

AI-Prompt:

Du bist ein Softwarearchitekt. Es soll ein Produkt mit Anwendungen für die Online Wahl zum deutschen Bundestag hergestellt werden. Die Anwendungen sollen ein Backend für die Registrierung der Wähler als Onlinewähler enthalten, so dass nur registrierte Wähler ihre Stimme im Frontend abgeben können. Die Wahl soll den Kriterien, die im Bundeswahlgesetz enthalten sind, genügen, also z.B. geheim sein. Um Manipulationen vorzubeugen, sollen alle wesentlichen Schritte in einer Blockchain festgehalten werden. Die Auszählung der Stimmen soll nach dem Ende der Stimmabgabe automatisch erfolgen und an den Wahlleiter gesendet werden. Bitte erstelle einen Entwurf mit Konzept.

Blockchain Voting System

Konzept

1. Einführung

Hier ist ein konzeptioneller Entwurf für ein System zur Online-Wahl zum Deutschen Bundestag, das die Anforderungen an Rechtssicherheit, Geheimhaltung, Transparenz und Manipulationsschutz erfüllt. Dabei wird moderne IT-Architektur mit Blockchain-Technologie kombiniert.

1.1. Zielsetzung

Ein sicheres, datenschutzkonformes und nachvollziehbares Online-Wahlsystem für den Bundestag, das:

- nur registrierten Wählern die Stimmabgabe ermöglicht,
 - die Anonymität der Stimme garantiert,
 - wesentliche Wahlvorgänge revisionssicher dokumentiert (Blockchain),
 - die Stimmen automatisiert nach Wahlschluss auszählt und dem Wahlleiter meldet,
 - den Anforderungen des Bundeswahlgesetzes entspricht.
-

1.2. Systemkomponenten (Architekturübersicht)

A. Frontend

Plattformen: Web & Mobile

Funktionen:

- Authentifizierung
- Anzeige des Wahlzettels
- Stimmabgabe
- Bestätigung (anonymisiert)

B. Backend

Microservice-Architektur, containerisiert (z.B. Docker/Kubernetes)

Wichtige Services:

- Registrierungsservice: Prüft Wahlberechtigung und erstellt verschlüsseltes Token für Wahlsystem
- Wählerauthentifizierung: Bindet z.B. eID, Personalausweis mit Online-Funktion oder ELSTER-Zertifikat ein
- Stimmabgabe-Service: Nimmt Stimme entgegen und verschlüsselt sie anonym
- Blockchain-Service: Persistiert Wahlereignisse (Registrierung, Token-Ausgabe, Stimmabgabe [nicht Inhalt!], Zählung etc.)
- Zähldienst: Führt nach Wahlschluss automatisiert die Auszählung durch
- Meldedienst: Übermittelt Ergebnis mit Prüfsummen an den Wahlleiter

C. Blockchain-Komponente

- Permissioned Blockchain (z.B. Hyperledger Fabric oder Quorum)
 - Knoten: Wahlkommission, Parteien, neutrale Beobachter
 - Einträge: Registrierungen (Hash), Stimmabgaben (anonym, nur Token + Zeitstempel), Zähl-Ergebnisse, Ereignisprotokolle
-

1.3. Wahlprozess (End-to-End)

1.3.1. Registrierung

- Nutzer authentifiziert sich mit amtlichem Ausweisdokument
- Prüfung durch Registrierungsservice (Anbindung an Melderegister)
- Token-Ausgabe an den Nutzer (wird später zur Stimmabgabe benötigt)
- Registrierung auf Blockchain protokolliert (nur Hash des Tokens)

1.3.2. Stimmabgabe

- Nutzer meldet sich mit Token im Frontend an
- Stimme wird lokal im Browser verschlüsselt (Ende-zu-Ende)
- Stimme (anonym) und Token (signiert) werden ans Backend gesendet
- Token wird entwertet (keine Mehrfachabgabe möglich)
- Blockchain: Eintrag mit Zeitstempel, Token-Hash, Transaktions-ID

1.3.3. Auszählung & Ergebnisübermittlung

- Nach Wahlschluss: Stimmen werden durch den Zähldienst aggregiert
 - Validierung gegen Blockchain (nur gültige, nicht doppelte Tokens)
 - Ergebnis + Blockchain-Verweis an Wahlleiter übermittelt
 - Ergebnisse öffentlich einsehbar über ein Dashboard (verifiziert über Blockchain)
-

1.3.4. Sicherheits- und Datenschutzmaßnahmen

- Ende-zu-Ende-Verschlüsselung der Stimmen
 - Tokenisierung zur Entkopplung von Identität und Stimme
 - Auditing durch unabhängige Instanzen über Blockchain
 - Zero-Knowledge-Proofs (optional) zur Gewährleistung der Anonymität bei gleichzeitiger Nachvollziehbarkeit
 - Keine Speicherung personenbezogener Daten in der Blockchain
 - DSGVO-konforme Verarbeitung
-

1.3.5. Technologiestack (Beispielhaft)

- Frontend: React / Vue + WebCrypto API
 - Backend: Node.js, Python oder Go (Microservices)
 - Blockchain: Hyperledger Fabric / Quorum
 - Datenbank: PostgreSQL (nicht für Stimmeninhalte, sondern für Nutzerverwaltung)
 - Authentifizierung: OIDC + eID / Online-Ausweisfunktion
-

1.3.6. Weitere Überlegungen

- Skalierbarkeit: Lastverteilung, Loadbalancer
- Barrierefreiheit: WCAG-konforme Oberflächen
- Offline-Backup: Möglichkeit zur hybriden (Papier + Online) Absicherung
- Audit-API: Externe Prüfer können Blockchain-Ereignisse verifizieren

Möchtest du auf einen bestimmten Teil näher eingehen, z. B. das Blockchain-Design, den Ablauf der Verschlüsselung oder das genaue Wähler-Token-Konzept?

2. Authentisierung und Geheimhaltung - das Wähler-Token-Konzept

Das Wähler-Token-Konzept ist zentral, um zwei entscheidende Anforderungen zu erfüllen:

1. Nur registrierte Wahlberechtigte dürfen wählen (Integrität der Wahl),
2. Die Stimmabgabe muss geheim sein (Geheimhaltung).

Das Token dient als anonymer Wahlausweis, der es erlaubt, genau einmal zu wählen – ohne dass die Identität des Wählers mit der abgegebenen Stimme in Verbindung gebracht werden kann.

2.1. Ziele des Wähler-Tokens

- Sicherstellung, dass nur berechtigte Personen wählen
- Verhinderung von Mehrfachabgaben (eine Stimme pro Token)
- Trennung von Identität und Stimmabgabe
- Nachvollziehbarkeit und Verifizierbarkeit ohne Datenschutzverletzung

2.2. Ablauf im Detail

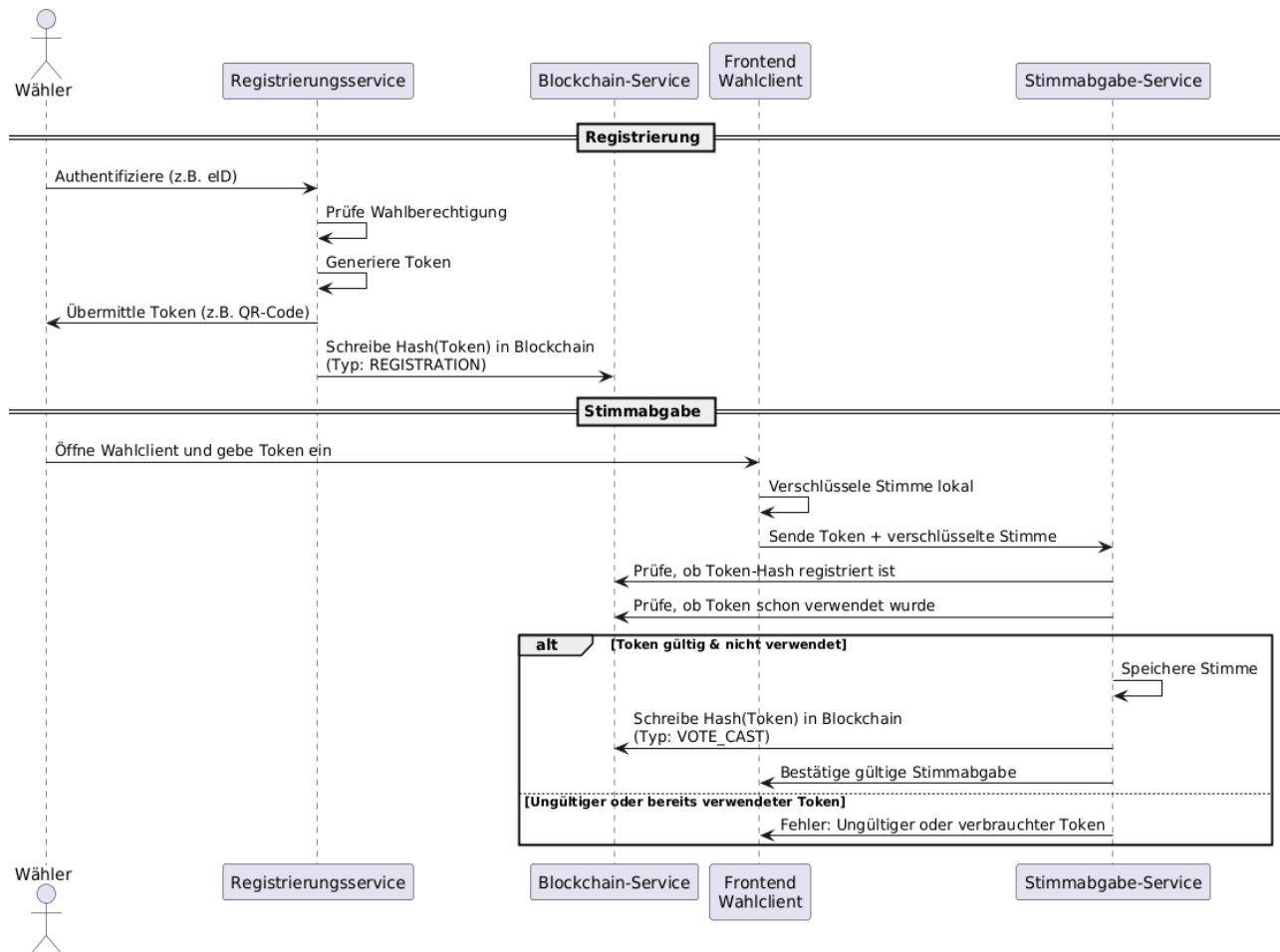


Schaubild 1: UML-Sequenzdiagramm: Wähler-Token-Flow

A. Registrierung und Token-Erzeugung

1. Der Wähler authentifiziert sich gegenüber dem Registrierungsservice (z. B. über eID oder ELSTER).
2. Der Dienst prüft die Wahlberechtigung (z. B. durch Anbindung an das Melderegister).
3. Wenn gültig, wird ein Wähler-Token generiert:
Ein zufälliger, kryptografisch starker Schlüssel (UUID + Zeitstempel, signiert).
Kein Bezug zu Name oder persönlichen Daten.
4. Der Token wird dem Nutzer gesichert übertragen (z. B. per verschlüsseltem Link oder QR-Code).
5. Im Backend wird ein Hashwert des Tokens auf der Blockchain gespeichert, z. B.:

```

Token-Hash = SHA-256(Token)
Blockchain-Eintrag: {
  type: "REGISTRATION",
  hash: <Token-Hash>,
  timestamp: <Zeit>,
  nonce: ...
}
  
```

}

Die Blockchain enthält nie den Token selbst, sondern nur einen Hash davon – für spätere Verifikation.

B. Stimmabgabe

1. Der Wähler gibt seinen Token im Frontend ein (z. B. als Login oder QR-Scan).
2. Der Token wird:
 1. lokal validiert (Format, Ablaufdatum etc.),
 2. anonymisiert verwendet zur Abgabe der Stimme.
3. Die Stimme wird lokal verschlüsselt (z. B. mit einem öffentlichen Schlüssel des Zählendienstes).
4. Der verschlüsselte Stimmzettel und der Token-Hash werden an den Stimmabgabe-Service gesendet.
5. Dieser prüft:
 1. Ob der Token-Hash in der Blockchain registriert ist,
 2. ob er noch nicht verwendet wurde.
6. Falls gültig:
 1. Stimme wird entgegengenommen,
 2. Token als "verbraucht" markiert (per Blockchain-Eintrag vom Typ "VOTE_CAST" mit gleichem Token-Hash),
 3. Stimme landet verschlüsselt im Speicherpool für Auszählung.

C. Auszählung

Nach Wahlschluss:

1. Alle gültigen Stimmen werden ausgewertet,
2. Stimmen werden entschlüsselt, aber der Token-Hash ist zu diesem Zeitpunkt irrelevant,
3. Blockchain wird herangezogen zur Prüfung, dass nur registrierte, einmalige Token verwendet wurden.

3. Sicherheit & Vorteile

- **Anonymität:** Die Token sind nicht rückverfolgbar auf Identitäten.
- **Integrität:** Ein Token kann nur einmal verwendet werden – doppelte Stimmen sind ausgeschlossen.
- **Verifizierbarkeit:** Externe Prüfer können anhand der Blockchain verifizieren, wie viele gültige Token registriert und verwendet wurden – ohne zu wissen, wer gewählt hat oder was gewählt wurde.
- **DSGVO-konform:** Keine Speicherung personenbezogener Daten in der Blockchain.

4. Erweiterungen

Optional kann das System um folgende Elemente ergänzt werden:

- **Zero-Knowledge-Proofs**, um zu beweisen, dass eine Stimme abgegeben wurde, ohne Token offenzulegen.
- **One-Time-Pad-Signaturen**, damit selbst die Plattform nicht weiß, ob ein Token zu einem bestimmten Zeitpunkt verwendet wurde.

Kann der Speicherpool für die abgegebenen verschlüsselten Stimmen auch aus einer Blockchain bestehen, ohne dass Geheimhaltung oder Sicherheit kompromittiert werden? Wie muss das obige Konzept abgeändert werden?

Ja, grundsätzlich kann auch der Speicherpool für verschlüsselte Stimmen als Blockchain realisiert werden, ohne die Geheimhaltung oder Sicherheit zu kompromittieren – unter bestimmten Voraussetzungen. Dabei ersetzt oder ergänzt die Blockchain die bisherige (zentrale oder verteilte) Speicherung der verschlüsselten Stimmzettel.

Vorteile einer Blockchain für verschlüsselte Stimmen

- **Unveränderlichkeit:** Keine Stimme kann nachträglich gelöscht oder verändert werden.
- **Transparenz:** Alle abgegebenen Stimmen sind öffentlich sichtbar (verschlüsselt), nachvollziehbar und zählbar.
- **Verteilte Sicherheit:** Keine zentrale Vertrauensinstanz erforderlich.

Wichtige Anforderungen, um Geheimhaltung zu gewährleisten

1. **Ende-zu-Ende-Verschlüsselung vor Speicherung auf der Blockchain.**
 - Die Stimme muss vor Einfügen in die Blockchain verschlüsselt werden (idealerweise im Browser/Client).
2. **Verwendung asymmetrischer Verschlüsselung.**
 - Verschlüsselung mit öffentlichem Schlüssel des Zähldienstes, der **nicht auf der Blockchain** bekannt ist.

3. **Keine Metadaten oder Token in Klartext auf der Blockchain.**
 - Nur verschlüsselte Stimme + Hash des Tokens (oder anonymisierte Referenz).
4. **Optional: Homomorphe oder MixNet-Verfahren**, falls Entschlüsselung ohne Rückverfolgung gewünscht ist.

Änderungen am bisherigen Konzept

Was wird ersetzt?

- Statt die verschlüsselten Stimmen in einer zentralen oder verteilten Datenbank zu speichern, wird **jede Stimme als Transaktion in die Blockchain geschrieben**.

Angepasster Ablauf (nur relevanter Teil)

Stimmabgabe mit Blockchain-Speicherpool

1. Der Wähler gibt seine Stimme ab (wie vorher), die **im Client verschlüsselt** wird.
2. Die verschlüsselte Stimme + Hash(Token) werden als **Transaktion in die Blockchain** geschrieben:

```
{
  "type": "VOTE",
  "encrypted_vote": "<ciphertext>",
  "token_hash": "<SHA-256(token)>",
  "timestamp": "2025-09-22T16:20:00Z"
}
```

3. Der Konsens-Mechanismus der Blockchain schreibt die Stimme irreversibel ins Ledger.
4. Der Zähldienst liest nach Wahlschluss **alle VOTE-Transaktionen**, prüft, ob Token gültig/nicht doppelt sind, entschlüsselt die Stimme, zählt sie.
5. Ergebnisse und Prüfsummen werden ebenfalls auf der Blockchain gespeichert.

Blockchain-Einträge (überarbeitet)

Typ	Zweck
REGISTRATION	Registrierter Token-Hash
VOTE	Verschlüsselte Stimme + Token-Hash
VOTE_CAST	Optional: Markierung, dass Token verwendet wurde (evtl. überflüssig, wenn VOTE ausreicht)
TALLY	Ergebnis + Prüfsumme

Sicherheitsgarantie

- Die **Geheimhaltung bleibt gewahrt**, da nur verschlüsselte Inhalte gespeichert werden.

- Solange der **private Schlüssel** zur Entschlüsselung sicher verwahrt bleibt (z. B. bei einer Wahlbehörde in einem HSM – Hardware Security Module), ist die Anonymität garantiert.
- Bei korrektem Design kann niemand, auch nicht durch Analyse der Blockchain, rekonstruieren, **wer** was gewählt hat.

Risiko

Re-Identification durch Metadatenanalyse

Schlüsselkompromittierung

Blockchain-Größe

Mitigation

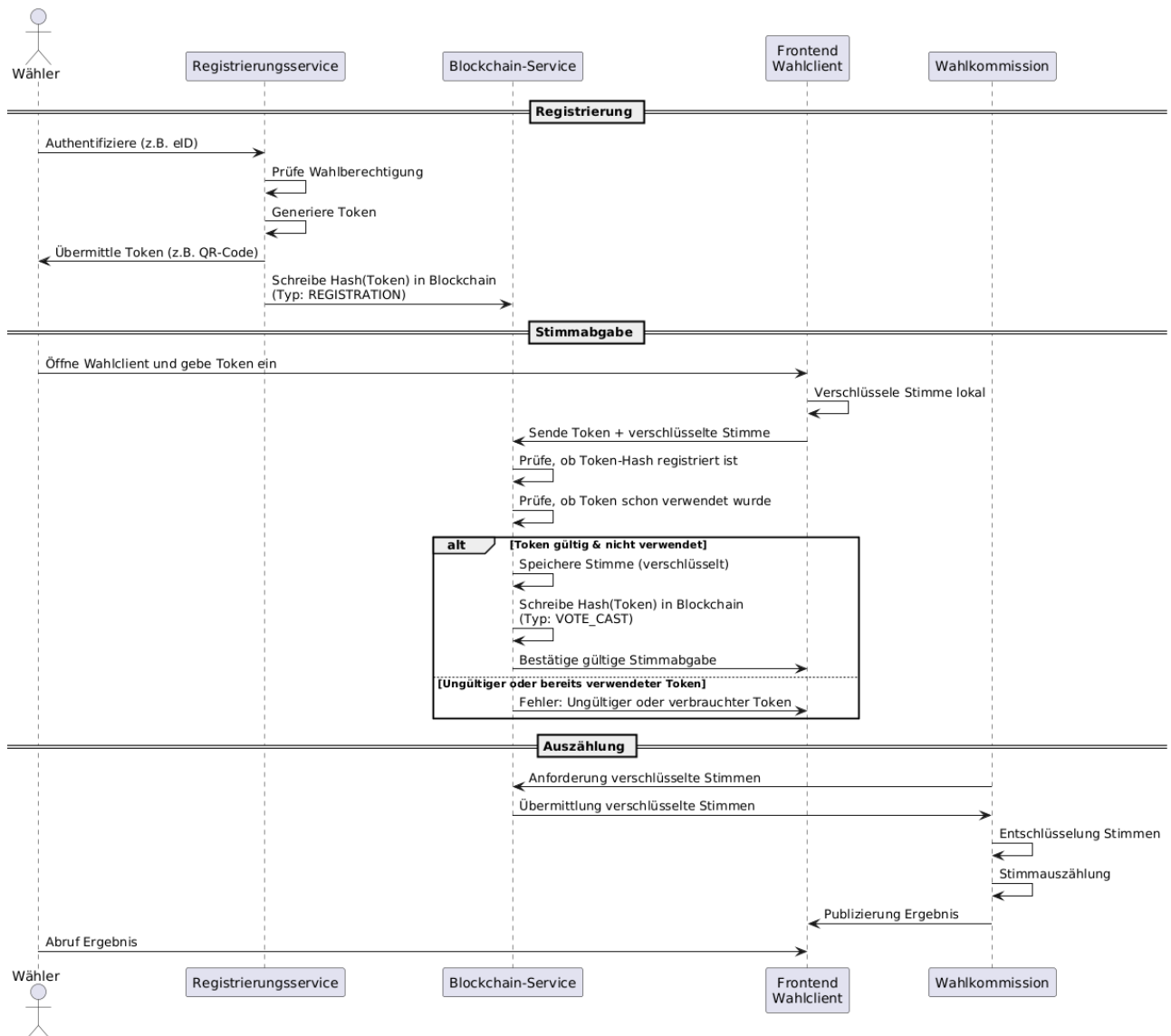
Minimale Metadaten, keine IP-Adressen, Zeit-Streuung

Einsatz von HSMs, Shamir Secret Sharing für Private Key

Komprimierung, Abstimmung auf effiziente Formate

Fazit

Ja, der Speicherpool für verschlüsselte Stimmen kann sicher durch eine Blockchain ersetzt werden – **wenn die Verschlüsselung clientseitig erfolgt, und wenn keine Metadaten oder Rückschlüsse auf Wähler enthalten sind.**



Möchtest du zu diesem Blockchain-Vote-Speicher ein technisches Beispiel oder ein angepasstes Diagramm?

Ja, ich möchte bitte sowohl ein technisches Beispiel als auch ein angepasstes Diagramm.

Technisches Beispiel

Hier ist ein **technisches Beispiel** für die Speicherung von Wähler-Token und verschlüsselten Stimmen in einer Blockchain, basierend auf einer JSON-basierten Transaktionsstruktur. Diese Architektur könnte mit einem privaten Blockchain-Framework wie **Hyperledger Fabric**, **Ethereum (mit Private Chain)** oder einer maßgeschneiderten Lösung umgesetzt werden.

1. Registrierungstransaktion (REGISTRATION)

Beim Generieren des Tokens wird der Hash des Tokens (nicht der Token selbst!) auf der Blockchain gespeichert.

Beispiel-Transaktion:

```
{
  "type": "REGISTRATION",
  "token_hash": "a4c67f2f82e14361df61d6fe7ef7cd228b3d3d2e8bc048ef278d4fa4a9b2e13c",
  "timestamp": "2025-09-22T10:15:30Z",
  "metadata": {
    "origin": "registration-service-1"
  }
}
```

- token_hash = SHA-256(Token)
- Keine Verbindung zur Wähleridentität
- Optional: Signatur der Registrierungsstelle

2. Stimmabgabe-Transaktion (VOTE)

Nach clientseitiger Verschlüsselung wird die verschlüsselte Stimme als Transaktion eingetragen.

Beispiel-Transaktion:

```
{
```

```

"type": "VOTE",
"encrypted_vote": "g2k14ab2==...==q9xz", // Base64-encoded ciphertext
"token_hash": "a4c67f2f82e14361df61d6fe7ef7cd228b3d3d2e8bc048ef278d4fa4a9b2e13c",
"timestamp": "2025-09-22T11:03:12Z",
"signature": "0xa91bc9a...47d", // Optional (z. B. für Node-Signatur)
"metadata": {
  "submitted_by": "vote-node-4"
}
}

```

- **Nur eine VOTE-Transaktion pro token_hash ist erlaubt** → Verhinderung von Mehrfachabgaben
- `encrypted_vote` enthält die mit dem öffentlichen Schlüssel verschlüsselte Stimme (z. B. "Partei X")

3. Auszählungs-Transaktion (TALLY)

Nach Wahlschluss erstellt ein Zähldienst ein signiertes Ergebnis:

Beispiel-Transaktion:

```

{
  "type": "TALLY",
  "results": {
    "Partei A": 142345,
    "Partei B": 130222,
    "Partei C": 53421
  },
  "counted_votes": 325988,
  "timestamp": "2025-09-23T18:00:00Z",
  "signature": "0xd4cf34d...aa1"
}

```

- Optional könnten auch **Merkle-Roots** der gezählten Stimmen beigefügt werden
- Die Entschlüsselung der Stimmen erfolgt lokal, nicht on-chain

Blockchain-Validierungslogik (vereinfacht)

Die Chain-Logik oder Smart Contracts müssen sicherstellen: (Python)

```
def validate_vote_submission(token_hash, blockchain):  
    # Prüfe, ob der Token-Hash registriert wurde  
  
    registered = any(tx for tx in blockchain if tx["type"] == "REGISTRATION" and tx["token_hash"]  
    == token_hash)  
  
    # Prüfe, ob für diesen Hash bereits gewählt wurde  
  
    already_voted = any(tx for tx in blockchain if tx["type"] == "VOTE" and tx["token_hash"] ==  
    token_hash)  
  
    if not registered:  
        raise Exception("Unbekannter Token")  
  
    if already_voted:  
        raise Exception("Token bereits verwendet")  
  
    return True
```

Wichtige technische Hinweise

Thema	Umsetzung
Kollisionsvermeidung	Token-Hash eindeutig, token selbst geheim
Verschlüsselung	Public-Key-Verschlüsselung, z. B. RSA-OAEP oder ECIES
Entschlüsselung	Nur autorisierte Stelle (Wahlleiter mit Private Key)
Zugriffsschutz	Blockchain nicht anonym schreibbar (nur autorisierte Nodes)

Wenn du möchtest, kann ich dir auch Beispiel-Smart-Contracts (z. B. in Solidity oder Go für Hyperledger) bereitstellen. Sag einfach Bescheid, welches Framework du bevorzugst.

Ich bevorzuge Quorum auf Ethereum-Basis, bitte erstelle die Beispiel-Smart-Contracts in Solidity.

Hier ist ein Beispiel für einen Satz von Solidity Smart Contracts, die auf Quorum (Ethereum-kompatible private Blockchain) laufen und die Wählerregistrierung, Stimmabgabe und Auszählung gemäß dem Token-basierten Konzept umsetzen.

1. Übersicht der Smart Contracts

- ElectionRegistry: Registrierung und Verifizierung von Token-Hashes
- VoteLedger: Speicherung der verschlüsselten Stimmen

- Tally: Einreichung und Veröffentlichung der Wahlergebnisse

1. ElectionRegistry.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract ElectionRegistry {
    mapping(bytes32 => bool) public registeredTokens;
    mapping(bytes32 => bool) public usedTokens;

    address public admin;

    constructor() {
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Only admin allowed");
        _;
    }

    function registerToken(bytes32 tokenHash) public onlyAdmin {
        require(!registeredTokens[tokenHash], "Token already registered");
        registeredTokens[tokenHash] = true;
    }

    function isTokenValid(bytes32 tokenHash) public view returns (bool) {
        return registeredTokens[tokenHash] && !usedTokens[tokenHash];
    }

    function markTokenUsed(bytes32 tokenHash) public onlyAdmin {
        require(registeredTokens[tokenHash], "Token not registered");
        require(!usedTokens[tokenHash], "Token already used");
        usedTokens[tokenHash] = true;
    }
}
```

2. VoteLedger.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "./ElectionRegistry.sol";

contract VoteLedger {
    struct Vote {
        bytes32 tokenHash;
        string encryptedVote; // e.g., Base64-encoded ciphertext
        uint256 timestamp;
    }

    ElectionRegistry public registry;
    address public admin;
    Vote[] public votes;

    constructor(address registryAddress) {
        registry = ElectionRegistry(registryAddress);
        admin = msg.sender;
    }

    function submitVote(bytes32 tokenHash, string memory encryptedVote) public {
        require(registry.isTokenValid(tokenHash), "Invalid or used token");
        votes.push(Vote({

```

```

        tokenHash: tokenHash,
        encryptedVote: encryptedVote,
        timestamp: block.timestamp
    }));
    registry.markTokenUsed(tokenHash);
}

function getVoteCount() public view returns (uint256) {
    return votes.length;
}

function getVote(uint256 index) public view returns (string memory, bytes32,
uint256) {
    Vote memory v = votes[index];
    return (v.encryptedVote, v.tokenHash, v.timestamp);
}
}

```

3. Tally.sol

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract Tally {
    address public admin;

    string public results;
    uint256 public submittedAt;

    constructor() {
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Only admin can submit results");
        _;
    }

    function submitResults(string memory resultJson) public onlyAdmin {
        require(submittedAt == 0, "Results already submitted");
        results = resultJson;
        submittedAt = block.timestamp;
    }

    function getResults() public view returns (string memory, uint256) {
        return (results, submittedAt);
    }
}

```

Deployment-Hinweise (für Quorum)

1. Stelle sicher, dass nur autorisierte Konten (admin) Transaktionen signieren können (z. B. Registrare, Wahlleiter).
2. Optional: Verwende **Quorum's privateFor-Funktion** zur privaten Stimmübertragung.
3. Die **verschlüsselte Stimme** wird on-chain gespeichert – Entschlüsselung erfolgt **off-chain** beim Wahlleiter.
4. Admin-Konten könnten durch ein Governance-Modell ersetzt werden (z. B. mit MultiSig).

Beispiel für Deployment und Nutzung (JavaScript/Hardhat/Scripts)

Hier ist ein vollständiges Beispiel, wie du mit Hardhat und ethers.js die Smart Contracts **ElectionRegistry**, **VoteLedger** und **Tally** auf einem Quorum-kompatiblen Netzwerk deployen und verwenden kannst.

Voraussetzungen

Stelle sicher, dass du Folgendes installiert hast:

```
npm install --save-dev hardhat @nomiclabs/hardhat-ethers ethers
```

Projektstruktur

my-election/

```
|— contracts/
|   |— ElectionRegistry.sol
|   |— VoteLedger.sol
|   |— Tally.sol
|— scripts/
|   |— deploy-and-use.js
|— hardhat.config.js
|— package.json
```

hardhat.config.js

```
require("@nomiclabs/hardhat-ethers");
```

```
module.exports = {
```

```
  solidity: "0.8.19",
```

```
  networks: {
```

```
    quorum: {
```

```
      url: "http://localhost:8545", // oder URL zu deinem Quorum-Knoten
```

```
      accounts: ["0x<PRIVATE_KEY>"], // deployender Admin-Account
```

```
    },
```

```
  },
```

```
};
```

```
  scripts/deploy-and-use.js
```

```
const hre = require("hardhat");
```

```
const ethers = hre.ethers;
```

```

async function main() {
  const [deployer] = await ethers.getSigners();
  console.log("Deploying contracts with:", deployer.address);

  // Deploy ElectionRegistry
  const Registry = await ethers.getContractFactory("ElectionRegistry");
  const registry = await Registry.deploy();
  await registry.deployed();
  console.log("ElectionRegistry deployed at:", registry.address);

  // Deploy VoteLedger with address of registry
  const Ledger = await ethers.getContractFactory("VoteLedger");
  const ledger = await Ledger.deploy(registry.address);
  await ledger.deployed();
  console.log("VoteLedger deployed at:", ledger.address);

  // Deploy Tally
  const Tally = await ethers.getContractFactory("Tally");
  const tally = await Tally.deploy();
  await tally.deployed();
  console.log("Tally deployed at:", tally.address);

  // --- Interaktion: Registrierung eines Token-Hashes ---
  const sampleToken = "abc123SECRET";
  const tokenHash = ethers.utils.keccak256(ethers.utils.toUtf8Bytes(sampleToken));
  const tx1 = await registry.registerToken(tokenHash);
  await tx1.wait();
  console.log("Token registered with hash:", tokenHash);

  // --- Interaktion: Stimmabgabe ---
  const encryptedVote = "Verschlüsselte Stimme base64==";

```



```
const tx2 = await ledger.submitVote(tokenHash, encryptedVote);
await tx2.wait();
console.log("Vote submitted.");
```

// --- Interaktion: Ergebnis veröffentlichen ---

```
const resultJson = JSON.stringify({
  "Partei A": 1000,
  "Partei B": 1200,
  "Partei C": 800,
});

const tx3 = await tally.submitResults(resultJson);
await tx3.wait();
console.log("Ergebnisse eingereicht.");
}
```

```
main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

Ausführen

```
npx hardhat run scripts/deploy-and-use.js --network quorum
```

Was dieses Skript macht:

- Deployt alle drei Verträge
- Registriert ein Token
- Reicht eine verschlüsselte Stimme ein
- Übermittelt ein Wahlergebnis

Hier ist ein einfaches, aber funktionales React + ethers.js UI-Frontend zur Interaktion mit deinen Smart Contracts auf Quorum. Es ermöglicht:

- Registrierung eines Token-Hashes (Admin)
- Abgabe einer verschlüsselten Stimme
- Abruf und Anzeige der Wahlergebnisse

Voraussetzungen

```
npx create-react-app quorum-vote-ui
```

```
cd quorum-vote-ui
```

```
npm install ethers
```

Projektstruktur (relevant)

```
quorum-vote-ui/
```

```
|— src/
|  |— App.js
|  |— contracts/
|     |— ElectionRegistry.json
|     |— VoteLedger.json
|     |— Tally.json
|  |— config.js
```

src/config.js

```
export const CONTRACT_ADDRESSES = {
  registry: "0x...", // Adresse aus Deployment
  ledger: "0x...",
  tally: "0x...",
};
```

```
export const PROVIDER_URL = "http://localhost:8545"; // oder RPC-URL deiner Quorum-Node
```

src/App.js

```
import React, { useState, useEffect } from "react";
import { ethers } from "ethers";
import { CONTRACT_ADDRESSES, PROVIDER_URL } from "../config";
import ElectionRegistry from "../contracts/ElectionRegistry.json";
import VoteLedger from "../contracts/VoteLedger.json";
import Tally from "../contracts/Tally.json";
```

```

const App = () => {
  const [provider, setProvider] = useState();
  const [signer, setSigner] = useState();
  const [registry, setRegistry] = useState();
  const [ledger, setLedger] = useState();
  const [tally, setTally] = useState();

  const [tokenInput, setTokenInput] = useState("");
  const [voteInput, setVoteInput] = useState("");
  const [results, setResults] = useState("");

  useEffect(() => {
    const init = async () => {
      const provider = new ethers.JsonRpcProvider(PROVIDER_URL);
      const signer = await provider.getSigner(0); // Für Test: erstes Konto
      setProvider(provider);
      setSigner(signer);

      const registry = new ethers.Contract(CONTRACT_ADDRESSES.registry,
ElectionRegistry.abi, signer);

      const ledger = new ethers.Contract(CONTRACT_ADDRESSES.ledger, VoteLedger.abi,
signer);

      const tally = new ethers.Contract(CONTRACT_ADDRESSES.tally, Tally.abi, signer);

      setRegistry(registry);
      setLedger(ledger);
      setTally(tally);
    };
    init();
  }, []);

  const registerToken = async () => {

```

```

const hash = ethers.keccak256(ethers.toUtf8Bytes(tokenInput));
const tx = await registry.registerToken(hash);
await tx.wait();
alert("Token registriert: " + hash);
};

```

```

const submitVote = async () => {
  const hash = ethers.keccak256(ethers.toUtf8Bytes(tokenInput));
  const tx = await ledger.submitVote(hash, voteInput);
  await tx.wait();
  alert("Stimme abgegeben");
};

```

```

const loadResults = async () => {
  const [resultText] = await tally.getResults();
  setResults(resultText);
};

```

```

return (
  <div style={{ padding: "2rem", fontFamily: "Arial" }}>
    <h2>Online-Wahl UI (Quorum)</h2>

    <h4>   Token registrieren (Admin)</h4>
      <input value={tokenInput} onChange={e => setTokenInput(e.target.value)}
placeholder="Token" />
      <button onClick={registerToken}>Registrieren</button>

    <h4>   Stimme abgegeben</h4>
      <textarea value={voteInput} onChange={e => setVoteInput(e.target.value)}
placeholder="Verschlüsselte Stimme" />
      <button onClick={submitVote}>Abgeben</button>

```

```
<h4>    Wahlergebnisse laden</h4>
```

```
<button onClick={loadResults}>Ergebnisse abrufen</button>
```

```
<pre>{results}</pre>
```

```
</div>
```

```
);
```

```
};
```

```
export default App;
```

Sicherheit und Authentifizierung

Für ein echtes System musst du:

- Benutzerrollen und -berechtigungen absichern (z. B. über Wallets, OAuth, Zertifikate)
- Den Verschlüsselungsteil im Client integrieren (z. B. RSA Public-Key-Verschlüsselung der Stimme)
- Wahrscheinlich zusätzliche Schutzmaßnahmen wie Rate-Limiting, Zeitfenster, Captchas einführen

Starten

```
npm start
```

Dann kannst du das UI unter `http://localhost:3000` aufrufen.

Möchtest du auch ein Modul, das die Verschlüsselung der Stimme im Browser mit RSA (z. B. mit `node-forge` oder `SubtleCrypto`) übernimmt?

1. Installation von `node-forge`

```
npm install node-forge
```

Strukturvorschlag

```
src/
```

```
|— crypto/
```

```
|  └— rsaEncryptor.js
```

2. `rsaEncryptor.js` – RSA-Verschlüsselung

```
import forge from 'node-forge';
```

```
/**
```

```
 * Verschlüsselt eine Nachricht (z. B. "Partei A") mit einem gegebenen öffentlichen Schlüssel.
```

```
 * @param {string} voteText - Die zu verschlüsselnde Stimme (Klartext).
```

```

* @param {string} publicKeyPem - Der öffentliche Schlüssel im PEM-Format.
* @returns {string} Base64-codierter verschlüsselter Text.
*/

```

```

export function encryptVote(voteText, publicKeyPem) {
  const publicKey = forge.pki.publicKeyFromPem(publicKeyPem);
  const encryptedBytes = publicKey.encrypt(voteText, 'RSA-OAEP', {
    md: forge.md.sha256.create(),
  });
  const encryptedBase64 = forge.util.encode64(encryptedBytes);
  return encryptedBase64;
}

```

3. Beispiel-Nutzung in App.js

Beispiel für einen eingebetteten öffentlichen PEM-Schlüssel:

```

const PUBLIC_KEY_PEM = `
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnEiX...snip...QIDAQAB
-----END PUBLIC KEY-----
`;

```

Ersetzen im submitVote-Handler:

```

import { encryptVote } from "./crypto/rsaEncryptor";

// ...

```

```

const submitVote = async () => {
  const hash = ethers.keccak256(ethers.toUtf8Bytes(tokenInput));
  const encrypted = encryptVote(voteInput, PUBLIC_KEY_PEM);
  const tx = await ledger.submitVote(hash, encrypted);
  await tx.wait();
  alert("Stimme abgegeben (verschlüsselt)");
};

```

Hinweise zur Sicherheit

- RSA-Schlüssel mindestens **2048 Bit**, idealerweise 3072+.
- Das **private Gegenstück** (für die Entschlüsselung) liegt **ausschließlich beim Wahlleiter, nicht im Browser**.
- Optional: Validierung der PEM-Struktur vor Nutzung.

Ergebnis

Du hast jetzt eine **vollständige clientseitige RSA-Verschlüsselung**, die:

- Deine Stimme sicher verschlüsselt
- Diese verschlüsselte Stimme in der Blockchain speichert
- Keine Möglichkeit bietet, den Klartext zurückzugewinnen, außer für die autorisierte Stelle mit dem privaten Schlüssel

Wenn du willst, kann ich dir auch ein Node.js-Modul für den Wahlleiter zur Entschlüsselung schreiben. Sag einfach Bescheid.

Ja, bitte schreib mir auch ein Beispiel für den Wahlleiter zur Entschlüsselung, danke!

Sehr gerne! Hier ist ein Beispiel in Node.js für den Wahlleiter, um die verschlüsselten Stimmen zu entschlüsseln, die aus der Blockchain (z. B. via Quorum-Smart-Contract) geladen wurden. Es verwendet den gleichen Algorithmus wie der Client: RSA-OAEP mit SHA-256, umgesetzt mit [node-forge](#).

Voraussetzungen

`npm install node-forge`

Beispielstruktur

election-tally/

├── decrypt.js

├── private.pem

└── votes.json

private.pem (Beispielschlüssel)

Hier muss der private RSA-Schlüssel liegen, der zur Entschlüsselung der Stimmen verwendet wird.

Beispiel: Generiere Schlüsselpaar (für Testzwecke)

`openssl genrsa -out private.pem 2048`

`openssl rsa -in private.pem -pubout -out public.pem`

votes.json (Stimmenliste von Blockchain)

Angenommen, du hast verschlüsselte Stimmen aus der Blockchain exportiert, z. B.:

```
[  
  "uLZqeCJ2NfPStLm5lZryAE7NZuD8ovZJxNiwn1g04gnKvpoL/n7H...==",  
  "dHVwcmVyV2FobGVudGV4dEJhc2U2NEVuY3J5cHRpb24uLi4=="  
]
```

decrypt.js – Wahlleiter-Tool zur Entschlüsselung

```
const forge = require('node-forge');  
const fs = require('fs');  
  
// Private Key laden  
const privateKeyPem = fs.readFileSync('private.pem', 'utf8');  
const privateKey = forge.pki.privateKeyFromPem(privateKeyPem);  
  
// Verschlüsselte Stimmen laden  
const votesBase64 = JSON.parse(fs.readFileSync('votes.json', 'utf8'));  
  
const decryptedVotes = votesBase64.map((base64, index) => {  
  try {  
    const encryptedBytes = forge.util.decode64(base64);  
    const decrypted = privateKey.decrypt(encryptedBytes, 'RSA-OAEP', {  
      md: forge.md.sha256.create(),  
    });  
    return {  
      index,  
      decrypted,  
    };  
  } catch (e) {  
    return {  
      index,  
      error: "Entschlüsselung fehlgeschlagen",  
    };  
  }  
});  
  
// Ergebnisse anzeigen  
console.log("  Entschlüsselte Stimmen:");  
console.table(decryptedVotes);
```

Ausführen

node decrypt.js

Erwartete Ausgabe (bei gültiger Verschlüsselung):

Entschlüsselte Stimmen:

(index)	index	decrypted	
0	0	'Partei A'	

	1		1		'Partei B'	

Zusammenfassung

Mit diesem Node.js-Skript kann der Wahlleiter:

- Die Stimmen als Base64-Strings laden (aus Quorum-Blockchain extrahiert)
- Die RSA-verschlüsselten Stimmen entschlüsseln
- Die Klartexte für die weitere Auswertung bereitstellen

Wenn du willst, kann ich dir auch ein Tool für die automatisierte Auszählung und Aggregation aus diesen Stimmen erstellen. Sag einfach Bescheid!

Anhang:

UML-Diagramm 1:

```
@startuml
actor Wähler
participant "Registrierungsservice" as RS
participant "Blockchain-Service" as BC
participant "Frontend\nWahlclient" as FE
participant "Stimmabgabe-Service" as VoteAPI

== Registrierung ==

Wähler -> RS : Authentifiziere (z.B. eID)
RS -> RS : Prüfe Wahlberechtigung
RS -> RS : Generiere Token
RS -> Wähler : Übermittle Token (z.B. QR-Code)
RS -> BC : Schreibe Hash(Token) in Blockchain\n(Typ: REGISTRATION)

== Stimmabgabe ==

Wähler -> FE : Öffne Wahlclient und gebe Token ein
FE -> FE : Verschlüssele Stimme lokal
FE -> VoteAPI : Sende Token + verschlüsselte Stimme

VoteAPI -> BC : Prüfe, ob Token-Hash registriert ist
VoteAPI -> BC : Prüfe, ob Token schon verwendet wurde
alt Token gültig & nicht verwendet
    VoteAPI -> VoteAPI : Speichere Stimme
    VoteAPI -> BC : Schreibe Hash(Token) in Blockchain\n(Typ: VOTE_CAST)
    VoteAPI -> FE : Bestätige gültige Stimmabgabe
else Ungültiger oder bereits verwendeter Token
    VoteAPI -> FE : Fehler: Ungültiger oder verbrauchter Token
end

@enduml
```