**Connecting A Wallet With Rainbowkit**

**Upgrading Your dApp's Wallet Connection**

We begin with a minimal Next.js application displaying a simple "Hi". Our ultimate goal is to implement functions like airdropERC20, but a robust wallet connection mechanism is a prerequisite.

Looking back at simpler implementations, like those built with plain HTML and TypeScript, reveals several limitations. For instance, a basic "Connect" button might not maintain its state; refreshing the page often resets the button text even if the user's wallet (e.g., MetaMask) remains connected. Furthermore, if the user disconnects directly within their wallet extension, the webpage typically remains unaware, leading to a disconnected user experience (UX).

To build a professional decentralized application (dApp), we need a more sophisticated approach. Consider platforms like Aave: they feature a persistent header with a "Connect wallet" button. Clicking this button opens a clean modal presenting various wallet options (MetaMask, WalletConnect QR code, etc.). Once connected, the button intelligently displays the user's truncated address and balance, and this state persists across page refreshes. This polished UX is what we aim to achieve.

Instead of manually building the complex logic for state management, UI components, and browser storage persistence, we'll leverage battle-tested libraries:

* wagmi: A powerful React Hooks library for Ethereum interactions. Built on viem, it simplifies tasks like reading/writing blockchain data and managing wallet connections.
* RainbowKit: Built specifically for wallet connections on top of wagmi, RainbowKit provides pre-built React components (like the connect button and modal) for an excellent developer and user experience. It's often described as "The best way to connect a wallet."

We choose RainbowKit for its ease of use, polished UI, and seamless integration with wagmi.

**Installing Wagmi and RainbowKit**

Before installing new dependencies, ensure your development server is stopped (usually Ctrl+C in the terminal where pnpm run dev or npm run dev is running).

First, add RainbowKit to your project using your preferred package manager. We'll use pnpm:

pnpm add @rainbow-me/rainbowkit@latest

*Note:* The original context used version 2.2.4. If you encounter issues with the latest version or wish to follow along exactly, you can install a specific version: pnpm add @rainbow-me/rainbowkit@2.2.4.

RainbowKit relies on several other packages, known as peer dependencies. Install them using the command recommended in the RainbowKit documentation (adapted for pnpm):

pnpm add wagmi viem@2.x @tanstack/react-query

This installs wagmi itself, viem (version 2 specifically), and @tanstack/react-query, which wagmi and RainbowKit often use internally for data fetching and caching.

**Configuring Wagmi and RainbowKit**

To configure these libraries, we'll create a dedicated file.

1. Create a new file: src/rainbowKitConfig.tsx (place it in src, but outside the app directory).
2. Add the "use client"; directive at the very top. Wallet connections are inherently client-side operations happening in the user's browser. This directive is crucial in Next.js App Router projects.

Now, populate the file with the core configuration using getDefaultConfig from RainbowKit:

"use client"; // Essential for client-side logic

​

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

import { anvil, zksync, mainnet } from "wagmi/chains"; // Import your desired chains

​

// Retrieve the WalletConnect Project ID from environment variables

const walletConnectProjectId = process.env.NEXT\_PUBLIC\_WALLETCONNECT\_PROJECT\_ID;

​

// Basic error handling for missing Project ID

if (!walletConnectProjectId) {

throw new Error("Error: NEXT\_PUBLIC\_WALLETCONNECT\_PROJECT\_ID is not defined. Please set it in your .env.local file");

}

​

// Define the configuration object

const config = getDefaultConfig({

appName: "TSender", // Your dApp's name, shown in wallet prompts

projectId: walletConnectProjectId, // WalletConnect Cloud Project ID

chains: [anvil, zksync, mainnet], // Array of chains your dApp supports

ssr: false, // Set to false for static sites or if not heavily using SSR with wagmi

});

​

export default config; // Export for use in Providers

Let's break down the getDefaultConfig parameters:

* appName: The name of your application. This will be displayed to users when they connect their wallet.
* projectId: **This is essential for enabling WalletConnect**. WalletConnect allows users to connect mobile wallets or other wallets without browser extensions using QR codes. You must obtain a free Project ID from [WalletConnect Cloud](https://cloud.walletconnect.com/" \t "_blank). Sign up, create a new project, and copy the generated ID.
* chains: An array specifying the blockchain networks your dApp will interact with. We import chain definitions (like anvil, zksync, mainnet) from wagmi/chains. Add all networks you intend to support.
* ssr: Server-Side Rendering. For this setup aiming for a primarily client-side experience, false is appropriate. Set this to true only if you plan to integrate wagmi deeply with Next.js server-side rendering features.

**Handling the WalletConnect Project ID Securely:**

The projectId is public but shouldn't be hardcoded directly into your source code. We use environment variables:

1. Create a file named .env.local in the root of your project (if it doesn't exist).
2. Add your WalletConnect Project ID to this file:

# .env.local

NEXT\_PUBLIC\_WALLETCONNECT\_PROJECT\_ID=YOUR\_WALLETCONNECT\_PROJECT\_ID\_HERE

**Important:** In Next.js, environment variables intended for browser access **must** be prefixed with NEXT\_PUBLIC\_.

1. Ensure your .gitignore file includes .env.local to prevent accidentally committing it.
2. The configuration code above (process.env.NEXT\_PUBLIC\_WALLETCONNECT\_PROJECT\_ID) reads this value. The added check ensures the application fails fast during development if the ID is missing. We avoid the non-null assertion (!) in favor of explicit checking for better safety.

**Setting Up Global Providers**

wagmi and RainbowKit use React Context to make wallet state and functions available throughout your application. We need to wrap our entire app with their respective Provider components.

1. Create a new file: src/app/providers.tsx.
2. Add the "use client"; directive, as these providers manage client-side state.

Add the following code to set up the providers:

"use client";

​

import \* as React from 'react';

import { useState, useEffect } from 'react';

import { RainbowKitProvider } from '@rainbow-me/rainbowkit';

import { WagmiProvider } from 'wagmi';

import { QueryClientProvider, QueryClient } from "@tanstack/react-query";

import config from '@/rainbowKitConfig'; // Import the configuration we created

​

// Import RainbowKit CSS for default styling

import '@rainbow-me/rainbowkit/styles.css';

​

// Create a single QueryClient instance

const queryClient = new QueryClient();

​

// Define the Providers component

export function Providers({ children }: { children: React.ReactNode }) {

// Hydration safety check: ensure component mounts on client before rendering children

const [mounted, setMounted] = useState(false);

useEffect(() => setMounted(true), []);

​

return (

<WagmiProvider config={config}>

<QueryClientProvider client={queryClient}>

<RainbowKitProvider>

{/\* Only render children after client-side mounting \*/}

{mounted ? children : null}

</RainbowKitProvider>

</QueryClientProvider>

</WagmiProvider>

);

}

Here's what's happening:

* WagmiProvider: Makes wagmi hooks and state available. It requires the config object we defined earlier.
* QueryClientProvider: Provides the context for @tanstack/react-query. A QueryClient instance is created and passed.
* RainbowKitProvider: Provides the context for RainbowKit components and sits *inside* the other two providers.
* children: Standard React pattern to allow this component to wrap other parts of your application.
* **Hydration Safety (**mounted**state):** The useState and useEffect hooks ensure that the children (your actual app UI, including RainbowKit components that need browser APIs) are only rendered *after* the component has successfully mounted on the client-side. This prevents hydration mismatch errors common in server-rendered frameworks like Next.js when dealing with client-only logic.
* **CSS Import:** import '@rainbow-me/rainbowkit/styles.css'; is crucial. It imports the default stylesheets necessary for the ConnectButton and modal to look correct. Without it, they will appear unstyled.

**Integrating Providers into Your Layout**

To make the wallet context available everywhere, we wrap the application's root layout with our Providers component.

Modify your src/app/layout.tsx file:

import type { Metadata } from "next";

import "./globals.css";

import { Providers } from "./providers"; // Import the Providers component

​

export const metadata: Metadata = {

title: "TSender",

description: "A simple ERC20 token sender dApp", // Example description

};

​

export default function RootLayout({

children,

}: Readonly<{

children: React.ReactNode;

}>) {

return (

<html lang="en">

<body>

{/\* Wrap the entire body content with Providers \*/}

<Providers>

{children}

</Providers>

</body>

</html>

);

}

By wrapping children within <Providers>, every page and component in your application will now have access to the context provided by wagmi, RainbowKit, and react-query.

**Adding the Connect Button to Your UI**

With the setup complete, adding the actual connection UI is remarkably simple. RainbowKit provides a pre-built ConnectButton component.

Go to the page where you want the button to appear, for example, src/app/page.tsx:

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

​

export default function Home() {

return (

<main style={{ padding: '2rem' }}> {/\* Added some padding for layout \*/}

<div style={{ display: 'flex', justifyContent: 'flex-end', marginBottom: '1rem' }}>

<ConnectButton /> {/\* Add the button here \*/}

</div>

<h1>Hi</h1>

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

</main>

);

}

Simply import ConnectButton from @rainbow-me/rainbowkit and render it (<ConnectButton />). This single component handles:

* Displaying "Connect Wallet" when disconnected.
* Opening the RainbowKit modal with wallet options when clicked.
* Guiding the user through the wallet's connection approval process.
* Displaying the connected network icon and name.
* Showing the user's balance and truncated address when connected.
* Providing options to switch networks or disconnect via a dropdown/modal.

**Testing the Wallet Connection Flow**

Run your development server (pnpm run dev or npm run dev) and open your application in the browser.

1. **Initial State:** You should see the ConnectButton rendered (likely in the top right, depending on your styling).
2. **Connect:** Click the button. The RainbowKit modal appears, listing available wallets (e.g., MetaMask, Coinbase Wallet if installed) and the WalletConnect option.
3. **Wallet Approval:** Select your preferred wallet (e.g., MetaMask). Your wallet extension will prompt you to authorize the connection to your dApp. Approve it.
4. **Connected State:** The button's appearance changes dramatically. It now shows:
   * The icon and name of the currently connected network (e.g., Anvil, Mainnet).
   * The user's balance of the native currency on that network (e.g., "0 ETH").
   * The user's truncated wallet address (e.g., "0x12...AbCd").
5. **Interaction:** Clicking the button now opens a small modal showing the full address, options to copy the address, change wallets, or disconnect.
6. **Network Switching:** If your wallet is connected to a network *not* listed in your rainbowKitConfig.tsx chains array (e.g., Sepolia), the button will display "Wrong network". Clicking it prompts you to switch to one of the supported networks. Approving the switch in your wallet updates both the wallet and the button UI.
7. **Persistence:** Refresh the page. You'll notice the button retains its connected state, reflecting the persistent UX we aimed for.

**Key Concepts Recap**

By integrating wagmi and RainbowKit, we've significantly upgraded our dApp's connection mechanism. Key concepts involved include:

* **Libraries:** Leveraging wagmi for core blockchain interaction hooks and RainbowKit for a streamlined wallet connection UI/UX.
* **React Context/Providers:** Using WagmiProvider, QueryClientProvider, and RainbowKitProvider to make global state (like connection status, address, network) and functionality accessible throughout the component tree.
* **Configuration:** Defining essential parameters like appName, supported chains, and the crucial WalletConnect projectId in a central config file (rainbowKitConfig.tsx).
* **Client-Side Rendering:** Recognizing that wallet interactions happen in the browser and using the "use client"; directive in Next.js, along with techniques (like the mounted state) to prevent hydration errors.
* **Environment Variables:** Securely managing configuration like the projectId using .env.local and the NEXT\_PUBLIC\_ prefix for browser exposure in Next.js.
* **WalletConnect:** Understanding its role in connecting non-extension wallets via QR codes and the necessity of a projectId.
* **User Experience (UX):** Achieving a persistent, informative, and professional connection flow similar to established dApps, vastly improving upon basic implementations.