BondVault Documentation

Table of Contents

Overview

BondVault is a modern full-stack Web3 application that enables tokenized government bonds as Real World Assets (RWA) on the blockchain. The platform allows users to purchase fractional ownership of government bonds using stablecoins, earn fixed interest, and securely redeem their bonds after maturity.

Key Highlights

- **Blockchain-Based**: Built on Ethereum using Solidity smart contracts
- **Fractional Ownership**: Buy bonds in any amount using USDC
- **Fixed Returns**: Earn predictable interest (e.g., 7% annually)
- **Time-Locked**: Automatic maturity enforcement via blockchain timestamps
- **Transparent**: All transactions visible on-chain

Use Cases

- Democratizing access to government bonds
- Enabling fractional bond investments
- Providing blockchain-based proof of ownership

- Creating programmable fixed-income instruments

Features

1. Tokenized Government Bonds (RWA)

Government bonds are represented as ERC20-like tokens (GBOND), where each token represents fractional ownership of the underlying bond. This enables:

- Easy transferability
- Fractional ownership
- On-chain tracking
- Programmable redemption

2. Fractional Bond Purchase with Stablecoin

Users can purchase bonds using MockUSDC (a 6-decimal stablecoin):

- Purchase any amount (no minimum threshold)
- Automatic conversion between USDC and bond tokens
- Instant settlement on-chain
- Transparent pricing

3. Time-Locked Bond Maturity

Bonds have a predetermined maturity date:

- Fixed maturity timestamp set at issuance
- Redemption blocked until maturity
- Blockchain-enforced time lock
- No manual intervention required

4. Interest Calculation & Payout

Fixed interest rate mechanism:

- Predetermined interest rate (e.g., 7%)
- Calculated on-chain at redemption
- Payout includes principal + accrued interest
- Transparent calculation logic

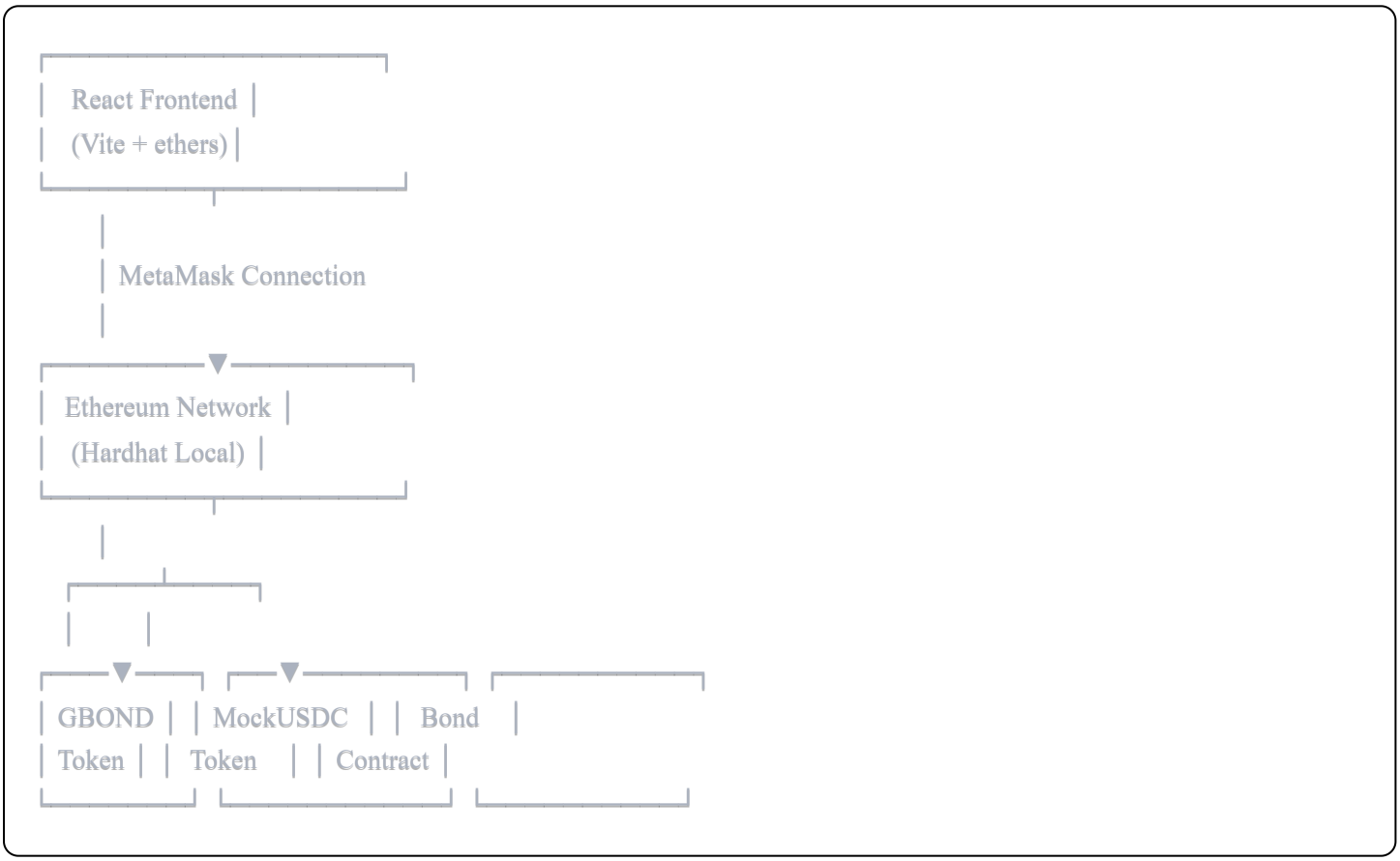
5. Secure Redemption & Token Burn

Bond redemption process:

- Bond tokens are burned upon redemption
- Prevents double-spending
- Maintains accurate token supply
- Ensures one-time payout per bond

Architecture

System Components



Data Flow

1. **User Interaction**: User connects wallet and interacts with frontend
2. **Transaction Creation**: Frontend creates transaction using ethers.js
3. **MetaMask Signature**: User signs transaction in MetaMask
4. **Blockchain Execution**: Transaction executed on Hardhat local network
5. **Smart Contract Logic**: BondContract processes purchase/redemption

6. **State Update**: Blockchain state updated, events emitted
7. **Frontend Update**: UI refreshes with new balances

Technology Stack

Component	Technology	Purpose
Smart Contracts	Solidity	Business logic and token standards
Development Framework	Hardhat	Smart contract development, testing, and deployment
Frontend Framework	React + Vite	Fast, modern UI with hot module replacement
Web3 Library	ethers.js	Blockchain interaction and wallet connection
Wallet Integration	MetaMask	User authentication and transaction signing
Local Blockchain	Hardhat Network	Local Ethereum node for development
Styling	CSS	Custom styling for dashboard UI
Version Control	Git/GitHub	Source code management

Why These Technologies?

- **Solidity**: Industry standard for Ethereum smart contracts
- **Hardhat**: Superior developer experience with built-in testing and debugging
- **React + Vite**: Fast build times and excellent developer experience
- **ethers.js**: Lightweight, comprehensive Web3 library
- **MetaMask**: Most popular Ethereum wallet with broad user base

---

Smart Contracts

Contract Architecture

1. BondToken.sol

ERC20-like token representing bond ownership.

Key Features:

- Standard ERC20 interface
- Mintable by contract owner
- Burnable on redemption
- 18 decimal precision

Core Functions:

```solidity
function mint(address to, uint256 amount)
function burn(address from, uint256 amount)
function balanceOf(address account)
function transfer(address to, uint256 amount)
```

2. MockUSDC.sol

Stablecoin used for bond purchases and redemptions.

Key Features:

- 6 decimal precision (matching real USDC)
- Mintable for testing purposes
- Standard ERC20 functionality
- Faucet functionality for development

Core Functions:

```solidity
function mint(address to, uint256 amount)
function balanceOf(address account)
function transfer(address to, uint256 amount)
function approve(address spender, uint256 amount)
```

3. BondContract.sol

Main contract orchestrating bond lifecycle.

Key Features:

- Bond issuance management
- Purchase processing
- Maturity enforcement

- Redemption logic
- Interest calculation

Core Functions:

```solidity
function buyBonds(uint256 usdcAmount)
function redeemBonds(uint256 bondAmount)
function calculateRedemptionAmount(uint256 bondAmount)
function getBondDetails()
```

Contract Interactions

Purchase Flow:

1. User approves USDC spending to BondContract
2. User calls `buyBonds(amount)`
3. Contract transfers USDC from user
4. Contract transfers GBOND tokens to user
5. Event emitted for tracking

Redemption Flow:

1. User calls `redeemBonds(amount)`
2. Contract checks maturity date
3. Contract calculates principal + interest
4. Contract burns GBOND tokens
5. Contract transfers USDC to user
6. Event emitted for tracking

Frontend Application

User Interface Components

1. Dashboard

Main interface showing:

- Wallet connection status

- User's Ethereum address
- ETH balance
- USDC balance
- GBOND token balance
- Bond details (rate, maturity)

2. Bond Purchase Section

- Input field for USDC amount
- Real-time conversion to GBOND
- Purchase button with transaction feedback
- Transaction status notifications

3. Bond Redemption Section

- Display of redeemable bonds
- Maturity countdown timer
- Redemption button (enabled after maturity)
- Expected payout calculation

State Management

The frontend manages several key states:

- **Wallet Connection**: Connected/disconnected state
- **User Balances**: ETH, USDC, GBOND balances
- **Contract Data**: Interest rate, maturity date, bond details
- **Transaction Status**: Pending, confirmed, failed
- **Loading States**: API calls and blockchain interactions

Web3 Integration

Connecting to MetaMask:

```javascript
const provider = new ethers.providers.Web3Provider(window.ethereum);
await provider.send("eth_requestAccounts", []);
const signer = provider.getSigner();
```

Reading Contract Data:

```javascript
const bondContract = new ethers.Contract(address, abi, provider);
const balance = await bondContract.balanceOf(userAddress);
```

Sending Transactions:

```javascript
const bondContractWithSigner = bondContract.connect(signer);
const tx = await bondContractWithSigner.buyBonds(amount);
await tx.wait();
```

Installation & Setup

Prerequisites

Before starting, ensure you have:

- **Node.js** (v16 or higher)
- **npm** or **yarn**
- **Git**
- **MetaMask** browser extension

Step-by-Step Installation

1. Clone the Repository

```bash
git clone https://github.com/ChanchalTaye/Bond-Vault.git
cd Bond-Vault
```

2. Install Dependencies

```bash
```

```
# Install Hardhat dependencies
npm install

# Navigate to frontend and install dependencies
cd frontend
npm install
cd ..
```

3. Start Local Blockchain

```bash
npx hardhat node
```

This starts a local Ethereum node on `http://localhost:8545` with 20 pre-funded accounts.

4. Deploy Smart Contracts

Open a new terminal window:

```bash
npx hardhat run scripts/deploy.js --network localhost
```

Important: Save the deployed contract addresses from the console output.

5. Configure Frontend

Update the frontend configuration file with deployed contract addresses:

```javascript
// src/config.js
export const BOND_CONTRACT_ADDRESS = "0x...";
export const USDC_ADDRESS = "0x...";
export const GBOND_ADDRESS = "0x...";
```

6. Start Frontend Application

```bash
cd frontend
npm run dev
```

The application will be available at `http://localhost:5173`

7. Configure MetaMask

Add Hardhat Network:

- Network Name: Hardhat Localhost
- RPC URL: http://localhost:8545
- Chain ID: 31337
- Currency Symbol: ETH

Import Test Account:

- Copy a private key from the Hardhat node output
- Import into MetaMask
- Switch to Hardhat Localhost network

Verification

After setup, verify:

- Frontend loads without errors
- MetaMask connects successfully
- Balances display correctly
- You can interact with the contract

User Guide

Getting Started

1. Connect Your Wallet

- Click "Connect Wallet" button
- Approve MetaMask connection
- Ensure you're on Hardhat Localhost network

2. Get Test USDC

For development, mint test USDC:

```
bash
```

```
npx hardhat run scripts/mintUSDC.js --network localhost
```

3. Purchasing Bonds

Steps:

1. Enter desired USDC amount
2. Click "Approve USDC" (first-time only)
3. Confirm approval in MetaMask
4. Click "Buy Bonds"
5. Confirm purchase in MetaMask
6. Wait for transaction confirmation
7. See updated GBOND balance

Important Notes:

- Each USDC spent equals 1 GBOND token
- Minimum purchase: 1 USDC
- Approval only needed once per wallet

4. Monitoring Your Investment

View your dashboard for:

- Current GBOND holdings
- Time until maturity
- Expected redemption value
- Current interest earned

5. Redeeming Bonds

Steps:

1. Wait for maturity date
2. Click "Redeem Bonds" (enabled after maturity)
3. Enter amount to redeem
4. Confirm transaction in MetaMask
5. Receive USDC (principal + interest)

Redemption Calculation:

```
Redemption Amount = Principal × (1 + Interest Rate)
Example: 1000 USDC × (1 + 0.07) = 1070 USDC
```

Common Operations

Check Bond Details

View on dashboard:

- Interest rate
- Maturity date
- Total supply
- Your holdings

Transfer GBOND Tokens

GBOND tokens are transferable before maturity:

1. Use standard ERC20 transfer
2. Recipient can redeem after maturity

Emergency Situations

If you encounter issues:

1. Check MetaMask network
2. Verify transaction status on block explorer
3. Ensure sufficient ETH for gas fees
4. Refresh page and reconnect wallet

Development Guide

Project Structure

```
Bond-Vault/
├── contracts/
│   ├── BondToken.sol
│   ├── MockUSDC.sol
│   └── BondContract.sol
```

```
├── scripts/
│   ├── deploy.js
│   └── mintUSDC.js
├── test/
│   └── BondContract.test.js
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   ├── App.jsx
│   │   ├── config.js
│   │   └── main.jsx
│   ├── public/
│   └── package.json
├── hardhat.config.js
└── package.json
```

Development Workflow

1. Smart Contract Development

Writing Contracts:

- Place contracts in `contracts/` directory
- Follow Solidity best practices
- Add comprehensive comments
- Use SafeMath for calculations

Testing Contracts:

```
bash

npx hardhat test
```

Compiling Contracts:

```
bash

npx hardhat compile
```

2. Frontend Development

Hot Reload: Vite provides instant hot module replacement:

```
bash
```

```
npm run dev
```

Building for Production:

```
bash

npm run build
```

Preview Production Build:

```
bash

npm run preview
```

3. Debugging

Hardhat Console:

```
bash

npx hardhat console --network localhost
```

Contract Events: Listen to events in frontend:

```
javascript

bondContract.on("BondsPurchased", (buyer, amount) => {
  console.log(`${buyer} purchased ${amount} bonds`);
});
```

Transaction Debugging: Use Hardhat's built-in tracing:

```
bash

npx hardhat run scripts/debug.js --network localhost
```

Testing Strategy

Unit Tests

Test individual contract functions:

```
javascript
```

```
describe("BondContract", function() {
  it("Should allow bond purchase", async function() {
    await usdc.approve(bondContract.address, amount);
    await bondContract.buyBonds(amount);
    expect(await gbond.balanceOf(buyer)).to.equal(amount);
  });
});
```

Integration Tests

Test complete user flows:

- Connect wallet → Purchase bonds → Wait maturity → Redeem
- Test edge cases and error conditions
- Verify event emissions

Frontend Tests

```
bash

npm run test
```

Best Practices

1. **Security First**: Always audit smart contracts
2. **Gas Optimization**: Minimize storage operations
3. **Error Handling**: Comprehensive try-catch blocks
4. **User Feedback**: Clear transaction status messages
5. **Code Comments**: Document complex logic
6. **Version Control**: Commit frequently with clear messages

---

Security Considerations

Smart Contract Security

1. Reentrancy Protection

Use checks-effects-interactions pattern:

```
solidity
```

```
function redeemBonds(uint256 amount) public {
  require(block.timestamp >= maturityDate, "Not matured");
  require(bondToken.balanceOf(msg.sender) >= amount, "Insufficient bonds");

  // Effects
  bondToken.burn(msg.sender, amount);

  // Interactions
  usdc.transfer(msg.sender, redemptionAmount);
}
```

2. Integer Overflow/Underflow

Use OpenZeppelin's SafeMath or Solidity 0.8+ built-in checks.

3. Access Control

Implement role-based access:

```solidity
modifier onlyOwner() {
  require(msg.sender == owner, "Not authorized");
  _;
}
```

4. Time Manipulation

Be aware that miners can manipulate `block.timestamp` by ~15 seconds.

Frontend Security

1. Input Validation

Validate all user inputs before sending to blockchain:

```javascript
if (amount <= 0 || isNaN(amount)) {
  throw new Error("Invalid amount");
}
```

2. Transaction Verification

Always verify transaction success:

```javascript
```

```
const tx = await contract.buyBonds(amount);
const receipt = await tx.wait();
if (receipt.status === 0) {
  throw new Error("Transaction failed");
}
```

3. Wallet Security

- Never expose private keys
- Use environment variables for sensitive data
- Implement connection timeout

Deployment Security Checklist

- [ ] Audit smart contracts
- [ ] Test on testnet thoroughly
- [ ] Verify contract source code
- [ ] Set appropriate gas limits
- [ ] Implement emergency pause mechanism
- [ ] Set up monitoring and alerts
- [ ] Document all admin functions
- [ ] Implement multi-sig for critical operations

Future Enhancements

Short-term Improvements

1. **Secondary Market**
   - Enable peer-to-peer bond trading
   - Implement order book or AMM
   - Dynamic pricing based on time to maturity
2. **Multiple Bond Series**
   - Different maturity dates
   - Varying interest rates
   - Risk-tiered offerings
3. **Enhanced UI**
   - Real-time price charts

- Transaction history
- Portfolio analytics

Medium-term Goals

4. **Multi-chain Support**
    - Deploy on Polygon, Arbitrum, etc.
    - Cross-chain bond transfers
    - Unified liquidity pools
5. **Advanced Features**
    - Automatic reinvestment
    - Bond laddering strategies
    - Yield optimization
6. **Institutional Features**
    - KYC/AML integration
    - Regulatory compliance tools
    - Bulk purchase discounts

Long-term Vision

7. **Real Government Bonds**
    - Partner with government entities
    - Legal framework compliance
    - Regulated offering
8. **DeFi Integration**
    - Use bonds as collateral
    - Integration with lending protocols
    - Liquidity mining rewards
9. **Mobile Application**
    - Native iOS/Android apps
    - Biometric authentication
    - Push notifications

Contributing

We welcome contributions! Please:

1. Fork the repository
2. Create a feature branch
3. Make your changes
4. Add tests
5. Submit a pull request

Contribution Guidelines

- Follow existing code style
- Write meaningful commit messages
- Add documentation for new features
- Ensure all tests pass
- Update README if needed

License

This project is for educational and demonstration purposes.

Support

For issues, questions, or contributions:

- GitHub Issues: Bond-Vault Issues
- Repository: ChanchalTaye/Bond-Vault

Acknowledgments

- OpenZeppelin for secure contract libraries
- Hardhat for excellent development tools
- Ethereum community for continuous innovation
- All contributors and supporters of the project

Built with ❤️ for the Web3 community