Token System Architecture Documentation

Overview

The token system is a comprehensive framework for creating, configuring, managing, and deploying various types of blockchain tokens within the application. The system supports multiple token standards (ERC20, ERC721, ERC1155, ERC1400, ERC3525, ERC4626) and provides both simple and detailed configuration interfaces for each standard.

Directory Structure

The token system is organized into the following directory structure:

```
src/components/tokens/
├── deployment/            # Token deployment components
├── essential/             # Essential configuration components
├── standards/             # Token standard components
│   └── ERC1400/           # ERC1400 specialized components
├── templates/             # Token template components
├── utils/                 # Utility functions
├── *.tsx                  # Core token components
├── config.ts              # Configuration settings
└── index.ts               # Main exports
```

Core Components and Relationships

Key Components

1. **TokenBuilder.tsx**
    - Main component for creating and editing tokens
    - Manages token creation, editing, and template selection
    - Handles form state and database interactions
    - Routes between token list, templates, and builder interfaces
2. **TokenList.tsx**
    - Displays a list of existing tokens and templates
    - Provides actions for editing, deleting, and cloning tokens
    - Filters and sorts tokens by various criteria
3. **TokenDetail.tsx**
    - Shows detailed view of a specific token
    - Displays token properties, deployment status, and history

4. **TokenManagement.tsx**
    - Provides administration interface for tokens
    - Handles token status updates and lifecycle management
5. **TokenForm.tsx**
    - Generic form component for token data entry
    - Used by multiple components for consistent UI
6. **index.ts**
    - Central export point for all token components
    - Conditionally exports simple or detailed configuration components based on `config.ts` settings
    - Controls which token standards are enabled

Configuration System

config.ts controls two key aspects of the token system:

```typescript
// Set to true to use SimpleConfig components, false to use DetailedConfig components
export const USE_SIMPLE_CONFIG = false;

// Configure which token standards to enable
export const ENABLED_STANDARDS = {
  ERC20: true,
  ERC721: true,
  ERC1155: true,
  ERC1400: true,
  ERC3525: true,
  ERC4626: true
};
```

This configuration allows:

- Switching between simple and detailed configuration components
- Enabling/disabling specific token standards throughout the application

Token Standards

The system supports multiple token standards, each with its own configuration component:

1. **ERC20 (Fungible Tokens)**
    - Standard for fungible tokens (currencies, utility tokens)
    - Properties: name, symbol, decimals, supply, minting, burning, pausing

2. **ERC721 (Non-Fungible Tokens)**
   - Standard for non-fungible tokens (unique assets, collectibles)
   - Properties: name, symbol, baseURI, metadata storage, minting methods
3. **ERC1155 (Multi-Token Standard)**
   - Standard for both fungible and non-fungible tokens
   - Properties: name, symbol, baseURI, batch operations, token types
4. **ERC1400 (Security Tokens)**
   - Standard for security tokens with compliance features
   - Properties: name, symbol, decimals, partitions, controllers, transfer restrictions
   - Specialized configurations for different security types:
     - Equity (EquityConfig.tsx)
     - Debt (DebtConfig.tsx)
     - Derivative (DerivativeConfig.tsx)
     - Fund (FundConfig.tsx)
     - REIT (REITConfig.tsx)
5. **ERC3525 (Semi-Fungible Tokens)**
   - Standard for semi-fungible tokens with value slots
   - Properties: name, symbol, decimals, slots, slot transferability
6. **ERC4626 (Tokenized Vaults)**
   - Standard for tokenized yield-bearing vaults
   - Properties: name, symbol, decimals, asset address, fees, deposit limits
   - Features: NAV panel, vault simulator

Essential Configuration System

The `essential` directory contains components that provide different configuration experiences:

1. **Simple Configuration**
   - Minimal UI with only essential options
   - Reduced complexity for quick token creation
   - Example: ERC20SimpleConfig.tsx
2. **Detailed Configuration**
   - Advanced UI with all possible options
   - Complete customization capabilities
   - Example: ERC20DetailedConfig.tsx

3. **Essential Configuration**
    - Bridge components that conditionally render simple or detailed configuration
    - Based on `USE_SIMPLE_CONFIG` in config.ts
    - Example: ERC20EssentialConfig.tsx

This system allows for different user experiences based on user expertise or project requirements.

Type System

The type system is defined in `essential/types.ts` and provides interfaces for:

1. **Base Configuration Types**
    - BaseTokenConfig - Common properties for all tokens
    - Specific configurations for each standard (ERC20Config, ERC721Config, etc.)
2. **Component Props**
    - Props interfaces for each configuration component
    - TokenFormProps - Common props for form components
3. **Complex Types**
    - ERC1155TokenType - For ERC1155 token types
    - TokenConfig - Union type of all possible token configs

Deployment System

The `deployment` directory contains components for deploying tokens to blockchain networks:

1. **TokenDeploymentWizard.tsx**
    - Multi-step wizard for token deployment
    - Handles token configuration and deployment stages
    - Supports all token standards
2. **DeploymentPanel.tsx**
    - Handles the actual deployment process
    - Connects to blockchain networks
    - Manages transaction signing and deployment status

The deployment workflow:

1. Configure token properties
2. Select network and deployment options
3. Review gas costs and contract details
4. Deploy and track transaction status

5. Store deployment results in database

Template System

The `templates` directory contains components and utilities for token templates:

1. **TokenTemplateBuilder.tsx**
    - Interface for creating and editing token templates
    - Allows saving configurations for reuse
2. **TokenTemplateSelector.tsx**
    - UI for selecting and applying templates
    - Filters templates by category and standard
3. **tokenTemplate.ts**
    - Defines template data structures
    - Provides default token form state
4. **Other Template Files**
    - buildingBlocks.ts - Reusable token features
    - productCategories.ts - Template categorization
    - tokenStandards.ts - Standard definitions
    - contractPreview.ts - Generate preview of contract code

Templates enable users to:

- Save common token configurations
- Create standardized token designs
- Quickly apply predefined settings
- Share token designs across projects

Data Flow

1. **Token Creation Flow**
    - User selects token standard in TokenBuilder
    - Configures token using standard-specific configuration component
    - Saves token to database
    - Optionally deploys token to blockchain
2. **Template Usage Flow**
    - User selects template in TokenTemplateSelector
    - Template populates form fields in TokenBuilder

- User modifies template as needed
- Creates token based on template

3. **Token Deployment Flow**
    - User opens TokenDeploymentWizard
    - Configures deployment settings
    - Confirms transaction details
    - Deploys token and receives confirmation

Database Integration

The token system integrates with Supabase for data persistence:

1. **Tables Used**
    - tokens - Stores token configurations
    - token_templates - Stores reusable templates
    - token_versions - Tracks version history
    - token_deployments - Records deployment details
2. **Service Functions**
    - getTokens() - Fetch all tokens for a project
    - getToken() - Fetch a specific token
    - createToken() - Create a new token
    - updateToken() - Update an existing token
    - deleteToken() - Delete a token
    - cloneToken() - Create a copy of a token

Web3 Integration

The token system integrates with blockchain networks:

1. **Deployment Operations**
    - Connect to wallet and network
    - Build contract with configured parameters
    - Estimate gas and transaction costs
    - Execute deployment transaction
    - Track transaction status
2. **Token Management**
    - Monitor token status on blockchain

    - Execute token-specific operations (mint, burn, transfer)
    - Update token status based on blockchain events

Key Workflows

1. Creating a New Token

1. Navigate to TokenBuilder
2. Select token standard
3. Configure token properties using standard-specific component
4. Save token to database
5. Optionally deploy to blockchain

2. Using a Template

1. Browse templates in TokenTemplateSelector
2. Select template that matches requirements
3. Modify template as needed
4. Create token from modified template

3. Deploying a Token

1. Open token in TokenManagement
2. Navigate to deployment section
3. Configure deployment settings
4. Execute deployment
5. Track transaction status

Configuration Options

The token system can be configured through:

1. **config.ts**
    - Simple/Detailed mode toggle
    - Enable/disable specific standards
2. **UI Settings**
    - Toggling between essential and detailed views
    - Project selection for multi-project scenarios

Future Extension Points

The token system is designed for extensibility:

1. **New Token Standards**
    - Add new standard to ENABLED_STANDARDS
    - Create configuration components
    - Update index.ts exports
2. **Enhanced Templates**
    - Add template categories and types
    - Create specialized template builders
    - Extend template data structures
3. **Additional Blockchain Networks**
    - Extend deployment system
    - Add network-specific configurations
    - Support cross-chain operations

Conclusion

The token system provides a comprehensive framework for token creation, configuration, and deployment. Its modular architecture supports multiple token standards, configuration styles, and deployment targets, making it adaptable to a wide range of blockchain token scenarios.