A Methodological Approach to Building a ReactJS Code Generation Website Builder

Xuan-Y T. Dam
University of Science, VNU-HCM
Ho Chi Minh, Vietnam
19120160@student.hcmus.edu.vn

Hoang-Vu Vo
University of Science, VNU-HCM
Ho Chi Minh, Vietnam
19120727@student.hcmus.edu.vn

Duc-Huy Nguyen
University of Science, VNU-HCM
Ho Chi Minh, Vietnam
ndhuy@fit.hcmus.edu.vn

Vi-Minh Luong
University of Science, VNU-HCM
Ho Chi Minh, Vietnam
lvminh@fit.hcmus.edu.vn

Minh-Triet Tran
University of Science, VNU-HCM
Ho Chi Minh, Vietnam
tmtriet@fit.hcmus.edu.vn

Abstract

Website builders have emerged as powerful tools, enabling users to create websites without extensive coding knowledge. However, there is a significant dearth of research and articles that delve into the essential methodological aspects of developing a website builder that empowers users to maintain complete ownership of their websites by having access to the source code for ongoing development and seamless deployment. This article aims to fill this gap by presenting developers with a comprehensive approach to constructing their own website builder, with a specific emphasis on leveraging ReactJS code generation. The website builder encompasses critical features such as multiple-page support, drag-n-drop functionality for pre-designed elements, event handling, theme customization, and dynamic data integration. The resulting ReactJS source code adheres to industry-standard practices, ensuring readability, customization, and reusability. This article serves as a valuable resource for developers aspiring to create a website builder that empowers users with full ownership and accessible source code.

CCS Concepts: • **Software and its engineering → Source code generation**.

Keywords: website builder, website development tool, source code generation, ReactJS generation

ACM Reference Format:
Xuan-Y T. Dam, Hoang-Vu Vo, Duc-Huy Nguyen, Vi-Minh Luong, and Minh-Triet Tran. 2023. A Methodological Approach to Building a ReactJS Code Generation Website Builder. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00

1 Introduction

In the digital era, having a professional website is crucial for promoting businesses, services, or personal brands. However, designing and building a website often requires specialized programming and design knowledge, making it challenging for those without experience or time constraints. Various approaches to generate frontend code already exist, including code generation from screen layouts, sketches, design tools like Figma, and website builders.

Machine learning and deep learning techniques have been explored in research papers, including pix2code: Generating Code from a Graphical User Interface Screenshot [1], Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps [5], and Generating webpages from screenshots [4]. These approaches show promise, but they require extensive data and are most effective for simple websites with common structures. Additionally, the generated source code can be complex, poorly organized, and difficult to understand, hindering future modifications and edits. Personalization options, such as custom themes, primary colors, and font families, are also limited.

Another approach involves generating code from design tools like Figma, where plugins [3] such as Anima, TeleportHQ, Locofy, and pxCode have gained recognition. However, using these plugins to produce high-quality source code demands meticulous attention to detail, adherence to specific rules, and an understanding of the underlying code generation mechanisms. This process can be time-consuming and challenging.

Recognizing these challenges, our team has chosen to leverage the power of website builders to address the need for accurate, efficient, and customizable code generation. Website builders offer intuitive interfaces and enable users to effortlessly drag-n-drop components while customizing various parameters. Several website builders, including Wordpress [10], Wix [9], Teleporthq [8], Builder.io [2], Quarkly.io [7], have already entered the market. However, existing

products exhibit certain drawbacks, such as complex interfaces, rigid drag-n-drop functionalities, non-user-friendly code generation in terms of source organization and naming, and limited customization options. Moreover, performance issues may arise due to feature overload, and the resulting source code may lack sophistication and cleanliness.

To overcome these limitations, our approach focuses on building a website builder that generates ReactJS code. This research aims to fill the existing gap in literature by providing developers with a comprehensive methodology for constructing their own website builder with a specific emphasis on ReactJS code generation. Our methodological approach entails developing a user-friendly website builder with predesigned components and templates. Users can effortlessly drag-n-drop components, customize parameters, and personalize visual aspects such as primary colors, font sizes, font weights, spacing, and button styles. Additionally, users can create dynamic data and seamlessly integrate it within components. The website builder supports multi-page designs and enables smooth navigation through routing.

For frontend code generation, our website builder facilitates the serialization of nodes in the frontend and their deserialization in the backend. Each component will have specific props, with values passed through custom user parameters. The resulting project adheres to the typical structure of ReactJS source code, granting users a high level of customization and reusability. Users can combine the website builder with their programming expertise to develop advanced and intricate features for their websites.

By presenting this methodology, our paper aims to equip developers with the necessary knowledge and tools to build their own website builders and integrate ReactJS code generation effectively.

2 Website Builder and Implementation

Our website builder implementation is inspired by the critical role played by the browser rendering engine in managing web page display. To replicate its functionalities, we have developed a powerful page editor that acts as a drag-and-drop canvas region. This **Editor** closely emulates a rendering engine, empowering users to effortlessly handle component rendering, updates, movement, and removal. We acknowledge the significant inspiration we derived from Craft.js [6] in the implementation of our website builder.

To enhance the user experience, our implementation features a display of the page tree/layout with **Layer** management. This functionality provides users with a clear and organized visualization of the element structure. Additionally, we offer convenient undo/redo functionality with **History** management, enabling users to easily revert or restore previous actions.

In the subsequent sections, we will provide detailed insights into the Editor, Layer, and History components of our website builder implementation. These components work harmoniously to facilitate a seamless and efficient web page creation process for our users.

2.1 Editor

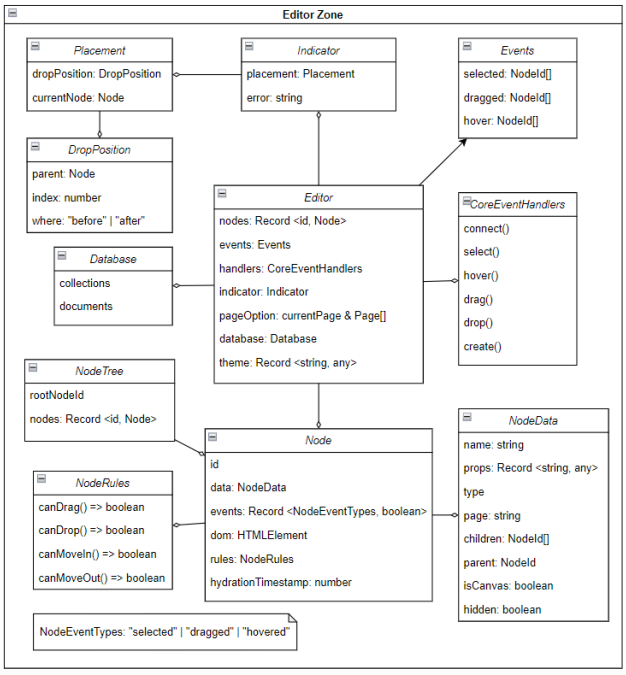

Figure 1. Editor Architecture

In Figure 1, we present the architecture of the Editor, showcasing its essential components:

1. **Node**: The Editor comprises a node list responsible for managing all the nodes within the editor. Each node possesses its own data, including properties (props), type, children, parent, and page. Nodes govern their behavior through state events such as 'selected', 'dragged', and 'hovered'. They also define rules, such as canDrag(), canDrop(), canMoveIn(), and canMoveOut(), to manage their interactions.
2. **Indicator**: During dragging operations, the indicator is displayed as a colored line, indicating the calculated drop position. It changes color to green if the drop position is valid; otherwise, it appears in red. The indicator stores the drop position, including the index of the current node among its parent's children and the position ('before' or 'after').
3. **Events**: The Editor maintains a list of NodeIds for the 'selected', 'dragged', and 'hovered' states. It includes core event handlers that facilitate the dispatching of appropriate actions in response to user interactions with nodes.
4. **Theme**: The Editor allows users to define their own themes, which consist of a list of key-value properties.

These properties can represent colors (e.g., primary color) or numerical values (e.g., padding, font size), empowering users to customize the visual appearance of their websites.

5. **Database**: To support dynamic data, the Editor seamlessly integrates with a NoSQL database system. This integration enables users to incorporate and manage dynamic data within their websites, enhancing their functionality.
6. **Page**: The Editor facilitates the creation of multiple pages. It maintains a list of pages and tracks the currently active page. Each node belongs to a single page, and every page includes a root node that serves as an automatically created container upon page creation.

These components work together to form the foundation of our website builder implementation. In the following subsections, we will delve into their functionalities and interactions, providing a comprehensive understanding of how the Editor architecture powers our website builder's inner workings.

2.1.1 Selector. The Selector is a key feature of our website builder, offering a collection of pre-designed elements that can be easily dragged and dropped into the Editor for web page creation. The Selector comprises two types:

1. **Basic Elements**: These elements form the fundamental building blocks of web pages, encompassing components such as Containers, Buttons, Text, Inputs, Links, Anchors, Images, Videos, and more. Each Basic Element is seamlessly **connected to the Editor** through its corresponding React Component, enabling the Editor to effectively manage and handle events associated with its Node. This integration empowers users to perform various actions and effortlessly **modify element properties** via **user Settings**.
2. **Built-in Templates**: Our website builder offers a selection of pre-designed templates that feature well-arranged combinations of multiple Basic Elements. By simply selecting and inserting templates like Headers, Footers, Menus, Introductions, Contents, Banners, and others, users can expedite the web page creation process. These templates are designed to streamline the workflow and save users valuable time.

To enhance the user experience, we provide a comprehensive toolbar setting that offers a variety of visualization options. The toolbar supports various input types, including number fields, text fields, dropdown menus, sliders, checkboxes, radios, color pickers, and image upload functionality. Additionally, we offer support for custom events, such as navigation actions, scrolling to a specific href, and displaying pop-up messages.

With these powerful features and customization options, our website builder empowers users to create visually stunning and interactive web pages with ease and efficiency.

2.1.2 Understanding the Core Event Flow. Within the Editor, we encounter two primary flows depicted in Figure 2 and Figure 3. The first flow involves the drag-and-drop process, where users can insert a Selector into the Editor to create web pages. The second flow entails moving a node to a new position within the Editor. After having grasped the overall flows, we will explore the crucial step that drives these interactions: computing the Indicator. This step is reliant on the findingPosition() function, which plays a vital role in determining the position of the Indicator.

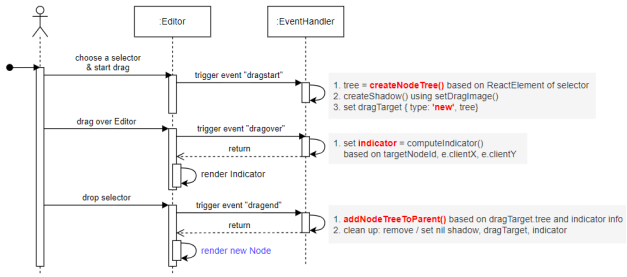

Figure 2. Drag-n-drop a Selector into the Editor

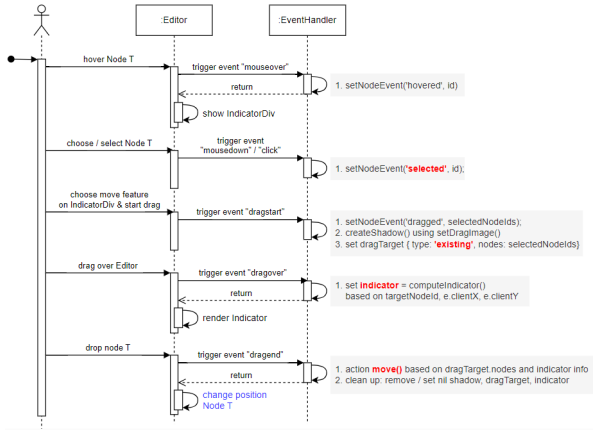

Figure 3. Move a Node to another Position in the Editor

The Indicator is an essential visual guide that aids users during dragging operations, offering real-time feedback on the calculated drop position. Its primary function is to assist users in accurately determining the optimal placement for the dragged node. To gain a deeper understanding of this vital step, we provide a suggested source code snippet that illustrates a handler for finding Position, enabling efficient computation of the Indicator:

The findPosition function, illustrated in listing 1, is responsible for determining the suitable index for the dragged-Node among the list of children nodes of the parentNode,

```
function findPosition(parent:Node, dims:NodeInfo[], x,y:number){
  result: DropPosition = { parent, index: 0, where: 'before'};
  leftLimit = rightLimit = bottomLimit = xCenter = yCenter = 0;
  for (let i = 0, len = dims.length; i < len; i++) {
    dim = dims[i];
    xCenter = dim.left + dim.outerWidth / 2;
    yCenter = dim.top + dim.outerHeight / 2;
    // Skip if over the limits
    if ((rightLimit && dim.left > rightLimit) ||
      (leftLimit && dim.right < leftLimit) ||
      (bottomLimit && yCenter >= bottomLimit))
      continue;

    result.index = i;

    if (dim.inFlow) {
      if (y < yCenter) {
        result.where = 'before';
        break; // yCenter covariate over loop
      } else result.where = 'after'; // after last element
    } else {
      if (y < dim.bottom) bottomLimit = dim.bottom;
      if (x < xCenter) {
        rightLimit = xCenter;
        result.where = 'before';
      } else {
        leftLimit = xCenter;
        result.where = 'after';
      }
    }
  }
  return result;
}
```

Listing 1. Find Position to Compute Indicator

based on the current mouse position (x, y). The approach employed by the function considers whether **the children nodes are spread vertically or not** in order to calculate the index and determine the position before or after that index. A child node is marked as `inFlow = true` if it satisfies the following conditions:

1. Parent style: The parent node's style should not be `'float'` or `'grid'` (as this could potentially cause child nodes to overlap). It can be `'flex'`, but `'flex-direction'` must be `'column'`.
2. Child style: The child node's position must not be `'fixed'` or `'absolute'`, and its display CSS property must introduce **line breaks before and after** the element.

For child nodes where `inFlow = true`, since the children of the parent node are listed in order from top to bottom, the `yCenter` value increases over the loop. Therefore, the loop can be terminated early if a child node with `y < yCenter` is found.

For child nodes where `inFlow = false`, overlapping cases can occur. Even though the children of the parent node are listed in order from left to right and top to bottom, it cannot be guaranteed that the next child node will have `next.xCenter > current.xCenter`. In this case, the loop continues while **limiting** the left, right, and bottom boundaries to find the correct index.

By examining the core event flow and focusing on the implementation of this crucial step, we aim to deepen your understanding of how the Indicator is computed and how it enhances the drag-and-drop experience within the Editor.

2.2 Layer

2.2.1 Layer architecture. Layers in a website builder play a vital role as they represent a hierarchical arrangement of elements within the website's structure. Traditionally, a straightforward approach to implementing layers involves recalculating and re-rendering them whenever changes occur in the element tree. However, this approach can lead to increased computational overhead and re-rendering processes. Furthermore, as the need for additional layer features arises, such as **drag-and-drop** functionality to reorder or reposition layers, or options to hide or show layers, this approach becomes inadequate. To address these limitations, alternative approaches are needed to indicate and manipulate each layer effectively, enabling enhanced user interactions and customization within the website builder interface, as shown in Figure 4.

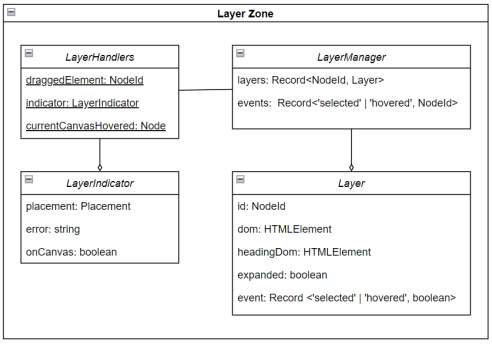

Figure 4. Layers Architecture

Layer component includes attributes such as id, dom, headingDom, expanded, and event to represent the corresponding node, DOM element, heading, expanded state, and selected/hovered state. This is depicted in the figure 5

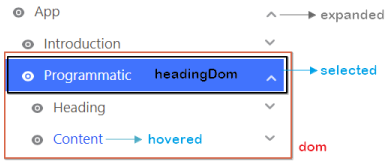

Figure 5. Layer visualization

We propose a **LayerManager** to manage the current layer states, binding node IDs to layers and tracking events for selection and hovering. This enables convenient control over layer events, such as easily turning off events for previously selected or hovered layers.

```
const currentNodeId = state.events[eventType];
layers[currentNodeId].event[eventType] = false;
```

To enable drag-n-drop functionality for layers, we introduce the **LayerHandlers** component. It includes attributes such as draggedElement (representing the ID of the currently dragged layer), indicator (indicating the drop position), and currentCanvasHovered (identifying the parent or adjacent Node that the user intends to drag the layer to by hovering over it). Additionally, we introduce the **LayerIndicator** component with attributes like placement (calculating the drop position), error (displaying a red indicator for incorrect actions, such as attempting to drop an element onto an undroppable parent Node), onCanvas (a flag determine whether the user is currently dragging inside the element or dragging enter its boundary). These components greatly enhance the user experience by providing seamless layer manipulation and visual feedback during drag-n-drop interactions.

2.2.2 Understanding drag-n-drop functionality.

The flow of the drag-n-drop layer is described in Figure 6, with a focus on four essential functions: DragStart, DragEnter, DragOver, and DragEnd.

The DragStart function is responsible for initiating the dragging process by assigning the draggedElement as the layer ID. On the other hand, the DragEnd function finalizes the action by moving the corresponding node based on the draggedElement and indicator information.

The responsibilities of DragEnter and DragOver functions are related to updating the indicator. It is important to note that we only call the computeIndicator function in DragEnter and not in DragOver. This is because the user cannot drag over an element without first dragging into its boundary immediately before.

In the DragEnter function, the new parent of the dragged Node is set as the parent of the currently hovered Node. As for the DragOver function, the parent is the currently hovered Node itself, and we simply append the dragged Node to the list of its children.

To provide a visual representation of these two cases, please refer to Figure 7 and Figure 8.

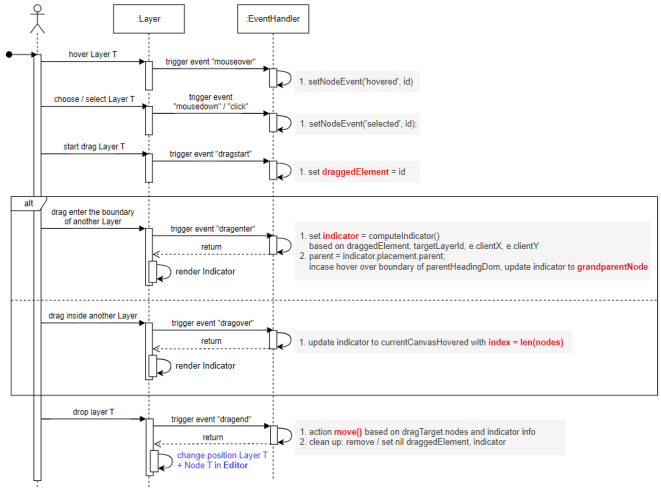

Figure 6. Drag-n-drop layer flow

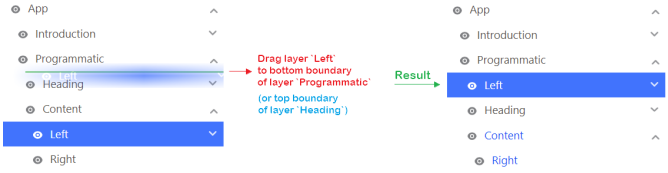

Figure 7. Drag a Layer enters the Boundary of another Layer

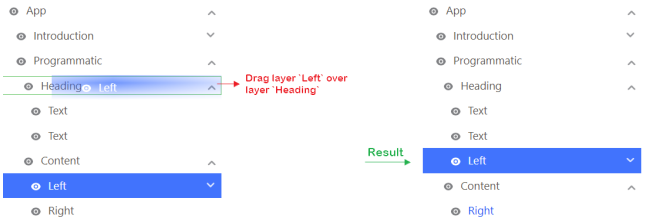

Figure 8. Drag a Layer Over another Layer

2.3 History

We manage all the information of the Editor using a state management system, utilizing multiple small actions to update each relevant piece of information. With each action triggered when user interacts directly to modify the Editor's information, we keep track of this editing history. The History management involves storing two pieces of information:

1. **Timeline**: This is an array containing [patches, inversePatches, timestamp]. The patches and inversePatches contain information related to the operations and are managed by the immer library. The timestamp indicates the time when the operations were performed.
2. **Pointer**: This is an index reference that points to the current item in the Timeline.

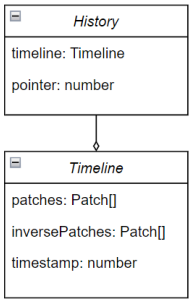

Figure 9. History Architecture

Immer is an open-source JavaScript library, is used for History's implementation. It enables easy and efficient creation and modification of immutable data. In the context of

the History feature, immer allows storing a list of applied changes in chronological order (Timeline) instead of only storing the final state. This enables tracking the history of changes and performing undo and redo operations based on this list.

When user requests an undo operation, immer retrieves the current state and applies the previous changes from the list in reverse order (inversePatches). Similarly, when the user requests a redo operation, immer retrieves the current state and applies the subsequent changes stored in the list (patches). The state transformation and storage of history (timeline and pointer) during the sequential execution of node addition operations in the Editor, as well as the method by which history records information for the purpose of supporting redo and undo operations, are illustrated using the example in Figure 10 (the data of a node in the example has been simplified for the purpose of describing the mechanism).

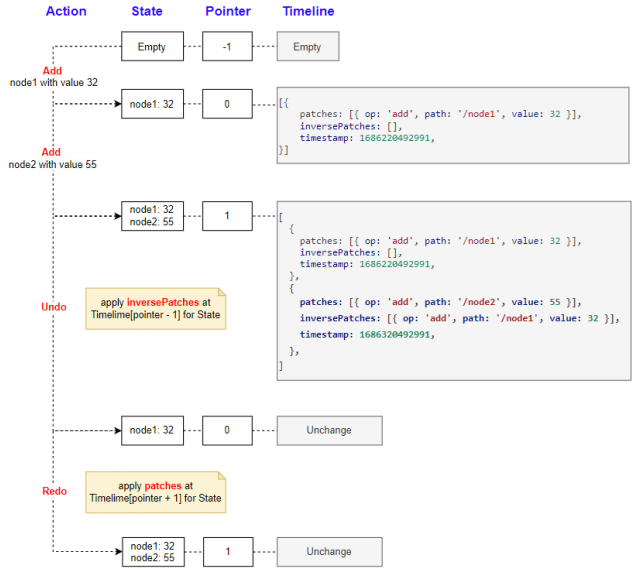

Figure 10. Example for History Mechanism

3 Generating ReactJS Source Code

We create a ReactJS app based on the information provided in the list of nodes, list of pages, theme, and databases of the user's website project. The goal is to generate the source code that powers the website.

To generate a page for the project, we follow a step-by-step process. We start by merging the nodes in a tree structure, beginning with the node that has an ID starting with 'ROOT'. Using a top-down recursive approach, we merge the remaining child nodes, ensuring the correct hierarchy and organization of the elements. Additionally, we create routing for the pages based on the information in the list of pages, mapping each page to its corresponding path.

In our approach, managing components presents a challenge due to the varying complexity between the frontend and backend. Frontend components require hooks for establishing connections with the Editor, managing event listeners, and performing essential actions, while the backend follows a lightweight component structure adhering to standard practices. To ensure consistency and minimize the need for modifications in both frontend and backend, we propose segregating the frontend component version. This involves maintaining a frontend component version that mirrors the backend structure, with more complex event handlers extracted into a separate file. Utilizing a Git submodule to store shared components allows for easy integration, enabling both frontend and backend to effortlessly pull the latest version. By adopting this approach, we enhance consistency and efficiency in code maintenance efforts.

In addition, we handle themes and dynamic data using an AppContext. This global context serves as a centralized hub for managing the theme and database. We utilize hooks to retrieve values from the Node props, ensuring seamless integration and access to the necessary data.

By implementing this approach, we can efficiently generate the ReactJS source code for the user's website project, ensuring consistent and well-structured components while facilitating easy management of themes and dynamic data through the AppContext and associated hooks.

3.1 Optimizing Source Code Generation Speed

To ensure an efficient and fast generation of the source code, we have implemented several strategies to optimize the process:

1. Pre-Configuration of a Source Code Base: We start by pre-configuring a base source code structure and configuration files that remain unchanged during the generation process. By copying the pre-configured code base, we eliminate the need to regenerate the entire code base for each project. This approach significantly reduces the time required to initiate the code generation process.
2. Leveraging Concurrency with Golang: Since each page can be generated independently, we can take advantage of the effective concurrency support provided by Golang. By utilizing concurrent processing, we can parallelize the generation of multiple pages, significantly improving the overall speed of the code generation process.
3. Smart Code Formatting with Prettier: Once the source code is generated, we optimize its formatting using the popular tool called Prettier. However, to avoid unnecessary formatting operations, we only format the files that have changed compared to the base source code. This smart code formatting approach minimizes the

time spent on formatting while ensuring consistent and clean code throughout the project.

4. Security and Payload Optimization: To address security concerns and optimize payload size, we provide the option to encode the information using Base64 before transmitting it to the backend. This encoding ensures data integrity during transmission while reducing the payload size. On the backend side, the information is decoded to retrieve the original data securely and efficiently.

By incorporating these optimization techniques, we can generate the source code for user website projects at an accelerated pace. This optimized approach enables us to deliver a seamless and responsive code generation experience, empowering users to quickly bring their website ideas to life.

4 Results

The implemented outcomes include the website builder interface, the structure of the generated source code, and the resulting interface of the website when running the source code.

Figure 11 shows the intuitive and visually appealing user interface of the website builder, ensuring ease of navigation and interaction

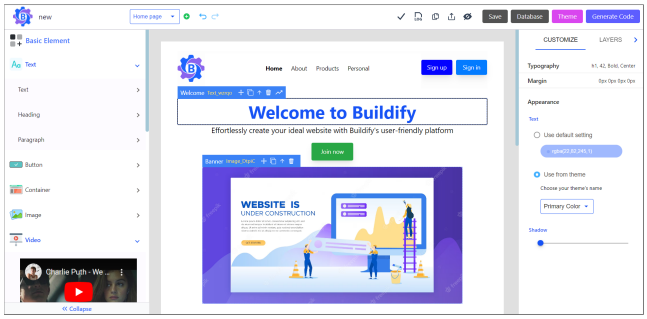

Figure 11. Website Builder

Figure 12 illustrates the source code of the website interface, specifically the list of Components and the details of a Component (Anchor) designed in ReactJS after being downloaded and extracted

The experimental results demonstrate that the source code generation is optimized, taking approximately 2 seconds.

In Figure 13, we can observe the generated JSX Component for each page.

After installing all the necessary packages and running the source code on localhost, you will be able to see the interface of the website built from the source code shown in Figure 14.

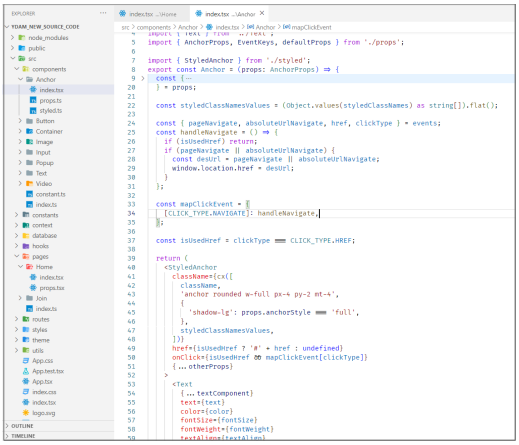

Figure 12. Generated Source Code in ReactJS Framework

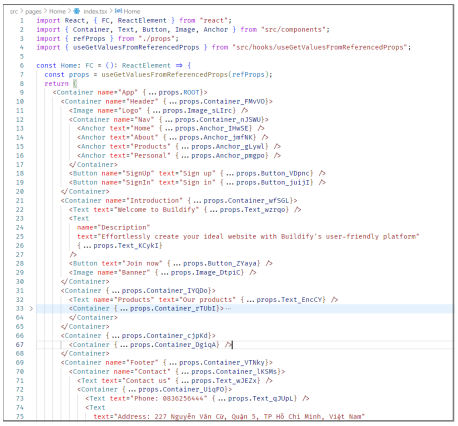

Figure 13. The source code for each page

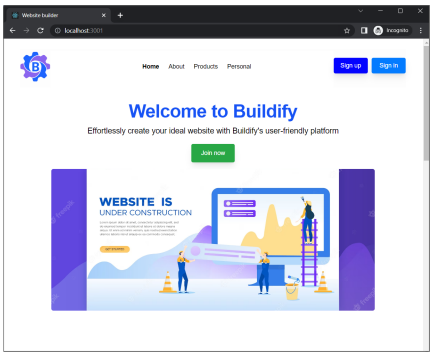

Figure 14. The local runtime interface of the generated source code website.

5 Discussions

Based on the achieved results, several notable contributions and strengths of the research can be observed:

1. We have developed a comprehensive Website Builder product. The feature of generating source code for a

JavaScript framework like ReactJS is relatively new and presents several challenges. The obtained results in section 4 demonstrate that the source code generation feature performs exceptionally well, meeting and even exceeding expectations.
2. The runtime user interface of the exported source code closely resembles the website design created by users in the Website Builder. The internal architecture of the source code's user interface is organized into separate pages, each consisting of a list of components along with their associated props.
3. The data architecture of the system is visually described, and the most important processing flows are represented, such as the drag-n-drop of a selector.
4. The selectors are systematically built, classified, and designed to be diverse.
5. The ReactJS source code generation feature is efficient and accurate compared to the user's design, generating UI component sets in the typical ReactJS code style. The generated source code is clean, concise, and easy to understand, with a standard and highly extensible code base.

Despite achieving significant results, the proposed method still has some limitations:

1. Only source code export for the ReactJS framework is supported, and generating pure CSS files is not yet possible, requiring the use of styled-components.
2. The existing set of pre-designed components is not visually appealing compared to other Website Builders in the market, such as Wix.
3. The exported source code consists of predefined components (selectors) and does not currently support user-defined selectors or the use of standard HTML tags like div, a, button, video, etc.

6 Conclusions and Future Work

6.1 Conclusions

In conclusion, this paper provides valuable insights and guidance for developers aiming to create a website builder that empowers users with full ownership and access to the underlying source code. The approach presented covers various essential aspects, including the implementation of key components such as the Editor, Layer, History, and event handling mechanisms. These components enable important features like drag-n-drop functionality, multi-page support, theming, and dynamic data integration. For ReactJS source generation, the paper proposes a method for consistent User Component management in both the frontend and backend, along with the technique to merge node trees for efficient code organization. Additionally, techniques to optimize ReactJS source code generation are presented, improving the speed and effectiveness of the website builder. By following the recommendations and strategies outlined in this paper, developers can create a website builder that allows users to effortlessly build and customize their websites while retaining control over the generated source code.

6.2 Future Work

To enhance the website builder's functionality and capabilities, the following areas will be explored:

1. Multi-framework support: Expand the source code generation to include popular frameworks like Angular and Vue, allowing users to export projects compatible with their framework of choice.
2. Plugin feature: Develop a Plugin feature with APIs that enable community contributions to the User Component library, fostering collaboration and customization.
3. User-defined components: Enable users to create their own components within the design interface, leveraging existing User Components for customization and maintaining reusability.
4. Database integration: Enhance the website builder's capabilities by allowing users to integrate data from various sources, including external APIs and third-party storage providers.
5. Diversify custom event capabilities: Incorporate additional custom event capabilities for nodes, such as hover and focus, enabling dynamic and interactive web experiences.

References

[1] Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 1–6.
[2] Builder.io. 2023. https://www.builder.io/.
[3] Figma. 2023. Design to Code Plugin. https://www.figma.com/community/tag/designtocode/plugins.
[4] Andrew Lee. 2018. Generating Webpages from Screenshots.
[5] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering* 46, 2 (2018), 196–221.
[6] Prev Wong, Andy Krings-Stern, Mateusz Drulis and others. 2023. A React Framework for building extensible drag and drop page editors. https://craft.js.org, https://github.com/prevwong/craft.js. Accessed: 2023-12-06.
[7] Quarkly.io. 2023. https://quarkly.io/.
[8] Teleporthq. 2023. https://teleporthq.io.
[9] Wix. 2023. https://www.wix.com/.
[10] Wordpress. 2023. https://wordpress.com/.