Voting System Backend Documentation

Welcome to the comprehensive backend documentation for the Secure Voting System. This document outlines the project structure, explains the purpose of each file, and provides a simple, detailed walkthrough of the code.

Project Structure

A quick look at how the code is organized:

```
voting-backend/
├── server.js                # The Heart: Main application entry point
├── createDb.js              # Setup: Script to create the database
├── check_db.js              # Debug: Script to view database data
├── config/
│   └── database.js          # Config: Database connection settings
├── models/                  # Database Schemas
│   ├── index.js             # Setup: Initializes models & relationships
│   ├── election.js          # Table: Stores election details
│   ├── candidate.js         # Table: Stores candidate info
│   └── registrationToken.js # Table: Manage voter tokens
├── controllers/             # Business Logic
│   ├── electionController.js # Logic: Manage elections & Merkle roots
│   └── tokenController.js    # Logic: Generate tokens & register voters
├── routes/                  # API Endpoints
│   ├── electionRoutes.js    # Routes: Election related URLs
│   └── tokenRoutes.js       # Routes: Token related URLs
├── services/
│   └── witnessService.js    # Cache: Fast access to Merkle proofs
└── utils/                   # Utilities
    ├── blockchain.js        # Blockchain: Ethers.js helper functions
    ├── merkleTree.js        # Cryptography: Merkle Tree generation
    └── reset_db.js          # Utility: Wipe and reset database
```

Root Files

1. `server.js`

Role: The Entry Point.
What it does: Starts the web server, connects to the database, and sets up the API routes so the frontend can talk to the backend.

Code Walkthrough:

- **Imports**: We start by bringing in `express` (web framework), `body-parser` (to read data Sent by users), and `cors` (to allow the frontend to connect). We also import our database (`db`) and our route files.
- **Setup**: We create the `app` and define the `PORT` (5001).
- **Middleware**: We tell the app to use CORS and to parse incoming JSON data.
- **Routes**: We plug in our route files directly into URL paths like `/api/elections` and `/api/tokens`.
- **Special Routes**: We define two specific routes manually (`/api/register` and `/api/merkle/witness`) to handle voter registration and proof fetching directly.

- **Start Server**: Finally, we sync with the database (updating tables if needed) and tell the app to start listening for requests.

The Code:

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const db = require('./models');
const electionRoutes = require('./routes/electionRoutes');
const tokenRoutes = require('./routes/tokenRoutes');
const tokenController = require('./controllers/tokenController');
const electionController = require('./controllers/electionController');

const app = express();
const PORT = process.env.PORT || 5001;

app.use(cors());
app.use(bodyParser.json());

// Routes
app.use('/api/elections', electionRoutes);
app.use('/api/tokens', tokenRoutes);

// Standalone routes
app.post('/api/register', tokenController.registerVoter);
app.post('/api/merkle/witness', electionController.getMerkleWitness);

// Sync Database and Start Server
db.sequelize.sync({ alter: true }).then(() => {
    console.log('Database synced');
    app.listen(PORT, () => {
        console.log(`Server is running on port ${PORT}`);
    });
}).catch(err => {
    console.error('Failed to sync database:', err);
});
```

2. `createDb.js`

Role: The Setup Script.
What it does: Checks if your database exists. If not, it creates it.

Code Walkthrough:

- **Imports**: We use `mysql2` to talk to the MySQL server and `dotenv` to read your password securely.
- **Connection**: We connect to the database server using the credentials from your `.env` file.
- **Creation**: We run a SQL command `CREATE DATABASE IF NOT EXISTS`, which ensures the database is ready.
- **Cleanup**: We close the connection.

The Code:

```
const mysql = require('mysql2/promise');
require('dotenv').config();

async function createDatabase() {
    try {
        const connection = await mysql.createConnection({
            host: process.env.DB_HOST,
            user: process.env.DB_USER,
            password: process.env.DB_PASSWORD
        });

        await connection.query(`CREATE DATABASE IF NOT EXISTS \`${process.env.DB_NAME}\`;`);
        console.log(`Database '${process.env.DB_NAME}' created or already exists.`);
        await connection.end();
    } catch (error) {
        console.error('Error creating database:', error);
    }
}

createDatabase();
```

⚙️ Configuration

3. `config/database.js`

Role: Database Settings.
What it does: Tells Sequelize (our database manager) how to connect to MySQL across different environments (development, test, production).

📜 **Code Walkthrough**:

- **Load Env**: We load environment variables.
- **Export Config**: We export an object with settings for `username`, `password`, `database`, `host`, and `dialect` (mysql).

📖 **The Code**:

```
require('dotenv').config();

module.exports = {
    development: {
        username: process.env.DB_USER,
        password: process.env.DB_PASSWORD,
        database: process.env.DB_NAME,
        host: process.env.DB_HOST,
        dialect: process.env.DB_DIALECT
    },
    // Test and Production use the same pattern...
    test: {
        username: process.env.DB_USER,
        password: process.env.DB_PASSWORD,
        database: process.env.DB_NAME,
        host: process.env.DB_HOST,
        dialect: process.env.DB_DIALECT
    },
    production: {
        username: process.env.DB_USER,
        password: process.env.DB_PASSWORD,
        database: process.env.DB_NAME,
        host: process.env.DB_HOST,
        dialect: process.env.DB_DIALECT
    }
};
```

🔧 Utilities

4. `utils/blockchain.js`

Role: The Bridge to Blockchain.
What it does: Connects your backend to the Ethereum network. It handles funding wallets and verifying data on-chain.

📜 Code Walkthrough:

- **Imports**: We bring in `ethers` to interact with the blockchain.
- **Config**: We define the blockchain URL (localhost:8545) and the admin's private key.
- **Addresses & ABIs**: We define the addresses for the Faucet and Registry contracts, and the "API" (ABI) of the functions we want to call on them (`fundUser`, `setMerkleRoot`, `registerAuthorities`).
- **Setup**: We creating a "Provider" (connection) and a "Wallet" (signer).
- **Contracts**: We create JavaScript objects that represent our smart contracts.
- **Functions**:
    - `setMerkleRootOnChain`: Takes an election ID and root, sends a transaction to the blockchain to save it, and waits for confirmation.
    - `registerAuthoritiesOnChain`: Takes a list of authority addresses and registers them.
    - `fundAddress`: Sends fake ETH to a user so they can pay for gas fees.

💻 The Code:

```
const { ethers } = require('ethers');
```

```
// Configuration
const PROVIDER_URL = 'http://localhost:8545';
const PRIVATE_KEY = '0x1576a9d1c3aa8f98b6871f516acc97d8821948244e3f3f7591970e309e4e50b5';

const FAUCET_ADDRESS = '0x717a3ce015a2933d26090bae53e7dd105058e820';
const REGISTRY_ADDRESS = '0x00edda4aad3ae0e145fc74df56add19c418176eb';

const FAUCET_ABI = [ "function fundUser(address user) external" ];
const REGISTRY_ABI = [
    "function setMerkleRoot(string calldata electionId, bytes32 merkleRoot) external",
    "function registerAuthorities(string calldata electionId, address[] calldata authorities)
external"
];

const provider = new ethers.JsonRpcProvider(PROVIDER_URL);
const wallet = new ethers.Wallet(PRIVATE_KEY, provider);

const faucetContract = new ethers.Contract(FAUCET_ADDRESS, FAUCET_ABI, wallet);
const registryContract = new ethers.Contract(REGISTRY_ADDRESS, REGISTRY_ABI, wallet);

const blockchainUtils = {
    setMerkleRootOnChain: async (electionId, merkleRoot) => {
        try {
            console.log(`[Blockchain] Setting Merkle Root for ${electionId}: ${merkleRoot}`);
            const tx = await registryContract.setMerkleRoot(electionId, merkleRoot);
            console.log(`[Blockchain] Transaction sent: ${tx.hash}`);
            await tx.wait();
            console.log(`[Blockchain] Transaction confirmed.`);
            return tx.hash;
        } catch (error) {
            console.error("[Blockchain] Error setting Merkle Root:", error);
            throw error;
        }
    },

    registerAuthoritiesOnChain: async (electionId, authorityAddresses) => {
        try {
            console.log(`[Blockchain] Registering authorities for ${electionId}:`,
authorityAddresses);
            const tx = await registryContract.registerAuthorities(electionId,
authorityAddresses);
            await tx.wait();
            return tx.hash;
        } catch (error) {
            console.error("[Blockchain] Error registering authorities:", error);
            throw error;
        }
    },

    fundAddress: async (userAddress) => {
        try {
            console.log(`[Blockchain] Funding address: ${userAddress}`);
            const tx = await faucetContract.fundUser(userAddress);
            await tx.wait();
            return tx.hash;
```

```
        } catch (error) {
            console.error("[Blockchain] Error funding address:", error);
        }
    }
};

module.exports = blockchainUtils;
```

5. `utils/merkleTree.js`

Role: Crytography Engine.
What it does: Takes a list of voter "secrets" (commitments) and builds a secure tree structure (Merkle Tree) to prove membership without revealing identity.

Code Walkthrough:

- **Imports**: We use `merkletreejs` for the tree logic and `crypto-js` for SHA256 hashing.
- **Class**: We define a class `MerkleTreeService`.
- **Constructor**: When you create a new tree, it takes a list of commitments, hashes each one to create "leaves", and builds the tree.
- **getRoot**: Returns the top hash of the tree (the Merkle Root).
- **getProof**: Takes a single commitment and returns the "path" up the tree (the Proof) needed to verify it matches the root.

The Code:

```
const { MerkleTree } = require('merkletreejs');
const SHA256 = require('crypto-js/sha256');

class MerkleTreeService {
    constructor(commitments = []) {
        this.leaves = commitments.map(c => SHA256(c));
        this.tree = new MerkleTree(this.leaves, SHA256);
    }

    getRoot() {
        return this.tree.getHexRoot();
    }

    getProof(commitment) {
        const leaf = SHA256(commitment);
        return this.tree.getHexProof(leaf);
    }
}

module.exports = MerkleTreeService;
```

Database Models

6. `models/index.js`

Role: Database Layout.
What it does: Initializes the connection and sets up how tables relate to each other.

📜 **Code Walkthrough**:

- **Setup**: We import Sequelize and our config.
- **Connect**: We create a new `Sequelize` instance connected to our database.
- **Load Models**: We import the Election, Token, and Candidate models.
- **Associations**: We define the relationships:
    - One Election -> Many Tokens.
    - One Election -> Many Candidates.

💻 **The Code**:

```
const Sequelize = require('sequelize');
const config = require('../config/database.js');
const db = {};

const sequelize = new Sequelize(config.development.database, config.development.username,
config.development.password, {
    host: config.development.host,
    dialect: config.development.dialect
});

db.sequelize = sequelize;
db.Sequelize = Sequelize;

db.Election = require('./election')(sequelize, Sequelize);
db.RegistrationToken = require('./registrationToken')(sequelize, Sequelize);
db.Candidate = require('./candidate')(sequelize, Sequelize);

// Associations
db.Election.hasMany(db.RegistrationToken, { foreignKey: 'election_id', sourceKey: 'election_id'
});
db.RegistrationToken.belongsTo(db.Election, { foreignKey: 'election_id', targetKey:
'election_id' });

db.Election.hasMany(db.Candidate, { foreignKey: 'election_id', sourceKey: 'election_id' });
db.Candidate.belongsTo(db.Election, { foreignKey: 'election_id', targetKey: 'election_id' });

module.exports = db;
```

7. `models/election.js`

Role: Election Table.
What it does: Tells the database how to store an Election.

📜 **Code Walkthrough**:

- We define fields for the unique `election_id`, `name`, `creator`, key times (`start`, `end`, `result`), current `status` (like 'voting' or 'ended'), and the critical `merkle_root`.

💻 **The Code**:

```
module.exports = (sequelize, DataTypes) => {
    const Election = sequelize.define('Election', {
        election_id: { type: DataTypes.STRING, primaryKey: true, allowNull: false },
        election_name: { type: DataTypes.STRING, allowNull: false },
        creator_name: { type: DataTypes.STRING, allowNull: true },
        start_time: { type: DataTypes.DATE, allowNull: true },
        end_time: { type: DataTypes.DATE, allowNull: true },
        result_time: { type: DataTypes.DATE, allowNull: true },
        status: {
            type: DataTypes.ENUM('created', 'registration', 'setup_completed', 'voting',
'ended'),
            defaultValue: 'created'
        },
        merkle_root: { type: DataTypes.STRING, allowNull: true }
    }, {
        timestamps: true,
        underscored: true
    });
    return Election;
};
```

8. `models/registrationToken.js`

Role: Voter Token Table.
What it does: Stores the one-time passwords (tokens) for voters.

📜 Code Walkthrough:

- We define fields for the secure `token`, `election_id`, voter info (`full_name`, `voter_id`), their secure vote `commitment`, and whether the token is `used` or `unused`.

📖 The Code:

```
module.exports = (sequelize, DataTypes) => {
    const RegistrationToken = sequelize.define('RegistrationToken', {
        id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
        token: { type: DataTypes.STRING, unique: true, allowNull: false },
        election_id: { type: DataTypes.STRING, allowNull: false },
        full_name: { type: DataTypes.STRING, allowNull: true },
        voter_id: { type: DataTypes.STRING, allowNull: true },
        commitment: { type: DataTypes.STRING, allowNull: true },
        status: { type: DataTypes.ENUM('unused', 'used'), defaultValue: 'unused' },
        used_at: { type: DataTypes.DATE, allowNull: true }
    }, {
        timestamps: true,
        underscored: true
    });
    return RegistrationToken;
};
```

🧠 Controllers (Business Logic)

9. `controllers/electionController.js`

Role: Election Manager.
What it does: Handles all the complex logic for creating and managing elections.

📜 Code Walkthrough:

- **Imports**: We import models, blockchain utils, and merkle services.
- `createElection`: Receives an ID and name, creates a basic entry in the database.
- `setupElection`: Updates time settings and adds candidates. It also talks to the blockchain to register election authorities.
- `completeSetup`: Marks election as ready for registration. It starts a timer (2 minutes) to automatically close registration and generate the Merkle Root.
- `generateMerkleRoot (Helper)`: This is the heavy lifter. It finds all *used* tokens, grabs their commitments, builds a Merkle Tree, saves the Root to the database, and publishes it to the Blockchain.
- `getMerkleWitness`: When a user wants to vote, they ask for a "witness" (proof). This function checks our cache or rebuilds the tree to give them their proof.

📖 The Code:

```
const db = require('../models');
const Election = db.Election;
const RegistrationToken = db.RegistrationToken;
const Candidate = db.Candidate;
const MerkleTreeService = require('../utils/merkleTree');
const blockchainUtils = require('../utils/blockchain');
const witnessService = require('../services/witnessService');

// Helper: Generates Merkle Root from registered voters
const generateMerkleRoot = async (election_id) => {
    try {
        const election = await Election.findByPk(election_id);
        if (!election) return;

        // Get all voters who have registered (used tokens)
        const tokens = await RegistrationToken.findAll({
            where: { election_id, status: 'used' }
        });

        const commitments = tokens.map(t => t.commitment).filter(c => c);
        let root = null;
        const witnessMap = {};

        if (commitments.length > 0) {
            // Build the tree
            const merkleService = new MerkleTreeService(commitments);
            root = merkleService.getRoot();

            // Save proofs for everyone
            commitments.forEach(commitment => {
                witnessMap[commitment] = merkleService.getProof(commitment);
            });
            witnessService.storeWitnesses(election_id, witnessMap);
        } else {
            root = '0x0000000000000000000000000000000000000000000000000000000000000000';
        }
```

```
        // Save to DB and Blockchain
        election.merkle_root = root;
        election.status = 'setup_completed';
        await election.save();
        await blockchainUtils.setMerkleRootOnChain(election_id, root);

    } catch (error) {
        console.error(`Error generating Merkle Root:`, error);
    }
};

exports.createElection = async (req, res) => {
    try {
        const { election_id, election_name, creator_name } = req.body;
        const election = await Election.create({ election_id, election_name, creator_name,
status: 'created' });
        res.status(201).json(election);
    } catch (error) { res.status(500).json({ message: error.message }); }
};

exports.setupElection = async (req, res) => {
    try {
        const { election_id, candidates, start_time, end_time, result_time, authorities } =
req.body;

        const election = await Election.findByPk(election_id);
        if (!election) return res.status(404).json({ message: 'Election not found' });

        election.start_time = start_time;
        election.end_time = end_time;
        election.result_time = result_time;
        await election.save();

        if (candidates && candidates.length > 0) {
            const candidateData = candidates.map(c => ({
                election_id, candidate_name: c.candidate_name, symbol_name: c.symbol_name
            }));
            await Candidate.bulkCreate(candidateData);
        }

        if (authorities) {
            const authAddresses = authorities.map(a => a.wallet_address);
            if (authAddresses.length > 0) await
blockchainUtils.registerAuthoritiesOnChain(election_id, authAddresses);
        }

        res.json({ message: 'Election setup updated', election });
    } catch (error) { res.status(500).json({ message: error.message }); }
};

exports.completeSetup = async (req, res) => {
    try {
        const { election_id } = req.body;
        const election = await Election.findByPk(election_id);
```

```
        election.status = 'registration';
        await election.save();

        // 2 Minutes Timer
        setTimeout(() => generateMerkleRoot(election_id), 2 * 60 * 1000);

        res.json({ message: 'Registration started. Merkle Root in 2 minutes.' });
    } catch (error) { res.status(500).json({ message: error.message }); }
};

exports.getMerkleWitness = async (req, res) => {
    try {
        const { election_id, commitment } = req.body;
        let proof = witnessService.getWitness(election_id, commitment);

        // If missing, try to regenerate
        if (!proof) {
             const tokens = await RegistrationToken.findAll({ where: { election_id, status:
'used' } });
             const commitments = tokens.map(t => t.commitment).filter(c => c);
             if (commitments.length > 0) {
                const merkleService = new MerkleTreeService(commitments);
                proof = merkleService.getProof(commitment);
             }
        }

        if (!proof) return res.status(404).json({ message: 'Witness not found' });
        res.json({ proof });
    } catch (error) { res.status(500).json({ message: error.message }); }
};
```

10. `controllers/tokenController.js`

Role: Token Manager.
What it does: Generates invite tokens and registers users when they redeem them.

📜 Code Walkthrough:

- `generateTokenForUser`: Creates a random 16-byte hex string (the token), saves it to the database linked to a specific user and election.
- `registerVoter`: This is called when a user submits their token and commitment.
    - It checks if the token exists and is unused.
    - It saves the voter's `commitment` (their ZKP identity).
    - It marks the token as `used`.
    - **Crucially**, it calls `blockchainUtils.fundAddress` to send the user some ETH so they can vote.

📖 The Code:

```
const db = require('../models');
const RegistrationToken = db.RegistrationToken;
const crypto = require('crypto');
const { ethers } = require('ethers');
const blockchainUtils = require('../utils/blockchain');

exports.generateTokenForUser = async (req, res) => {
    try {
        const { election_id, full_name, voter_id } = req.body;
        const tokenString = crypto.randomBytes(16).toString('hex');

        const token = await RegistrationToken.create({
            token: tokenString, election_id, full_name, voter_id, status: 'unused'
        });

        res.status(201).json({ message: 'Token generated', token: token });
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
};

exports.registerVoter = async (req, res) => {
    try {
        const { election_id, token, commitment, walletAddress } = req.body;

        const registration = await RegistrationToken.findOne({ where: { token, election_id } });

        if (!registration) return res.status(404).json({ message: 'Invalid token' });
        if (registration.status === 'used') return res.status(400).json({ message: 'Token
already used' });

        // Update token
        registration.commitment = commitment;
        registration.status = 'used';
        registration.used_at = new Date();
        await registration.save();

        // Fund the user on Blockchain
        if (walletAddress) {
            blockchainUtils.fundAddress(walletAddress).catch(err => console.error("Faucet
failed:", err));
        }

        res.json({ message: 'Voter registered successfully' });
    } catch (error) { res.status(500).json({ message: error.message }); }
};
```