IJNRD.ORG ISSN : 2456-4184
INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD) | IJNRD.ORG
An International Open Access, Peer-reviewed, Refereed Journal

Drag And Drop Website Builder in Next.JS

Name of the student(s):

Nitish Kumar (2100331530033)

Dheeraj Mathpal (2100331530016)

Amaan Ahmed (2100331530009)

Keshav Kaushik (2100331530020)

Name of the Supervisor:

Mrs. Leena Mam

Department of Artificial Intelligence & Machine Learning

Abstract

This project presents the design and implementation of a drag-and-drop form builder developed using Next.js, a popular React framework. The form builder enables users to create customized forms through an intuitive visual interface without requiring coding knowledge. The system architecture comprises four main components: Navigation, Droppable Elements, Editor View, and Editing Area, working together to provide a seamless form creation experience.

State management is handled through Redux Toolkit, with specific focus on Editor State and History State to facilitate real-time editing and undo-redo functionality respectively. This implementation addresses the growing demand for tools that democratize web development by allowing non-technical users to create interactive forms independently.

The research examines architectural decisions, implementation approaches, and solutions to challenges faced during development, such as performance optimization, cross-browser compatibility, and accessibility concerns. Performance benchmarks indicate that the optimized implementation maintains responsive interactions even with complex forms containing numerous elements.

This project contributes to the field of web development by demonstrating an effective architecture for implementing drag-and-drop interfaces with implications for broader application in website construction, data visualization, and interactive document creation. The findings suggest that the described architecture provides a scalable and user-friendly solution for form building, with potential for future enhancements including collaborative editing and advanced validation systems.

Keywords: Next.js, Form Builder, Drag-and-Drop Interface, Redux, State Management, Web Development, React

CHAPTER 1: INTRODUCTION

1.1 Introduction to Form Builders

Web forms are essential components of interactive websites, enabling data collection, user registration, feedback gathering, and various other interactions between users and web applications. Traditionally, creating forms required knowledge of HTML, CSS, and JavaScript, limiting the ability of non-technical users to design and implement forms according to their specific requirements.

Form builders have emerged as solutions to this problem, offering interfaces that allow users to create forms without writing code. These tools have evolved significantly over time, from basic HTML form generators to sophisticated drag-and-drop interfaces that provide visual representations of the form being created.
The proliferation of web applications has led to an increasing demand for customizable form solutions that empower non-technical users to create and manage forms independently. This demand is particularly evident in contexts where:

1. Organizations need to rapidly deploy forms for data collection
2. Small businesses with limited technical resources need to create interactive websites
3. Marketing teams require forms for lead generation and customer feedback
4. Content creators need to embed forms in their websites or blogs

The significance of form builders extends beyond mere convenience, as they democratize web development and enable a wider range of users to engage in digital creation. By removing the technical barriers associated with form creation, these tools contribute to a more inclusive digital ecosystem where ideas can be implemented without the intermediation of technical specialists.

1.2 Objective & Scope of the Project

1.2.1 Primary Objectives

The primary objectives of this project are to:

1. Design and implement a drag-and-drop form builder using Next.js that enables users without programming knowledge to create customized forms
2. Develop an intuitive interface that provides real-time visual feedback during the form creation process
3. Implement a robust state management system that supports real-time editing and undo-redo functionality
4. Ensure cross-browser compatibility and responsiveness across different device types
5. Address accessibility concerns to ensure that both the form builder and the generated forms are usable by people with disabilities

1.2.2 Scope of the Project

The scope of this project encompasses:

1. **Component Structure**: Development of four main components:
   - Navigation Component for mode switching and form publishing

- Droppable Elements Component for providing a library of form elements
- Editor View Component for rendering the form being created
- Editing Area Component for modifying element properties

2. **State Management**: Implementation of a Redux-based state management system with:
   - Editor State for maintaining the current state of the form
   - History State for enabling undo-redo functionality
3. **Element Library**: Creation of a comprehensive library of form elements including:
   - Headings and text blocks
   - Text inputs and text areas
   - Checkboxes and radio buttons
   - Dropdown menus
   - Image placeholders
   - Buttons
4. **User Interface Features**:
   - Drag-and-drop functionality for adding and arranging elements
   - Real-time property editing with immediate visual feedback
   - Preview mode for testing the form's appearance and functionality
   - Export options for deploying the created form
5. **Technical Implementation**:
   - Use of Next.js for rendering and routing
   - Redux Toolkit for state management
   - React DnD or similar library for drag-and-drop functionality
   - Implementation of accessibility features

1.3 Problem Statement

Despite the proliferation of web applications and the increasing need for user-friendly form solutions, several challenges persist in the domain of form creation:

1. **Technical Barrier**: Traditional form creation requires knowledge of HTML, CSS, and JavaScript, excluding non-technical users from the process.
2. **Limited Flexibility**: Many existing form builders offer limited customization options, constraining users' ability to create forms that match their specific requirements or branding.
3. **Performance Issues**: Complex forms with numerous elements can lead to performance degradation, resulting in poor user experience.
4. **Accessibility Concerns**: Forms created using existing builders may not adhere to accessibility standards, excluding users with disabilities.
5. **State Management Complexity**: Managing the state of forms during creation, including undo-redo functionality, presents significant technical challenges.
6. **Cross-Browser Compatibility**: Ensuring consistent behavior across different browsers and devices remains a challenge, particularly for drag-and-drop interfaces.

This project aims to address these challenges by developing a form builder that combines user-friendliness with technical robustness, providing an intuitive interface while ensuring optimal performance, accessibility, and cross-browser compatibility.

1.4 Project Overview

The drag-and-drop form builder project leverages Next.js, a React framework known for its performance and developer experience, to create a user-friendly interface for form creation. The system follows a component-based architecture with state management handled through Redux Toolkit.

The form builder enables users to:

1. **Select Elements**: Choose from a library of pre-designed form elements, including headings, text inputs, buttons, and more.
2. **Drag and Drop**: Place elements onto a canvas and arrange them in the desired order.
3. **Edit Properties**: Modify element properties such as text content, styles, and validation rules.
4. **Preview Forms**: Switch to a preview mode to test the form's appearance and functionality.
5. **Publish Forms**: Generate a deployable version of the form for integration into websites.

The implementation employs modern web development techniques, including:

1. **React Components**: Modular, reusable components that encapsulate specific functionality.
2. **Redux State Management**: Centralized state management for maintaining form data and supporting undo-redo functionality.
3. **HTML5 Drag and Drop API**: Enhanced with specialized libraries for smooth drag-and-drop interactions.
4. **WYSIWYG Interface**: Real-time visual feedback during form creation.
5. **Accessibility Features**: Implementation of ARIA attributes and keyboard navigation to ensure accessibility.

The project's significance lies in its potential to democratize form creation, enabling non-technical users to create sophisticated forms while adhering to best practices in web development. By combining an intuitive interface with robust technical implementation, the form builder aims to bridge the gap between user-friendliness and technical excellence.

CHAPTER 2: LITERATURE REVIEW

2.1 Evolution of Form Builders

Form builders have evolved significantly from simple HTML generators to sophisticated drag-and-drop interfaces. Patel & Johnson (2018) trace this evolution, noting that early form builders required users to have knowledge of HTML and CSS, limiting their accessibility to technical users. The introduction of WYSIWYG (What You See Is What You Get) editors represented a significant advancement, enabling users to visualize their forms during creation.

According to Chen et al. (2020), modern form builders like Wufoo, JotForm, and Typeform have popularized the drag-and-drop paradigm, making form creation accessible to non-technical users. These platforms offer rich libraries of pre-designed elements and templates, along with extensive customization options. Their research indicates that these tools have contributed significantly to the democratization of web development, allowing small businesses and individual creators to implement interactive forms without requiring dedicated technical resources.

However, existing form builders often present limitations in terms of flexibility, performance, and integration capabilities. Many commercial solutions operate within closed ecosystems, limiting users' ability to customize forms beyond predefined templates or integrate them with specific technologies. This has created a need for more open, flexible form-building solutions that can be tailored to specific requirements while maintaining user-friendliness.

2.2 Drag-and-Drop Interfaces in Web Applications

Drag-and-drop interfaces have become increasingly prevalent in web applications due to their intuitive nature. Research by Nielsen (2019) demonstrates that drag-and-drop interactions can reduce cognitive load and increase user efficiency compared to traditional form inputs or coding interfaces. The direct manipulation paradigm, where users can physically "grab" and "move" elements, aligns closely with natural human interaction patterns, making it particularly suitable for visual design tasks.

Several libraries have emerged to facilitate the implementation of drag-and-drop functionality in web applications. Wang et al. (2021) survey these libraries, finding that React DnD, react-beautiful-dnd, and SortableJS are among the most popular solutions for React-based applications. These libraries provide abstractions that simplify the handling of drag events and state management, enabling developers to create smooth and responsive interfaces.

The implementation of drag-and-drop functionality presents several technical challenges, including:

1. **Cross-browser compatibility**: Different browsers implement the HTML5 Drag and Drop API with varying levels of support and behaviors.
2. **Mobile support**: Touch interfaces require different handling compared to mouse-based interactions.
3. **Accessibility**: Drag-and-drop interfaces must provide keyboard alternatives and proper ARIA attributes to ensure accessibility.

4. **Performance**: Complex drag-and-drop operations with many elements can lead to performance issues, particularly on less powerful devices.

These challenges have prompted the development of specialized libraries and techniques for implementing drag-and-drop functionality, with a focus on performance optimization and cross-device compatibility.

2.3 State Management in Interactive Web Applications

State management is a critical aspect of interactive web applications, particularly those involving complex user interactions. As Li and Zhang (2022) note, applications with rich interactions like drag-and-drop interfaces require sophisticated state management solutions to maintain consistency and enable features like undo-redo.

Redux has emerged as a popular solution for managing application state in React-based applications, offering a predictable state container that facilitates features like time-travel debugging and undo-redo functionality (Abramov, 2018). Redux follows a unidirectional data flow pattern, where state changes are handled through actions and reducers, providing a clear and predictable pattern for state mutations.
Li and Zhang's research compares different state management approaches in React applications, finding that Redux provides superior performance and maintainability for applications with complex state requirements. The study also highlights the benefits of using Redux Toolkit, which simplifies common Redux patterns and reduces boilerplate code.

For form builders, state management must address several specific requirements:

1. **Real-time updates**: Changes to element properties must be reflected immediately in the interface.
2. **History tracking**: The system must maintain a history of state changes to enable undo-redo functionality.
3. **Performance**: State updates must be optimized to prevent performance degradation, especially for complex forms.
4. **Persistence**: The state must be serializable to enable saving and loading forms.

The combination of Redux with specialized middleware for handling these requirements has proven effective in implementing state management for interactive web applications like form builders.

2.4 Next.js for Web Application Development

Next.js has gained significant traction as a framework for building React applications, offering features like server-side rendering, static site generation, and API routes (Vercel, 2023). Studies by Kumar et al. (2022) demonstrate that Next.js applications typically outperform client-side rendered React applications in terms of initial load time and search engine optimization.

The framework's hybrid rendering approach, which combines server-side rendering with client-side hydration, provides an optimal balance between performance and interactivity. This is particularly valuable for applications like form builders, which require both immediate responsiveness and good initial load performance.
According to Mehta (2023), several commercial website builders, including Framer and Editor.js, have adopted Next.js as their underlying technology due to its performance characteristics and developer experience. The framework's file-based routing system and built-in API routes simplify the implementation of complex applications, while its support for static site generation enables excellent performance for published forms.

For form builders, Next.js offers several specific advantages:

1. **Server-side rendering**: Improves initial load time and SEO for published forms.
2. **API routes**: Simplifies the implementation of form submission endpoints.
3. **File-based routing**: Facilitates the organization of different views (builder, preview, etc.).
4. **Image optimization**: Enhances performance for forms containing images.
5. **Development experience**: Hot module replacement and other developer tools speed up the development process.

These advantages make Next.js a compelling choice for implementing a form builder, particularly one that aims to offer both excellent user experience and technical robustness.

CHAPTER 3: DESIGN OF PROJECT MODEL

3.1 System Architecture

The form builder's architecture follows a component-based design pattern, with clear separation of concerns between different parts of the application. The overall system architecture consists of four primary components that work together to provide a seamless form creation experience:

1. **Navigation Component**: Controls the overall application mode and provides access to form-level operations.
2. **Droppable Elements Component**: Serves as a library of available form elements.
3. **Editor View Component**: Functions as the canvas where users construct their forms.
4. **Editing Area Component**: Provides the interface for modifying element properties.

These components are orchestrated through a centralized state management system based on Redux Toolkit, which maintains two main state slices:

1. **Editor State**: Represents the current state of the form being edited.
2. **History State**: Maintains a history of previous states to enable undo-redo functionality.

The architecture follows a unidirectional data flow pattern, where user interactions trigger actions that update the state, which in turn causes the UI to re-render. This pattern ensures predictability and helps maintain consistency between the application state and the user interface.

The system also incorporates several cross-cutting concerns:

1. **Accessibility**: Implementation of ARIA attributes and keyboard navigation.
2. **Performance Optimization**: Strategies for maintaining responsiveness with complex forms.
3. **Error Handling**: Mechanisms for detecting and recovering from errors.
4. **Persistence**: Functionality for saving and loading forms.

Figure 1 provides a visual representation of the system architecture, illustrating the relationships between components and the flow of data through the application.

3.2 Component Structure

The form builder's component structure follows a hierarchical organization, with each component encapsulating specific functionality. This section details the four primary components and their interactions.

3.2.1 Navigation Component

The Navigation component serves as the control center for the form builder, providing functionality to:

1. Switch between editing mode and live mode, allowing users to preview their form as it would appear to end-users
2. Publish the completed form, generating a URL that can be shared with users
3. Access form settings and configuration options

The component maintains a clean, minimal interface to avoid distracting users from the form-building process. It communicates with the global state management system to trigger mode changes and publishing actions.

Implementation details include:

```
// Navigation component structure
const Navigation = () => {
  const dispatch = useDispatch();
  const isPreviewMode = useSelector(selectIsPreviewMode);

  const togglePreviewMode = () => {
    dispatch(setPreviewMode(!isPreviewMode));
  };

  const publishForm = () => {
    dispatch(publishFormAction());
  };

  return (
    <nav className="navigation">
      <div className="logo">Form Builder</div>
      <div className="actions">
        <button onClick={togglePreviewMode}>
          {isPreviewMode ? "Edit Mode" : "Preview Mode"}
        </button>
        <button onClick={publishForm}>Publish</button>
        <button>Settings</button>
      </div>
    </nav>
  );
};
```

3.2.2 Droppable Elements Component

The Droppable Elements component functions as a repository of available form elements that users can add to their forms. It includes:

1. A categorized library of elements (headings, text inputs, text areas, images, links, buttons)
2. Visual representations of each element type
3. Drag handles that enable users to drag elements onto the Editor View

Each element in the library is associated with a predefined set of properties and default styles, which are applied when the element is dropped onto the canvas. The component implements drag event listeners using React DnD or a similar library to initiate the drag operation.

The component structure includes:

```
// Droppable Elements component structure
const DroppableElements = () => {
  const elementTypes = [
    { id: "heading", label: "Heading", icon: "heading-icon" },
    { id: "text-input", label: "Text Input", icon: "input-icon" },
    { id: "text-area", label: "Text Area", icon: "textarea-icon" },
    { id: "button", label: "Button", icon: "button-icon" },
    { id: "image", label: "Image", icon: "image-icon" },
    { id: "link", label: "Link", icon: "link-icon" }
  ];

  return (
    <div className="droppable-elements">
      <h2>Elements</h2>
      <div className="elements-list">
        {elementTypes.map(element => (
          <DraggableElement
            key={element.id}
            type={element.id}
            label={element.label}
            icon={element.icon}
          />
        ))}
      </div>
    </div>
  );
};
```

3.2.3 Editor View Component

The Editor View component serves as the canvas where users construct their forms. It:

1. Renders all form elements in a WYSIWYG format
2. Provides visual feedback during drag operations (e.g., highlighting valid drop zones)
3. Allows users to select elements for editing
4. Displays visual aids such as alignment guides and element boundaries

The component subscribes to the Editor State in the Redux store, ensuring that it reflects the current state of the form. When elements are selected, the component updates the Selected Element property in the state, triggering the Editing Area to display the relevant properties.

Implementation details include:

```
// Editor View component structure
const EditorView = () => {
  const dispatch = useDispatch();
  const elements = useSelector(selectElements);
  const selectedElementId = useSelector(selectSelectedElementId);
  const isPreviewMode = useSelector(selectIsPreviewMode);

  const handleDrop = (item, index) => {
    dispatch(addElement({ type: item.type, index }));
  };

  const selectElement = (id) => {
    if (!isPreviewMode) {
      dispatch(setSelectedElement(id));
    }
  };

  return (
    <div className="editor-view">
      {elements.map((element, index) => (
        <div key={element.id}>
          <DropZone onDrop={(item) => handleDrop(item, index)} />
          <ElementRenderer
            element={element}
            isSelected={element.id === selectedElementId}
            onClick={() => selectElement(element.id)}
            isPreviewMode={isPreviewMode}
          />
        </div>
      ))}
      <DropZone onDrop={(item) => handleDrop(item, elements.length)} />
    </div>
  );
};
```

3.2.4 Editing Area Component

The Editing Area component provides an interface for modifying the properties of the selected element. It:

1. Displays all editable properties of the selected element
2. Provides appropriate input controls for each property type (e.g., color pickers, text inputs, dropdowns)
3. Updates the Editor State in real-time as properties are modified
4. Groups related properties into logical sections for improved usability

The component receives the ID of the selected element from the Editor State and retrieves the corresponding element's properties. As users modify these properties, the component

dispatches actions to update the state, causing the Editor View to re-render with the updated styles and content.

Implementation details include:

```
// Editing Area component structure
const EditingArea = () => {
  const dispatch = useDispatch();
  const selectedElementId = useSelector(selectSelectedElementId);
  const selectedElement = useSelector(selectSelectedElement);

  if (!selectedElementId || !selectedElement) {
    return <div className="editing-area empty">Select an element to edit</div>;
  }

  const updateProperty = (property, value) => {
    dispatch(updateElementProperty({
      id: selectedElementId,
      property,
      value
    }));
  };

  return (
    <div className="editing-area">
      <h2>Edit {selectedElement.type}</h2>

      <div className="property-group">
        <h3>Content</h3>
        {selectedElement.hasContent && (
          <div className="property">
            <label>Text</label>
            <input
              type="text"
              value={selectedElement.content}
              onChange={(e) => updateProperty("content", e.target.value)}
            />
          </div>
        )}
      </div>

      <div className="property-group">
        <h3>Appearance</h3>
        <div className="property">
          <label>Width</label>
          <select
            value={selectedElement.styles.width}
            onChange={(e) => updateProperty("styles.width", e.target.value)}
          >
            <option value="100%">Full Width</option>
```

```
            <option value="75%">Three Quarters</option>
            <option value="50%">Half Width</option>
            <option value="25%">Quarter Width</option>
          </select>
        </div>
        {/* Additional style properties would be rendered here */}
      </div>
    </div>
  );
};
```

3.3 State Management Architecture

The state management architecture is built on Redux Toolkit, which provides a standardized approach to defining and updating state. The state is structured into two main components: Editor State and History State.

3.3.1 Editor State

The Editor State maintains the current state of the form being edited, including:

1. A unique identifier for the form
2. An array of elements, each with its own ID, type, styles, and content
3. The ID of the currently selected element
4. A Boolean flag indicating whether preview mode is active

This state structure enables real-time editing by providing a single source of truth for the form's composition and appearance. When users modify an element's properties, the Editor State is updated, triggering re-renders of the affected components.

The implementation uses Redux Toolkit's *createSlice* function to define the state and its reducers:

```
// Editor State slice
const editorSlice = createSlice({
  name: 'editor',
  initialState: {
    formId: uuidv4(),
    elements: [],
    selectedElementId: null,
    isPreviewMode: false
  },
  reducers: {
    addElement: (state, action) => {
      const { type, index } = action.payload;
      const newElement = createDefaultElement(type);
      state.elements.splice(index, 0, newElement);
      state.selectedElementId = newElement.id;
```

```
  },
  removeElement: (state, action) => {
    state.elements = state.elements.filter(el => el.id !== action.payload);
    state.selectedElementId = null;
  },
  updateElementProperty: (state, action) => {
    const { id, property, value } = action.payload;
    const element = state.elements.find(el => el.id === id);
```

3.3.2 History State

The History State facilitates undo-redo functionality by maintaining a history of the form's state at different points in time. It consists of:

1. An array of historical Editor States
2. An index pointer indicating the current position in the history

When users perform actions that modify the form, a copy of the updated Editor State is appended to the history array, and the index is incremented. Undo operations involve decrementing the index and restoring the corresponding historical state, while redo operations increment the index.

History State slice code snippet

```
const historySlice = createSlice({
  name: 'history',
  initialState: {
    past: [],
    future: [],
    saving: false
  },
  reducers: {
    addToHistory: (state, action) = >{
      state.past.push(action.payload);
      state.future = [];
    },
    undo: (state) = >{
      if (state.past.length > 0) {
        const previous = state.past.pop();
        state.future.unshift(previous);
      }
    },
    redo: (state) = >{
      if (state.future.length > 0) {
        const next = state.future.shift();
        state.past.push(next);
      }
    },
```

```
clearHistory: (state) = >{
  state.past = [];
  state.future = [];
}
```

References

[1] D. Abramov, *Redux: A predictable state container for JavaScript apps*. GitHub Repository, 2018.

[2] L. Chen, Q. Zhang, and H. Wang, "A comparative analysis of no-code development platforms," *Journal of Web Engineering*, vol. 19, no. 4, pp. 363–396, 2020.

[3] A. Kumar, V. Singh, and R. Patel, "Performance comparison of server-side rendering frameworks for React applications," in *Proc. IEEE Int. Conf. Web Services*, 2022, pp. 112–125.

[4] J. Li and T. Zhang, "State management patterns in React applications: A systematic review," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 1–34, 2022.

[5] S. Mehta, "The adoption of Next.js in commercial website builders: Case studies and analysis," *Int. J. Web Develop.*, vol. 7, no. 2, pp. 89–104, 2023.

[6] J. Nielsen, "Usability of drag-and-drop interfaces: A longitudinal study," *User Experience Magazine*, vol. 14, no. 2, pp. 45–52, 2019.

[7] R. Patel and M. Johnson, "The evolution of web-based form builders: From HTML to visual interfaces," *Int. J. Hum.-Comput. Interact.*, vol. 34, no. 5, pp. 423–442, 2018.

[8] Vercel, *Next.js Documentation*. Available: https://nextjs.org/docs (2023).

[9] R. Wang, Y. Li, and G. Zhang, "A survey of drag-and-drop libraries for React applications," *ACM Comput. Surv.*, vol. 54, no. 3, pp. 1–28, 2021.

[10] K. Zhang and H. Chen, "Accessible form design: Guidelines and implementation strategies," *Web Accessibility Journal*, vol. 5, no. 1, pp. 78–95, 2023.