**Input Fields AI**

**Building Reusable UI Components with AI Assistance in Next.js**

This lesson guides you through building foundational UI components for a Next.js web application, specifically focusing on creating a reusable header and input fields. We'll leverage AI assistance for generating boilerplate code and integrate these components effectively within the Next.js App Router structure.

Our starting point is a basic Next.js page displaying minimal content. We aim to replicate the header and form structure seen on sites like t-sender.com.

**Creating a Reusable Header Component**

**The Problem:** Our initial page.tsx might only contain specific page content and perhaps a connect button. However, most web applications require a consistent header across all pages, containing elements like the site title, navigation links (like a GitHub link), and user authentication status (like a wallet connect button). Repeating this header code on every single page is inefficient and hard to maintain.

**The Solution: React Components:** React allows us to break down our UI into reusable pieces called components. A component encapsulates its own HTML structure (using JSX/TSX), styling, and logic. By creating a dedicated Header component, we can define it once and use it wherever needed, ensuring consistency and making updates easier.

**Component Structure:** Standard practice dictates placing reusable components in a dedicated folder.

1. Create a components folder within your src directory: src/components.
2. Inside src/components, create a new file for our header: Header.tsx.

**Leveraging AI for Code Generation:** Generating standard UI elements like headers is often a task well-suited for AI code assistants (like DeepSeek or Claude). They can quickly produce boilerplate code based on your requirements.

**Prompting Strategy for the Header:**

To get useful code from an AI, provide clear context and requirements:

1. **Provide Existing Code:** Copy the relevant parts of your current page.tsx, especially imports and usage related to elements you want in the header (like ConnectButton).
2. **State the Goal:** Clearly ask the AI to create a reusable header component. Example: "Can you turn this into a reusable header component with:"
3. **List Specific Elements:** Enumerate the required features:
   * A title (e.g., 'tsender').
   * A link/button to a GitHub repository.
   * The existing ConnectButton positioned on the right side.
4. **Specify Libraries/Imports:** Guide the AI by listing allowed imports. This helps constrain the output and ensures compatibility with your project's dependencies. For example:

Can you only use the following imports?

import { ConnectButton } from "@rainbow-me/rainbowkit";

import { FaGithub } from "react-icons/fa";

import Image from "next/image"; // (If using Next/Image for logos)

1. **Install Dependencies:** If the AI uses libraries you don't have, install them. For the GitHub icon (FaGithub), you'll need react-icons:

pnpm add react-icons

**Reviewing and Integrating AI Code:**

* **Context Awareness:** Note that AI might infer the context (like JSX) but might not know specifics (like TypeScript/TSX) unless explicitly told. More context usually yields better results.
* **Critical Review:** *Always* treat AI-generated code as if it came from a junior developer. Review it carefully to ensure it's correct, secure, and aligns with your project's standards. Understand every line before using it.
* **Integration:**
  1. Copy the generated TSX code block for the Header functional component.
  2. Paste it into your src/components/Header.tsx file.
  3. Adjust any placeholders, such as updating the GitHub link URL to your specific repository.
  4. Observe the styling: The AI likely used utility classes (e.g., from Tailwind CSS if prompted or inferred) for layout (flex, justify-between, items-center) and appearance (p-4, bg-white, shadow-md).

**Using the Header Component:**

Initially, let's add the header to our main page to test it:

1. Open src/app/page.tsx.
2. Import the Header component:

import Header from "@/components/Header"; // Adjust path if necessary

*(Note: @/ implies configured path aliases in tsconfig.json)*

1. Use the component within the Home function, replacing any standalone elements now handled by the header:

export default function Home() {

return (

<div>

<Header />

{/\* Rest of the page content \*/}

Hi

</div>

);

}

1. Run your development server (pnpm run dev). You should see the header rendered above your page content.

**Making the Header Global with**layout.tsx**:**

The header shouldn't just be on the home page; it needs to be on *every* page. Next.js's App Router provides the layout.tsx file for this exact purpose.

1. **Concept:** The layout.tsx file acts as a template wrapping around your page content (props.children). Any components placed *outside* of {props.children} within the layout's return statement will render on every page governed by that layout.
2. **Action:**
   * Open src/app/layout.tsx.
   * Import the Header component:

import Header from "@/components/Header";

* + Place the <Header /> component inside the <body> tag, usually *before* the {props.children}:

<body>

<Providers> {/\* Assuming you have Providers wrapper \*/}

<Header /> {/\* Add the Header here \*/}

{props.children}

</Providers>

</body>

* + Go back to src/app/page.tsx and *remove* the <Header /> component and its import. The page should only contain its specific content now.

export default function Home() {

return (

<div>

Hi

</div>

);

}

1. **Result:** Refresh your browser. The header now appears persistently, managed by the layout, while page.tsx remains clean and focused on its unique content.

**Creating Reusable Input Fields for a Form**

With the header in place, let's focus on building the main interaction area – an airdrop form. This form will require several input fields (Token Address, Recipients, Amounts). To avoid repetition, we'll create a reusable InputField component.

**Planning the Form Structure:**

1. Create a component specifically for the form: src/components/AirdropForm.tsx.
2. Recognize the need for multiple similar input fields (some single-line, some multi-line). This justifies creating a generic, reusable input component.
3. Create a conventional ui subfolder for general-purpose UI elements: src/components/ui/.
4. Create the input field component file: src/components/ui/InputField.tsx.

**Using AI for the Input Field Component:**

Generating form elements is another good use case for AI.

**Prompting Strategy for the Input Field:**

1. **State the Goal & Tech:** "I'm looking to make an input field React component with tsx."
2. **Specify Props:** Define the necessary properties (props) the component should accept and their types/purpose:
   * label (string): Text label displayed above the input.
   * placeholder (string): Placeholder text inside the input.
   * value (string): The current value of the input (controlled component).
   * type (string): The HTML input type (e.g., "text", "number").
   * large (boolean): A flag to indicate whether to render a multi-line <textarea> instead of a single-line <input>.
   * onChange (function): A callback function to execute when the input's value changes.
3. **Specify Styling:** "I'm using tailwind for styling."

**Reviewing and Integrating the Input Field:**

1. **AI Output:** The AI should generate an InputField component, likely including:
   * A Props interface (e.g., InputFieldProps) defining the expected properties and their types (using ChangeEventHandler from React for onChange).
   * Conditional rendering logic: if (large) { return <textarea ... />; } else { return <input ... />; }.
   * Tailwind CSS classes for styling the label, input/textarea.
   * Correct wiring of props (label, placeholder, value, type, onChange) to the underlying HTML elements.
2. **Integration:** Copy the generated TSX code into src/components/ui/InputField.tsx. You might slightly adjust type definitions (like the onChange event type to explicitly handle both HTMLInputElement and HTMLTextAreaElement) based on preference or stricter typing needs, but the AI's version is often functional.

// Example type for onChange prop in InputFieldProps

onChange: (e: React.ChangeEvent<HTMLInputElement | HTMLTextAreaElement>) => void;

**Using the**InputField**in**AirdropForm**:**

Now, let's use our new InputField component within the AirdropForm.

1. Open src/components/AirdropForm.tsx.
2. **Add Client Directive:** Since this component will manage state (useState) and handle user events (onChange), mark it as a Client Component by adding "use client"; at the very top of the file.
3. **Import Dependencies:**

import { useState } from "react";

import InputField from "@/components/ui/InputField"; // Adjust path if needed

1. **Manage State:** Use the useState hook to manage the data for each input field. Start with the token address:

export default function AirdropForm() {

const [tokenAddress, setTokenAddress] = useState<string>("");

// State for other fields (recipients, amounts) will be added later

​

// ... rest of the component

}

1. **Render the Input Field:** Use the InputField component in the return statement, passing the required props:

export default function AirdropForm() {

const [tokenAddress, setTokenAddress] = useState<string>("");

​

return (

<div className="p-4 space-y-4"> {/\* Add some padding/spacing \*/}

<InputField

label="Token Address"

placeholder="Enter token contract address (e.g., 0x...)"

value={tokenAddress}

onChange={(e) => setTokenAddress(e.target.value)} // Update state on change

type="text" // Explicitly set type

// 'large' prop is omitted (defaults to false or not needed for single line)

/>

​

{/\* We will add InputFields for Recipients and Amounts here later \*/}

​

{/\* Button to trigger the airdrop will be added later \*/}

</div>

);

}

​

1. **Integrate**AirdropForm**into the Page:** Finally, use the AirdropForm component in your main page (src/app/page.tsx or another relevant page).

// In src/app/page.tsx

import AirdropForm from "@/components/AirdropForm";

​

export default function Home() {

return (

<main className="p-4"> {/\* Add some padding for content area \*/}

{/\* The Header is already handled by layout.tsx \*/}

<AirdropForm />

</main>

);

}

**Result:** When you run your application, you should now see the global header and the beginning of your airdrop form, featuring the "Token Address" input field rendered by your reusable InputField component. The input field will update its state within the AirdropForm as you type.

This lesson demonstrated how to create modular, reusable UI components in React/Next.js, utilize layout.tsx for global elements, and effectively employ AI assistance for generating boilerplate code, significantly speeding up the UI development process. Remember to always review and understand AI-generated code before integrating it.