Project Report: Nouniverse Portal

A Modern, Responsive Web Portal for the Nouns DAO Ecosystem

Abstract

The NounsNet Portal is a comprehensive decentralized application (dApp) designed to serve as the primary, user-friendly interface for the Nouns DAO ecosystem. The project's core objective is to provide a unified and seamless platform for community members to engage with the DAO's key activities. It integrates a real-time NFT auction system, a robust governance portal for proposal creation and voting, and secure wallet management through MetaMask. Built on a cutting-edge technology stack featuring React 18, TypeScript, and Ethers.js, the portal is designed for performance, security, and scalability. By leveraging modern web technologies and direct blockchain integration, the NounsNet Portal aims to lower the barrier to entry for participation and empower users to "Own the Pixel and Steer the Future" of the Nouns organization.

Introduction

Nouns DAO represents a novel approach to decentralized governance and community building, centered around the daily auction of unique, algorithmically generated Noun NFTs. Each Noun grants its owner membership and one vote in the Nouns DAO treasury. As this ecosystem has grown, the need for a centralized, modern, and highly functional hub for all community interactions has become paramount.

The NounsNet Portal project was initiated to address this need. It replaces fragmented user experiences with a single, cohesive dApp that consolidates the most critical functions of the DAO. The portal's mission is to provide an intuitive, responsive, and powerful interface that facilitates:

- Real-time participation in the daily Noun auctions.
- Active engagement in DAO governance through proposal submission and voting.
- Secure and straightforward management of user assets and identity via wallet integration.

This document provides a detailed overview of the project, covering its multi-layered architecture, the specific technologies and methodologies used in its implementation, and a comprehensive breakdown of its core functionalities.

NounsNet Portal Architecture

The NounsNet Portal is built upon a robust, layered architecture designed for modularity, scalability, and a clear separation of concerns. Each layer handles a distinct aspect of the application, from user interface rendering to direct blockchain communication.

The architecture follows a four-layer design pattern that ensures maintainability, performance, and seamless user experience:
See the images provided below – Nouniverse_Documentation

Architecture Layers:

1. Frontend Layer (Presentation)

The user-facing layer responsible for rendering the UI and handling user interactions

Core Technologies:

- **React 18 & TypeScript**: Modern React framework utilizing Concurrent Rendering features for enhanced performance. TypeScript provides compile-time type safety, reducing runtime errors and improving developer productivity.
- **Tailwind CSS**: Utility-first CSS framework enabling rapid development of a custom, responsive design system that maintains consistency with the Nouns brand identity.

Key Features:

- Component-based architecture for reusability
- Server-side rendering capabilities
- Responsive design optimized for all devices
- Accessibility compliance (WCAG 2.1)

2. State Management Layer

Manages application data flow and state consistency across components

Core Technologies:

- **React Hooks & Context API**: Custom hooks combined with React's Context API manage both local component state and global application state (user authentication, network settings, UI preferences).
- **LocalStorage Integration**: Browser storage for persisting user preferences, wallet connection state, and session data to enhance user experience across visits.

State Management Patterns:

- Centralized state for critical data (user session, blockchain connection)
- Local state for component-specific data
- Optimistic updates for improved UX
- Error boundary handling for graceful error recovery

3. Blockchain Integration Layer (Data Access)

Bridge between frontend application and blockchain infrastructure

Core Technologies:

- **Ethers.js v6**: Comprehensive Ethereum library handling all blockchain interactions including smart contract calls, transaction management, and event parsing.
- **MetaMask Integration**: Primary wallet provider enabling secure wallet connections, transaction signing, and identity management without exposing private keys.
- **The Graph Protocol**: Decentralized indexing service for efficient querying of complex blockchain data, avoiding slow direct blockchain calls for historical data.

Integration Features:

- Multi-wallet support (MetaMask, WalletConnect, Coinbase Wallet)
- Real-time event monitoring
- Transaction status tracking
- Gas optimization strategies

4. External Services Layer (Infrastructure)

External infrastructure and APIs supporting the application ecosystem

Core Services:

- **Ethereum Mainnet**: Primary blockchain network hosting Nouns DAO smart contracts and transaction records. Configurable for alternative networks and custom RPC endpoints.
- **IPFS Network**: Decentralized storage for Noun metadata and assets, ensuring censorship resistance and permanent availability.
- **Nouns API**: RESTful API providing structured access to Noun data, proposal information, and community metrics.

Additional Services:

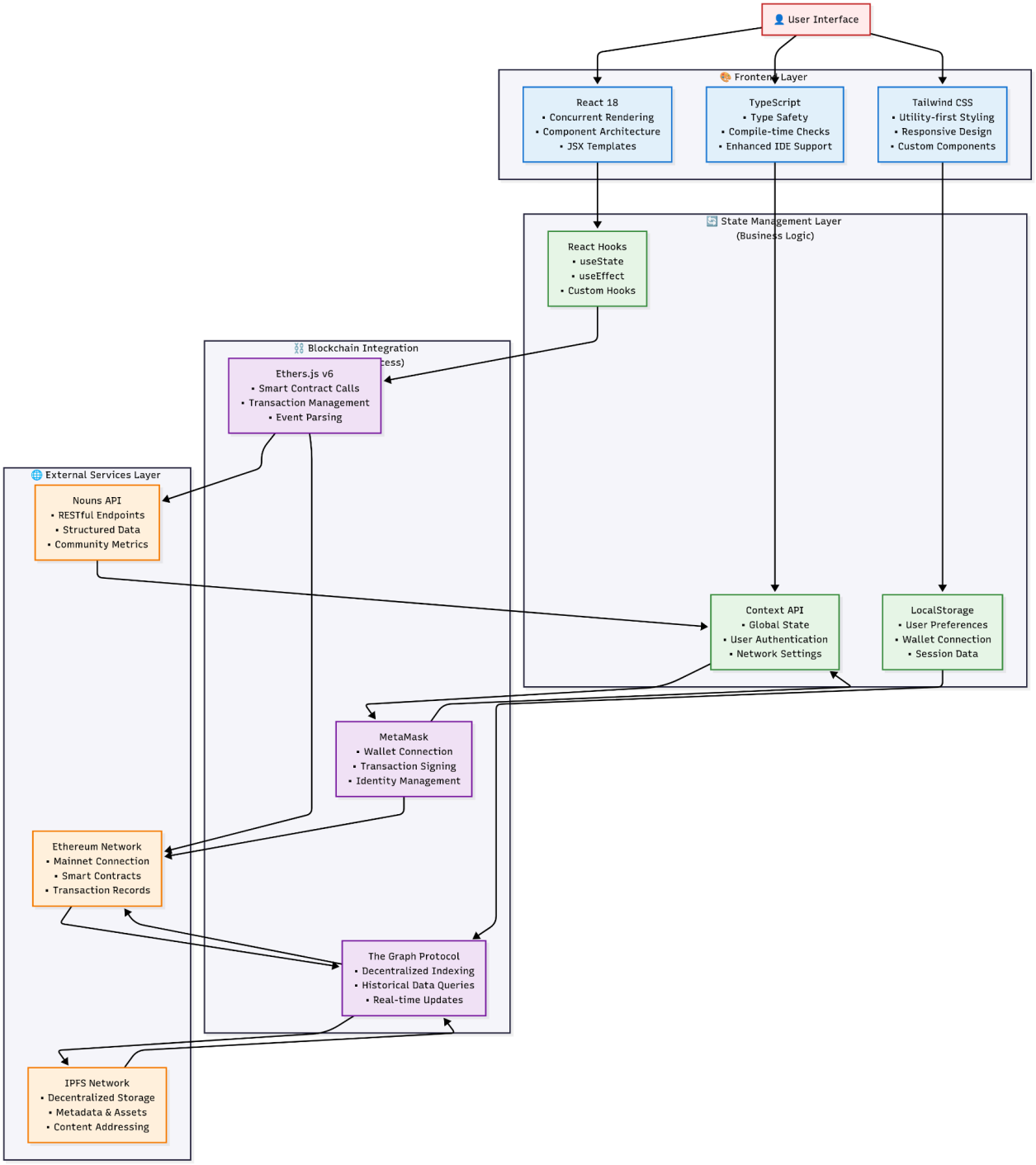
- CDN for static asset delivery
- Analytics and monitoring services
- Error tracking and performance monitoring
- Rate limiting and DDoS protection

Data Flow

1. **User Interaction**: User interacts with React components in the Frontend Layer
2. **State Updates**: Actions trigger state changes managed by the State Management Layer
3. **Blockchain Queries**: State changes initiate blockchain calls through the Integration Layer
4. **External Data**: External services provide additional context and metadata
5. **UI Updates**: Results flow back through the layers to update the user interface

This architecture ensures a robust, scalable, and maintainable application that can evolve with the growing needs of the Nouns DAO community.

Architecture of Nouniverse



Implementation:

To begin with the implementation part, First we have to address the file structure of Nouniverse to get the basic understanding

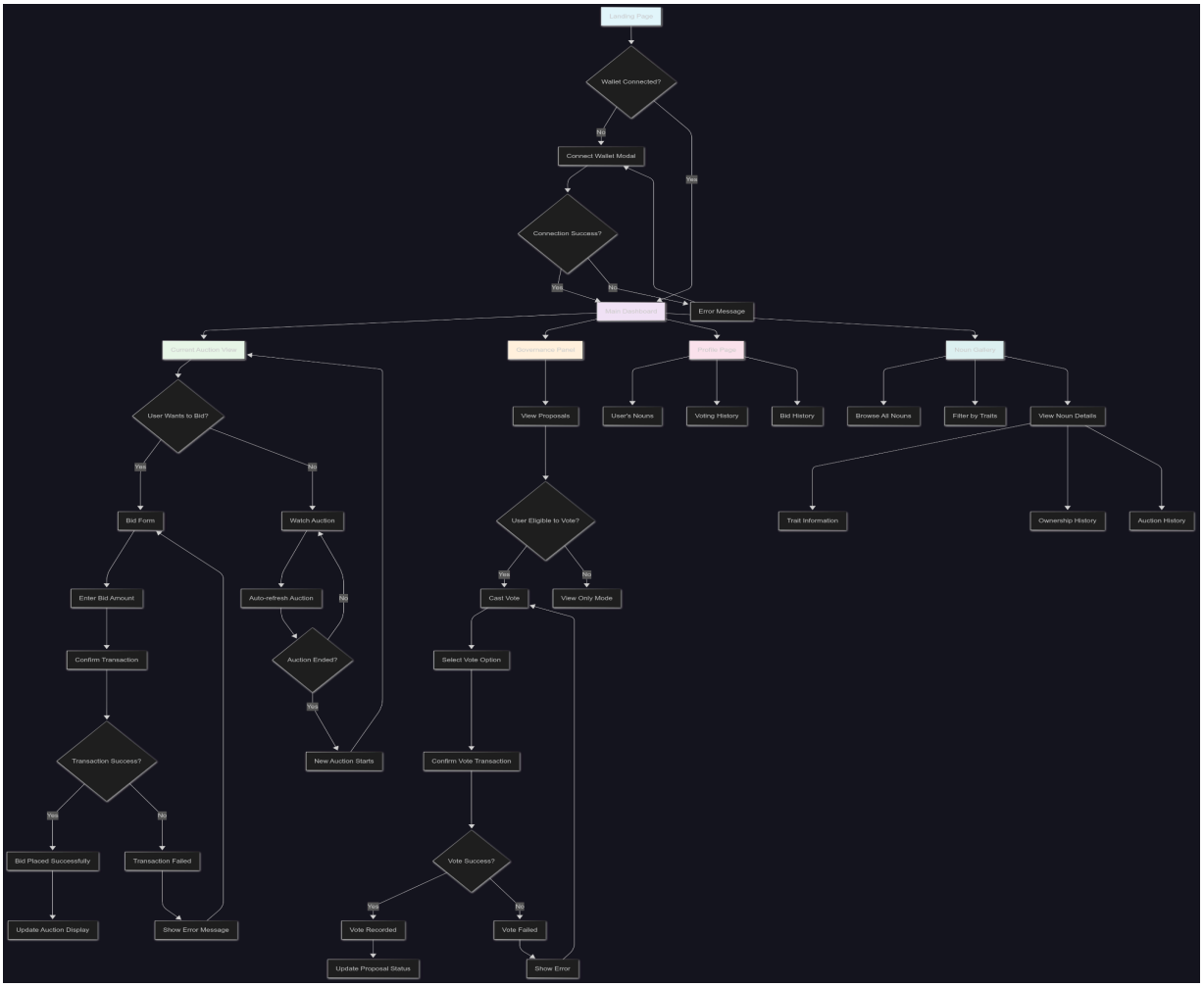
```
Nouniverse/
├── 📂 public/                # Static assets and metadata
│   ├── vite.svg              # Default Vite favicon
│   └── index.html            # Main HTML template
├── 📂 src/                   # Source code directory
│   ├── 📂 assets/            # Static assets and images
│   │   ├── head-earth.png    # Favicon and logo assets
│   │   ├── noun-*.png        # Noun character images
│   │   └── 📂 Noggles/       # Noggles glasses collection
│   │       ├── 0-glasses-hip-rose.png
│   │       ├── 1-glasses-square-black.svg
│   │       └── ... (40+ variations)
│   ├── 📂 components/        # Reusable React components
│   │   ├── Navigation.tsx    # Header navigation with mobile menu
│   │   ├── NounDisplay.tsx   # 3D Noun showcase component
│   │   └── ThemeToggle.tsx
│   ├── 📂 pages/             # Main application pages
│   │   ├── HomePage.tsx      # Landing page with hero section
│   │   ├── AuctionPage.tsx   # Live auction interface
│   │   ├── ProposalsPage.tsx # DAO governance portal
│   │   └── SettingsPage.tsx  # User configuration panel
│   ├── 📂 hooks/             # Custom React hooks
│   │   ├── useAuctionBids.ts # Auction data management
│   │   ├── useNounData.ts    # Noun metadata fetching
│   │   ├── useNounTraits.ts  # Trait data processing
│   │   ├── useGraph.ts       # Graph API integration
│   │   └── NotFoundPage
│   ├── 📂 lib/               # Utility libraries
│   │   ├── ether.ts          # Ethereum provider setup
│   │   └── graph.ts          # GraphQL query handling
│   │   └── contracts.ts # GraphQL query handling
│   ├── 📂 config/            # Configuration files
│   │   └── endpoint.tsx      # RPC and API endpoint management
│   ├── App.tsx               # Main application component
│   ├── main.tsx              # Application entry point
│   └── index.css             # Global styles and Tailwind imports
├── 📂 configuration files    # Build and development configuration
│   ├── package.json          # Dependencies and scripts
│   ├── tsconfig.json         # TypeScript configuration
│   ├── tailwind.config.js    # Tailwind CSS customization
│   ├── vite.config.ts        # Vite build configuration
│   ├── eslint.config.js      # ESLint rules and settings
│   └── postcss.config.js     # PostCSS plugin configuration
└── README.md                 # This documentation file
```

Now the implementation of Nouniverse could be understood by the activity diagram provided below and the content for better understanding

Application Workflow Overview

The diagram on the left illustrates the core user flows and system logic of our platform, outlining the interactions between users, the blockchain wallet, and the decentralized auction and voting mechanisms. Below is a detailed explanation of each major section of the flow:

Wallet Connection Flow
Main Dashboard Navigation Auction Participation Flow Governance Voting Flow



- **Understanding the Functionalities of the Folders**

`Components Folder`

`Navigation.tsx` - Main Navigation Component

- MetaMask Integration: Connects to user wallets via MetaMask
- Responsive Design: Mobile-friendly hamburger menu
- Navigation Items: Links to all major sections
- Wallet Status: Shows connected wallet address (shortened format)
- Logo Display: NounsNet branding

`NounDisplay.tsx` - Interactive Noun Visualization

- 3D Model: Embeds Sketchfab 3D model of a specific Noun (#4521)
- Animations: Floating particle effects using Framer Motion
- Responsive: Adapts to different screen sizes
- Visual Effects: Gradient backgrounds and backdrop blur effects

⚙️ `Config Folder`

`endpoint.tsx` - Configuration & Settings Management

- Environment Variables: Manages API keys and RPC endpoints
- Context Provider: Global settings state management
- Local Storage: Persists user settings across sessions
- Default Endpoints: Fallback URLs for The Graph API and Ethereum RPC
- Settings Hook: useSettings() for accessing configuration throughout the app

🪝 Hooks Folder - Custom React Hooks for Data Fetching

useGraph.ts - GraphQL Data Fetching

- Proposals Query: Fetches latest governance proposals
- GraphQL Integration: Uses The Graph Protocol for decentralized data
- State Management: Loading, error, and data states
- Customizable: Accepts parameters for query customization

useNounData.ts - Current Auction Data

- Blockchain Integration: Connects to Ethereum via ethers.js
- Auction Contract: Fetches current auction information
- Real-time Data: Current Noun being auctioned, bid amount, timing
- Error Handling: Comprehensive error management

useNounTraits.ts - Noun Metadata & Traits

- Trait Fetching: Gets specific Noun characteristics (background, body, accessories, head, glasses)
- GraphQL Queries: Uses subgraph for trait data
- Dynamic Loading: Fetches data based on Noun ID

useAuctionBids.ts - Bid History & Analytics

- Bid History: Fetches recent auction bids
- User Recognition: Identifies known community members
- Token Balance: Shows if bidders own other Nouns
- Rich Data: Includes bidder details, amounts, timestamps

Lib Folder - Core Utilities

ether.ts - Ethereum Blockchain Integration

- Provider Setup: Configures Ethereum RPC connections
- Contract Interaction: Nouns Auction contract interface
- Environment Support: Uses environment variables for addresses
- ethers.js: Modern Ethereum library integration

graph.ts - GraphQL Query Engine

- The Graph Protocol: Decentralized data indexing
- Query Execution: Generic GraphQL query function
- Error Handling: Comprehensive error management
- Flexible Endpoints: Supports multiple subgraph endpoints

Pages Folder - Main Application Views

HomePage.tsx - Landing Page

- Hero Section: Welcome message and branding
- Animated Noggles: Rotating display of Noun glasses
- Quick Navigation: Links to main sections
- Visual Appeal: Gradient backgrounds and animations
- Responsive Design: Mobile-first approach

AuctionPage.tsx - Live Auction Interface

- Current Auction: Real-time auction data display
- Bid History: Recent bids with user information
- Noun Traits: Visual representation of current Noun's characteristics
- Wallet Integration: Connect and bid functionality
- Timer: Auction countdown display
- Trait Analysis: Uses @nouns/assets for trait naming

`ProposalsPage.tsx - Governance Dashboard`

- Proposal Listing: All governance proposals
- Voting Interface: Prepared for voting functionality
- Status Tracking: Proposal states (active, executed, etc.)
- Detailed View: Proposal descriptions and actions
- Community Interaction: Voting history and statistics

`SettingsPage.tsx - User Configuration`

- Profile Management: User information and preferences
- Wallet Settings: Connected wallet management
- API Configuration: Custom RPC and Graph endpoints
- Notification Preferences: Alert settings
- Theme Options: UI customization
- Security Settings: Privacy and security controls

Smart Contract Integration Implementation

This document outlines the complete blockchain functionality implementation for bidding, voting, and proposal creation in the Nouniverse application.

📋 Contract Addresses

All contracts are deployed on Ethereum Mainnet:

- Nouns Auction House: 0x830BD73E4184ceF73443C15111a1DF14e495C706
- Nouns DAO Governor: 0x6f3E6272A167e8AcCb32072d08E0957F9c79223d

⚡ Key Features Implemented

🏷️ Auction Bidding

// Real contract interaction for placing bids

```
// Real contract interaction for placing bids
const txHash = await placeBid(nounId, bidAmount);
```

- Minimum Bid Validation: Ensures bids meet minimum requirements
- Balance Checks: Validates sufficient ETH balance
- Gas Optimization: Proper gas limit settings
- Gas Optimization: Proper gas limit settings

🗳️ Governance Voting

// Cast vote with optional reasoning

```
// Cast vote with optional reasoning
const txHash = await castVote(proposalId, support, reason);
```

- Voting Power Validation: Checks if user owns Nouns
- Duplicate Vote Prevention: Prevents multiple votes on same proposal
- Reasoning Support: Optional vote reasoning for transparency
- Vote Types: Support for For(1), Against(0), Abstain(2)

📝 Proposal Creation

// Create new governance proposal

```
// Create new governance proposal
const txHash = await createProposal(targets, values, signatures, calldatas, description);
```

- Multi-Action Support: Proposals can contain multiple contract calls
- Ownership Validation: Ensures proposer owns at least one Noun
- Rich Descriptions: Markdown-formatted proposal descriptions
- ction Validation: Validates all proposal actions before submission

🔧 Technical Implementation Details

// Auction Contract

```
Smart Contract ABIs

const AUCTION_ABI = [

    "function createBid(uint256 nounId) payable",

    "function auction() view returns (uint256 nounId, uint256
amount, uint256 startTime, uint256 endTime, address bidder, bool
settled)"

];
```

// Governor Contract

```
const GOVERNOR_ABI = [

    "function propose(address[] targets, uint256[] values,
string[] signatures, bytes[] calldatas, string description)
returns (uint256 proposalId)",

    "function castVote(uint256 proposalId, uint8 support) returns
(uint256 weight)",

    "function castVoteWithReason(uint256 proposalId, uint8
support, string reason) returns (uint256 weight)"

];
```

Error Handling

- Transaction Rejection: Handles user-rejected transactions
- Insufficient Funds: Validates balances before transactions
- Contract Reverts: Displays meaningful error messages
- Network Issues: Graceful handling of connectivity problems

🚀 Deploying the Nouns Frontend Using PinMe

For deploying my Nouns frontend on a decentralized network, I used PinMe — a free and simple tool that uploads static sites to IPFS and automatically assigns a decentralized ENS subdomain.

🔧 Why PinMe?

- Decentralized Hosting: My site is hosted on IPFS, removing any reliance on centralized servers
- Free ENS Subdomain: I instantly get a .pinit.eth.limo ENS subdomain without purchasing one
- Easy Deployment: Just one CLI command to deploy.

📂 Step 1: Build the Frontend

First, I built my React app (or any frontend framework) to generate the dist/ folder:

```
npm run build
```

This command creates the optimized production files inside the dist/ directory.

📤 Step 2: Upload to IPFS with PinMe

I installed PinMe globally to simplify usage (you can also use npx):

```
npm install -g pinme
```

Then, I deployed the dist/ folder to IPFS:

```
pinme upload dist/
```

✔️ After a few seconds, PinMe successfully uploaded my project and gave me a unique link

---

END