає з ним взаємодіяти. Кожен додатковий перехід між сторінками не запитує більше файлів з веб-сервера, а замість цього використовує раніше завантажений JavaScript, щоб відрендерити нову сторінку. Крім того, кожна додаткова взаємодія користувача також обробляється клієнтом. У цій сучасній версії сервер передає по мережі переважно JavaScript з одним мінімальним HTML-файлом. Після цього файл HTML виконує на стороні клієнта зв'язаний JavaScript, щоб відрендерити всю HTML-розмітку додатка і щоб обробляти на JavaScript всі взаємодії з користувачем.

Бібліотека React, серед інших SPA-рішень, робить це можливим. По суті, SPA — це множина інструкцій JavaScript, яка елегантно організована в папки та файли, щоб створити цілий додаток, тоді як SPA- фреймворк або бібліотека (наприклад, React) надає всі інструменти для його створення. Цей додаток, орієнтований на JavaScript, доставляється один раз по мережі до браузера, коли користувач відвідує URL-адресу веб-додатка. Після цього React або будь-який інший фреймворк SPA бере на себе рендеринг всього в браузері та взаємодію з користувачем.

Компоненти. Компоненти React – це самодостатні елементи, які, поряд зі звичайними тегами HTML, можна використовувати на веб-сторінці будь-яку кількість разів [2]. Кожен компонент визначає своє представлення на екрані за допомогою специфічної для React мови JSX, яка має XML-подібний синтаксис і являється комбінацією елементів традиційних для програмування Інтернету мов HTML, CSS та JavaScript. Після визначення компонента він може бути використаний в дереві компонентів для створення повноцінного додатка практично любої складності. Незважаючи на те, що React приділяє велику увагу компонентам як елементам бібліотеки, супроводжуюча екосистема робить її гнучким інструментом.

Концептуально компоненти React є подібними до функцій JavaScript. Вони приймають довільні вхідні дані в формі об'єкта з даними props (скорочено від properties - властивості) і повертають React-елементи, які описують те, що повинно з'явитися на екрані. Компоненти можна визначати різними способами.

Перший спосіб визначення компонентів React – за допомогою класу ES6, наприклад:

```
class Welcome extends Component {
  render() {
    return <h1>Привіт, {this.props.name}!</h1>
  }
}
```

Тут було визначено компонент з ім'ям Welcome, що подібно визначенню нового тегу HTML з цим же ім'ям. Всі імена компонентів мають починатися з великої літери. Наведений вище компонент виводить в розмітку HTML елемент з повідомленням «Привіт, світ!».

Для генерації HTML-розмітки для відображення в браузері користувача (коротше - рендерингу) в класі компонента обов'язково має бути визначений метод render(), який повертає створюваний компонентом елемент на JSX. Елементи — це найменші будівельні блоки React-додатка. Елемент описує те, що користувач хоче бачити на екрані. На відміну від DOM-елементів, елементи React — звичайні програмні об'єкти, створення яких займає дуже мало часу.

Другий спосіб визначення компонентів - функціональний:

```
function Welcome(props) {
  return <h1>Привіт, {props.name}!</h1>
}
```

Для визначення компонентів також можна використовувати стрілочні функції (arrow functions), наприклад:

```
var Welcome = (props) => {
  return <h1>Привіт, {props.name}</h1>
}
```

Загалом, всі три способи визначення компонентів React є рівноцінними. Але порівняння кодів наведених вище трьох прикладів показує, що коди обох функціональних компонентів простіше за код аналогічного компонента, заснованого на класі. Окрім того, для використання функціонального компонента не потрібно створювати екземпляр класу, не потрібно викликати спеціальну функцію render(). Достатньо просто викликати основну функцію з відповідним ім'ям.

Компоненти можуть посилатися на інші компоненти під час виведення. Це дозволяє використовувати одну і ту ж абстракцію компонентів для будь-якого рівня деталізації. Кнопка, форма, діалогове вікно, екран: у React-додатках всі вони зазвичай визначаються як компоненти.

Щоб створити додаток на базі компонентів React, можна скористатись середовищем розробки Create React App. Файл index.js проекту на базі Create React App може містити наступний код :

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import App from "./App";
const root =
  ReactDOM.createRoot(document.getElementById("root"));
root.render(<App />);
```

Окрім цього, в файлі App.js, який імпортується JavaScript-кодом попереднього файлу, буде міститися логіка додатка на JavaScript з використанням JSX, формуючого ієрархію компонентів React, яка становить візуальний інтерфейс додатка. В цьому файлі додатково створимо компонент App, який відрендерить компонент Welcome декілька разів:

```
import * as React from "react";
function Welcome(props) {
  return <h1> Привіт, {props.name}</h1>;
}
function App() {
```

```
  return (
    <div>
      <Welcome name="Марія" />
      <Welcome name="Петро" />
      <Welcome name="Василь" />
    </div>
  );
}
export default App;
```

Після запуску на виконання додатка командою менеджера пакетів Node.js:

```
npm start
```

бібліотекою React буде автоматично сформована HTML-сторінка, в якій будуть відображені три привітання, відренлерені екземплярами компонента Welcome (рис. 1).

Привіт, Марія

Привіт, Петро

Привіт, Василь

Рисунок 1 - Привітання, сгенеровані React на HTML-сторінці за допомогою модулів App та Welcome

Як правило, React-додатки мають єдиний компонент App, що знаходиться зверху дерева компонентів. Однак, якщо React інтегрується з уже існуючим додатком, можна почати знизу вгору з невеликим компонентом, наприклад Button, і поступово нарощувати до верхньої частині дерева компонентів сторінки.

Потоки даних між компонентами React. Дані між елементами дерева React-компонентів можуть передаватись як зверху вниз за допомогою об’єктів props, як це було показано вище, так і знизу вверх, використовуючи функції-обробники зворотного виклику (callbacks).

Концепція функції-обробника зворотного виклику проста. Ця функція, маючи реалізацію в батьківському компоненті, передається від батьківського компонента до дочірнього компонента в дереві компонентів як один із елементів об’єкта props, а потім ця функція викликається в дочірньому компоненті. По суті, коли обробник події передається як властивість від батьківського компонента до його дочірнього компонента, він стає обробником зворотного виклику. Властивості або реквізити JSX завжди передаються вниз по дереву компонентів, а обробники зворотного виклику, які передаються як функції в props, можна використовувати для передачі даних вгору по дереву компонентів.

Стан React. На відміну від пропсів (елементів об’єкта props), стан React використовується, щоб зробити додатки інтерактивними. Обидві структури даних, пропси і стан, мають чітко визначені цілі: пропси використовуються для передачі інформації по дереву компонентів, стан використовується для зміни інформації з часом. Як пропси, так і стан можуть також працювати сумісно для досягнення спільної мети.

Стан - це збірне поняття для будь-якої інформації, що стосується змінної частини додатка. Це можуть бути як дані, що використовуються в математичній моделі додатка або інтерфейсі користувача, такі як список завдань або список користувачів, так і стан як такий, наприклад, стан завантаження або стан форми. В класних React-компонентах стан оголошується в конструкторі класу як об’єкт state, наприклад:

```
this.state = {welcome: "Ласкаво просимо на сайт!"};
```

і оновлюється за допомогою вбудованої функції setState, наприклад:

```
this.setState({welcome: " Привіт, React!"});
```

Для управління елементами стану функціональних компонентів призначений хук useState():

```
const [state, setState] = useState(initialValue);
```

який повертає масив із двома елементами: поточним значенням стану state та сеттером setState() – функцією для оновлення цього значення. В якості аргументу initialValue хука вказується початкове чи дефолтне значення стану. За згодою, назва сеттера повинна починатися з "set" + назва першого елемента з великої літери ([count, setCount], [text, setText] тощо).

В обох випадках оновлення стану, тобто виклик функції setState(), призводить до повторного рендерингу компонента, в зв’язку з чим оновлюється вся веб-сторінка.

Умовно, стан можна розділити на локальний та глобальний стани. Під локальним станом зазвичай розуміється стан окремо взятого компонента, наприклад, стан форми, як правило, є локальним станом відповідного компонента. У свою чергу, глобальний стан правильніше називати розподіленим або спільно використовуваним, маючи на увазі те, що такий стан використовується більш ніж одним компонентом додатка. Умовність цієї градації виражається в тому, що локальний стан цілком може використовуватися декількома компонентами (наприклад, стан, визначений за допомогою useState(), може у вигляді пропсів передаватися дочірнім компонентам), а глобальний стан не обов'язково використовується всіма компонентами додатка.

Щоб правильно побудувати додаток, спочатку потрібно продумати необхідний набір даних змінюваного стану цього додатка [3]. Головне, дотримуватися принципу розробки DRY (Don’t Repeat Yourself - не повторюйся): визначається мінімально необхідний стан, який потрібен розроблюваному додатку, все інше

обчислюється при необхідності. Після цього з'ясовується, який з компонентів володіє станом або змінює його. Треба пам'ятати, що у React односторонній потік даних, який сходить згори вниз в ієрархічному порядку [4]. Оскільки може бути не зрозуміло, який з компонентів повинен володіти яким станом, для кожної частини стану в додатку треба:

- визначити компоненти, які рендерять щось на основі цього стану;
- знайти спільний батьківський компонент (компонент, розташований над іншими компонентами, яким потрібен цей стан). Або спільний батьківський компонент, або будь-який компонент, що стоїть вище за ієрархією, повинен містити стан;
- якщо не вдається знайти відповідний компонент, треба створити ще один компонент виключно для стану та розмістити його вище за ієрархією над загальним спільним батьківським компонентом.

Хуки. В React часто доводиться підтримувати компоненти на базі класів, сукупність яких була простою на початку, але переросла у якийсь некерований безлад логіки стану та побічних ефектів [5]. Методи життєвого циклу деяких компонентів цієї сукупності часто містять суміш не пов'язаної між собою логіки. Наприклад, компоненти можуть виконувати вибірку деяких даних у методах життєвого циклу componentDidMount() та componentDidUpdate(). При цьому у методі componentDidMount() може міститися якась не пов'язана логіка реєстрації обробників подій, відміна підписки яких виконується в componentWillUnmount(). Взаємно пов'язаний код, який спільно змінюється, опиняється розділеним, зате зовсім не пов'язані між собою частини зв'язуються в одному методі. І це призводить до виникнення помилок та невідповідностей в додатках, побудованих з використанням таких компонентів.

У багатьох випадках неможливо розбити ці компоненти на більш дрібні, оскільки логіка стану присутня повсюдно. Тестувати їх також важко. Це одна з причин, чому багато людей вважають за краще поєднувати React з окремою бібліотекою управління станом.

При використанні класних компонентів зазвичай використовують компоненти вищого порядку (High Order Components) або рендер-пропси (render props). Однак це часто вводить занадто багато абстракцій, вимагає переходу між різними файлами і ускладнює повторне використання компонентів.

Щоб вирішити цю проблему глобально, в React були добавлені хуки [6], які націлені на вирішення всіх цих проблем, дозволяючи писати функціональні компоненти, які мають доступ до стану, контексту, методів життєвого циклу, рефів та іншого, без написання класів. Хуки дають можливість розділяти один компонент на декілька з меншими функціями на основі пов'язаних між собою частин (таких, як налаштування підписки або отримання даних), а не примусово розбивати на основі методів життєвого циклу. Щоб поведінка була більш передбачуваною, можна також контролювати внутрішній стан за допомогою функцій-редюсерів.

Хуки — це функції, які дозволяють підключатися (hook into) до функцій стану та життєвого циклу React із функціональних компонентів. Хуки не працюють всередині класів — вони дозволяють використовувати React без класів для побудови ефективних інтерфейсів користувача. React надає кілька вбудованих хуків, а також можливість створювати користувацькі хуки з метою повторного використання поведінки з відслідковуванням стану в різних компонентах.

Як уже було описано вище хук стану useState() — це новий спосіб використання тих можливостей в функціональних компонентах, які змінна this.state надає у класних компонентах. Зазвичай, змінні зникають після виходу з функції, але змінні стану між викликами зберігаються за допомогою React.

Хук ефекту useEffect() додає можливість виконувати побічні ефекти з функціонального компонента. Він слугує тим же цілям, що й методи життєвого циклу componentDidMount(), componentDidUpdate() та componentWillUnmount() у класних компонентах React, але об'єднаних в єдиний API. Коли викликається useEffect(), React запускає на виконання задану в якості аргументу функцію, яка викликає побічний ефект, після реалізації змін у DOM. Ці функції ефектів оголошуються всередині компонента, так що вони мають доступ до його властивостей і стану. За замовчуванням React запускає ефекти після кожного рендерингу, включаючи перший рендеринг. Функція ефекту також може необов'язково вказувати, як «очищатися» після неї, повертаючи іншу функцію. Хуки дають можливість організовувати побічні ефекти як цілісні одиниці функціональності (наприклад, додавання і скасування підписки), замість того, щоб ділити все згідно з методами життєвого циклу.

Окрім цих двох основних є ще декілька додаткових вбудованих хуків, призначення яких можна зрозуміти з імен, за допомогою яких вони викликаються на виконання: useContext(), useCallback(), useMemo(), useRef(), useImperativeHandle(), useLayoutEffect() та useDebugValue(). Хуки представляють собою звичайні функції JavaScript, але на них накладаються деякі додаткові обмеження: хуки слід викликати тільки на найвищому рівні; не можна викликати хуки всередині циклів, умов або вкладених функцій; хуки слід викликати тільки з функціональних React-компонентів; не можна викликати хуки зі звичайних функцій JavaScript.

Рендеринг компонентів. Рендеринг - це процес, в рамках якого React опитує компоненти додатка, отримуючи від кожного актуальний опис тієї секції інтерфейсу користувача, за яку він відповідає, ґрунтуючись на поточній комбінації значень пропсів і стану [4].

Компоненти додатка зазвичай представляються у вигляді JSX-коду, який потім компілюється і розгортається як JavaScript-код, набуваючи вигляду серії викликів React.createElement(), що показано в табл. 1.

Таблиця 1. Відповідність JSX-коду скомпільованому JavaScript-коду

Скриптова частина додатка з використанням JSX	*Скриптова частина додатка після компіляції JSX в JavaScript*

import * as React from "react";
function Welcome(props) {
  return   Привіт, {props.name} ;
}
function App() {
  return (
 	
  <Welcome name="Марія" />
  <Welcome name="Петро" />
  <Welcome name="Василь" />
  );
}
export default App;
	import * as React from "react";
	function Welcome(props) {
	  return React.createElement("h1", null, " Привіт", props.name);
	}
	function App() {
	  return (React.createElement(("div",
	  null,
	  React.createElement(Welcome, { name: "Марія" }),
	  React.createElement(Welcome, { name: "Петро" }),
	  React.createElement(Welcome, { name: "Василь" }))
	  );
	}
	export default App;

Виклики функції createElement() повертають відповідні React-елементи, що є простими JavaScript-об'єктами, які описують бажану структуру візуального інтерфейсу користувача і є основою для формування дерева об’єктної моделі документа.

Об’єктна модель документа, або DOM (Document Object Model), — це внутрішнє представлення веб-сторінки в програмах React або в браузері, в якому HTML, стилі та вміст представляються як дерево вузлів, якими можна управляти за допомогою JavaScript.

React починає процес рендерингу з кореня дерева компонентів і рекурсивно спускається вниз, щоб знайти всі компоненти, які вимагають оновлення інтерфейсу користувача. Для кожного знайденого компонента React викликає або classComponentInstance.render() (для класових компонентів) або FunctionComponent() (для функціональних компонентів) і зберігає результат рендерингу.

Зібравши результати рендерингу всього дерева компонентів, React робить порівняння з попередньо зібраним деревом об'єктів (віртуальною DOM) і складає список усіх змін, які потрібно зробити в DOM браузера, щоб привести її до бажаного в даний момент виду. Процес зіставлення двох дерев та обчислення різниці між ними називається узгодженням. Концепції, які лежать в основі процесу узгодження, такі як віртуальна DOM та алгоритм розрізнення (diffing), суттєво прискорюють роботу додатків React.

При традиційному рендерингу без React браузер створює DOM браузера та відображає будь-які дані в DOM, навіть якщо дані такі ж, як вони були при попередньому відображенні цієї ж сторінки. Цей рендеринг, виконуваний браузером, має послідовність кроків і є досить витратним за своєю природою. Концепція віртуальної DOM, яку використовує React, робить рендеринг набагато швидшим.

При рендерингу з React виконуються такі дії:

- Зберігається копія браузерної DOM, яка називається віртуальною DOM.

- Коли вносяться зміни або додаються інші дані, React створює оновлену віртуальну DOM і проводить її порівняння з попередньою.

- Порівняння здійснюється з використанням алгоритму розрізнення. Варто зазначити, що всі пов’язані з цим порівняння відбуваються в оперативній пам'яті React, а в браузері поки що нічого не змінюється.

- Після порівнянь React продовжує роботу і створює нову віртуальну DOM з внесеними змінами. Слід зазначити, що типові алгоритми роботи з деревами мають порядок складності щонайменше $O(N^2)$, так що за секунду на сучасному комп’ютері можна обробити не більше декількох сотень вузлів DOM.

- Потім React оновлює браузерну DOM, дотримуючись стратегії внесення в дерево найменшої кількості змін по можливості без повторного рендерингу всієї DOM (рис. 2), що суттєво покращує ефективність візуального інтерфейсу.

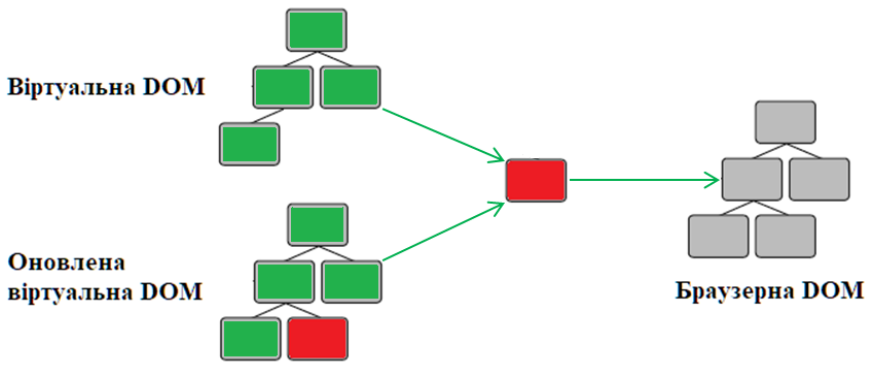

Рисунок 2 – Обчислення різниці між деревами віртуальних DOM і оновлення браузерної DOM

При роботі з віртуальною DOM в алгоритмі розрізнення при находженні змінених вузлів застосовується пошук в ширину, оскільки, якщо вузол буде знайдено як змінений, він повторно відрендерить все піддерево, отже, пошук в глибину не є оптимальним. При порівнянні двох елементів одного типу враховуються лише зміни в атрибутах або стилях. React використовує й інші подібні оптимізації, щоб мінімальну різницю між деревами за допомогою цього алгоритму можна було ефективно обчислити за O(N) операцій.

Висновки. React є відмінним інструментом для створення візуальних інтерфейсів масштабованих веб-додатків, особливо в тих ситуаціях, коли додаток представляє собою SPA (односторінковий додаток). За допомогою React можна створювати інкапсульовані компоненти, які управляють своїми даними та уникають впливів на інші стани та дії компонентів у DOM-дереві. Це також допомагає повторно використовувати код, розділяти відповідальність між розробниками та уникати повторень того ж коду в одному додатку. Ще одна особливість React – використання мови JSX. JSX представляє собою комбінацію кодів JavaScript та XML і надає простий та інтуїтивно зрозумілий спосіб визначення коду візуального інтерфейсу.

Компоненти React можуть бути двох типів: засновані на класах або на функціях. Спочатку в React використовувалися переважно компоненти, засновані на класах. Застосування таких компонентів вимагало надмірних зусиль у ході розробки веб-додатка або веб-сайту, оскільки програмісту постійно доводилось переключатися між класами, компонентами вищого порядку, властивостями рендерингу. Тому через деякий час перевага почала віддаватися функціональним компонентам, коди яких набагато простіші за коди таких же компонентів, але заснованих на класах. Для використання функціонального компонента не потрібно створювати екземпляр класу, не потрібно викликати функцію render(). А поява в React хуків і використання їх в додатках, основаних на функціональних компонентах, дало можливість вирішувати всі ті ж задачі, які вирішувались за допомогою класів, але при цьому дозволило швидше та ефективніше реалізовувати схожий функціонал, наприклад, працювати зі станом компонентів та з методами їх життєвого циклу. Концептуально, React-компоненти завжди були ближчими до функцій; хуки забезпечують доступ до функціонала, але не знецінюють досвід використання React.

React забезпечує детермінований рендеринг візуальних представлень компонентів, в основі якого лежить односпрямована прив'язка даних та іммутабельний стан компонентів. При рендерингу використовується віртуальна DOM, яка разом з алгоритмом розрізнення дає можливість оновлювати браузерну DOM за O(N) операцій, а не $O(N^2)$ операцій, які були б потрібні без використання цієї проміжної DOM та інших функціональних особливостей React.

Література

1. Larsen-Disney S. Elevating React Web Development with Gatsby / S. Larsen-Disney. - Birmingham: Packt Publishing Ltd., 2022. - 315 p.
2. Wieruch R. The Road to React. / R. Wieruch. - Victoria, BC: Lean Publishing, 2021. - 253 p.
3. Boduch A. React and React Native. Third Edition / A. Boduch, R. Derks. - Birmingham: Packt Publishing Ltd., 2020. - 505 p.
4. React, v18.0.0. Docs. Main Concepts [Електронний ресурс] / React. - Режим доступу: https://reactjs.org/docs/hello-world.html.
5. React, v18.0.0. Docs. Advanced Guids [Електронний ресурс] / React. - Режим доступу: https://reactjs.org/docs/accessibility.html.
6. React, v18.0.0. Docs. Hooks [Електронний ресурс] / React. - Режим доступу: https://reactjs.org/docs/hooks-intro.html.

References

1. Larsen-Disney S. Elevating React Web Development with Gatsby / S. Larsen-Disney. - Birmingham: Packt Publishing Ltd., 2022. - 315 p.
2. Wieruch R. The Road to React. / R. Wieruch. - Victoria, BC: Lean Publishing, 2021. - 253 p.
3. Boduch A. React and React Native. Third Edition / A. Boduch, R. Derks. - Birmingham: Packt Publishing Ltd., 2020. - 505 p.
4. React, v18.0.0. Docs. Main Concepts [Electronic resource] / React. - Access mode: https://reactjs.org/docs/hello-world.html.
5. React, v18.0.0. Docs. Advanced Guids [Electronic resource] / React. - Access mode: https://reactjs.org/docs/accessibility.html.
6. React, v18.0.0. Docs. Hooks [Electronic resource] / React. - Access mode: https://reactjs.org/docs/hooks-intro.html.

React is a popular frontend library from the JavaScript environment. It is known for its ease of use and readability of the code of web applications and websites developed with its use. The React Library is another solution for developing SPA applications on JavaScript, which was released by Facebook in 2013. React has a well-designed API, a stable, thriving ecosystem, and a large user community that allows organizations of different sizes to successfully implant this library. React is based on the component concept. The React

components are reusable building blocks for creating various kinds of visual interfaces of application, including SPA application. The data between the elements of the React component tree of the application is transmitted both from top to bottom using props objects (short for properties) and from bottom to top, using callback handlers. State is another central concept of React. This is where application data is stored - something that may change over time. Initially, the React ecosystem used mostly class-based components. The use of such components usually requires additional effort during development, as the programmer constantly has to switch between classes, higher-order components and rendering props. React components based on common JavaScript functions do not have these shortcomings. Due to transition of functional components use in the modern library React, a mechanism was introduced to launch asynchronously executed functions, so-called hooks. Hooks are functions that allow to connect to the React state and life cycle functions from functional components. React updates the browser DOM, following the strategy of making the least amount of changes to the tree, if possible, without re-rendering the entire DOM, which significantly improves the efficiency of the visual interface of the web application.
Keywords: *frontend, SPA, component, DOM, virtual DOM, JSX, props, state, data flow, hooks, rendering.*

Щербаков Є.В. – канд. тех. наук, доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: gkvarc@gmail.com

Щербакова М.Є. – канд. тех. наук, доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: m.shcherbakova432@gmail.com