УДК 004.415.2 DOI: 10.30977/BUL.2219-5548.2024.104.0.15

ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ РІШЕНЬ ПОБУДОВАНИХ ЗА ДОПОМОГОЮ БІБЛІОТЕК *REACTJS* ТА *D3*

Чуб І. М., Демченко К. В.
Державний біотехнологічний університет

Анотація. *У статті проаналізовано продуктивність React-компонентів щодо розроблення та використання компонента, що візуалізує графік, показники якого змінюються в реальному часі. Досліджено актуальні поточні рішення, що стосуються оптимізації React-компонентів. Установлено, що ці способи лише частково вирішують зазначену проблему. Знайдено альтернативний шлях мінімізації рендерів компонента в умовах інтеграції з бібліотекою D3. Надано неоптимізовану версію компонента. Проведено його оптимізацію за допомогою розробленого методу. Порівняно роботи обох версій. Експериментально доведено ефективність запропонованого методу щодо оптимізації та надано код і графіки.*

Ключові слова: *ReactJS, React, D3, web-застосунок, оптимізація, Javascript, рендеринг, мемоїзація, хуки, компонент*.

Вступ

Останнім часом проблема продуктивності web-застосунків постає особливо гостро. Це зумовлено ускладненням програмного забезпечення та одночасним погіршенням якості коду. Оскільки якість програмного коду є лише суб'єктивним фактором, подальший матеріал буде розглянуто в контексті ускладнення програмного забезпечення, а саме застосування різноманітних фреймворків та бібліотек. У цій статті розглядатимуться важливі аспекти продуктивності клієнтської частини web-застосунку, побудованої за допомогою бібліотек *ReactJS* та *D3* [1, 2].

Оскільки були вказані бібліотеки *ReactJS* та *D3*, неважко здогадатись, що наш предметний напрям стосуватиметься візуалізації даних. Перед тим, як розпочати розглядання поточної проблеми безпосередньо, варто зупинитися на деяких аспектах продуктивності *ReactJS* застосунку загалом.

По-перше, *ReactJS* – одна з найпопулярніших бібліотек щодо створення клієнтської частини web-застосунку. По-друге, рівень початкових умінь фахівця в цій технології відносно низький порівняно, скажімо, з фреймворком *Angular*, та як наслідок – створення неоптимального з погляду продуктивності коду в багатьох випадках. Попри це *ReactJS* містить низку потужних засобів щодо покращення продуктивності [3]. Річ у тому, що рішення продуктивності *ReactJS* тісно пов'язане з таким поняттям, як кількість рендерів, а точніше з мінімізацією цієї кількості. Це надважливо, оскільки кожен рендер (перемальовування частини екрана, пов'язану з компонентом) – це додаткова робота з боку як програмного, так і апаратного забезпечення. Як наслідок – уповільнення рішення загалом, втрата часу користувача на очікування результату та втрати електроенергії.

Аналіз публікацій

Базові способи вирішення проблеми продуктивності *ReactJS*-компонента основані на мемоїзації та хуках *useMemo* [4] і *useCallback* [5]. Автор роботи [4] пропонує *React.memo*, що забезпечує високу продуктивність і відсутність лагів. Але в *React.memo* є обмеження – він мемоїзує результат компонента та перевіряє пропси. Це корисно, коли компонент рендериться часто з однаковими пропсами або має складний інтерфейс. Проте не варто надмірно використовувати *React.memo()*, оскільки це може призвести до надмірного навантаження і як наслідок – до погіршення продуктивності. Щодо застосування хуків *useCallback()* та *useMemo()* [5], то вони допомагають мемоїзувати результати функцій та значень. Є певні обмеження в їх використанні, а саме: застосовувати їх розумно тоді, коли є залежності від інших хуків чи важких обчислень, які можна мемоїзувати. Навіть у цьому разі добре було б обмежити мемоїзацію лише тими значеннями, що справді потребують оптимізації.

Автор статті [6] зупиняється на деяких важливих способах оптимізації продуктивності програми *React*, зокрема методах попередньої оптимізації, та пропонує використовувати *Reselect* у *Redux* для оптимізації візуалі-

зації. Але *Redux* часто призводить до проблем із продуктивністю через непотрібний повторний рендеринг під час зміни стану.

У процесі аналізу встановлено, що запропоновані рішення використовуються в переважній більшості оптимізаційних рішень, але вони не дають змоги розв'язати проблему, обумовлену застосуванням бібліотеки *D3*. Тобто потрібно запровадити більш комплексне рішення, що бере до уваги особливість *D3*.

Необхідність застосування бібліотеки *D3* обґрунтована її перевагами над аналогами.

1. У мережі багато прикладів використання *D3.js*. На *GitHub* [2, 3] є підібрані списки прикладів *D3*, які можна переглядати разом із кінцевим результатом.

2. *D3* був використаний спільнотою розробників понад 9 тис. разів, із великою кількістю безкоштовних ресурсів для розширень і сторонніх бібліотек-оболонок, щоб зробити час, витрачений на створення візуалізацій, набагато легшим.

3. *D3* має гнучкість для відображення великого прошарку різноманітних даних.

4. Високий попит на ринку праці.

5. Низькі ключові навички веброзробника. Потрібне лише розуміння *HTML5*, *SVG* та *Javascript*.

Мета й постановка завдання

Метою статті є розроблення *React*-компонента (оптимізованого новим методом) у вигляді графіка, показники якого змінюються в реальному часі. Побудова *React*-компонента дасть змогу використовувати його в різних web-застосунках, побудованих за допомогою бібліотеки *React*. Це дозволить використовувати всю можливість функціоналу *D3* у проєктуванні широкого спектра моніторингових і білінгових систем та систем автоматизованого управління процесами.

Виклад основного матеріалу

Для розв'язання проблеми продуктивності пропонується така реалізація. Створення *ReactJS*-компонента який містить графік, створений за допомогою *D3*, що змінюється в реальному часі.

Наведемо базове неоптимізоване рішення щодо компонента *React DemoChart*.

```
    import React, { useEffect, useRef }
from "react";
    import * as d3 from "d3";

    const transform = "translate(50,50)";

    export default function DemoChart({
points, width, height, pointer }) {
      const chartAreaRef = useRef();

      const renderSvg = () => {
        const CHART_WIDTH = width - 200;
        const CHART_HEIGHT = height - 200;

        const svgObj =
d3.select(chartAreaRef.current);

        svgObj.selectAll("*").remove();

        const scaleX =
d3.scaleLinear().domain([0, 100]).range([0,
CHART_WIDTH]);
        const scaleY =
d3.scaleLinear().domain([0,
200]).range([CHART_HEIGHT, 0]);

        const gObj =
svgObj.append("g").attr("transform", trans-
form);

        gObj.append("g")
          .attr("transform", `trans-
late(0,${CHART_HEIGHT})`)
          .call(d3.axisBottom(scaleX));


gObj.append("g").call(d3.axisLeft(scaleY));

        svgObj
          .append("g")
          .selectAll("dot")
          .data(points)
          .enter()
          .append("circle")
          .attr("cx", function (d) {
            return scaleX(d[0]);
          })
          .attr("cy", function (d) {
            return scaleY(d[1]);
          })
          .attr("r", 3)
          .attr("transform", transform)
          .style("fill", "#CC0000");

        const line = d3
          .line()
          .x(function (d) {
            return scaleX(d[0]);
          })
          .y(function (d) {
            return scaleY(d[1]);
          })
          .curve(d3.curveMonotoneX);

        svgObj
          .append("path")
          .datum(points)
          .attr("class", "line")
          .attr("transform", transform)
          .attr("d", line)
          .style("fill", "none")
          .style("stroke", "#CC1100")
```

```
          .style("stroke-width", "2");

        if (pointer) {
          svgObj
            .append("svg:line")
            .attr("transform", transform)
            .attr("stroke", "#00cc00")
            .attr("stroke-linejoin",
"round")
            .attr("stroke-linecap",
"round")
            .attr("stroke-width", 2)
            .attr("x1", scaleX(pointer))
            .attr("y1", 200)
            .attr("x2", scaleX(pointer))
            .attr("y2", 0);
        }
      };

      useEffect(() => {
        renderSvg();
      }, [width, height, points, pointer]);

      if (!width || !height || !points) {
        return <></>;
      }

      return <svg ref={chartAreaRef}
width={width} height={height} />;
    }
```

Компонент *DemoChart* приймає такі вхідні показники:

– *points* – інформація у вигляді двовимірного масиву цілих чисел для *X* та *Y* вимірів;

– *width* – кількість пікселів компонента завширшки;

– *height* – кількість пікселів компонента заввишки;

– *pointer* – координата *X* маркерної вертикальної лінії, що рухається в реальному часі.

Далі буде наведений батьківський компонент, що містить посилання на вказаний вище та двовимірний масив даних для передачі компонента.

```
  import React, { useState, useEffect }
from "react";
  import DemoChart from "./DemoChart";
  import "./style.css";

  const points = [
    [1, 0],
    [12, 23],
    [24, 32],
    [32, 54],
    [40, 72],
    [50, 102],
    [55, 106],
    [65, 127],
    [72, 138],
    [75, 132],
    [82, 131],
    [88, 137],
    [100, 148],
  ];

  const STEP = 5;

  export default function App() {
    const [pointer, setPointer] =
useState(STEP);

    useEffect(() => {
      const intId = setInterval(() => {
        setPointer((prevPointer) =>
          prevPointer + STEP > 100 ? STEP
: prevPointer + STEP
        );
      }, 1000);

      return () => {
        clearInterval(intId);
      };
    }, []);

    const currentTime = new
Date().toISOString();

    return (
      <div id="root-container">
        <DemoChart points={points}
width={500} height={400} pointer={pointer}
/>
      </div>
    );
  }
```

Важливо зауважити, що рух маркерної лінії кожну секунду досягнуто за допомогою функції *setInterval*.

Питання підрахунку рендерів вирішено в певний спосіб. У файл *public/index.html* додаються змінні *renders* та *timeStart*, що наведено в наступному прикладі.

```
    <script>
      var renders = 0;
      var timeStart = new
Date().toISOString();
    </script>
    <div id="root"></div>
```

У такий спосіб змінюється компонент *DemoChart*.

```
    export default function DemoChart({
data, width, height, marker }) {
      // тут нема змін

      // збільшити кількість рендерів
      renders++;

      // тут нема змін

      return <svg ref={svgRef}
width={width} height={height} />;
    }
```

Також виправлено батьківський компонент *App* таким чином:

```
    // без змін
    export default function App() {
      // без змін
      const currentTime = new
Date().toISOString();

      return (
   <div id="root-container">
        <div style={{ marginTop: 20,
marginLeft: 20 }}>
          рендерів: {renders}
          <br />
          час старту: {timeStart}
          <br />
          поточний час: {currentTime}
        </div>
        <DemoChart points={points}
width={500} height={400} pointer={pointer}
/>
      </div>
      );
    }
```

Наслідком змін у коді, який наведено вище, є присутність на екрані показників, необхідних для подальшого оцінювання продуктивності:

- кількість рендерів за годину;
- час першого рендеру компонента;
- поточний час.

Після запуску рішення, наведеного вище, маємо 666 рендерів за 5,7 хв (результат наведено на рис. 1).

рендерів: 666
час старту: 2024-03-05T13:51:43.504Z
поточний час: 2024-03-05T13:57:16.761Z

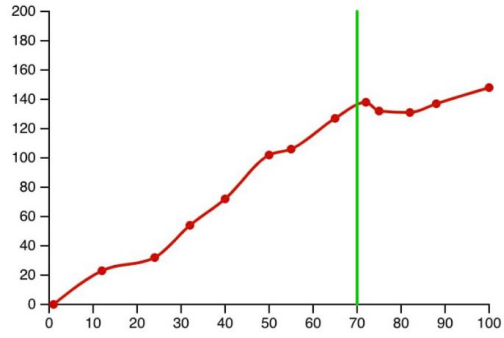

Рис. 1. Приклад роботи рішення без оптимізації

Цей результат свідчить про значні проблеми щодо продуктивності. Далі опишемо способи вирішення цієї проблеми.

Фундаментальною властивістю бібліотеки *D3* є те, що її об'єкти мають змогу зберігати свій власний стан. Із цього можна зробити висновок про недоречність зміни станів *ReactJS* і *D3* водночас. У прикладі вище є факт зміни обох станів – спочатку *ReactJS*, потім *D3*. Оскільки безпосередньо графік відображується за допомогою засобів *D3*, рендери *ReactJS* у цьому разі зайві. Ця проблема вирішується за допомогою хука *useImperativeHandle* [7].

Завдяки передачі частини логіки під керування батьківського компонента останній має змогу оперувати функціоналом компонента напряму *DemoChart* без рендерів. Саме в цьому й полягає оптимізація рішення загалом.

Далі наведемо оптимізований код компонента *DemoChart*, що містить коментарі-пояснення суттєвих змін порівняно з неоптимізованою версією.

```
import React, {
  useEffect,
  forwardRef,
  useImperativeHandle,
  useRef,
} from "react";
import * as d3 from "d3";

const transform = "translate(50,50)";

// forwardRef є необхідним для коректної
роботи useImperativeHandle
// змінна ref є екземпляром посилання на
компонент та передається батьківським ком-
понентом
const DemoChart = forwardRef(({ points,
width, height }, ref) => {
  const chartAreaRef = useRef();
  let svgObj;
  let scaleX;

  // зовнішнє керування
  useImperativeHandle(ref, () => ({
    // встановлення маркера було переміщено
з функції renderSvg
    // у функцію setPointer що керується
ззовні
    setPointer: (value) => {
      if (isNaN(value)) {
        return;
      }
      // прибрати попередній маркер
      svg.selectAll(".pointer").remove();

      svg
        .append("svg:line")
        .attr("transform", transform)
        // фіктивний CSS-клас дозволяє кон-
тролювати стан маркера
        .attr("class", "pointer")
        .attr("stroke", "#00ff00")
        .attr("stroke-linejoin", "round")
   .attr("stroke-linecap", "round")
        .attr("stroke-width", 2)
        .attr("x1", scaleX(value))
        .attr("y1", 200)
        .attr("x2", scaleX(value))
        .attr("y2", 0);
```

```
    },
  }));

  const renderSvg = () => {
    const CHART_WIDTH = width - 200;
    const CHART_HEIGHT = height - 200;

    svgObj =
d3.select(chartAreaRef.current);

    svgObj.selectAll("*").remove();

    scaleX = d3.scaleLinear().domain([0,
100]).range([0, CHART_WIDTH]);
    const yScale =
d3.scaleLinear().domain([0,
200]).range([CHART_HEIGHT, 0]);

    const g =
svgObj.append("g").attr("transform", trans-
form);

    g.append("g")
      .attr("transform", `trans-
late(0,${CHART_HEIGHT})`)
      .call(d3.axisBottom(scaleX));


g.append("g").call(d3.axisLeft(yScale));

    svgObj
      .append("g")
      .selectAll("dot")
      .data(points)
      .enter()
      .append("circle")
      .attr("cx", function (d) {
        return scaleX(d[0]);
      })
      .attr("cy", function (d) {
        return yScale(d[1]);
      })
      .attr("r", 3)
      .attr("transform", transform)
      .style("fill", "#CC0000");

    const line = d3
      .line()
      .x(function (d) {
        return scaleX(d[0]);
      })
      .y(function (d) {
        return yScale(d[1]);
      })
      .curve(d3.curveMonotoneX);

    svgObj
      .append("path")
      .datum(points)
      .attr("class", "line")
      .attr("transform", transform)
      .attr("d", line)
      .style("fill", "none")
      .style("stroke", "#CC0000")
      .style("stroke-width", "2");
  };
  renders++;

  useEffect(() => {
    renderSvg();
  }, [width, height, points]);

  if (!width || !height || !points) {
    return <></>;
  }

  return <svg ref={chartAreaRef}
width={width} height={height} />;
});

export default DemoChart;
```

Наведемо оптимізований батьківський компонент *App*, який має коментарі-пояснення щодо суттєвих змін порівняно з неоптимізованою версією.

```
import React, { useState, useEffect,
useMemo, useRef } from "react";
import DemoChart from "./DemoChart";
import "./style.css";

const points = [
  // без змін
];

const STEP = 5;

export default function App() {
  // Зміна стану що описує маркер
  const [pointer, setPointer] =
useState(STEP);
  // Посилання на DemoChart
  const demoChartRef = useRef();

  useEffect(() => {
    // Якщо стан маркера змінено необхідно
викликати setPointer безпосередньо в
DemoChart
    // Це можливо завдяки
useImperativeHandle

demoChartRef.current.setPointer(pointer);
  }, [pointer]);

  useEffect(() => {
    const intId = setInterval(() => {
      setPointer((prevPointer) =>
        prevPointer + STEP > 100 ? STEP :
prevPointer + STEP
      );
    }, 1000);

    return () => {
      clearInterval(intId);
    };
  }, []);

  const currentTime = new
Date().toISOString();

  // Також для мініміцації кількості ренде-
рів використано стандартну мемоїзацію
  // Тобто рендер DemoChart буде виконано
тільки за умови зміни вхідних показників
```

```
  const chart = useMemo(() => {
    return <DemoChart ref={demoChartRef}
points={points} width={500} height={400}
/>;
  }, [points]);

  return (
    <div id="root-container">
      <div style={{ marginTop: 20,
marginLeft: 20 }}>
        рендерів: {renders}
        <br />
        час старту: {timeStart}
        <br />
        поточний час: {currentTime}
      </div>
      {chart}
    </div>
  );
}
```

Зазначимо, що на рис. 1 подано результати роботи неоптимізованого компонента, а на рис. 2 – графік, отриманий після проведення оптимізації.

Кількість рендерів вказана на кожному графіку. На рис. 1 кількість рендерів дуже велика (666) й постійно зростає в часі. А на рис. 2 кількість рендерів стабільна, дорівнює 2 і не збільшується в часі.

Отже, результатом роботи оптимізованого компонента є лише два рендери, які взагалі не залежать від часу.

рендерів: 2
час старту: 2024-03-05T13:58:03.402Z
поточний час: 2024-03-05T14:07:14.689Z

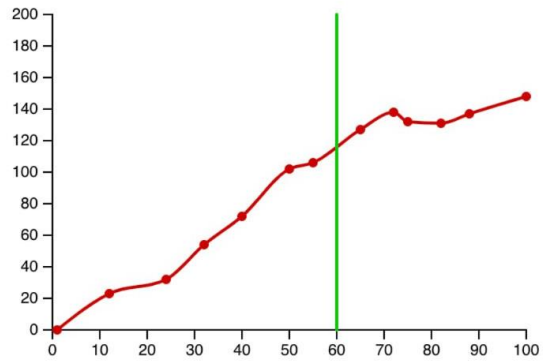

Рис. 2. Приклад роботи рішення з оптимізацією

Також є можливість зменшити цю кількість до одного. Але це не рекомендовано з деяких технологічних *ReactJS* міркувань.

Висновки

У статті розглянуто проблему оптимізації компонента, побудованого за допомогою *ReactJS* та *D3*. Його особливістю є зміна стану в реальному часі.

1. Наведено базову неоптимізовану реалізацію.

2. Проаналізовано способи оптимізації з подальшим втіленням.

3. Аналіз результату показав, що запропоновані рішення не тільки дали змогу мінімізувати кількість рендерів, але й прибрати їх залежність від часу.

Запропоновані рішення дозволять використовувати всю можливість функціоналу *D3* в проєктуванні широкого спектра моніторингових і білінгових систем та систем автоматизованого керування процесами.

Література

1. Adopting D3 in React apps. 2022. URL: https://rajeshnaroth.medium.com/adopting-d3-in-react-apps-a6237a61b59f (дата звернення: 28.02.2024).
2. React.js + D3.js. 2019. URL: https://2019.wattenberger.com/blog/react-and-d3 (дата звернення: 28.02.2024).
3. D3.js with React.js: An 8-step comprehensive manual. 2021. URL: https://blog.griddynamics.com/using-d3-js-with-react-js-an-8-step-comprehensive-manual/ (дата звернення: 28.02.2024).
4. Оптимізація продуктивності в React-застосунку. 2023. URL: https://dou.ua/forums/topic/45406/ (дата звернення: 28.02.2024).
5. Оптимізація продуктивності React додатків. 2024. URL: https://www.linkedin.com/pulse/%D0%BE%D0%BF%D1%82%D0%B8%D0%BC%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F-%D0%BF%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%96-react-%D0%B4%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D1%96%D0%B2-yevhen-rudofylov-gwwve/ (дата звернення: 28.02.2024).
6. Optimizing performance in a React app. 2023. URL: https://blog.logrocket.com/optimizing-performance-react-app/ (дата звернення: 28.02.2024).
7. API-довідник хуків. 2023. URL: https://uk.legacy.reactjs.org/docs/hooks-reference.html#useimperativehandle (дата звернення: 28.02.2024).

References

1. Adopting D3 in React apps. Retrived from: https://rajeshnaroth.medium.com/adopting-d3-in-react-apps-a6237a61b59f (accessed: 28.02.2024).

2. React.js + D3.js. 2019. Retrived from: https://2019.wattenberger.com/blog/react-and-d3 (accessed: 28.02.2024).
3. D3.js with React.js: An 8-step comprehensive manual. 2021. Retrived from: https://blog.griddynamics.com/using-d3-js-with-react-js-an-8-step-comprehensive-manual/ (accessed: 28.02.2024).
4. Optymizatsiia produktyvnosti v React-zastosunku. [Optimizing performance in a React application]. Retrived from: https://dou.ua/forums/topic/45406/ (accessed: 28.02.2024).
5. Optymizatsiia produktyvnosti React dodatkiv. [Optimizing the performance of React applications]. Retrived from: https://www.linkedin.com/pulse/%D0%BE%D0%BF%D1%82%D0%B8%D0%BC%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F-%D0%BF%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%96-react-%D0%B4%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D1%96%D0%B2-yevhen-rudofylov-gwwve/ (accessed: 28.02.2024).
6. Optimizing performance in a React app. Retrived from: https://blog.logrocket.com/optimizing-performance-react-app/ (accessed: 28.02.2024).
7. API-dovidnyk hukiv [Hooks API Guide.] Retrived from: https://uk.legacy.reactjs.org/docs/hooks-reference.html#useimperativehandle (accessed: 28.02.2024).

Чуб Ірина Миколаївна, к.т.н., ст. викладач каф. автоматизації та комп'ютерно-інтегрованих технологій, тел. +38-066–170–27–32, chub.irina.nik@gmail.com,
Демченко Катерина Вікторівна, к.т.н., доц. каф. автоматизації та комп'ютерно-інтегрованих технологій, тел. +380969586977, Yayaska31@gmail.com,
Державний біотехнологічний університет, 61002, Україна, м. Харків, вул. Алчевських, 44.

Optimizing the productivity of solutions built with the help of ReactJS and D3 libraries

Abstract. Problem. *Recently, the problem of the performance of web applications has become particularly acute. This is due to the complexity of the software and the simultaneous deterioration of the code quality. The work considers important aspects of performance of the client part of the web application, built using the ReactJS and D3 libraries. React is known to deliver a faster user interface than most competitors, thanks to its in-house, performance-oriented development approach. However, when your React application starts to scale, you may notice some performance issues or lag. Internal optimization methods may not be sufficient to support the growing traffic and complexity of a fast-paced, enterprise-level application. This is where the question arises: "How to improve performance in ReactJS?". Next, the article will consider important aspects of the performance of the client part of the web application, built using the ReactJS and D3 libraries.* ***Goal.*** *The purpose of the article is to develop a React component (optimized by a new method) in the form of a graph, whose data changes in real time.* ***Methodology****. Analytical research methods and functional programming methodology were used.* ***Results.*** *The paper considers the problem of optimizing a component built using ReactJS and D3. Its feature is the change of state in real time. A new code optimization method is proposed, which allows to minimize the number of renders. The proposed solutions will enable to more effectively use all the capabilities of the D3 functionality when designing a wide range of monitoring and billing systems and automated process management systems.* ***Originality****. A method is proposed that will help improve the performance of the React application and reduce the number of re-renders. The method is developed using additional React hooks and using a reference to the current graphic component from the parent. The method is based on the fundamental property of the D3 library - saving its own state this allows you to get rid of unnecessary renders.* ***Practical value.*** *The proposed solutions will allow you to use all the functionality of D3 when designing a wide range of monitoring and billing systems and automated process management systems. D3 has the flexibility to display a large layer of diverse data and allows data visualization using HTML, SVG and CSS.*

Key words: *ReactJS, React, D3, Web application, optimization, Javascript, rendering, memoization, hooks, component*

Chub Iryna, Ph.D., Assoc. Prof. Department of automation and computer-integrated technologies, tel. +38-066–170–27–32, chub.irina.nik@gmail.com,
Demchenko Kateryna, Ph.D., Assoc. Prof. Department of automation and computer-integrated technologies, tel. +380969586977,
Yayaska31@gmail.com
[1]State Biotechnological University, 44, Alchevsky str., Kharkiv, 61002, Ukraine.