ChatGPT

Next.js Live UI/UX Editing Engine Starter Project

Overview

This starter project implements an **Edit Mode** feature in a Next.js (TypeScript) app, enabling live in-context editing of UI content for a personal/business finance management app. In Edit Mode, authorized users can modify text content, component props, and design tokens (like colors) directly on the page, with changes applied in real-time as **draft** edits. Key capabilities include:

- **Inline Editing of Text and Props:** Users can click on text or UI elements to edit their content or properties. Edits are captured as JSON Patch operations (per RFC-6902) so that only the changes are stored [1], not whole copies of the data. This makes updates precise and efficient.
- **Color Token Management:** Design color tokens (e.g. brand colors, theme colors) are editable with support for modern color standards. The system uses **OKLCH** color values for perceptual consistency [2] and validates contrast using **APCA** (Advanced Perceptual Contrast Algorithm) to ensure accessibility [3]. If a chosen foreground/background color combination fails a contrast threshold (e.g. APCA score below 60), the editor can warn the user.
- **Draft/Publish Workflow:** Edits go into a *draft state* that can be previewed in a **diff viewer** before publishing. Users with the proper role can publish drafts to make them live, or roll back to previous versions. Every change is tracked and versioned for auditability. Undo/redo functionality is provided so that editors can revert mistakes or re-apply changes easily.
- **Role-Based Access Control:** The editing tools respect user roles. For example, a user with an **Editor** role may create and save drafts, but only a **Publisher** role can publish those changes live. All edit actions (edit, preview, publish, rollback) check the current user's role before proceeding, enforcing a simple Role-Based Access Control policy.
- **Modular & Typed Architecture:** The project is organized into clear modules – an `EditModeProvider` context for state management, a set of reusable inline editor components, a diff/preview component, and mock backend API routes – all written in TypeScript for type safety. JSON Schemas define the shape of editable data (text copy, colors, component props), and these schemas are used to validate edits on the fly. Test coverage is provided for core flows (e.g. editing and undo, publishing, role restrictions) to ensure reliability. Minimal styling is applied using Tailwind CSS utility classes just to make the editing UI clear, keeping the focus on functionality.

Overall, this starter project provides a foundation that is **self-contained and easily extendable**. Developers can build on it to support more component types or integrate a real backend later, without needing to rewrite the core editing engine.

Project Structure and Design

The project is a standard Next.js application (using the Pages Router for simplicity) with a focus on the editing engine modules. The key parts of the codebase are organized as follows:

- **Context Provider (`EditModeProvider`):** Located in, for example, `context/EditModeContext.tsx`. This uses React Context to hold global editing state (draft data, patches, etc.) and provides actions for modifying that state. It wraps the app (e.g. in `_app.tsx`) so all components can access editing functionality.
- **Editable Components:** A set of React components for inline editing:
- `InlineTextEditable` – for in-place editing of text content (like headings, labels, paragraph text).
- `PropEditor` – a generic form control to edit a component's props (numbers, booleans, enums, etc.).
- `ColorTokenEditor` – an interface (e.g. color picker or input) to edit design token values (stored as color strings, possibly in OKLCH format). These components reside in a folder like `components/editable/` and are used in the app wherever editable content appears. Each is aware of Edit Mode (via context) and conditionally renders either a read-only view or an input control. They also perform JSON Schema validation of inputs before submitting changes.
- **Diff Viewer:** A component (in `components/DiffViewer.tsx`) that shows the differences between the published content and the current draft. This might be a modal or page that lists changes in text or props side-by-side (original vs edited) or with plus/minus highlights. Under the hood it can leverage a diff library (for example, using `jsondiffpatch` to compare two JSON states and generate a human-readable diff [4]).
- **Mock Backend API Routes:** Under `pages/api/`, a set of API route handlers simulate backend endpoints for saving drafts, publishing, and retrieving version history. For instance, `pages/api/content.ts` might handle GET (fetch current published content and any draft) and PATCH (apply a JSON Patch to the draft data) [5] , and `pages/api/publish.ts` handles finalizing a draft to published. These routes use an in-memory store (such as a module-scoped variable or simple object) to hold data – meaning no database is required to run the project. On each publish, a new "version" is appended to a history log with timestamp and user, to emulate audit logging.
- **Schemas and Validation:** A `/schemas` directory contains JSON Schema definitions for the editable data structures (e.g. schema for copy text fields, schema for color tokens, schema for component props). The application uses these schemas both client-side (via a library like AJV for JSON Schema validation) and server-side to validate incoming patches or draft data. This ensures that invalid data (e.g. a number out of range, or an incorrectly formatted color) is caught early.
- **Utilities:** Utility modules include functions for applying JSON Patches, computing APCA contrast scores, converting color formats (OKLCH ↔ HEX/RGB), etc. For example, a `colorUtils.ts` might wrap an APCA calculation algorithm (using known formulas or an open-source library) and expose a function like `checkContrast(color1, color2)` that returns an APCA score. Another utility might handle generating JSON Patch operations given an original and new value (if not using an existing diff library).
- **Styling:** Tailwind CSS is included for convenient styling of the editing UI. For example, inline editors can show a subtle border or background on hover, and invalid inputs can be highlighted with Tailwind classes (e.g. a red border). Tailwind is configured minimally (mostly using default utility classes) since the UI styling is not the primary focus.

This structure ensures a clear separation of concerns: context/state vs presentation vs backend logic. The app remains **highly modular** – for instance, one could drop in a new inline editor component for a different data type, or replace the mock API with real endpoints, without disrupting other parts. TypeScript types and JSON Schemas provide a single source of truth for what content and props are allowed, making it easier to extend the system confidently.

EditModeProvider: Context for Editing State

The **EditModeProvider** is the heart of the editing engine. It leverages React Context to share editing state and actions across the component tree. Internally, it may use `useReducer` or `useState` hooks to manage a complex state that includes:

- **Edit Mode Status:** A boolean flag indicating if Edit Mode is active. When off, the UI is in normal view mode; when on, editable components render their input controls and editing UI. This can be toggled by the user (e.g. via an "Enter Edit Mode" button) if the user has editing privileges.
- **Content State:** Both the *current published content* and the *draft content* being edited. This content can be stored as a JSON object representing all editable data (for example, a big object with keys for texts, colors, etc.). On entering Edit Mode, the draft state might start as a clone of the published state. Edits are applied to the draft only, so the published state remains unchanged until a publish action.
- **Draft Patches (Change Set):** A collection of JSON Patch operations representing all unsaved edits made in this session. Each patch follows the RFC-6902 format (e.g. `{ op: "replace", path: "/homepage/heroTitle", value: "New Title" }`) which precisely defines the document location and change [6]. Using JSON Patch for drafts has several benefits: it is **precise** (targets a specific field), can express multiple changes as an array of operations, and is **atomic/rollback-able** (the patch list can be unapplied or reapplied as needed) [7]. The context can store patches in the order they were made; this serves as a "draft diff" that can be sent to the backend or used for preview.
- **Undo/Redo History:** The provider maintains history stacks to support undo/redo of edits. A simple approach is to keep two stacks of patches: an **undo stack** and a **redo stack**. When the user makes a change (e.g. edits text), a patch is generated and pushed onto the undo stack, and the redo stack is cleared. On **Undo**, the last patch from the undo stack is popped and reverted – meaning we apply the inverse of that patch to the draft state. JSON Patch makes it straightforward to invert operations: for a `replace` operation, the inverse is another `replace` swapping the old value back; for an `add`, the inverse is a `remove`, etc. (In fact, libraries exist to assist with this inversion, treating JSON Patch as a reversible log of changes [8] [9].) The reverted patch is pushed onto the redo stack. Similarly, a Redo pops from redo stack and re-applies that change (moving it back to undo stack). This mechanism allows editors to step backward or forward through their draft modifications, much like undo/redo in a text editor. Internally, the context might generate and store **inverse patches** automatically when changes occur, to facilitate easy undo [10] [11].
- **User Role & Permissions:** The context also holds information about the current user's role (e.g. `role: "EDITOR"` or `"PUBLISHER"`). This is used to guard certain actions. For instance, the context could expose a `publishDraft()` function that will check the role and only proceed if the user is a Publisher. Similarly, entering edit mode or saving a draft might require an Editor or higher role. By centralizing this in the provider, the UI components can stay simple – e.g. a "Publish" button can call `editMode.publishDraft()` without worrying about the role logic (the provider will no-op or throw if not allowed). Role enforcement ensures that even if someone manages to toggle an edit

UI, the critical actions are still protected on the client, and of course the backend will double-check roles as well.

Implementation: `EditModeProvider` is implemented as a React component that wraps its children with a Context provider. It likely uses something like:

```
interface EditModeState {
  draftData: ContentData;
  originalData: ContentData;
  patches: JSONPatchOperation[];
  undoStack: JSONPatchOperation[][];
  redoStack: JSONPatchOperation[][];
  userRole: Role;
  // ...other flags or settings
}
```

The `patches` could be an array of operations or an aggregated JSON Patch document (which is itself an array). The undo/redo stacks might store arrays of ops or single ops depending on whether we batch multiple changes at once. For simplicity, assuming each edit action produces a single JSON Patch operation, we can push that single-op array to the undo stack. Alternatively, if we want to allow grouping (e.g. multiple fields before pressing "Save"), we might accumulate patch ops and treat the whole set as one transaction for undo. This design is flexible.

The context provides methods such as:

- `startEditing()` / `stopEditing()`: to toggle the Edit Mode flag. When starting, we might clone current content to `draftData`. When stopping (without publishing), we might drop the draft changes or leave them for later.
- `applyPatch(patch: JSONPatchOperation)`: to apply a single change. This will update the `draftData` (we can use a library like `fast-json-patch` or a custom function to apply the patch to the JSON object) and update the patch history (push to undo stack, etc.). If we have a JSON Schema for the target of the patch, we can validate here: e.g. if patch is replacing `/settings/maxItems` with value 15, check schema for `maxItems` (maybe it must be ≤ 10) – if invalid, reject the patch (do not apply and maybe report an error to the UI).
- `undo()` / `redo()`: as described, manage moving patches between stacks and applying the inverse operations to `draftData`. For example, an undo might take the last patch or patch set, generate an inverse (if not stored already), and apply it. The result is that `draftData` returns to a previous state.
- `publishDraft()`: contacts the backend to save the draft as the new published data. This might call a Next.js API route (via `fetch` or using SWR/mutation). Before sending, we can present a confirmation (e.g. open the diff viewer). On successful publish, the provider might update `originalData` to the new content and clear out `patches` and history (since draft is now committed). It would also log the action (e.g. in console or state) for audit.
- `discardDraft()` (optional): to abandon the draft changes – simply reset `draftData` to `originalData` and clear patch lists and history, leaving everything as it was.

Security-wise, these methods check roles. For instance, `applyPatch` and `startEditing` may require at least Editor role, `publishDraft` requires Publisher. This can be as simple as `if (userRole !== 'PUBLISHER') return alert("Not authorized")` on the client, but also the API route will verify the role from the request (in a real app, via auth token or session).

Using **JSON Patch** as the core diff format is very beneficial in this context. It provides a standardized way to express changes, and is lightweight enough to send over HTTP if needed (using the HTTP PATCH method with `application/json-patch+json` content type is the standard for partial updates [5]). It also aligns with the undo/redo approach: since each operation is an explicit, granular change, you can invert or reapply them systematically [12]. The JSON Patch standard supports operations like *add*, *remove*, *replace*, *move*, etc., which means even complex changes can be captured. In our use-case, most edits will be simple "replace this value" operations, possibly some "add" if adding new list items or so, which are straightforward to handle.

Finally, the EditModeProvider can also integrate with React DevTools or logging in development mode, to make it easier to debug the sequence of patches. But by and large, it serves as an **encapsulated editing state manager**. Components subscribe via the context hook (e.g. `useEditMode()` that provides state and the actions) and then act accordingly.

Inline Editable Components

To make the editing experience intuitive, the UI uses specialized components that render either display-only content or an interactive editor depending on context. These **inline editable** components all share some common behavior: they check if Edit Mode is active and if the current user is allowed to edit the specific content. If yes, they render an input control (text field, select box, etc.) with appropriate validation; if not, they just render the content normally. The components also use the context's methods to save changes (typically by producing a JSON Patch and calling `applyPatch`). Let's go through the main ones:

InlineTextEditable

This component allows inline editing of text content, such as headings, labels, or paragraph text in the app. In view mode, it might render a `<span>` or `<p>` with the text. In edit mode (for authorized users), it renders an input field (like `<input type="text">` or a `<textarea>` for multi-line) pre-filled with the current text. Key features of `InlineTextEditable` include:

- **Activation & UI:** Often, the component visually indicates it's editable on hover or with an edit icon when in Edit Mode (e.g. a pencil icon or dashed underline to hint the text can be clicked). Once clicked or focused, it turns into a text input. We use minimal Tailwind styling here, for example: a subtle border and background on focus, so the editor knows they are editing this text.
- **Editing & Validation:** The component likely accepts props specifying the **JSON Pointer** path to the text in the content object (for example `path="/homepage/heroTitle"`). When the user types a new value, we can do two forms of validation:
- **JSON Schema validation:** We have a schema for copy/text fields. For example, it might require that the text is a string within a certain length (min 1 character, max 100 or 200 characters, etc.), or match a regex (if, say, it should not contain certain symbols). We can compile this schema with a library like

Ajv and use it to check the value as the user types or on blur. If invalid, the component can show an error message (and refrain from applying the patch until fixed).
- **Content-specific validation:** Perhaps certain text fields in a finance app should be non-empty or numeric-only if it's an amount, etc. These can be handled via either the schema or additional custom checks passed as props (for a starter, schema alone might suffice).
- **Saving Changes:** When editing is complete (e.g. on blur of the input, or pressing Enter), the component will construct a JSON Patch operation: `{ op: "replace", path: <the JSON pointer>, value: <new text> }`. Before dispatching it, it might attach a `test` operation if needed to ensure it's patching the expected old value (though in a single-user scenario this might be overkill – but it's an available feature of JSON Patch for concurrency control [13]). The patch is then sent to the context via something like `editMode.applyPatch(patch)`. The context updates the draft data and tracks the change for undo. The UI immediately reflects the new text (since it's bound to draft data state).
- **Reusability:** This component can be used for any text field. It might accept a `label` prop for accessibility (e.g. screenreader labels when editing), and it could also accept a `multiline` prop to use a textarea for longer text blocks. The JSON Schema for text could be generic, or we might pass in a specific schema override for certain fields (for example, a field that expects a number in a string could use a different schema or validation function).

By using InlineTextEditable everywhere for text content, we ensure a consistent editing experience. For instance, an `<h1>` that is editable would actually be `<InlineTextEditable as="h1" path="/homepage/heroTitle" />` – the component could use the `as` prop to render a `h1` in view mode and an appropriate input in edit mode to preserve styling.

PropEditor

This component (or set of components) handles editing of component **properties** (props) in the UI. In a live-editable interface, you might want to adjust how a component behaves – for example, toggling a feature on or off, changing a layout option, or updating a numeric setting. The PropEditor can be implemented in a few ways. One approach is a contextual toolbar or sidebar: when you focus or select a component in Edit Mode, a small popover or side panel shows the editable props for that component, each with an appropriate input control. However, to keep things simple, we can also embed PropEditor controls in the component's JSX, visible only in edit mode.

Features of `PropEditor` include:

- **Dynamic Form Controls:** It uses the JSON Schema for the component's props to decide what inputs to render. For example, suppose we have a `BudgetChart` component with props `{ showTrendline: boolean, maxMonths: number, chartType: "bar" | "line" }`. The PropEditor for this component would:
- Render a checkbox for `showTrendline` (boolean).
- Render a number input (perhaps with min/max) for `maxMonths`.
- Render a dropdown select for `chartType` with options "bar" and "line". This mapping from data types to form controls is often manual or using a library. For a starter kit, we might manually code a small mapping (booleans -> checkbox, number -> `<input type="number">`, enum -> `<select>`).

- **Inline or Panel UI:** The PropEditor could appear inline (e.g. beneath the component or overlayed) whenever Edit Mode is on. For example, a card component could, in edit mode, show a small editable section at the bottom with its props. Alternatively, we could make a floating panel that shows props for the last clicked component. Either approach is fine; inline keeps everything in one place for a starter project. We'd likely conditionally render it only when editing, e.g.:

  ```
  {editMode.isActive && <PropEditor path="/components/BudgetChart/123"
  schema={budgetChartSchema} />}
  ```

  where the `path` might point to the JSON object storing that component's props. The schema is used to know what fields to edit.
- **Validation and Type Safety:** Each prop's input will enforce the type defined in schema. If the schema says a number has a minimum of 1, the number input can have `min={1}` and we can validate on blur that it's >=1. For strings with an allowed set, we use a select. We ensure that after editing, the value conforms so that applying the patch doesn't violate the data constraints. JSON Schema validation can run on the whole props object as well, not just field-by-field, to catch any interdependent rules.
- **Applying Changes:** Similar to text, when a prop value is changed, we create a patch. The path might look like `/components/BudgetChart/123/maxMonths` for example, and operation "replace" with the new value. We send it through `applyPatch` so the context updates the draft state. The UI component using that prop will get the new value from draft data and update instantly, allowing the editor to *see the effect live*. (For instance, if `maxMonths` is bound to chart data length, the chart might immediately update to show fewer/more months.) This real-time feedback is a big UX win for such an editor.
- **Role and Access:** If a user does not have permission to edit (or if a particular prop is deemed not editable), the PropEditor can either not render or render a disabled control. For example, maybe some advanced prop is only for admins; the schema or a config could mark it as admin-only and the PropEditor checks user role to decide. This level of detail can be added as needed.

Overall, `PropEditor` provides a flexible way to manipulate component behavior without leaving the page or editing code. It's essentially a mini form that is context-aware. The use of JSON Schema for props means the editor is not hardcoded to specific components; adding a new component type to be editable is as simple as writing a schema for its props and including a `PropEditor` for it in the JSX.

ColorTokenEditor

The **ColorTokenEditor** is responsible for managing design tokens related to color (e.g. theme colors like primary, secondary, background, text colors). In a personal finance app, these might control the branding or theming (for instance, accent colors for charts or backgrounds for cards). This editor typically presents color values and allows changing them via a color picker or inputs, with immediate effect on the UI's theme. Key aspects are:

- **Displaying Tokens:** The editor could list all editable color tokens in a form, each with a label and a color input. For example, tokens like *primaryColor*, *secondaryColor*, *textColor*, *backgroundColor* would each have an entry. Alternatively, the editor might focus on one token at a time (for context-specific editing). A simple approach: have a panel (maybe in the app's settings or in a theme editor page) where all tokens are shown together for editing.

- **Color Format (OKLCH):** This project embraces modern color standards by using OKLCH for color values. OKLCH is a perceptually uniform color space, meaning adjustments in its parameters (especially lightness) correspond more closely to human vision differences [2]. In practice, we can store colors as strings like `oklch(0.6 0.1 230)` (which represents a color in OKLCH notation). The ColorTokenEditor can let users input colors in multiple ways:
- They can use a color picker (which might output hex or RGB by default). We then convert that to OKLCH under the hood.
- They can type an OKLCH string directly (for advanced users), or choose from presets.
- We can also display the color in a swatch for easy visualization. We will need a conversion utility. For example, if a user picks a hex `#A00000`, we convert it to `oklch(L C H)` format for storage, and vice versa for showing a hex value if needed. Libraries or formulas are available for this conversion (e.g. the Evil Martians "**apcach**" library and other tools can convert between OKLCH, HEX, RGB, etc [14]).
- **APCA Contrast Validation:** Ensuring accessible contrast is crucial, and we use the APCA algorithm for this. APCA (Accessible Perceptual Contrast Algorithm) is a newer method proposed in WCAG 3.0 that calculates contrast in a more accurate, perceptually-based way than the old ratio method [15] [3]. APCA produces a contrast score roughly between 0 and 100 (higher is more contrast) instead of a ratio [16]. Our ColorTokenEditor will use APCA to validate combinations like text on background:
- For example, if the token being edited is a text color and we know the background color (from another token or default background), we compute the APCA score for text vs background. If the score is below a threshold (e.g. APCA < 60 which is the minimum for body text readability [17]), we can flag a warning to the user (maybe show a message or outline the color input in red). This guides the user to pick a color that is sufficiently legible. Because APCA considers polarity (light text on dark background vs dark on light), we will ensure to calculate it correctly depending on which is foreground.
- Similarly, for background tokens, if there is a standard text color that goes on them, check that contrast.
- We might also provide suggestions: e.g. if contrast is too low, bump the OKLCH lightness up or down until it meets 60 and show that as a suggestion. Thanks to OKLCH, adjusting lightness is straightforward without hue shifting. (This is how tools like *Harmonizer* and *Polychrom* ensure consistent contrast by working in OKLCH and using APCA [18] [19].)
- **Applying Color Changes:** When the user selects a new color, similar flow: generate a patch like `{ op: "replace", path: "/designTokens/colors/primaryColor", value: "oklch(0.6 0.1 230)" }`. The context will apply it, updating the draft theme. We likely have the app's CSS hooked up to these tokens (e.g. via CSS variables or a Tailwind theme that reads from a JSON). For simplicity, we can update a `<style>` tag or CSS variables context when the draft changes. The user immediately sees, for example, the new primary color reflected across the UI (buttons, etc.), allowing them to preview if it looks good. If not, they can tweak further or undo.
- **Schema for Colors:** The JSON Schema for a color token could ensure the format is correct. It might be as simple as a regex pattern that matches either a hex or an `oklch(...)` string. For example:

```
{ "type": "string", "pattern": "^(#([0-9A-Fa-f]{3}|[0-9A-Fa-f]{6})|oklch\\([^)]*\\))$" }
```

This pattern allows 3 or 6 hex digits with an optional #, or an OKLCH function. We can validate user input against this. Additionally, if we accept color names or other formats, we adjust accordingly. With a proper color library, we might even parse and re-serialize to enforce a canonical format.

- **User Experience:** We should make it easy to use – possibly integrate with the native color picker input (`<input type="color">`) for a quick UI. However, the native color input only handles sRGB hex, so behind the scenes we convert that to OKLCH. Another nice touch is showing the current color and the new color side by side if editing, or showing the contrast score live.

Using OKLCH and APCA demonstrates a forward-looking approach to design tokens. OKLCH ensures our color adjustments are **predictable and uniform** (for instance, increasing the L value always makes the color perceptibly lighter without weird hue shifts, which isn't true in HSL or hex), and APCA ensures **accessibility** by not relying solely on old contrast ratios. In fact, APCA is touted as *"far more accurate than [the] current WCAG implementation"* for color contrast [3]. This foundation allows the design system to be more easily tuned for light/dark modes and a wide range of user vision differences.

Diff Viewer for Previewing Changes

Before publishing a set of edits, it's important to review what exactly has been changed. The Diff Viewer component provides a clear **before-and-after comparison** of the content. When the user triggers a "Preview Changes" action (say, clicking a *Preview* button in the UI), the Diff Viewer will appear (either as a modal overlay or navigating to a `/preview` page) and show all modifications in the current draft compared to the last published version.

How it works: The Diff Viewer obtains two versions of the data from the EditMode context – the original (published) content and the draft (edited) content. It then computes the differences. This can be done simply by using the list of JSON Patch operations accumulated (since they essentially *are* the differences). However, to present it nicely, we might want to show the actual values before/after, especially for text content. We have a couple of approaches:

- **JSON Patch to Human Readable Diff:** We can iterate through the patch list and for each operation, retrieve the value *before* the patch from the original data and the value *after* from the draft. For example, a patch `{ op: "replace", path: "/profile/username", value: "alice99" }` – the diff viewer can look up the original `profile.username` ("alice") and show a line like: *Username: ~~alice~~ alice99 (with strike-through for old value and bold or colored text for new). If multiple changes occurred in one field (less likely since each patch is separate), each patch is handled individually. Additions (* `op: "add"` *) can be shown as added content*, maybe highlighted in green, and deletions (`op: "remove"`) as *removed*, highlighted in red. Moves or complex operations can be broken down but those might not be common in our use-case.
- **Using a Diff Library:** We can leverage existing libraries like `jsondiffpatch` which can compute a structured diff between two JSON objects. In fact, `jsondiffpatch` can even generate an HTML representation of the differences. There is a React wrapper available that can take a `left` (original) and `right` (edited) object and render a diff view with annotations [20] [21]. For example, unchanged fields can be hidden or shown, changed fields will be displayed with markings. This saves us from writing the diff logic manually. For our starter, we could include this library and use its default styling for the diff output, which typically shows deletions in red and additions in green, nested appropriately according to the JSON structure.
- **Focus on Key Changes:** We might not want to dump the entire JSON; rather, we show a list of changed items. A user-friendly approach is to present a summary like a bullet list:
- "Changed **Hero Title** from "*Welcome to MyApp*" to "*Welcome to MyApp (Pro Edition)*""
- "Set **Show Trendline** to "*true*" on BudgetChart component"

- "Updated **Primary Color** from `#0033AA` to `#0055CC` (contrast vs background improved to APCA 75)" Each of these lines corresponds to a patch or related group of patches. This format is easy to read. Implementing this requires mapping JSON paths to human-friendly names (we might maintain a dictionary of field labels, e.g. `/homepage/heroTitle` -> "Hero Title"). For the color example, we even calculated APCA to mention the outcome. Such niceties can be added as needed.

In the interest of time, using a ready diff component might be the quickest route. If we use `jsondiffpatch-react`, for instance, it can show the diff with some annotations explaining additions/removals [4] . The Diff Viewer component could simply wrap that and also provide a Publish button.

UI integration: The Diff Viewer likely has a "Cancel" (go back to editing) and "Publish Changes" button for convenience. Only a Publisher-role user will see the Publish option enabled. If an Editor opens it, they might see the diff but not be able to confirm publish (or that button is hidden/disabled). This final review step helps avoid mistakes and also provides a chance to do one more validation pass (the code can run validation on the draft before publishing, ensuring everything is consistent with schemas, although ideally those were enforced already on each edit).

In summary, the Diff Viewer ensures transparency of what will change. It leverages the accumulated knowledge of changes (patches) to present a clear before/after snapshot. By seeing the differences side-by-side, users can confidently proceed to publish, knowing exactly what updates they're making.

Mock Backend: Draft Storage, Versioning, and Publishing Flow

To keep this starter project simple yet realistic, we implement a **mock backend** using Next.js API routes. This simulates how a real server would store draft changes, handle publishing, and keep an audit log of versions. All data is kept in-memory (so it resets on server restart), which is sufficient for development and demo purposes. The main parts of this backend simulation are:

- **Content Data Store:** We use a module-scoped variable in the API route files to hold the current state of the content. For example, in `pages/api/content.ts` we might have:

  ```
  let currentContent: ContentData = { ...initialData... };
  let draftContent: ContentData | null = null;
  const history: VersionEntry[] = [];
  ```

  The `ContentData` type corresponds to the JSON structure of all editables (texts, colors, props). `history` could be an array of version objects, where each `VersionEntry` contains maybe an `id` or version number, a timestamp, the content state (or the patch applied), and who made the change. This simulates a version control or audit log.
- **API Endpoints:** We create endpoints for relevant actions:
- `GET /api/content`: Returns the current published content, and possibly any draft if it exists (or we could keep draft separate). In a simple design, when Edit Mode starts, the front-end could call this to load the latest content. (In Next.js, you can also fetch this on the server side as initial props, but an API gives flexibility for client polling.)
- `PATCH /api/content`: This could be used to save a draft incrementally. In a real scenario, using HTTP PATCH with JSON Patch is ideal: the client sends the patch document (list of ops) and the server

applies it to its stored draft. For our mock, we can support receiving a JSON Patch in the request body and do:

```
import { applyPatch } from 'fast-json-patch';
// ...
if (!draftContent) draftContent = structuredClone(currentContent);
draftContent = applyPatch(draftContent, req.body.patch).newDocument;
// maybe store the patch or validate it as needed
return res.status(200).json({ status: 'draft updated' });
```

We would validate `req.body.patch` against the JSON Patch schema (to ensure it's a valid patch format) and also perhaps ensure it doesn't modify disallowed fields. JSON Patch can be **validated using JSON Schema** as well [22] [23], which is a good practice. This route should also check user role (perhaps via a token or header; for the mock, the role might be passed in the request for simplicity).

- `POST /api/publish`: This endpoint finalizes the draft. It might simply take the current `draftContent` and promote it to `currentContent`. It would create a new entry in `history` with an incremented version number, timestamp, and maybe a summary of changes (we could store the patch list or the diff here for auditing). Then it clears `draftContent` (or we keep it until next edit). The response could be the new version or success status. This action must verify that the requester is a Publisher-role – in a mock, we might check a query param like `?role=publisher` or some hardcoded check.
- `GET /api/history`: Returns the list of past versions (for display or potential rollback).
- `POST /api/rollback`: Allows reverting to a prior version by ID. This would find the entry in `history`, take its content snapshot, set `currentContent` to that snapshot, and perhaps push the current content as a new version if we want to record the rollback as another entry (with a note that it was a rollback). This again would require a Publisher or Admin role.

All these routes are defined in plain Next.js API route style (functions that take `req, res` and send a JSON response) [24] [25]. They run on the server side (Node.js), but since we are not using a database, the data is reset every run – which is acceptable for a starter demo. - **Data Validation and Logging on Server:** On the server, we mirror the validation using the same JSON Schemas to double-check that incoming data is valid. For example, if the client inadvertently or maliciously tries to set a string where a number should be, the server can catch it. This is part of a robust system (defense in depth). Our mock API can use Ajv to validate `draftContent` before saving it as current content on publish. If invalid, it rejects the publish (which should rarely happen if client did its job).

Audit logging in the mock can be as simple as console logs or pushing to the `history` array. Each entry might contain:

```
interface VersionEntry { id: number; timestamp: number; authorRole: string;
patches: JSONPatchOperation[]; contentSnapshot: ContentData; }
```

Storing the `patches` that led to this version is useful for insight, and `contentSnapshot` is the full content after applying those patches. The array acts like a version control timeline. The `id` could be autoincrement or just use the array index. This is not a full git-like diff storage, but enough for viewing and rollback.

- **Role Enforcement:** Although our app primarily relies on front-end role checks (since this is a closed environment), we also enforce roles in the API to mimic real security. This could be done by requiring an authorization header or token (beyond our scope to implement full auth here). For demonstration, one might include the role in the request (e.g. as part of the request body or a special header). The API then does:

  ```
  if (req.body.role !== 'PUBLISHER') {
    return res.status(403).json({ error: 'Forbidden' });
  }
  ```

  for the publish route, for instance. This double-check ensures that even if someone called the API directly, they couldn't publish unless allowed.

With this mock backend in place, the **flow** works as follows: 1. When the app loads, it fetches the current content via `GET /api/content`. This populates the context's `originalData`. 2. Editor enters Edit Mode, context clones `originalData` to `draftData`. 3. User makes edits; context may call `PATCH /api/content` for each change or batch them. In a simple offline-first approach, we actually might not call the API for each edit (we can keep it local until saving). We could choose to only call the API on publish or to autosave drafts. For now, assume we only send on publish to reduce complexity. 4. User clicks "Preview" – no API call, just showing diff of context states. 5. User clicks "Publish" – the front-end calls `POST /api/publish`. The server sets `currentContent = draftContent`, logs the version, and returns success. 6. The context on the client receives success, updates its `originalData` to match the new published content (which is the same as draft it already had), and clears the draft state (patches, etc.). Edit Mode can exit. 7. If needed, the app could now fetch `GET /api/history` to display a version list (or we keep the returned version info from publish). 8. Rollback would be another call, etc., which we handle similarly.

This cycle provides a realistic editing loop. Although in-memory, the structure is analogous to a real setup where `currentContent` is a database record and `history` is a versions table. The separation of **draft vs published** content ensures that incomplete changes never leak to regular users until explicitly published. And since every publish goes through a controlled endpoint, we have a single choke point to apply final validation and logging.

In a more advanced scenario, we could also implement *auto-draft saving* (periodically POST patches to server to save your work in case of refresh) and *multi-user editing* (which would require some locking or real-time sync). Those are beyond our starter scope, but the foundation laid with JSON Patch would actually facilitate real-time collaboration if needed, because JSON Patch is designed for expressing partial updates and could be used in WebSocket messages for collaborative editing.

Role-Based Access Control (RBAC)

As mentioned, the system distinguishes user roles to control who can edit content and who can publish it (and possibly who can perform other actions like rollback). We'll define a simple RBAC scheme with at least two roles:

- **Editor:** Can enter Edit Mode and make changes (text, props, colors). They can save drafts (which in our case means just leaving the draft in the system or maybe hitting a "Save Draft" which is akin to keeping their work for later). Editors **cannot** publish the changes to live.
- **Publisher:** Can do everything an Editor can, and additionally can publish drafts to make them live (and potentially rollback changes or approve others' drafts). This role might be for a manager or admin who reviews content before it goes live.

(We could also imagine a Viewer role with no edit rights, but in practice that would just mean they never enable Edit Mode. The UI could simply not show any edit controls for such users.)

Implementation of RBAC in the App: We keep the current user's role in the `EditModeProvider` (or possibly a separate Auth context, but to reduce complexity, one context can hold it). This is probably set based on a login or a config (for the demo, we might hardcode a user as Publisher for illustration, or allow switching role in a debug panel to test behavior). The role is then used in various places:

- **UI Access:** For example, the button that toggles Edit Mode is only shown if `userRole` is Editor or Publisher. If a Viewer somehow navigates to an edit URL, the provider would not allow editing. Similarly, the Publish button in the diff viewer only renders for Publishers. We might also visually indicate the user's role (like "You are in Edit Mode (Editor)" vs "(Publisher)").
- **Context Methods:** As described, methods in EditModeProvider check role. If an Editor calls `publishDraft()`, it could throw an error or simply do nothing and maybe set a state like `error: "Not authorized to publish"`. These errors can be displayed to the user.
- **API Routes:** Each API route can read the role (from a token or param). For the mock, we might include the role in requests. In real life, we'd have an authentication system and the server would derive role from the user's session or JWT. Nonetheless, our mock publish handler will enforce the rule (return 403 Forbidden if not Publisher) to simulate the backstop.

Using a context for role is straightforward and aligns with typical React patterns [26] [27]. For example, one could create an `AuthContext` or merge it in EditModeContext, and use a hook `useAuth()` to get the current user and role. The role can be a simple string union type in TypeScript (`type Role = "EDITOR" | "PUBLISHER" | "VIEWER"`), which makes it easy to restrict in code (like `if (role !== "PUBLISHER") { ... }`).

Testing role enforcement: We'll include tests (discussed later) to ensure that when an Editor is active, the Publish button is not shown and any attempt to call the publish API is blocked, whereas a Publisher can do so. This ensures our RBAC rules are effective.

By clearly separating roles, the project makes it safe to deploy an in-context editor in production – only authorized personnel can actually push changes to all users, whereas content editors can work on drafts without fear. This mimics real content governance processes.

JSON Schema Examples for Content

The project uses JSON Schema to define and validate the shape of editable content. By having explicit schemas for different data types (copy text, colors, component props), we gain a few advantages: **validation** (ensuring data integrity), **documentation** (anyone can see what structure and rules the data follows), and **tooling** (we can potentially auto-generate forms or use schema-driven UI if desired). Here are some starter schema examples:

- **Schema for Copy Text:** For simple text fields, a schema can enforce basic rules like type and length. For example:

```
{
  "$id": "https://example.com/schemas/copyText.schema.json",
  "type": "string",
  "minLength": 1,
  "maxLength": 200,
  "title": "Copy Text",
  "description": "A user-editable text string for UI copy.",
  "examples": ["Welcome to FinancePro!", "Your balance is below minimum."]
}
```

  This schema says any copy text must be a string between 1 and 200 characters. We can reference this schema in others or use it directly for all text fields. If some fields have more specific requirements (e.g. an email field), we'd have a different schema or an override with a pattern.

- **Schema for Color Token:** We want to accept either a hex code or an OKLCH function string. A schema can express this using a regex or an enumeration of formats. One approach is using a regex pattern as mentioned earlier:

```
{
  "$id": "https://example.com/schemas/colorToken.schema.json",
  "type": "string",
  "title": "Color Token",
  "description": "Color value in hex or OKLCH format.",
  "pattern": "^(#(?:[0-9A-Fa-f]{6}|[0-9A-Fa-f]{3})|oklch\\([^)]*\\))$"
}
```

  This pattern allows 3 or 6 digit hex with a leading `#`, or an `oklch(...)` string with any content inside parentheses (we could tighten that inner pattern to exactly numbers and commas if we want). We might also allow CSS color names or `rgb()` by extending the pattern or using an OR with multiple schemas (oneOf). For our scope, hex and OKLCH are enough. We use this schema to validate any color token value. Additionally, we might incorporate a constraint that certain tokens must meet a contrast requirement with others – but JSON Schema alone doesn't easily handle cross-field criteria, so that's handled in logic rather than schema.

- **Schema for Component Props:** This will vary per component type. We can define a schema for each editable component's props. For instance, consider a `BudgetChartProps` schema:

```
{
  "$id": "https://example.com/schemas/budgetChartProps.schema.json",
  "type": "object",
  "title": "BudgetChart Component Props",
  "properties": {
    "showTrendline": { "type": "boolean" },
    "maxMonths": { "type": "integer", "minimum": 1, "maximum": 24 },
    "chartType": { "type": "string", "enum": ["bar", "line"] }
  },
  "required": ["showTrendline", "maxMonths", "chartType"],
  "additionalProperties": false
}
```

This schema dictates that `showTrendline` must be boolean, `maxMonths` an integer between 1 and 24, and `chartType` one of two strings. By marking `additionalProperties: false`, we ensure no unknown prop gets added. The PropEditor can directly read this schema to know what fields to show and even the valid ranges or options. For another component, say a `TransactionList` component, we'd have a different schema. All these schemas can be indexed by component type in a lookup (like a mapping from component name to schema).

- **Composite Schema:** We may also have a top-level schema for the entire content JSON. For example:

```
{
  "$id": "https://example.com/schemas/content.schema.json",
  "type": "object",
  "properties": {
    "texts": {
      "type": "object",
      "properties": {
        "heroTitle": { "$ref": "copyText.schema.json" },
        "welcomeMessage": { "$ref": "copyText.schema.json" },
        "...": {}
      },
      "additionalProperties": false
    },
    "colors": {
      "type": "object",
      "properties": {
        "primaryColor": { "$ref": "colorToken.schema.json" },
        "secondaryColor": { "$ref": "colorToken.schema.json" },
        "textColor": { "$ref": "colorToken.schema.json" },
        "backgroundColor": { "$ref": "colorToken.schema.json" }
      },
      "additionalProperties": false
    },
    "components": {
      "type": "object",
      "properties": {
        "budgetChart1": { "$ref": "budgetChartProps.schema.json" },
        "transactionList1": { "$ref": "transactionListProps.schema.json" }
      },
      "additionalProperties": true
    }
  }
}
```

This is a rough idea: we divide content into sections (text copy, color tokens, components). Under `components`, keys like `budgetChart1` identify a particular component instance and we use the appropriate subschema for its props. In a real scenario, we might have an array of components, each with a `type` field that decides which schema to apply (this gets into polymorphic schemas). For a starter, a simpler approach is fine, even hardcoding known component IDs.

These schemas serve as the contract for our editing system. Both the front-end and back-end can use them. On the front-end, we use them in the editors to validate user inputs (as discussed, e.g., Ajv can validate a JSON Patch or the resulting draft against a schema and report errors [28] [23]). On the back-end, we validate the final draft on publish by running the `content.schema.json` validation on the draft content object – ensuring that no rules are violated before making it official. This prevents any edge cases or programmatic errors from introducing bad data.

Testing and Extensibility

We aim for this starter project to be reliable and easy to extend. To that end, we include **test coverage** for critical flows and have structured the code to allow new content types and rules without heavy refactoring.

Testing Strategy: We use **Jest** and **React Testing Library** for our tests (since Next.js is React-based). Key things to test:

- **EditModeProvider functionality:** unit tests for the reducer or state management. For example:
- Test that `applyPatch` correctly updates the draft data and records the patch in the undo stack.
- Test that multiple `applyPatch` calls accumulate patches and that `undo()` reverses the last change (the draft data returns to previous value, the patch moves to redo stack).
- Test that `redo()` reapplies a change after an undo.
- Test role restrictions: if we set `userRole` to Editor, calling `publishDraft()` in the context should result in an error or no state change; with Publisher role, it should proceed to call the API (which we can mock).
- **Editable Components:** using React Testing Library, we render components like `<InlineTextEditable>` in a test harness wrapped with EditModeProvider (with a sample content). Then:

- Simulate entering Edit Mode (maybe by toggling a value in context or having a context value where `isEditing=true`).
- Find the rendered input and simulate user typing. Assert that after blur/submit, the context's draft data was updated and the patch history grew. Also assert that invalid input is handled (e.g. try input exceeding max length and ensure an error message appears or patch not applied).
- Similar approach for `<PropEditor>`: e.g. provide a dummy component props in context, render PropEditor, toggle a checkbox or change a number input, and assert the context state updates accordingly. Ensure out-of-range values are blocked.
- `<ColorTokenEditor>`: simulate choosing a new color. Because color editing might involve an actual color picker dialog which is hard to test, we can instead simulate entering a new hex code in a text input. Then check that the context draft updated with the corresponding OKLCH string (meaning our conversion function was invoked). Also, if we have a function that computes APCA, we can unit test that function with known color pairs (there are known expected scores for certain pairs to verify correctness).
- **Diff Viewer:** We can test the diff generation logic independently. For example, given a known original and edited object, feed them into the diff utility and verify that the output contains expected strings or DOM elements indicating the changes. If using `jsondiffpatch`, perhaps we test that it highlights a changed field. If using our own summarizer, test that it lists correct before/after values.
- **API Routes:** Since these are simple, we might not need extensive tests, but we could use Next.js' testing approach or just call our handler functions directly with dummy requests. For example, test that if a non-PUBLISHER calls publish, the response is 403. Or test that applying a patch actually changes the stored data. However, given it's in-memory, we must reset the module state between tests (we can export the store from the module to manipulate it in tests, or better, structure the logic in pure functions that we can test).
- **End-to-End Smoke Test:** As a final integration test (maybe using a framework like Playwright or Cypress if needed), ensure that the whole flow works in a browser-like environment: an Editor can edit a field, see the change, preview diff, and a Publisher can publish. This might be out of scope for the starter but is a good goal to confirm the pieces integrate.

Extensibility: The project is designed to allow adding new editable content types or components with minimal fuss: - To add a new editable text field, you just include an `InlineTextEditable` in the JSX bound to the new content path and update the content JSON and schema accordingly. - To add a new component type with editable props, you create a new JSON schema for it, and use a `PropEditor` (or multiple specific editors) in that component's JSX for each prop (or a generic PropEditor that reads the schema dynamically). - New color tokens can be added by extending the color schema properties and adding fields in the ColorTokenEditor UI list. - The context and diff viewer don't need changes when new fields are added, because they work with the generic JSON data and patches. The diff viewer will automatically catch any new field changes as long as they appear in the draft data. - If one wanted to integrate a database or a real API, you could replace the Next.js API routes with calls to that external service. The front-end doesn't care as long as the contract is similar (e.g. instead of hitting `/api/publish` locally, you'd call your backend endpoint). - The use of JSON Patch means even real-time collaboration features could be introduced later (e.g. using websockets to sync patches among multiple editors). The current design doesn't implement that, but it's a natural extension of using a diff/patch model for state changes.

The codebase remains **modular**: context logic is in one place, UI components in another, schemas in another. This separation aligns with best practices and ensures that, for example, designers could tweak a

JSON schema for content limits without touching the React component code, or developers could swap out the patch library without affecting the UI.

Finally, minimal styling with Tailwind means developers can easily restyle the editing interface to match their app's look. Tailwind utility classes in the components can be replaced or augmented by custom CSS if desired, but as a starter, it avoids spending too much time on CSS.

Conclusion

This Next.js starter project provides a comprehensive starting point for a live UI editing engine. It brings together robust **state management** (via context and JSON Patch [1]), modern **design token handling** (OKLCH color space and APCA contrast checks for accessibility [3]), and solid **workflow support** (draft previews, version history, role-based publishing). The architecture is geared toward clarity and growth: new content pieces can be onboarded by defining their schema and adding appropriate editor components, and the system's core (EditModeProvider + backend) will handle the rest (tracking changes, ensuring validity, enabling undo/redo, etc.). With the included test coverage and modular design, teams can confidently build on this foundation to create a full-fledged content editing experience tailored to their app's needs, without risking regressions or maintenance headaches.

Overall, this project demonstrates how an in-app editing mode can be implemented with modern web technologies, balancing ease of use for content editors with technical rigor (like JSON schema validation and proper contrast enforcement) to maintain app integrity and accessibility. It's a powerful starting kit for any application that requires live content updates and a controlled publishing workflow.

Sources: The design and techniques referenced here draw on web standards and best practices, such as the JSON Patch RFC 6902 for expressing changes in JSON documents [1], the upcoming WCAG 3.0 APCA guidelines for accessible color contrast [3], and modern color management insights for using OKLCH color space [2]. These ensure that the solution is not only effective but also aligned with the broader direction of web development and design system engineering.

---

[1] [5] [6] [7] [13] [22] [23] [28] Unlocking the Power of JSON Patch | Zuplo Learning Center
https://zuplo.com/learning-center/unlocking-the-power-of-json-patch

[2] [3] [14] [18] [19] Exploring the OKLCH ecosystem and its tools—Martian Chronicles, Evil Martians' team blog
https://evilmartians.com/chronicles/exploring-the-oklch-ecosystem-and-its-tools

[4] [20] [21] GitHub - bluepeter/jsondiffpatch-react: Diff & patch JavaScript objects... in React
https://github.com/bluepeter/jsondiffpatch-react

[8] [9] [10] [11] [12] GitHub - johnpaulrusso/json-patch-history
https://github.com/johnpaulrusso/json-patch-history

[15] [16] [17] WCAG 3 and APCA | Dan Hollick
https://typefully.com/DanHollick/wcag-3-and-apca-sle13GMW2Brp

[24] [25] Pages Router: Creating API Routes | Next.js
https://nextjs.org/learn/pages-router/api-routes-creating-api-routes

[26] [27] The document outlines implementing access control in a React app using RBAC and ABAC methods, highlighting utilities for verifying user roles, permissions, and attributes to ensure authorized access based on roles or a set of user/environmental attributes. · GitHub
https://gist.github.com/Bonny-kato/3f6ef42c68324a16a463c182d9d3b81a