Page Builder with live preview

Трајче Проданов 221164
Раде Перовановиќ 221018

Abstract

This report documents the design, architecture, and implementation of the Page Builder project. It explains how the system evolved from a basic Vite + React + TypeScript scaffold into a schema-driven visual editor capable of composing pages out of reusable elements (text, button, image, container, input, icon), persisting projects, and exporting a clean, production-ready HTML snapshot. The report covers the core domain model, state management, rendering pipeline, editing UX, persistence strategy, export pathway, and a probable implementation chronology. It also includes representative code excerpts and references to images to aid understanding.

Overview and goals

The Page Builder enables users to construct web page layouts without hand-writing HTML and CSS. Users select elements from a palette, place and arrange them on a canvas, and configure properties via a dynamic, schema-driven properties editor. Key goals:

- Build pages by composing typed, reusable components.
- Provide an ergonomic editor experience: select, rename, reorder, and edit properties with immediate visual feedback.
- Persist work to the browser for a projects workflow.
- Export clean HTML that excludes editor-only affordances.

Technology stack

- React 19 + TypeScript
- Vite for dev build tooling
- React Router for page routing
- Lucide for icons
- Tailwind (utility classes and design tokens), with custom CSS for editor polish
- Sonner for toast notifications

High-level architecture

The architecture splits concerns across three layers:

- Types and configuration: strongly-typed definitions for components, instances, and editable properties, plus schemas that describe each element's configurable fields and how they map to CSS/attributes/content.
- State and actions: a central `useBuilder` hook that owns the current project (builder) and provides mutation APIs, plus supporting hooks for saving/loading and exporting/importing.
- Rendering and editor UI: a canvas renderer that maps high-level component data to real React components via a `componentMap`, alongside an editor shell with sidebars and a properties panel.

Key Modules

- `src/types/builder.ts` — foundational types for elements, instances, and schemas.
- `src/builder-elements/*` — primitive, visual components and a `componentMap` mapping domain types to React components.
- `src/renderer/CanvasRenderer.tsx` — editor-mode renderer with selection and placement hooks.
- `src/renderer/ExportRenderer.tsx` — export-only renderer that strips editor affordances.
- `src/hooks/useBuilder.ts` — the central state manager for add/update/remove/move, style updates, time travel (undo/redo), and advanced behaviors.
- `src/components/*` — editor UI: `EditorLayout`, `ComponentsList`, `PropertiesEditor`, and the palette/menubar.
- `src/config/elementSchemas.ts` — properties schemas defining editable fields and how they apply to the rendered element.
- `src/data/documentRepository.ts` + `src/hooks/useDocumentActions.tsx` — persistence and export/import actions.

Domain Model and Schemas

The domain model distinguishes between an element's type (what it is) and an instance's props (how it is configured on the canvas). The model evolved to support schema-driven property editing, enabling generic property panels that can render controls for any element type.

- **ComponentType** — the element's category, e.g., `text`, `button`, `image`, `container`, `input`, `icon`.

- **Component** — an instantiated element on the canvas with `id`, `type`, `props`, and optional `children`.
- **Builder** — the full document: `id`, `name`, `components` array, and `styles` for the canvas.
- **PropertyDefinition** and **ComponentSchema** — declarative metadata telling the editor how to render property controls and how to apply values to the rendered component as styles, attributes, or content.

Example: a text element schema defines a `content` property (applied as inner text) and typography fields applied as inline styles.

```12:75:src/config/elementSchemas.ts
export const textElementSchema: ComponentSchema<typeof textElementProperties> =
{
  type: "text",
  label: "Paragraph",
  properties: textElementProperties,
  defaultProps: buildDefaultProps(textElementProperties),
  getContentProp: (props) => "content",
  getStyleProps: (props) => {
    const styles: React.CSSProperties = {};
    for (const key in props) {
      const def = textElementProperties[key as keyof typeof textElementProperties];
      if (def?.applyAs === "style" && def.styleKey && props[key as keyof typeof props] !==
undefined) {
        styles[def.styleKey] = props[key as keyof typeof props] as any;
      }
    }
    return styles;
  },
};
```

This design allows the `PropertiesEditor` to be generic: it reads the schema for the selected component type and renders the appropriate controls (text, number, color, select, button-groups, and box shorthand) without hard-coding per-element UI.

State Management and Editing Actions

The core state hook, `useBuilder`, exposes a set of actions for manipulating the page and its components. It also tracks undo/redo via an internal history stack.

- `setBuilder` — initialize/reset the active builder document.
- `addComponent` — add an element either at root or as a child of a container.
- `removeComponent` — delete an element by id, recursively.
- `updateComponent` — update an element's props or partial fields.
- `moveComponent` — reorder among siblings.
- `renameComponent` — change the human-readable label.
- `updateChildPlacement` — adjust child placement hints (e.g., order, alignSelf) for flex layouts.
- `setStyles` — update global canvas styles.
- `undo` / `redo` — time travel through previous states.

Representative excerpt:

```1:120:src/hooks/useBuilder.ts
const updateRecursive = (
  components: Component[],
  targetId: string,
  updates: Partial<Component>,
): Component[] => {
  return components.map((comp) => {
    if (comp.id === targetId) {
      if (updates.props && !updates.type && !updates.children) {
        const { ...restProps } = updates.props;
        return { ...comp, props: { ...comp.props, ...restProps } };
      }
      return { ...comp, ...updates };
    }
    if (comp.children && comp.children.length > 0) {
      return { ...comp, children: updateRecursive(comp.children, targetId, updates) };
    }
    return comp;
  });
};
```

The hook also contains specialized behaviors that emerged from UX needs, like promoting an `image` to a container background or restoring it back to a child. These features are implemented as targeted transformations of the tree with careful checks to maintain referential integrity.

Undo/redo is maintained by a stack of previous `Builder` snapshots and a moving index; the `isTimeTraveling` guard suppresses history recording while traversing.

Rendering Pipeline

Two renderers share the same `componentMap` but differ in intent:

- `CanvasRenderer` — editor-mode renderer that passes selection handlers, placement callbacks, and editor-only flags down to components. It shows affordances like empty-canvas hints.
- `ExportRenderer` — a simplified renderer that passes only props and children. This is used to produce clean HTML snapshots without editor wiring or event handlers.

Editor-mode view:

```1:31:src/renderer/CanvasRenderer.tsx
<ComponentToRender
  key={componentConfig.id}
  id={componentConfig.id}
  props={componentConfig.props}
  children={componentConfig.children}
  onSelect={onSelectComponent}
  isSelected={selectedComponentId === componentConfig.id}
  selectedComponentId={selectedComponentId}
  onAddComponentRequest={onAddComponentRequestToContainer}
  updateChildPlacement={updateChildPlacement}
  isEditorMode={isEditorMode}
/>
```

Export view:

```1:14:src/renderer/ExportRenderer.tsx
return (
```

```
  <ComponentToRender
    id={component.id}
    props={component.props}
    children={component.children}
    isEditorMode={false}
  />
);
```

Editor UX

The editor UI is structured with `EditorLayout`, placing three main areas:

- Menubar/topbar — palette and save status
- Sidebar — hierarchical components list
- Canvas — the page under construction
- Properties — dynamic property editor and page settings

The components list supports selection, in-place rename, reorder (up/down), and collapse/expand for nested containers. It also supports keyboard navigation (Arrow keys) and quick-rename with Enter.

```170:237:src/components/ComponentsList.tsx
window.addEventListener('keydown', handleGlobalKeyDown);
// ArrowUp/Down navigate, ArrowRight expands or enters first child, ArrowLeft collapses or goes to parent
```

The properties editor supports two modes:

- Component mode — shows schema-driven groups/controls for the selected element, with debounced updates.
- Page mode — when no component is selected, shows page name and global style controls.

It includes specialized controls like a compact BoxControl for padding/margin shorthands with three modes (overall, axis, individual), easing common spacing edits.

Persistence and Projects

Persistence leverages `localStorage` in two ways:

- A projects index that lists all created documents with metadata (id, name, last modified) to support a Projects list view.
- Document storage under per-document keys for loading and deleting individual projects.

```1:22:src/data/documentRepository.ts
export const saveDocument = (document: Builder): void => {
  const documentKey = `builder_doc_${document.id}`;
  localStorage.setItem(documentKey, JSON.stringify(document));
  // update index with name and lastModified
};
```

`useDocumentActions` adds user-facing commands: save/load the active builder, export/import JSON, and export a print-ready HTML snapshot. The HTML export uses `ExportRenderer` to render a clean DOM into a temporary, hidden container, then serializes it into a downloadable HTML file.

```69:118:src/hooks/useDocumentActions.tsx
const exportAsHtml = useCallback(() => {
  if (!builder) return;
  const tempContainer = document.createElement('div');
  document.body.appendChild(tempContainer);
  const root = createRoot(tempContainer);
  root.render(
    <ExportRenderer components={builder.components} globalStyles={builder.styles} />
  );
  setTimeout(() => {
    const htmlContent = tempContainer.innerHTML;
    root.unmount();
    document.body.removeChild(tempContainer);
    const fullHtml = `<!DOCTYPE html>...${'${htmlContent}'}`;
    // trigger download
  }, 100);
}, [builder]);
```

Component Map and Element Library

The `componentMap` is the bridge between abstract component instances and real React components. Each entry renders itself using the provided `props` and optional `children`.

```1:26:src/builder-elements/componentMap.ts
export const componentMap: ComponentMapType = {
  button: ButtonElement,
  text: TextElement,
  image: ImageElement,
  container: ContainerElement,
  input: InputElement,
  icon: IconElement,
};
```

This encourages adding new elements incrementally without changing core editor logic. A new element requires:

- Implement the visual component.
- Add a schema to `elementSchemas` specifying editable properties and defaults.
- Register the component in `componentMap`.
- Add palette metadata so it appears in the topbar/menubar.

Styling Strategy

The editor chrome uses a mix of Tailwind utilities and custom inline styles for fine-grained control over shadow, radius, borders, and gradient backgrounds. The canvas itself accepts `globalStyles` from the Builder document, so end-users can control page-level layout and appearance (e.g., padding, min-height, background color) via the Page properties pane.

Clean Export Pathway

The export process renders components outside the editor context, ensuring the output is not contaminated by selection handlers, drag/drop affordances, or editor-only CSS

classes. This immutable representation is ideal for publishing or handing off to a downstream system.

Keyboard-centric Navigation

The hierarchical components list supports efficient navigation. With arrow keys, users can traverse siblings and container boundaries; Enter focuses rename. This significantly speeds up micro-adjustments and naming conventions during complex compositions.

Build, Run and Usage

- Install dependencies:

npm install

- Run the dev server:

npm run dev

- Open the app at the URL printed by Vite, usually `http://localhost:5173`.

- Create a new project (or load an existing one), add elements from the palette, select them in the sidebar, and adjust properties in the right panel. Use the topbar save/export actions (or keyboard shortcuts, if implemented in your version) to persist or export your work.

Extensibility

- New elements: define a component, add a schema entry, register in `componentMap`, and add palette metadata.
- New property editors: extend the control renderer with a new control type and plug it into the schema.
- Alternate persistence: swap `documentRepository` for an API-backed repository without touching editor internals.
- Theming: centralize editor chrome styles and expose a theme provider.

Known Limitations and Future Work

- Drag-and-drop placement is not covered here; containers accept children by request handlers rather than direct DnD UX. Integrating a DnD library (e.g., Dnd Kit) would be a natural follow-up.
- Constraints and responsive rules could be elevated into the schema layer to ensure consistent spacing and layout at different breakpoints.
- Asset management for images could move from URLs to a managed media library with uploads and transformations.
- Collaborative editing would require a shared backend, conflict resolution, and presence indicators.

Conclusion

The Page Builder delivers a robust foundation for a schema-driven, component-based page editing experience in the browser. The separation of concerns across types/schemas, state/actions, rendering, and editor UI enables rapid iteration and extensibility. The clean export pathway and local persistence make it practical both for quick prototyping and for handing off static snapshots. With further investment in DnD, responsive design tooling, and collaborative persistence, this architecture can scale into a professional-grade site builder.

Short Code Excerpts

Types excerpt:

```1:59:src/types/builder.ts
export type PropertyEditorType =
  | "text"
  | "textarea"
  | "number"
  | "boolean"
  | "select"
  | "color"
  | "slider"
  | "fontFamily"
  | "imageUpload";
```

Canvas empty state styling:

```51:71:src/renderer/CanvasRenderer.tsx
<div className="canvas" style={{ ...globalStyles, borderRadius: 12 }} ref={ref} onClick={unselectComponent}>
  {components.length === 0 && (
     <div style={{ textAlign: 'center', color: '#6b7280', padding: '60px 24px', border: '1px dashed rgba(0,0,0,0.08)', margin: '16px', borderRadius: 12, background: '#f9fafb', boxShadow: 'inset 0 1px 0 rgba(255,255,255,0.6), 0 4px 10px rgba(0,0,0,0.05)' }}>
      The canvas is empty. Add components from the palette.
    </div>
  )}
  {components.map(renderComponent)}
</div>
```

Properties editor debounced updates:

```174:183:src/components/PropertiesEditor.tsx
useEffect(() => {
  if (JSON.stringify(component.props) === JSON.stringify(currentProps)) return;
  const handler = setTimeout(() => {
    onUpdateComponent(component.id, { props: currentProps });
  }, 500);
  return () => clearTimeout(handler);
}, [component.id, component.props, currentProps, onUpdateComponent]);
```

Reordering siblings:

```370:399:src/hooks/useBuilder.ts
const newSiblings = [...siblings];
[newSiblings[index], newSiblings[newIndex]] = [
  newSiblings[newIndex],
  newSiblings[index],
];
// update parent.children or root components accordingly
```

To export this report to PDF: open `docs/report.html` in your browser and use Print to PDF. Optionally, import the Markdown into a word processor (docx/odt) and export to PDF.

Creation Process: From Scaffold to Schema-Driven Builder

This section narrates the end-to-end creation process chronologically, explaining the purpose of each key piece, why it's implemented this way, and how it fits the overall architecture. Code snippets are taken directly from the repository to ground the explanations.

1) Scaffold and Routing

We began by scaffolding a React + TypeScript project with Vite, then wiring React Router to support a home screen, projects list, per-project editor, and preview routes. The router shell is intentionally minimal, delegating real UI to pages and the editor layout.

```1:20:src/App.tsx
import { Route, Routes } from "react-router-dom";
import Home from "./pages/Home";
import ProjectsList from "./pages/ProjectsList";
import Project from "./pages/Project";
import Preview from "./pages/Preview";

function App() {
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/projects" element={<ProjectsList />} />
      <Route path="/project/:projectId" element={<Project />} />
      <Route path="/preview/:projectId" element={<Preview />} />
      {/* Optional: Add a 404 Not Found Route */}
      <Route path="*" element={<div>Not Found</div>} />
    </Routes>
  );
}
```

export default App;
```

The entrypoint mounts router and toasts early to enable navigation and global notifications across the editor experience.

```1:15:src/main.tsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.tsx'
import { BrowserRouter } from 'react-router-dom'
import { Toaster } from 'sonner'

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <BrowserRouter>
      <Toaster />
      <App />
    </BrowserRouter>
  </StrictMode>,
)
```

Why this way: Using route-based separation from day one allows the editor to remain a focused page while preview and list views evolve independently. Adding `Toaster` globally avoids repetitive plumbing.

2) Domain Model: Components, Builder, and Schemas

We modeled a page as a `Builder` document that owns an array of component instances and canvas-level `styles`. Components are typed by `ComponentType` and carry `props` and optional `children` for nesting (e.g., containers).

This file also includes a forward-looking schema layer, foreshadowing the later pivot to schema-driven property editing.

```19:58:src/types/builder.ts
export interface Component {
  id: string;

```
  name?: string;
  type: "button" | "input" | "container" | "text" | "image" | "icon" | "input";
  props: {
    [key: string]: string | number | boolean | undefined | null;
    position?: string;
    label?: string;
    src?: string;
    altText?: string;
    width?: string;
    height?: string;
    content?: string;
    fontSize?: string;
    color?: string;
    backgroundColor?: string;
    padding?: string;
    placeholder?: string;
    name?: string;
    size?: number;
    justifyContent?: JustifyContentType;
    flexDirection?: "row" | "column";
    kind?: "text" | "checkbox" | "radio";
  };
  placement?: {
    order?: number;
    alignSelf?: AlignSelfType;
  };
  children?: Component[];
}
```

Why this way: A tree of `Component` instances closely mirrors how actual DOM trees behave, which makes rendering and manipulation intuitive. Optional `placement` anticipates future layout constraints without over-committing the type early.

The schema section introduces `PropertyDefinition` and `ComponentSchema`, which define how editors should render controls and how props map to CSS/attributes/content. This is pivotal for generic property editors.

```118:138:src/types/builder.ts
export interface ComponentSchema<
```

```
  PropsDef extends Record<string, PropertyDefinition> = Record<
    string,
    PropertyDefinition
  >,
> {
  type: ComponentType;
  label: string;
  icon?: string;
  properties: PropsDef;
  defaultProps: Readonly<ComponentProps<PropsDef>>;
  getStyleProps?: (props: ComponentProps<PropsDef>) => React.CSSProperties;
  getAttributeProps?: (
    props: ComponentProps<PropsDef>,
  ) => Record<string, string>;
  getClassNameProps?: (props: ComponentProps<PropsDef>) => string;
  getContentProp?: (props: ComponentProps<PropsDef>) => keyof PropsDef | null;
}
```

Why this way: Decoupling the editor UI from element implementations makes the system scalable—adding a new element often becomes “write a schema + render function,” not “rewrite editor logic.”

3) Element Library and Component Map

We created primitive building blocks—`container`, `text`, `image`, `button`, `input`, `icon`—and a registry mapping each `ComponentType` to its respective React component. This enables a single, generic renderer to instantiate correct components by type.

```18:25:src/builder-elements/componentMap.ts
export const componentMap: ComponentMapType = {
  button: ButtonElement,
  text: TextElement,
  image: ImageElement,
  container: ContainerElement,
  input: InputElement,
  icon: IconElement,
};
```

Why this way: A registry isolates the coupling between domain types and UI implementations. It simplifies both the Canvas and Export renderers and encourages extension via new components.

4) Canvas Rendering (Editor Mode)

The canvas reads the `components` array and renders each instance via `componentMap`. In editor mode, it passes selection handlers and placement updates and renders helpful hints when the canvas is empty.

```17:49:src/renderer/CanvasRenderer.tsx
const CanvasRenderer = ({
  ref,
  components,
  globalStyles,
  onSelectComponent,
  selectedComponentId,
  onAddComponentRequestToContainer,
  updateChildPlacement,
  unselectComponent,
  isEditorMode = true,
}: CanvasRendererProps) => {
  const renderComponent = (componentConfig: Component) => {
    const ComponentToRender = componentMap[componentConfig.type];

    if (!ComponentToRender) {
      return <div key={componentConfig.id}>Unknown component type: {componentConfig.type}</div>;
    }

    return (
      <ComponentToRender
        key={componentConfig.id}
        id={componentConfig.id}
        props={componentConfig.props}
        children={componentConfig.children}
        onSelect={onSelectComponent}
        isSelected={selectedComponentId === componentConfig.id}
        selectedComponentId={selectedComponentId}
```

```
            onAddComponentRequest={onAddComponentRequestToContainer}
            updateChildPlacement={updateChildPlacement}
            isEditorMode={isEditorMode}
        />
    );
};
```

Why this way: The renderer remains agnostic of specific element internals while still enabling editor-only affordances by forwarding standardized props (`onSelect`, `isSelected`, etc.). The “unknown component” fallback protects against registry drift.

5) Export Rendering (Clean Output)

For exports, we reused the same registry but omitted editor-only props to generate clean, static HTML.

```
```9:23:src/renderer/ExportRenderer.tsx
const ExportedComponent: React.FC<ExportComponentProps> = ({ component }) => {
    const ComponentToRender = componentMap[component.type];
    if (!ComponentToRender) return null;

    return (
        <ComponentToRender
            id={component.id}
            props={component.props}
            children={component.children}
            isEditorMode={false}
        />
    );
};
```

Why this way: Keeping the export path close to the normal render path reduces divergence and bugs. A boolean `isEditorMode` flag lets element implementations strip editors’ UI.
```

6) Centralized State: useBuilder

We consolidated all editing actions into a single hook that owns the `Builder` document and its history for undo/redo.

- Adds/removes/updates components recursively.
- Moves siblings up/down.
- Renames components.
- Updates global canvas styles.
- Manages time-travel history with `stack` and `index`.

Recursive update pattern:

```223:255:src/hooks/useBuilder.ts
const updateRecursive = (
  components: Component[],
  targetId: string,
  updates: Partial<Component>,
): Component[] => {
  return components.map((comp) => {
    if (comp.id === targetId) {
      if (updates.props && !updates.type && !updates.children) {
        const { ...restProps } = updates.props;
        return { ...comp, props: { ...comp.props, ...restProps } };
      }
      return { ...comp, ...updates };
    }
    if (comp.children && comp.children.length > 0) {
      return {
        ...comp,
        children: updateRecursive(comp.children, targetId, updates),
      };
    }
    return comp;
  });
};
```

Why this way: A pure recursive map keeps tree transformations immutable and predictable, crucial for React state and time-travel debugging.

Undo/redo with a guarded history index:

```306:333:src/hooks/useBuilder.ts
const canUndo = history.index > 0;
const canRedo = history.index < history.stack.length - 1;

const undo = useCallback(() => {
  if (canUndo) {
    isTimeTraveling.current = true;
    const newIndex = history.index - 1;
    setHistory((prev) => ({ ...prev, index: newIndex }));
    setBuilderInternal(history.stack[newIndex]);
    setTimeout(() => { isTimeTraveling.current = false; }, 50);
  }
}, [canUndo, history]);
```

Why this way: Guarding time travel prevents pollution of the history while navigating it, which would otherwise lead to confusing states and broken redo chains.

Special behaviors (e.g., image-as-background): these reflect UX patterns where an `image` inside a `container` can be promoted to a `backgroundImage` style, or restored as a child later. Implemented as targeted tree rewrites inside `updateComponent` to keep behavior localized.

7) Schema-Driven Properties Editor

A key pivot was adopting an abstract schema to describe per-element property groups and controls—text, number, color, select, box shorthands, and button-groups. This schema powers a generic properties panel.

```36:71:src/utils/editor-schema.tsx
export const componentEditorSchema: Record<ComponentType, GroupDefinition[]> = {
    container: [
        {
            title: 'Layout',
            controls: [
                { prop: 'display', label: 'Display', control: 'select', options: [{ value: 'flex', label: 'Flex' }, { value: 'grid', label: 'Grid' }, { value: 'block', label: 'Block' }] },
```

```
            { prop: 'flexDirection', label: 'Direction', control: 'select', options: [{ value: 'row', label: 'Row' }, { value: 'column', label: 'Column' }] },
                { prop: 'justifyContent', label: 'Justify', control: 'select', options: [{ value: 'flex-start', label: 'Start' }, { value: 'center', label: 'Center' }, { value: 'flex-end', label: 'End' }, { value: 'space-between', label: 'Space Between' }] },
            { prop: 'alignItems', label: 'Align', control: 'select', options: [{ value: 'flex-start', label: 'Start' }, { value: 'center', label: 'Center' }, { value: 'flex-end', label: 'End' }, { value: 'stretch', label: 'Stretch' }] },
            { prop: 'gap', label: 'Gap', control: 'text' },
          ],
      },
```

Why this way: We avoid bespoke forms per element. Adding a new property is editing a schema entry rather than writing new component logic, accelerating iteration and reducing bugs.

The `PropertiesEditor` reads the active component's schema to render matched controls and applies debounced updates. Debouncing prevents spammy state writes while a user types.

```174:183:src/components/PropertiesEditor.tsx
useEffect(() => {
  if (JSON.stringify(component.props) === JSON.stringify(currentProps)) {
    return;
  }
  const handler = setTimeout(() => {
    onUpdateComponent(component.id, { props: currentProps });
  }, 500);
  return () => clearTimeout(handler);
}, [component.id, component.props, currentProps, onUpdateComponent]);
```

Why this way: Debounce balances responsiveness with performance, especially on larger trees where frequent renders would degrade UX.

The `BoxControl` enables compact editing of 4-value CSS shorthands with three modes (overall, axis, individual), crucial for ergonomics around padding/margin.

8) Component Palette and Menubar

We implemented a flat, horizontally scrollable palette for quick insertion of components with sensible defaults. Defaults are set to produce immediate visual feedback on insertion.

```13:31:src/utils/palette.tsx
export const paletteItems: PaletteItem[] = [
  {
    type: 'container',
    label: 'Container',
    icon: <Box size={16} />,
    defaultProps: {
      padding: '20px',
      display: 'flex',
      flexDirection: 'column',
      gap: '10px',
    },
  },
  {
    type: 'text',
    label: 'Text',
    icon: <Type size={16} />,
    defaultProps: { content: 'Type something...', fontSize: '16px' },
  },
```

Why this way: A single array drives both UI and behavior and is easy to extend. Providing defaults reduces friction—added elements look “right” immediately.

The menubar encapsulates common document commands (new/open/save/import/export), editor toggles (panel visibility), and preview/publish stubs. It also integrates a hidden `<input type="file">` for JSON import.

```81:111:src/components/Menubar.tsx
<MenubarMenu>
  <MenubarTrigger>File</MenubarTrigger>
  <MenubarContent>
                    <MenubarItem onClick={onNewPage}>New Page
<MenubarShortcut>⌘N</MenubarShortcut></MenubarItem>
```

```
  <MenubarItem onClick={() => { navigate("/projects") }}>Open...</MenubarItem>
  <MenubarSeparator />
                                        <MenubarItem onClick={onSave}>Save
<MenubarShortcut>⌘S</MenubarShortcut></MenubarItem>
  <MenubarSeparator />
  <MenubarItem onClick={handleImportClick}>Import from JSON...</MenubarItem>
  <MenubarSub>
    <MenubarSubTrigger>Export</MenubarSubTrigger>
    <MenubarSubContent>
          <MenubarItem onClick={() => { if (onExportJson) onExportJson() }}>as
JSON</MenubarItem>
          <MenubarItem onClick={() => { if (onExportHtml) onExportHtml() }}>as
HTML/CSS</MenubarItem>
    </MenubarSubContent>
  </MenubarSub>
  <MenubarSeparator />
            <MenubarItem onClick={onDeleteProject} className="text-red-600
hover:text-red-800">Delete Project...</MenubarItem>
 </MenubarContent>
</MenubarMenu>
```

Why this way: A consistent, discoverable place for document commands reduces cognitive load. Shortcuts and predictable menus mirror professional design tools.

9) Sidebar: Layers and Prebuilt Components

The sidebar toggles between a hierarchical “Layers” list (with selection, rename-in-place, reorder, collapse/expand, keyboard nav) and a “Components” tab for prebuilt snippets.

```26:44:src/components/Sidebar.tsx
return (
 <div style={{ display: 'flex', flexDirection: 'column', height: '100%' }}>
   <div style={{ display: 'flex', borderBottom: '1px solid #e5e7eb', marginBottom: '12px'
}}>
            <button style={tabStyle(activeTab === 'layers')} onClick={() =>
setActiveTab('layers')}>Layers</button>
           <button style={tabStyle(activeTab === 'components')} onClick={() =>
setActiveTab('components')}>Components</button>
```

```
    </div>

    <div style={{ flex: 1, minHeight: 0 }}>
      {activeTab === 'layers' && (
        <ComponentsList
          components={props.components} selectedId={props.selectedId}
          onSelect={props.onSelect} onRemove={props.onRemove}
          onRename={props.onRename}
          onMove={props.onMove}
        />
      )}
                          {activeTab === 'components' && <PrebuiltComponents
onAddComponent={props.onAddPrebuilt} />}
    </div>
  </div>
);
```

Why this way: The layers-first UX emphasizes structure and hierarchy, while a separate components tab encourages reuse and speed.

10) Persistence: Project Repository and Actions

We implemented a `documentRepository` backed by `localStorage` to save, load, list, create, and delete projects with an index for `ProjectsList` sorting by `lastModified`.

```22:41:src/data/documentRepository.ts
export const saveDocument = (document: Builder): void => {
  const documentKey = `builder_doc_${document.id}`;
  localStorage.setItem(documentKey, JSON.stringify(document));

  const index = getIndex();
  const existingEntry = index.find((p) => p.id === document.id);
  const now = new Date().toISOString();

  if (existingEntry) {
    existingEntry.name = document.name;
    existingEntry.lastModified = now;
  } else {
    index.push({ id: document.id, name: document.name, lastModified: now });
```

```
  }
  saveIndex(index);
};
```

Why this way: For an MVP, `localStorage` is sufficient, requires no backend, and keeps the developer velocity high. The index enables a simple project dashboard.

On top of that, `useDocumentActions` centralizes user-triggered commands like save/load/export/import. The HTML export renders the clean component tree and wraps it with basic boilerplate before downloading.

```69:117:src/hooks/useDocumentActions.tsx
const exportAsHtml = useCallback(() => {
    if (!builder) return;

    const tempContainer = document.createElement('div');
    document.body.appendChild(tempContainer);

    const root = createRoot(tempContainer);
    root.render(
        <ExportRenderer components={builder.components} globalStyles={builder.styles}
/>
    );

    setTimeout(() => {
        const htmlContent = tempContainer.innerHTML;
        root.unmount();
        document.body.removeChild(tempContainer);

            const fullHtml = `<!DOCTYPE html>\n<html lang="en">\n<head>\n  <meta
charset="UTF-8">\n  <title>${builder.name || "Exported Page"}</title>\n <style>\n body
{ margin: 0; font-family: sans-serif; }\n* {\n  overflow-wrap: break-word;\n  word-break:
break-word;\n}\n
</style>\n</head>\n<body><div>${htmlContent}</div></body>\n</html>`;
        const blob = new Blob([fullHtml], { type: 'text/html' });
        const url = URL.createObjectURL(blob);
        const a = document.createElement('a');
        a.href = url;
        a.download = 'index.html';
```

```
      a.click();
      URL.revokeObjectURL(url);
    }, 100);
}, [builder]);
```

Why this way: Rendering first ensures the exported HTML mirrors the current visual output. Delaying 100ms lets React flush the render to the temporary container.

11) Auto-Save UX

We added a small, reusable `useAutoSave` hook that debounces save operations and surfaces a simple status machine for a Save status bar.

```6:13:src/hooks/useAutoSave.ts
export const useAutoSave = (
  data: Builder | null,
  onSave: (data: Builder) => Promise<void> | void,
  delay: number = 1500,
) => {
  const [status, setStatus] = useState<SaveStatus>("idle");
  const [savedAt, setSavedAt] = useState<Date | null>(null);
```

```21:35:src/hooks/useAutoSave.ts
const handler = setTimeout(async () => {
  setStatus("saving");
  await onSave(data);
  setStatus("saved");
  setSavedAt(new Date());
  setTimeout(() => setStatus("idle"), 3000);
}, delay);
```

Why this way: This UX pattern reduces cognitive load—users see clear feedback without manually hitting “Save” constantly. The hook is independent of the repository layer for reuse.

12) Editor Layout and Visual Polish

The editor shell (`EditorLayout`) composes menubar, optional topbar, sidebar, canvas, and properties panels with a clean aesthetic. While not business logic, this polish improves perceived quality and usability.

Why this way: A comfortable, modern layout mirrors users' expectations from design tools (Figma, Notion, etc.) and makes features more discoverable.

Rationale Behind Key Decisions

- Schema-driven properties: Central for maintainability and speed. It unlocks non-breaking growth in element complexity.
- Single state hook for builder: Easier to reason about changes, ensures consistent undo/redo, and simplifies testing later.
- Two renderers sharing the same registry: Minimizes divergence between on-screen and exported output.
- Local persistence first: Ideal for a self-contained MVP. An API-backed repository can be added later behind the same interface.
- Keyboard navigation and rename-in-place: Power-user features that compound productivity as documents grow.

Practical Guidance: Extending and Maintaining

- To add a new element, implement the React component, declare its schema, register in `componentMap`, and add to the `paletteItems` with sensible defaults.
- To add a new property control, extend the `ControlType` set and update `PropertiesEditor`'s renderer. Most elements will then benefit with zero changes to their UI.
- To integrate a backend, replace `documentRepository` with API calls; keep the hook signatures unchanged to avoid UI churn.

What's Next

- Drag-and-drop placement between containers.
- Responsive breakpoints and constraint systems at the schema layer.
- Asset management with uploads and transforms.
- Real-time collaboration over WebSocket or CRDT/state-sync layer.
- Test coverage for reducers and schema application logic.

With this creation-process lens, every file in the repository plays a defined role in a coherent architecture that values extensibility, clarity, and a clean export story. This structure positions the project to grow into a more capable page/site builder with limited rework.