1. Jakub PRZYDALSKI, 2. Radosław WAJMAN

ORCID: 2. 0000-0002-6372-5960

DOI: 10.15199/48.2025.05.07

Renaming a JavaScript object key while preserving its original structure on the example of a graphical JSON editor application

Zmiana nazwy klucza obiektu JavaScript przy zachowaniu jego oryginalnej struktury na przykładzie graficznej aplikacji edytora JSON

***Abstract**. The article presents the assumptions and operation of an innovative JavaScript algorithm that allowed solving the problem of renaming of object keys while preserving their position in the JSON object tree. Achieving such functionality became possible through an imperative approach, bypassing the built-in mechanisms of the JS language. The algorithm performs restructuring and reconstructs the document structure, avoiding time-consuming data copying. The implementation of the algorithm enables free editing of any nested JSON structure. Based on it, a web application was designed, which, thanks to an additional abstraction layer, allows in the graphical view navigation through the loaded JSON object structure, editing its keys, and changing their order while safeguarding against damaging the object's structure.*

***Streszczenie.** Artykuł przedstawia założenia i działanie nowatorskiego algorytmu JavaScript, który pozwolił na rozwiązanie problemu zmiany nazwy klucza obiektu, z zachowaniem jego pozycji w drzewie obiektu JSON. Uzyskanie takich funkcjonalności stało się możliwe dzięki imperatywnemu podejściu pomijając wbudowane mechanizmy języka JS. Algorytm dokonuje restrukturyzacji oraz ponownego budowania struktury dokumentu unikając czasochłonnego kopiowania danych. Implementacja algorytmu umożliwiła swobodną edycję dowolnej zagnieżdżonej struktury JSON. Na jego podstawie przygotowano aplikację webową, która dzięki dodatkowej warstwie abstrakcji oferowanej w widoku graficznym umożliwia nawigowanie po wczytanej strukturze obiektu JSON, edytowanie jej kluczy oraz zmianę ich kolejności zabezpieczając przed uszkodzeniem struktury obiektu.*

Keywords: JavaScript dictionaries, JSON, web application, graphical user interface
Słowa kluczowe: Słowniki JavaScript, JSON, aplikacja internetowa, interfejs graficzny

Introduction

Dynamic web applications require data for their operation. Additionally, the flexibility of the solutions being developed has necessitated the implementation of configuration files that influence system behavior based on needs or constraints. As these solutions have become more standardized, they encountered challenges in standardizing the syntax of the transmitted information. The lack of a standard posed a risk to the stability of applications using data from external systems, which might not be aware of how their data is utilized by other heterogeneous applications in a specific way.

Currently, the official standard for data representation remains XML (Extensible Markup Language)[1], [2]. However, it is not an ideal solution, as reflected in the multitude of alternative approaches and its declining popularity. A detailed analysis of the popularity of various formats - XML (the official standard), JSON (the current leading solution) [3], [4], and other offered formats (CSV, SOAP, YAML, ODATA, AVRO) - clearly indicates the growing popularity of JSON, significantly surpassing other alternatives.

JSON is a lightweight data exchange format. It boasts a low entry barrier. The reading and writing data in this format are easy to grasp compared to other formats like XML. JSON was defined based on a subset of the JavaScript language, following the Standard ECMA-262 3rd Edition - December 1999 [5]. A thorough examination of the format is covered in the following chapters of this work.

The established position of JSON notation has increased demand for solutions that allow free editing and document creation. While there are many editors currently available for editing JSON-compliant document structures, all existing solutions are text-based editors [6]. As JSON-based applications continue to evolve, the need for editors keeps growing, and many required functionalities are not supported by the existing tools.

This situation inspired the development of a JSON document editor with a fresh perspective - an additional graphical user interface. The mechanism is based on extending current solutions, allowing hybrid work between the text-based document and the graphical view. Thanks to an additional layer of abstraction provided in the graphical view, users can navigate through the loaded structure and edit its keys, reorder them without fear of damaging the JSON object's structure. The system simplifies certain operations from multi-step instructions to single clicks or mouse drags, and the necessity of tracking multi-level nesting to add a new item does not hinder document transparency.

Achieving these functionalities became possible through an imperative approach that bypasses built-in JavaScript mechanisms. A new algorithm for restructuring and reconstructing the document structure was developed, avoiding time-consuming data copying. The implemented solution introduces the ability to change the name and order of object keys while preserving the content of the object. An internet application was built, allowing users to reorder array elements or object fields with a single mouse drag, and the result is reflected in both the textual structure and the saved data. The implementation leverages the Svelte library, known for its fast display of changing data in browsers, as opposed to popular solutions that require maintaining auxiliary structures [7].

JavaScript limitations

The JavaScript programming language [8] lacks a built-in mechanism for editing object keys. After analyzing the issue and creating prototypes, several conclusions were drawn:

The only solution that allows working directly with the original structure after converting it to an object involves creating a new object key, copying the modified value, and deleting the old key. While this approach ensures the correctness of the resulting data, it disrupts its original structure when transformed back from text to JSON format. Consequently, this solution was rejected due to usability concerns.

Another potential solution would be to create a copy of the original object field by field until encountering the edited key, which would be created under a new name. However,

this process would engage the entire (sometimes large) object for each key renaming operation, resulting in highly inefficient performance. Likewise, this approach did not meet the requirements of the intended system due to its computational complexity. The computational time for renaming the same key increases as the structure size grows.

A solution without these drawbacks emerged through the implementation of a restructuring and JSON object-building mechanism based on a more imperative approach than the built-in JSON prototype solutions in JavaScript. The structure is created when switching to graphical mode, and the reverse transformation to JSON format occurs when returning to text mode or saving to a file. The architecture developed for the project purposes allows storing information about object fields in the appropriate structure fields.

```
interface IDataKey {
    key?: string,
    type?: Types,
    value?: any,
    children?: IDataKey[],
    id?: string;
}
```

Fig. 1. Object field structure in the implemented architecture

By storing information about the key in a separate field (“key”), the edition process does not affect its position or disrupt the order of other keys. All object fields (and array elements) are stored in an array under the key “children.” This design allows us to avoid the need to build and maintain paths to individual contexts. A detailed analysis of the structure and the argumentation of the specific fields’ definition can be found in the further part of this work.

Data structure

Developing the algorithm that allows for free editing of the JSON structure became the core of the entire project and required a detailed analysis of the issue from many perspectives. At the initial stage of work preparation, two solutions turned out to be not good enough to allow full implementation or were not optimal. The final solution, as it was implemented, developed with new mechanisms to cover the requirements of the implemented system.

The structure initially passed to the application is of string type. The conversion between the string and the JavaScript object and vice versa is done by using the built-in methods, respectively: JSON.parse() and JSON.stringify().

Restructuring

Working directly on a JavaScript object would prevent the optimal implementation of the system's assumptions. The object received after conversion from text had to be remapped to a structure compatible with the prepared algorithm.

All keys of the original object are renamed to the values of the corresponding fields of the objects in the array:

Key - the value of the object key or the index number of the array,

Type – one of the values from: null, array, object, string, number, boolean. Checked in a specific order to distinguish between all types present, with appropriate distinction. In JavaScript, null, array, and object are all object type values [9].

Each of these values forces a different handling of data, which required the extension of the type checking algorithm to pre-filter out null values and arrays.

Value – according to the following scheme:

- for null values, it is a value representing the intentional absence of any value (null),
- for the array it is an empty array ([]),
- for the object, it is a value representing an empty object ({}),
- for other types (simple types) it is the value of the variable under the key.

Children – an array of values of children, objects with the same fields as the parent context, or elements of an array. It allows to define a hierarchical data structure.

```
static getObjectKeys(obj, previousPath = 'root', key = 'root') {
    const dataKey: IDataKey = {
        type: Utils.getType(obj),
        key: key,
        value: Utils.getValue(obj),
        id: Utils.createGuid()
    };
    if (obj && Object.keys(obj).length) {
        if (!dataKey.children) {
            dataKey.children = [];
        }
        Object.keys(obj).forEach((key) => {
            const currentPath = previousPath ? `${previousPath}.${key}` : key;
            if (typeof obj[key] === 'object' && obj[key] !== null) {
                const innerKeys = this.getObjectKeys(obj[key], currentPath, key);
                if (innerKeys?.children?.length > 0) {
                    dataKey.children.push(innerKeys);
                }
            } else {
                dataKey.children.push({
                    key,
                    type: Utils.getType(obj[key]),
                    value: Utils.getValue(obj[key]),
                    id: Utils.createGuid()
                });
            }
        });
    }
    return dataKey;
}
```

Fig. 2. The restructuring method

Example input structure

```
{
  "arr": [
    1,
    2,
    3
  ],
  "bool": true,
  "nullable": null,
  "num": 123,
  "Obj": {
    "a": "b",
    "c": "d"
  },
  "string": "Hello World"
}
```

```
{
    "type": "object",
    "key": "root",
    "value": {},
    "id": "5d4354b7-0d04-7319-6102-e062957341f2",
    "children": [
        {
            "type": "array",
            "key": "arr",
            "value": [],
            "id": "28cddbb9-c247-bcff-7470-513de84336bc",
            "children": [
                {
                    "key": "0",
                    "type": "number",
                    "value": 1,
                    "id": "cee564a5-5413-fee5-7b24-91c3ad5d3b04"
                },
                {},
                {}
            ]
        },
        {
            "key": "bool",
            "type": "boolean",
            "value": true,
            "id": "e43ce6ff-d452-db00-5663-52cc370ac56a"
        },
        {
            "key": "nullable",
            "type": "unknown",
            "value": null,
            "id": "1a8142f7-55e0-46f0-8fb5-71813f12efca"
        },
        {
            "key": "num",
            "type": "number",
            "value": 123,
            "id": "6f7b79bf-ad71-8dfa-dfc0-f23ba08de7a4"
        },
        {
            "type": "object",
            "key": "Obj",
            "value": {},
            "id": "0b9a14e9-e29a-8a93-3a78-d9a6337ae9d7",
            "children": [
                {
                    "key": "a",
                    "type": "string",
                    "value": "b",
                    "id": "9634b33e-766c-55a6-d602-2c6c0a4d7954"
                },
                {}
            ]
        },
        {
            "key": "string",
            "type": "string",
            "value": "Hello World",
            "id": "ad5c7f1f-fb3e-d22d-c0f1-3d2dc97889e5"
        }
    ]
}
```

Fig. 3. Example structure before and after restructuring

Identifier (id) is a value represented by a Globally Unique Identifier. A field used when reorganizing elements of an array or object, as a pointer to an element that distinguishes individual elements (an index that is a dynamic value is not suitable for this purpose).

Building the output

Working on a restructured object allows for flexible data modification. When switching to text mode or writing to file, the input is rebuilt. The algorithm iterates over the values of the structure recursively. For keys that are objects, the initial value of the output object is set to: "{}". Then, for each child key, a lower level of the tree is built, and its value is assigned under the key in the parent context.

For keys that are arrays, the initial value of the output object is set to: "[]". Then, for each child element, a lower level of the tree is built, and its value is pushed to the end of the parent context array.

For other types, the initial value of the output object is directly rewritten from the input object – this is equivalent to reaching the leaf of the structure and finishing building a given branch of the tree.

```
static buildObject(obj: IDataKey): unknown {
    const result = obj.value;

    if (obj.type === Types.Array && Array.isArray(result)) {
        obj.children.forEach((child: IDataKey) => {
            result.push(this.buildObject(child));
        });
    } else if (obj.type === Types.Object) {
        obj.children.forEach((child: IDataKey) => {
            result[child.key] = this.buildObject(child);
        });
    }
    return result;
}
```

Fig. 4. Method to build the output

When all the branches of the tree reach their leaves, the finished structure is returned to a text editor and is passed to a text file for writing.

Graphical JSON editor

The result of the work was a system that allows to edit JSON files and structures. The solution in the form of a web application is offered to a wide range of users of documents in the above format, both for commercial and non-commercial use. The system offers two cooperating views that allow for parallel operation in both modes, depending on the user's needs.

The figure 5 presents a standard view offering two data inputs on one screen as a standard JSON text document editor.

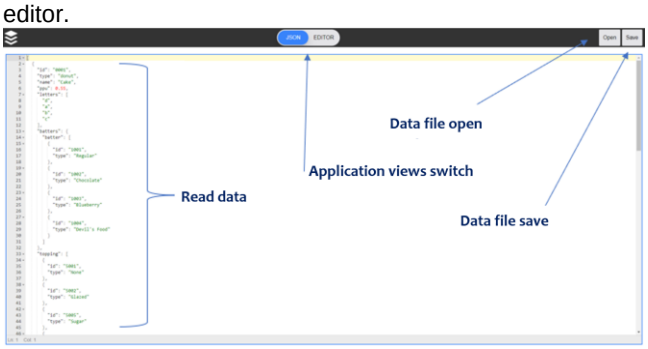

Fig. 5. Text editor view

In the case of damage to the data structure in the text mode, the user will be informed about syntax errors in the structure, which at the same time limits other system functionalities.

Nevertheless, many solutions based on a text editor allow to freely work on uncomplicated data structures. As the content grows, working in the graphic mode will be much more convenient. Therefore, at any stage of operation, the user can switch between modes, depending on his needs – see figure 6. Then, a tree of the structure of the loaded data will be displayed in the left navigation panel.

The graphical editing mode provides additional functionalities. When working on long structures with many nests, it is possible to constrain the displayed elements with chevrons to improve readability.

In the case of copying flattened structures (without indentations), their readability for humans is disturbed. The system offers the possibility of displaying data with the use of appropriate indentations to increase readability, regardless of the formatting of the entered data in the graphical mode.

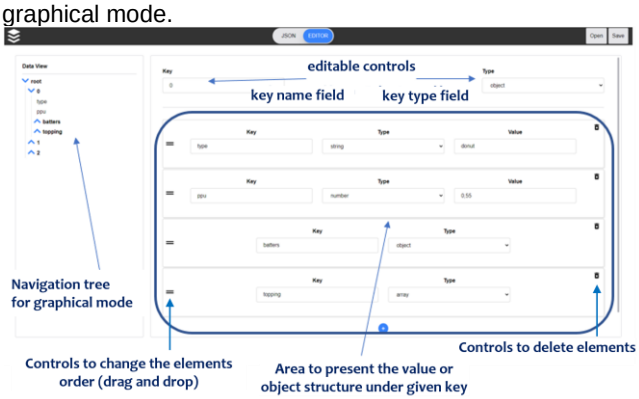

Fig. 6. Graphical editor view

In the event of distorting the structure with redundant or missing indents and additional rows, the system allows to reformat the data.

When working with complex structures, their complexity can interfere with the ability to quickly analyze content. In the prepared system, it is possible to set a context that displays the details of only a given nesting, allowing to fully focus and review the data.

The content of the main section of the application has been stored in the so-called "store". This is a functionality of the Svelte library, which allows to set the value of a given variable and access it from anywhere in the system. The value of such a section is a kind of Singleton, i.e. it will have consistent data for any place of the call. This type of object allows to change data, subscribe – i.e. notify about the change of content and allows to define methods that limit access and freedom to influence the shape of the section. In addition to the two-way binding functionality and event dispatcher, it is one of the three functionalities of the Svelte library, used to transfer data between application components.

One of the initial challenges of the developed system was to design an architecture that would enable an optimal algorithm for renaming the key, while maintaining the order of keys in the object. The system offers such a mechanism. By implementing the new data structure, changing the name of the key consists in overwriting the value of the "key" key for the edited context. This functionality does not require any additional maintenance and additional data copying in the prepared architecture. It is also possible to change the value and type under lock and key.

Objects that are structures with variable architecture require the ability to expand their content with additional fields. In the case of this system, this functionality is implemented with one click. The added base element is immediately ready to be edited by key name, type or assigned value.

According to the prepared architecture, adding a field to an object comes down to adding an array element under the key "children" in the edited context. This solution allows

to skip the tedious process of analyzing paths for later building the output. At the same time, the mechanism maintains the declared order of the object's fields.

Similarly, arrays, which are dynamic structures for storing data, require the functionality to quickly add elements. In this system, the user adds a base element with the possibility of adapting it to his requirements with one click.

In addition, standard JSON text editors do not support changing the order of array elements, which is a desirable functionality due to the ordered nature of such a structure. In the graphical mode, the user can change the order of elements using the drag and drop technique. Therefore, it was necessary to define the "id" field storing a unique identifier for each element. This is the only use for this field. On its basis, the positions of the dragged elements are identified, both in the state of holding over the zone (consider) and after dropping (finalize).

Summary

The aim of the work was to develop an innovative JavaScript algorithm that allowed to solve the problem of renaming the objects' keys while maintaining its position in the JSON object tree. In addition to this problem, the algorithm allowed for free editing of any nesting of the structure, which allowed for full coverage of the requirements for functional assumptions. An appropriate data structure has been designed to protect against unwanted copying of large objects. On its basis, a web application written in TypeScript using the Svelte library was prepared. A further goal was set to develop the system with functionalities such as a toolbox or a TypeScript interface generator.

The developed solution can be made available to companies and private users for editing and creating complex documents, ensuring the correct file structure at every stage of shaping JSON formats. The system does not limit the functionalities offered by solutions based on text editors but is an additional layer cooperating in parallel to the text editor. Text editors require the user to carefully edit and control the actions performed for fear of damaging the structure. The system has simplified some of the mechanisms from multi-step instructions to single mouse clicks or drags. The need to keep track of multi-level nesting to add a new item will not be an obstacle to achieving document clarity.

***Authors**: Jakub Przydalski, BSc is a graduate of Lodz University of Technology. His interests lie in exploring and advancing the field of JavaScript programming, with a particular focus on low-code and no-code solutions. He is passionate about leveraging these platforms to democratize software development, enabling more people to create robust applications without extensive coding knowledge. Additionally, he enjoys implementing various process automations, constantly seeking innovative ways to streamline workflows and enhance productivity.*

Radosław Wajman, DSc. Ph.D. Eng. is a professor at the Institute of Applied Computer Science at the Lodz University of Technology. In his work, he deals with the issues in the field of software engineering, machine learning, fuzzy inference for industrial approach and also electrical capacitance tomography, two-phase gas-liquid flow recognition, image reconstruction, analysis and processing.

REFERENCES

[1] T. Arnold and L. Tilton, "Data Formats," in *Humanities Data in R: Exploring Networks, Geospatial Data, Images, and Text*, Cham: Springer International Publishing, 2024, pp. 249–275. doi: 10.1007/978-3-031-62566-4_12.

[2] P. Drzymała and H. Welfle, "Przetwarzanie dużych zbiorów danych XML z użyciem struktur relacyjno-hierarchicznych w systemie IBM DB2," *Przegląd Elektrotechniczny*, vol. 6, p. 33, 2017.

[3] A. Šimec and M. Magličić, "Comparison of JSON and XML Data Formats," in *Central European Conference on Information and Intelligent Systems*, Varaždin, 2014. Accessed: Jul. 24, 2024. [Online]. Available: https://www.researchgate.net/publication/329707959_Comparison_of_JSON_and_XML_Data_Formats

[4] P. Drzymała and H. Welfle, "Mechanizmy wsparcia standardu JSON w celu wydajnego przechowywania i przetwarzania danych w środowisku IBM DB2," *Przegląd Elektrotechniczny*, vol. 2, p. 182, 2023.

[5] "ECMA International. (n.d.). ECMA-404 The JSON Data Interchange Standard." [Online]. Available: http://www.json.org/

[6] T. Marrs, *JSON at work : practical data integration for the web*, O'Reilly Media. 2017.

[7] R. Harris, "Svelte 3: Rethinking reactivity." Accessed: Jul. 24, 2024. [Online]. Available: https://svelte.dev/blog/svelte-3-rethinking-reactivity

[8] D. Herman, *Effective JavaScript*, Pearson Education. Addison Wesley, 2012.

[9] "JavaScript data types and data structures," MDN Web Docs, Mozilla Foundation. Accessed: Jul. 24, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures