FH MÜNSTER ETI FB Elektrotechnik und Informatik
Department of Electrical Engineering and Computer Science

Fachhochschule Münster

Fachbereich Elektrotechnik und Informatik

Votify - Voting DApp

Autoren Rebecca Zyla
Markus Esmann
Niklas Tasler

Abgabedatum 21. Juli 2022

Betreuer Dipl.-Ing. Sven Seydler
Prof. Dr.-Ing. Peter Glösekötter

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Idee

Bei der Durchführung von Wahlen wurden in den letzten Jahren eine Vielzahl von Ergebnissen gefälscht. Die Wähler müssen sich bei einer klassischen Papierwahl auf die korrekte Auszählung der Stimmen verlassen. Der gesamte Prozess der Auszählung wird von einer unabhängigen und zentralen Instanz durchgeführt, ohne dass sich das Endergebnis von den Wählern verifizieren lässt. Bei Betrugsfällen wurden Stimmen beispielsweise mehrmals abgegeben oder aufgekauft. Länder wie Brasilien, Indien und die USA nutzen E-Wahlen, bei denen sich die Vorwürfe von Wahlmanipulationen häufen. Fälle von falschen Auszählungen und Angriffen auf das System führen dazu, dass die Menschen dem Wahlsystem nicht mehr vertrauen. [Canessane et al., 2019]

Um Fehlzählungen und nachträgliche Manipulationen von Wahlstimmen zu vermeiden, wird eine Anwendung mit transparentem Wahlsystem vorgestellt. Mithilfe der Blockchain-Technologie sind Änderungen des Netzwerks für alle Personen einsehbar. Jede Node, die sich am Blockchain Netzwerk beteiligt, erhält eine Kopie der gesamten Transaktionshistorie in Form eines verteilten Protokolls. In der Blockchain wird deshalb keine zentrale Autorität benötigt, welche die Transaktionen bewilligt oder Operationen ausführt. Die Informationen werden mit der Technologie dezentralisiert und lassen sich so vor Manipulation von Außen schützen. In Ethereum läuft die Blockchain auf selbstausführenden Anwendungen, den *Smart Contracts*. [Canessane et al., 2019]

Die Anwendung *Votify* ist eine dezentralisierte Wahl-Applikation mit elektronischer Stimmabgabe auf der Blockchain. Mehrere Wahlen lassen sich mit zugehörigen Kandidaten in Votify verwalten. Jede wahlberechtigte Person kann eine Stimme bei einer Wahl beantragen, welche nur für einen Kandidaten dieser Wahl verwendet werden kann. Mit der Stimme kann der Wähler selbst abstimmen oder sie an eine andere Person delegieren. Die Delegation ermöglicht dem Wahlberechtigten mehrere Stimmen bei einer Wahl abzugeben. Die Stimmauszählung kann vom Wähler in Echtzeit abgelesen werden. Nach dem Beenden der Wahl gewinnt der Kandidat mit den meisten Stimmen.

2 Anforderungen

Die Benutzergruppen der Anwendung Votify lassen sich in Administratoren und Wahlberechtigte unterteilen. Die Administratoren sollen die Abstimmungen mit den Kandidatenlisten verwalten. Eine wahlberechtigte Person soll eine Stimme beantragen und für einen Kandidaten abgeben können. Die Anforderungen lassen sich in funktionale und nichtfunktionale Anforderungen aufteilen, die im Folgenden vorgestellt werden.

2.1 Funktionale Anforderungen

Die funktionalen Anforderungen der Anwendung Votify werden in Tabelle 2.1 aufgelistet.

Tabelle 2.1: Funktionale Anforderungen

Obligatorisch	Optional
Flexible Erstellung mehrerer Wahlen	Bereitstellung einer Benutzeroberfläche
Eindeutige Zuordnung von Wahlen und den zugehörigen Stimmen	Delegation einer Stimme an einen anderen Wähler
Abgabe einer Stimme nur zu der zugehörigen Wahl möglich	Einsicht der aktuellen Stimmverteilung während einer laufenden Wahl
Hinzufügen von Kandidaten durch den Administrator	Entfernung einer Wahl aus dem aktuellen Zustand
Starten und Beenden einer Wahl durch den Administrator	
Keine Stimmabgabe vor Wahlbeginn	
Keine Stimmabgabe nach Wahlende	
Keine mehrfache Stimmbeantragung zu einer Wahl	
Erstellung mehrerer Smart Contracts	

Die Abbildung 2.1 zeigt, die möglichen Aktionen eines Wählers grafisch. Wie bereits in der tabellarischen Auflistung der Anforderungen vorgestellt, soll ein Wähler eine Stimme einer laufenden Wahl beantragen können. Mit der Stimme soll der Wähler einen Kandidaten der zugehörigen Abstimmung wählen können oder diese an eine andere Person delegieren können.

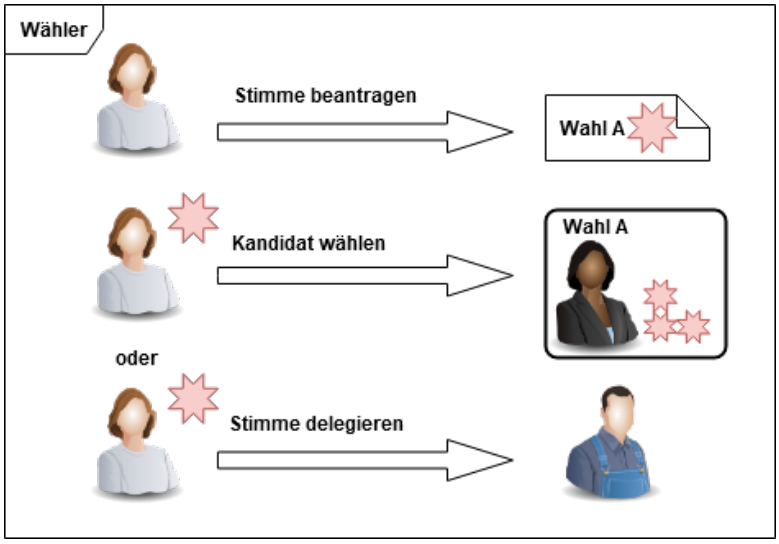


Abbildung 2.1: Benutzeraktionen des Wählers

2.2 Nichtfunktionale Anforderungen

Da zu den Benutzern der Anwendung auch Personen mit geringen Technikkenntnissen zählen, wird die Anwendung möglichst benutzerfreundlich gestaltet. Um die Wähler nicht mit hohen Transaktionskosten zu belasten, die bei jeder Transaktion auf der Blockchain anfallen, und allen eine Teilnahme an der Wahl zu ermöglichen, werden die Kosten so gering wie möglich gehalten. In Tabelle 2.2 werden diese und weitere nichtfunktionale Anforderungen aufgeführt.

Tabelle 2.2: Nichtfunktionale Anforderungen

Obligatorisch	Optional
Einsatz einer Versionsverwaltung	Grafisch ansprechende Benutzeroberfläche
Regelmäßige Projekttreffen	Automatische Tests in CI-Pipeline
Regelmäßige Design-Reviews mit anderen Gruppen	Minimierung der Transaktionskosten
Dokumentation des Projekts	
Spezifizierung eines Coding-Styles	

3 Projektorganisation

Nach der Einführung der Projektidee und der Spezifikation der Anforderungen wird in diesem Kapitel die Organisation zur Realisierung der dezentralen Anwendung Votify betrachtet. Dadurch sollen Arbeitspakete definiert und voneinander abgegrenzt werden, um die einzelnen Aufgaben auf die verschiedenen Projektmitglieder verteilen zu können. Nachfolgend ist diesbezüglich in Abbildung 3.1 der Projektplan dargestellt, der beim Kick-Off entworfen und auf den Termin der Abschlusspräsentation abgestimmt wurde.

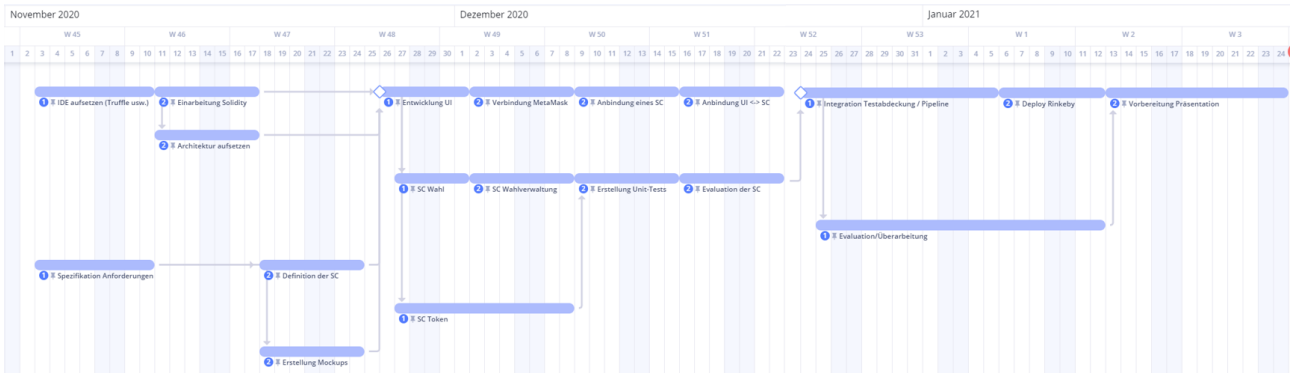


Abbildung 3.1: Projektplan zur Organisation der Arbeitspakete

Zunächst ist dabei zu erkennen, dass die einzelnen Arbeitspakete, bis auf wenige Ausnahmen, auf eine Woche ausgelegt sind. Im Zusammenspiel mit einem wöchentlichen Projektmeeting bestand dadurch die Möglichkeit, einzelne Arbeitsschritte abzuschließen und Probleme rechtzeitig im gesamten Team zu diskutieren. Abschließend ist anzumerken, dass es sich im dargestellten Projektplan um eine initiale Version handelt, die in Abstimmung der Projektmeetings und der Code-Reviews mit einer anderen Produktgruppe regelmäßig angepasst wurde.

Insgesamt ermöglicht es der Projektplan somit die eigentliche Implementierung der Smart Contracts, der Tests und des Frontends zu strukturieren und aufeinander abzustimmen. Darüber hinaus geben die einzelnen Arbeitspakete eine Richtung vor, die innerhalb der Projektmeetings evaluiert und angepasst werden kann.

4 Architektur

In diesem Kapitel wird die Architektur der dezentralisierten Anwendung vorgestellt und einzelne Komponenten detailliert aufgegriffen.

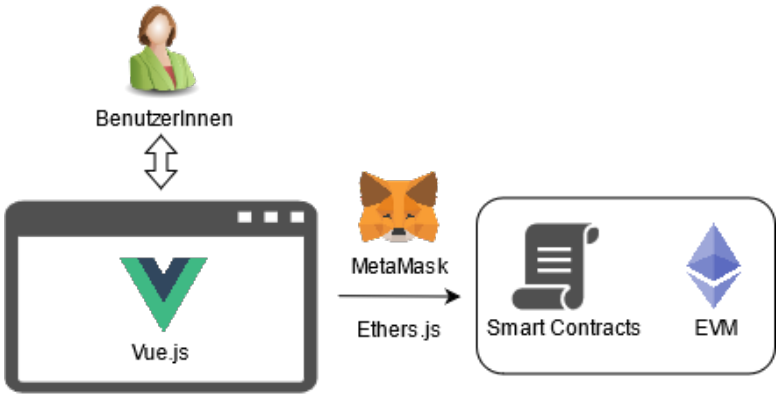


Abbildung 4.1: Architektur von Votify

Das Frontend wird mit dem Framework Vue.js erstellt, das noch genauer in Kapitel 7 vorgestellt wird. Um Transaktionen auf dem Ethereum Netzwerk durchzuführen und mit den Smart Contracts zu interagieren, wird das Webseiten-Plugin MetaMask und die Bibliothek ethers.js verwendet. Mithilfe der EVM (Ethereum Virtual Machine) wird der Code der Smart Contracts auf die Blockchain transformiert.

4.1 Ethereum Virtual Machine

Ethereum basiert auf der Blockchain-Technologie, mit der Entwickler dezentrale Anwendungen (dApps) bereitstellen können. Die dApps werden nach dem Hochladen so ausgeführt, wie sie programmiert wurden. Bei den einzelnen Transaktion führt jeder Knoten im Netzwerk Funktionen des Smart Contracts im Austausch für etwas Ether, der Währung von Ethereum, aus. Diese Transaktionskosten werden als Gas bezeichnet. Die Smart Contracts werden in einem vertrauenslosen System ausgeführt, da diese deterministisch ein Ergebnis ermitteln und keine zentrale Instanz benötigt wird, um Transaktionen zu genehmigen.

Die EVM [1] ist in jedem Knoten des Netzwerks eingebettet. Der Code, der auf der EVM läuft, ist vollständig von dem Netzwerk, Dateisystem und Prozessen des Hostsystems isoliert. Die Smart Contracts werden typischerweise in Solidity geschrieben und von der EVM zu Bytecode kompiliert, der von den Nodes gelesen und ausgeführt wird.

4.2 MetaMask und Ethers.js

MetaMask[2] ist eine Erweiterung für den Webbrowser, welche die Verwaltung von Cryptocurrencies mit einer Wallet ermöglicht. Mit MetaMask kann eine sichere Verbindung zu Blockchain-basierten Anwendungen aufgebaut werden, um Cryptocurrencies und Token zu verkaufen, zu tauschen und zu versenden. Nur Anwender haben Zugriff auf die Konten und Daten, da MetaMask Passwörter und Schlüssel auf dem Endgerät generiert. Ein Anwender erhält bei der Erstellung einer Wallet einen privaten und einen öffentlichen Schlüssel. Während der private Schlüssel Zugriff auf alle Cryptocurrencies und Token in der Wallet gibt und geheim bleiben soll, wird mit dem öffentlichen Schlüssel die Adresse generiert, die bei Transaktionen verwendet wird.

Die Bibliothek Ethers.js[3] zielt darauf ab, eine vollständige Bibliothek für die Interaktion mit der Ethereum Blockchain zu stellen. Dafür bietet Ethers weitere Komfortfunktionen und vereinfacht so die Kommunikation mit der Ethereum Blockchain.

4.3 Smart Contracts

Im Folgenden werden die implementierten Smart Contracts `Election`, `VoteToken` und `ElectionFactory` grob vorgestellt. Votify verwendet Produkte von OpenZeppelin[4], um die dezentrale Anwendung sicherer zu gestalten und die Implementierung zu vereinfachen und zu standardisieren. OpenZeppelin stellt unter anderem Smart Contracts und Interfaces für das Ethereum Netzwerk bereit. Die Produkte sind in der Programmiersprache Solidity geschrieben und sichern den Aufbau, die Automatisierung und den Betrieb von Anwendungen.

Jeder Wahl sind Stimmen mit gleicher Identifikationsnummer zugeordnet. Eine Stimme wird als Token umgesetzt. Ein Token[5] kann die Darstellung von Geld, einem virtuellen Tier oder irgendetwas anderem in der Blockchain sein. Token interagieren mit Smart Contracts und können erstellt, ausgetauscht und zerstört werden. Für verschiedene Wahlen werden Token mit

[1] `https://ethereum.org/en/developers/docs/evm/`
[2] `https://metamask.io/`
[3] `https://docs.ethers.io`
[4] `https://openzeppelin.com/`
[5] `https://docs.openzeppelin.com/contracts/3.x/tokens`

unterschiedlichen IDs angelegt. Die Stimmen einer Wahl sind somit gleich, während sich Stimmen von unterschiedlichen Wahlen in ihrer ID unterscheiden und nicht austauschbar sind.

Mit dem Smart Contract `VoteToken` lassen sich die Stimmen der Wahlen verwalten. Für jede Abstimmung stellt der `VoteToken` Vertrag Funktionen bereit, um neue und eindeutig zugeordnete Stimmen zu erstellen. Ein Element des VoteTokens beinhaltet eine ID mit der sich das Token zu einer Wahl zuordnen lässt. Um multiple Token in einem Smart Contract zu verwalten, erbt der Contract von der Klasse `ERC1155`. Die Klasse stellt Funktionen bereit, um Token mit verschiedenen IDs zu erstellen und diese zu versenden. Des Weiteren lässt sich für eine Adresse die Anzahl der im Besitz stehenden Token einer ID ausgeben. Als Datentyp für die ID Generierung von Token wurde der Zähler `Counters` von OpenZeppelin verwendet, der nur inkrementiert und dekrementiert werden kann.

Der Smart Contract `Election` umfasst die Funktionalitäten einer Wahl. Zur Wahl gehört eine Kandidatenliste, die um Kandidaten erweitert werden kann, solange die Wahl noch nicht gestartet wurde. Für jeden Kandidaten wird die Anzahl der bereits dafür abgegebenen Stimmen gespeichert. Eine Wahl kann in dem Zustand *Kreiert*, *Geöffnet* oder *Geschlossen* sein. Nur eine offene Wahl kann Token als Stimmen annehmen. Um Stimmen des Smart Contracts `VoteToken` zu empfangen, erbt `Election` von dem `ERC1155Receiver`. Der OpenZeppelin Contract bietet Funktionen um Token eines `ERC1155` Contracts zu erhalten.

Die `ElectionFactory` ist ein Smart Contract, welcher die Verwaltung mehrerer Wahlen vereinfacht. Mit dem Contract lassen sich Wahlen mit zugehörigen Token erstellen und aus dem aktuellen Zustand der Blockchain entfernen. Dieser Smart Contract greift dabei auf Funktionen der Smart Contracts `Election` und `VoteToken` zu.

5 Realisierung der Smart Contracts

Nach der Spezifikation der Anforderungen und einer Einführung der Architektur folgt in diesem Kapitel die eigentliche Realisierung der Smart Contracts auf Basis der objektorientierten Programmiersprache *Solidity*. Die Smart Contracts bilden hierbei die zuvor definierten Funktionalitäten auf der Blockchain ab und stellen eine Schnittstelle für die anwendungsspezifische Interaktion zur Verfügung. In diesem Zusammenhang besteht für die clientseitige Applikation die Möglichkeit, den aktuellen Status der Blockchain abzufragen oder diesen durch bezahlte Transaktionen zu verändern.

Inhaltlich wird die Realisierung der einzelnen Smart Contracts anhand der verwendeten Designmuster und der Umsetzung der Architektur erläutert. Dies beinhaltet sowohl den Entwicklungsprozess als auch die finale Implementierung.

5.1 Erstellung und Verwaltung der Wahlen

Damit die Nutzer ihre Stimme abgeben können, muss der Administrator im ersten Schritt die Wahl mit den zugehörigen Kandidaten anlegen und starten. Bei diesen Anforderungen steht zunächst die Fragestellung im Vordergrund, wie die einzelnen Wahlen auf der Blockchain erstellt und verwaltet werden. Neben einer Erläuterung der Implementierung und einer Evaluation der zugehörigen Vor- und Nachteile beinhaltet dieser Abschnitt dabei einen Vergleich zu alternativen Lösungsansätzen. Dies soll es ermöglichen, die ausgewählten Konzepte qualitativ zu bewerten und auf neue Anwendungsfälle zu übertragen.

Gemäß der Architektur aus dem vorherigen Kapitel wird die Funktionalität einer Wahl in dem Smart Contract `Election` definiert. Letzterer kann dabei individuell nach dem jeweiligen Abstimmungsverfahren implementiert werden. Eine Besonderheit dieser Lösung ist hierbei, dass für jede Wahl eine neue Instanz des zugehörigen Smart Contracts auf der Blockchain instanziiert wird. Damit die Nutzer beziehungsweise die clientseitige Applikation jedoch die einzelnen Wahlen nachverfolgen können, müssen die zugehörigen Adressen zentralisiert und von außen zugänglich verwaltet werden. Hierfür dient der Smart Contract `ElectionFactory`, der gemäß dem Entwurfsmuster der Fabrikmethode entwickelt wurde und somit zusätzlich die Instanziierung der einzelnen Wahlen für den Administrator kapselt.

Im Vergleich zur Verwaltung aller Wahlen in einem gemeinsamen Smart Contract hat der dargestellte Lösungsansatz den Vorteil, dass die Datenstrukturen in den Verträgen durch die entsprechende Aufteilung vereinfacht werden. Dies ist vor allem dann entscheidend, wenn der ursprüngliche Smart Contract die maximale Größe von 24 KB überschreitet.

Dennoch ergibt sich das Problem, dass bei jeder Wahl zu einem Großteil immer wieder der gleiche Bytecode des Smart Contracts, wie zum Beispiel die Implementation der Funktionen, bereitgestellt und veröffentlicht wird. Dadurch entstehen unnötige Gaskosten für den entsprechenden Speicherbedarf auf der Blockchain. Aus diesem Grund instanziiert die `ElectionFactory` bei jeder Wahl lediglich einen sogenannten Proxyvertrag. Die zugehörige Architektur ist nachfolgend in Abbildung 5.1 dargestellt.

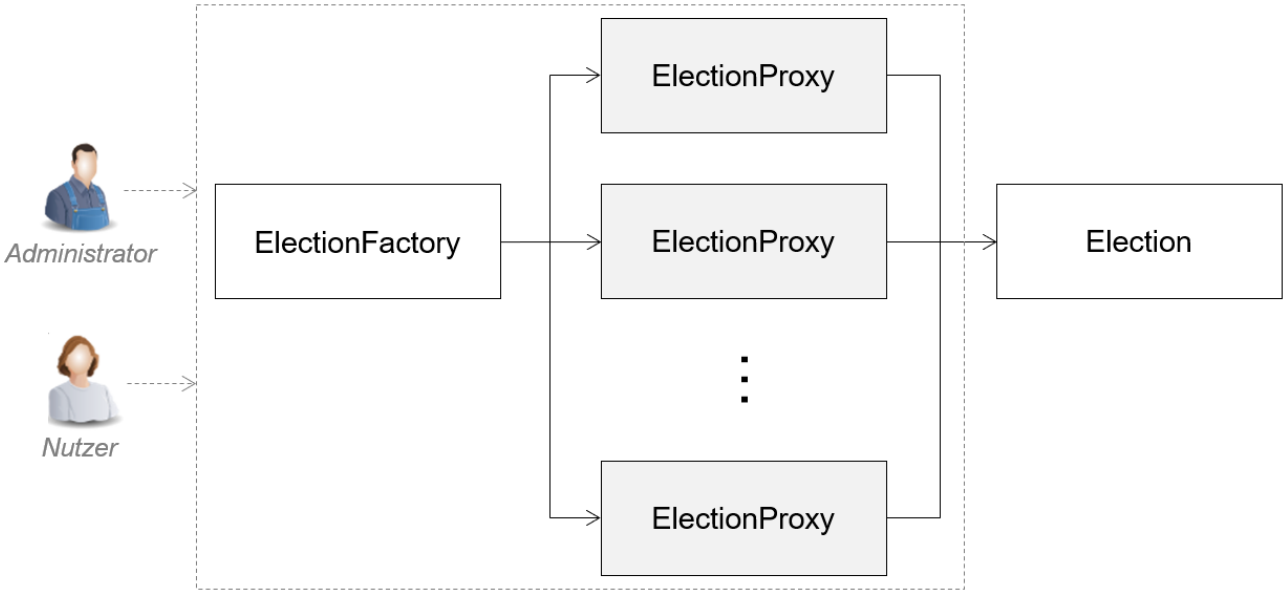


Abbildung 5.1: Architektur zur Verwaltung der Wahlen (vgl. [Lifanova, 2020])

Aus Sicht der Nutzer und Administratoren kapseln die Proxyverträge hierbei die eigentliche Realisierung des Abstimmungsverfahrens. Letztere leiten dabei alle Funktionsaufrufe per *Delegatecall* an den Smart Contract `Election` im Kontext der jeweiligen Wahl weiter. Zusammenfassend beinhaltet somit jeder Proxyvertrag seinen eigenen Status und verwendet die eigentliche Implementation als Bibliothek. [Lifanova, 2020] Dadurch wird es ermöglicht, den Speicherbedarf und die Gaskosten bei der Erstellung einer neuen Wahl zu verringern.

Darüber hinaus hat dieser Lösungsansatz im Vergleich zur klassischen Fabrikmethode noch ein weiteren Vorteil. Demnach kann durch die dargestellte Kapselung die Implementation des Abstimmungsverfahrens jederzeit ausgetauscht werden. Hierfür ist lediglich die im Proxy und der *ElectionFactory* hinterlegte Adresse über eine entsprechende Methode anzupassen. Somit ergibt sich insgesamt eine gewisse Flexibilität, die mit einem einzelnen Smart Contract nicht erreicht werden kann.

Für die Erstellung der Proxyverträge wurde in diesem Projekt die `CloneFactory`[1] verwendet, die von Optionality.io bereitgestellt wird. Diese definiert einen festen Bytecode, der alle Aufrufe an die zu übergebende Adresse weiterleitet. Abschließend ist anzumerken, dass zusätzlich die Möglichkeit besteht, diese Funktionalität beziehungsweise den zugehörigen Bytecode mithilfe der Vorlage `UpgradableProxy`[2] von OpenZeppelin selbst zu erstellen.

Zusammenfassend konnte mit der dargestellten Lösung somit eine flexible Architektur erstellt werden, die es ermöglicht, die Datenstrukturen in den Verträgen zu vereinfachen und die Abstimmungsverfahren nachträglich anzupassen. Dennoch sollte beachtet werden, dass dies mit zusätzlichem Overhead, wie zum Beispiel der Erstellung neuer Verträge und damit verbundenen Gaskosten, einhergeht.

5.2 Konzepte der Datenhaltung

Im nächsten Schritt wird die Datenhaltung innerhalb der einzelnen Smart Contracts betrachtet. Ziel ist es dabei, die Performance zu optimieren und die Gaskosten bei einer Transaktion, wie zum Beispiel dem Hinzufügen eines neuen Kandidaten, zu minimieren.

Als primäre Datenstruktur wird in diesem Projekt das sogenannte Mapping verwendet, das Schlüssel-/Wert-Paare beinhaltet und es ermöglicht, Objekte in konstanter Zeit zu extrahieren. Dagegen müssen bei einem Array im schlechtesten Fall alle vorhandenen Elemente durchsucht werden, wodurch die Laufzeit und die Gaskosten proportional zur Größe des Datensatzes ansteigen.

Darüber hinaus hat das Mapping jedoch die Einschränkung, dass es nicht möglich ist, über die einzelnen Datensätze zu iterieren. Da dies allerdings eine wichtige Voraussetzung für die Visualisierung innerhalb der clientseitigen Applikation darstellt, wird jedem Mapping zusätzlich ein Array zugeordnet, das alle verwendeten Schlüssel beinhaltet. Listing 5.1 zeigt die entsprechenden Datenstrukturen am Beispiel der Kandidaten einer Wahl.

```
bytes32[] public candidateNames;
mapping(bytes32 => Candidate) public candidates;
```

Listing 5.1: Konzept zur Datenhaltung am Beispiel der Kandidaten einer Wahl

Darauf aufbauend kann die clientseitige Applikation mit einer Abfrage der Feldgröße über das Array iterieren und im Anschluss die einzelnen Elemente aus dem Mapping abfragen. Da es sich hierbei um einen lesenden Zugriff handelt, werden auch keine Gaskosten verbraucht. Dagegen hat der Smart Contract auch innerhalb einer Transaktion und bei Übergabe des jeweiligen

[1] `https://github.com/optionality/clone-factory`
[2] `https://docs.openzeppelin.com/contracts/3.x/api/proxy`

Schlüssels die Möglichkeit, direkt das gewünschte Element aus dem Mapping zu extrahieren, sodass die Gaskosten unabhängig von der Anzahl der Datensätze konstant bleiben.

Neben dem Speichermanagement wird als nächstes ein Konzept für die Aktualisierung der clientseitigen Applikation benötigt. Solidity stellt hierfür Events bereit, die innerhalb des jeweiligen Smart Contracts im Rahmen einer Transaktion und einer Veränderung des Datenbestands aktiviert werden können. Dadurch bekommen alle Nutzer, die dieses Event abonniert haben, eine Benachrichtigung und können umgehend ihren aktuellen Status oder die zugehörige Visualisierung anpassen. Insgesamt ermöglicht es dieses Vorgehen somit eine zyklische Abfrage des Smart Contracts zu vermeiden.

Zusammenfassend konnte mit den erläuterten Datenstrukturen und dem Konzept der Events eine Schnittstelle bereitgestellt werden, die es der clientseitigen Applikation ermöglicht, alle vorhandenen Datensätze zu iterieren, über Änderungen benachrichtigt zu werden und die Gaskosten bei Transaktionen unabhängig von der Größe der beteiligten Datensätze konstant zu halten.

5.3 Zustandsabhängige Transaktionen

Als nächstes wird für die erfolgreiche Durchführung einer Wahl ein Mechanismus benötigt, der sicherstellt, dass ausgewählte Transaktionen nur unter bestimmten Bedingungen aufrufbar sind. So darf die jeweilige Stimme beispielsweise nur abgegeben werden, wenn die Wahl bereits durch den Administrator freigeben und noch nicht beendet wurde.

Zur Realisierung der dargestellten Anforderung wird ein zustandsbasiertes Vorgehen verwendet, das zu jedem Zeitpunkt die Phase der jeweiligen Wahl definiert. Nachfolgend ist in Abbildung 5.2 das zugehörige Diagramm dargestellt, das die einzelnen Zustände und Übergänge modelliert.

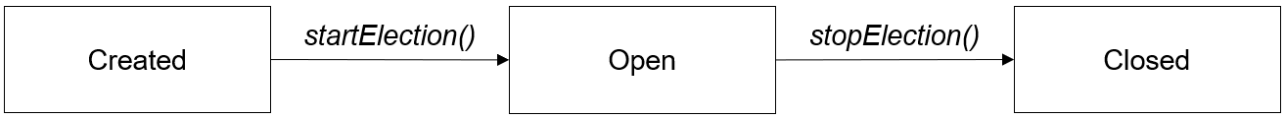


Abbildung 5.2: Zustandsdiagramm einer Wahl

Demnach wird zwischen einer erstellten, einer offenen und einer geschlossenen Wahl unterschieden. Des Weiteren kann der Administrator mithilfe der Transaktionen `startElection` und `stopElection` zwischen den einzelnen Zuständen wechseln. Wichtig zu beachten ist hierbei, dass es sich um unidirektionale Verbindungen handelt und es daher nicht möglich ist, in eine vorherige Phase der Wahl zurückzukehren.

Darauf aufbauend muss im nächsten Schritt das erläuterte Modell mithilfe von Solidity im Smart Contract abgebildet werden. Die einzelnen Zustände werden dabei gemäß Listing 5.2 innerhalb einer Enumeration definiert.

```
enum ElectionState { Created, Open, Closed }
ElectionState public currentState;
```

Listing 5.2: Modellierung der Zustände einer Wahl

Des Weiteren wird die aktuelle Phase der Wahl innerhalb der Variable `currentState` festgehalten. Dadurch kann zu Beginn einer Transaktion geprüft werden, ob die jeweilige Aktion, wie zum Beispiel die Abgabe einer Stimme, im aktuellen Zustand durchgeführt werden darf. Insgesamt wird es durch die dargestellte Realisierung somit ermöglicht, die aktuelle Phase der Wahl nach außen weiterzugeben und die Validierung innerhalb des Smart Contracts zu vereinfachen.

5.4 Stimmabgabe über den VoteToken

Mit den bisher eingeführten Konzepten besteht für den Administrator die Möglichkeit, eine Wahl innerhalb einer eigenen Smart Contract Instanz zu erstellen, Kandidaten der erläuterten Datenstruktur hinzuzufügen und die Abstimmung über eine Zustandsänderung für die einzelnen Nutzer freizugeben. In diesem Abschnitt wird darauf aufbauend die Realisierung und die Einbindung des *VoteToken* betrachtet.

Inhaltlich stellt der *VoteToken* hierbei eine eigens definierte Legitimation dar, die sicherstellen soll, dass nur autorisierte Wähler, die einen entsprechenden Token erhalten haben, an der jeweiligen Wahl teilnehmen können. Die zugehörige Implementierung basiert dabei auf dem *ERC-1155*[3] Standard. Letzterer definiert eine Schnittstelle, mit der mehrere Tokentypen mit eigenen Metadaten und Attributen innerhalb eines einzelnen Smart Contracts verwaltet werden können. Zur Unterscheidung wird bei jeder Transaktion eine *_id* übergeben, die den Tokentyp eindeutig spezifiziert. [Radomski, 2018] Insgesamt ermöglicht dieser Standard somit eine hohe Flexibilität, um verschiedenste Konzepte innerhalb eines einzelnen Smart Contracts zu verwalten.

Im vorliegenden Projekt wird die dargestellte Eigenschaft verwendet, um für jede neue Wahl einen eigenen Tokentyp zu definieren und dadurch die Stimme eindeutig zuzuordnen. Damit hierbei nicht ein Nutzer mehrfach abstimmen kann, ist innerhalb des *VoteToken* wichtig zu beachten, dass jede Person pro Wahl nur eine Stimme erhält. In einem realen Anwendungsfall könnte diese Aufgabenstellung alternativ durch die zuständige Behörde übernommen werden. Letztere müsste dann die entsprechenden Kriterien, die zu einer Teilnahme berechtigen, selbständig prüfen.

[3] `https://docs.openzeppelin.com/contracts/3.x/erc1155`

Nach der Einführung des `VoteToken` steht nun die Interaktion mit den anderen Smart Contracts zur Erstellung eines neuen Tokentyps und zur eigentlichen Abgabe und Delegation der Stimme im Vordergrund. Die zugehörige Verarbeitungskette ist nachfolgend in Abbildung 5.3 skizziert.

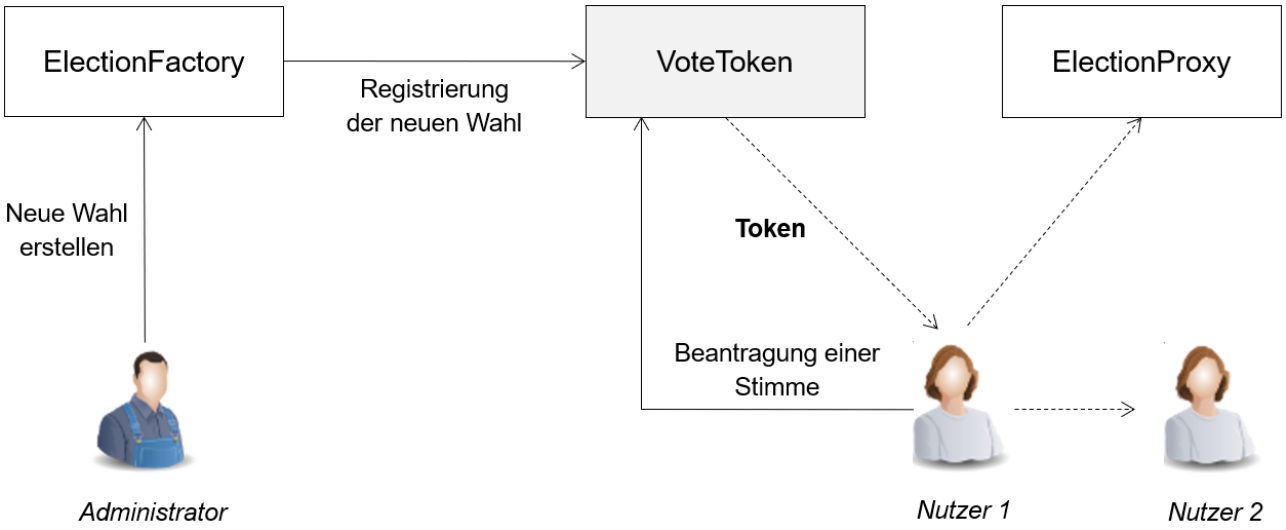


Abbildung 5.3: Verarbeitungskette zur Abgabe/Delegation einer Stimme

Demnach muss der Administrator im ersten Schritt die jeweilige Wahl bei der *ElectionFactory* anlegen. In diesem Kontext wird ein neuer Proxyvertrag instanziiert und die zugehörige Adresse einer internen Liste, die von den Nutzern abgefragt werden kann, hinzugefügt. Des Weiteren beinhaltet dieser Verarbeitungsschritt die Registrierung der neuen Wahl beim Smart Contract des `VoteToken`. Letzterer erstellt gemäß der ERC1155 Spezifikation eine neue ID für den entsprechenden Tokentyp und verknüpft diesen mit der zugehörigen Adresse des Proxyvertrags.

Mit den bisherigen Schritten wurden alle Voraussetzungen geschaffen, damit ein Nutzer seine Stimme beantragen kann. Zum aktuellen Zeitpunkt wird dabei mithilfe des Attributs `msg.sender` geprüft, dass jeder nur eine Stimme pro Wahl erhält. Letzteres kann allerdings mit zusätzlichen Kriterien, wie zum Beispiel einer Gruppenzugehörigkeit, erweitert werden. Sofern in diesem Zusammenhang alle Validierungen erfolgreich waren, wird ein neuer Token der Wahl erstellt und an den jeweiligen Nutzer transferiert.

Sobald der Nutzer seine Stimme für einen Kandidaten abgeben möchte, sendet die clientseitige Applikation den Token mithilfe der Funktion `safeTransferFrom` direkt an den zugehörigen Proxyvertrag der jeweiligen Wahl. Damit dieser allerdings Token empfangen kann, muss die Schnittstelle `IERC1155Receiver`[4] aus Listing 5.3 implementiert werden.

```
onERC1155Received(operator, from, id, value, data)
onERC1155BatchReceived(operator, from, ids, values, data)
```

Listing 5.3: Schnittstelle zum Empfang von Token des ERC1155-Standards

[4] `https://docs.openzeppelin.com/contracts/3.x/api/token/erc1155#IERC1155Receiver`

Innerhalb der Funktion `onERC1155Received`, die beim Empfang eines einzelnen Tokentyps aufgerufen wird, ist dabei zunächst zu überprüfen, ob die ID mit der der jeweiligen Wahl übereinstimmt. Darüber hinaus muss der Smart Contract als Empfänger wissen, welchem Kandidaten die Stimme zugeordnet werden soll. Hierfür dient das Bytearray `data`, in dem die ID des jeweiligen Kandidaten abgelegt wird. Sofern die erläuterten Parameter korrekt übergeben wurden und die Transaktion erfolgreich war, befindet sich der Token im Besitz des jeweiligen Proxyvertrags und die Stimme des jeweiligen Nutzers kann nicht erneut verwendet werden.

Abschließend soll es gemäß der einleitenden Anforderungsanalyse möglich sein, den beantragten Token an einen anderen Nutzer zu delegieren. Hierfür wird die Transferfunktion des ERC1155 Standards verwendet, mit der ein Token an eine andere Adresse geschickt werden kann. Anschließend besteht für den Empfänger die Möglichkeit, diesen Token für die jeweilige Abstimmung zu verwenden.

Zusammenfassend ermöglicht es der Ansatz über den eigenen Token, die Ausgabe der Stimmen und die damit verbundenen Kriterien vollständig von der eigentlichen Durchführung der Wahl zu entkoppeln. Somit ergibt sich eine höhere Flexibilität, um beispielsweise die Kriterien für eine Teilnahme an der Abstimmung zu verändern. Des Weiteren kann dadurch die Delegation einer Stimme auf den Transfer eines Tokens beschränkt werden.

5.5 Entfernung einer abgeschlossenen Wahl

Abschließend soll der Administrator die Möglichkeit haben, eine bereits beendete Wahl zu entfernen, sodass die Abstimmungsergebnisse nicht mehr eingesehen werden können. Des Weiteren ist es dadurch das Ziel, weitere Transaktionen der zugehörigen Token zu verhindern. Im folgenden Abschnitt wird die Vorgehensweise zur Realisierung dieser Anforderung erläutert.

Der erste Schritt beinhaltet hierbei die Entfernung des Proxyvertrags aus der Datenstruktur aller Wahlen innerhalb der `ElectionFactory`. In diesem Kontext ist allerdings wichtig zu beachten, dass es aufgrund der Eigenschaften einer Blockchain nicht möglich ist, Daten bereits abgeschlossener Blöcke zu löschen. Aus diesem Grund wird beim Schlüsselwort `delete` der Speicher im aktuellen Zustand als unbrauchbar markiert und auf den jeweiligen Standardwert zurückgesetzt. Da dieses logische Löschen allerdings keine Neuorganisation beinhaltet, können innerhalb von Feldern sogenannte Lücken auftreten, die bei anderen Funktionen oder Transaktionen zu Problemen führen. Deshalb wird in diesem Fall das letzte Element an die Position der jeweiligen Lücke verschoben und im Anschluss die Größe des Feldes um eins verringert.

Mit den bisherigen Aktionen konnte die Adresse des Proxyvertrags aus der Liste entfernt werden, sodass die zugehörige Wahl von der clientseitigen Applikation nicht mehr angezeigt wird. Aktuell ist es jedoch weiterhin möglich, auf den eigentlichen Smart Contract zuzugreifen, um Daten

abzufragen oder Transaktionen durchzuführen. Aus diesem Grund wird im nächsten Schritt auch der Proxyvertrag mithilfe der Routine `selfdestruct` aus dem aktuellen Zustand der Blockchain entfernt, sodass keine Interaktion mehr möglich ist.

Zum Abschluss der Transaktion ist die Wahl beziehungsweise die zugehörige ID aus dem Smart Contract des *VoteToken* zu entfernen. Dadurch soll sichergestellt werden, dass die Nutzer keine Möglichkeit mehr haben, ihren Token zu beantragen.

Insgesamt hat die dargestellte Vorgehensweise gezeigt, dass es nicht möglich ist, die Historie der zugehörigen Wahl zu entfernen. Durch entsprechende Anpassungen im aktuellen Zustand können allerdings zukünftige Transaktionen verhindert werden, die unnötig Speicherplatz auf der Blockchain und Gaskosten der Nutzer verbrauchen.

6 Qualitätssicherung

Nach der Realisierung der einzelnen Smart Contracts wird darauf aufbauend eine automatisierte Qualitätssicherung benötigt. Dadurch soll die korrekte Funktionalität gemäß der zu Beginn definierten Anforderungen sichergestellt werden. Gerade im Zusammenhang mit dezentralen Anwendungen auf der Blockchain ist dies ein wichtiger Aspekt, da es gemäß Abschnitt 5.5 mit erhöhtem Aufwand und entsprechenden Gaskosten verbunden ist einen bereits veröffentlichten Vertrag im Nachhinein zu ersetzen. Des Weiteren besteht durch diese Validierung und die damit verbundene Gewährleistung einer funktionsfähigen Schnittstelle die Möglichkeit, den Entwicklungsprozess der clientseitigen Applikation zu vereinfachen.

Inhaltlich ist es im folgenden Kapitel das Ziel, eine Architektur zum Test der einzelnen Smart Contracts in das vorhandene Projekt zu integrieren. In diesem Zusammenhang werden die Technologien und die Vorgehensweise der Implementierung dargestellt.

6.1 Komponententests

Die automatisierte Validierung der einzelnen Smart Contracts beinhaltet im ersten Schritt die Entwicklung von Unit-Tests. Letztere ermöglichen es, die einzelnen Funktionen unter verschiedenen Zuständen und Parametrisierungen aufzurufen und anschließend das Ergebnis beziehungsweise den Folgezustand im Vergleich mit den Anforderungen zu überprüfen. Implementiert wurden die einzelnen Tests hierbei in JavaScript mit dem Framework Mocha[1], das von Truffle bereitgestellt und auf Basis von Node.js ausgeführt wird.

Um die Vorgehensweise der Entwicklung zu veranschaulichen, sind in Listing 6.1 zwei beispielhafte Testfälle abgebildet. Dabei wird der Start einer Wahl, bei der zuvor bereits Kandidaten definiert wurden, überprüft.

```
describe("start initialized election with candidates", async () => {
    it("normal user failes", async () => {
      try{
        await this.election.startElection({ from: this.user });
```

[1] https://mochajs.org/

```
        assert.fail("exception excepted");
      }
      catch(error){
        assert.include(error.message, "function only for owner",
        "error message with wrong value");
      }
    })
    it("owner successful", async () => {
      const transaction = await this.election.startElection();
      assert.equal(transaction.logs[0].args.newState, 1);
    })
  })
```

Listing 6.1: Testfall zur Start einer mit Kandidaten initialisierten Wahl

Zunächst werden die einzelnen Testfälle einer Funktion oder eines bestimmten Vertragszustands dabei mithilfe des Schlüsselworts `describe` gruppiert. Wichtig anzumerken ist dabei, dass diese Gruppierung keinen Einfluss auf die eigentliche Ausführung der Tests hat. Es ermöglicht allerdings eine beliebige Strukturierung, um dadurch, gerade bei größeren Verträgen, die Übersichtlichkeit für den Entwickler zu erhöhen.

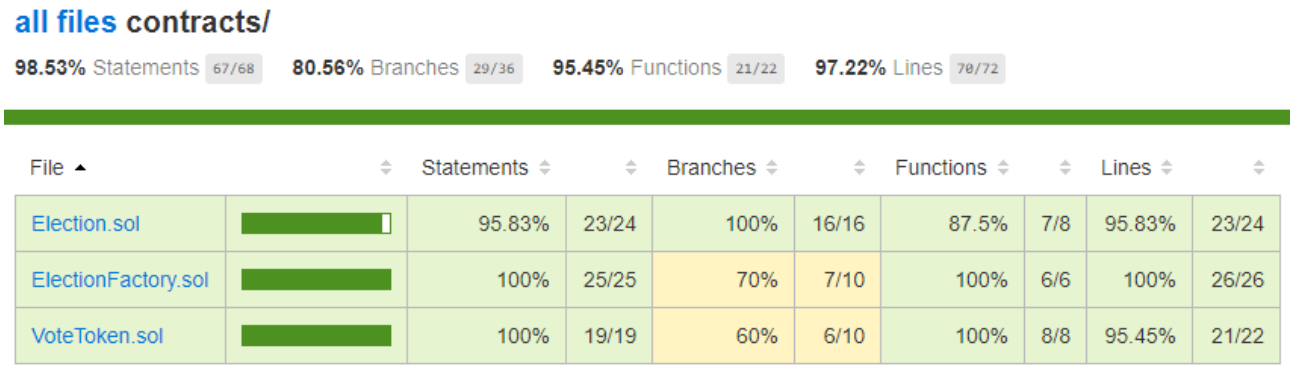
Anschließend können dem zuvor erläuterten Block mithilfe von `it` verschiedene Testfälle untergeordnet werden. Hierbei wird typischerweise der benötigte Datensatz abgefragt oder die zugehörige Transaktion ausgeführt und im Anschluss die Rückgabe mit dem erwarteten Ergebnis überprüft. Im betrachten Beispiel entspricht dies dem Aufruf der Funktion `startElection`, wobei zwischen den Rollen des Nutzers und des Administrators unterschieden wird. Die zugehörigen Tests sind dabei erfolgreich, wenn der normale Nutzer eine Fehlermeldung bekommt und der Zustand der Wahl sich beim Administrator erfolgreich verändert.

Die am Beispiel dargestellte Vorgehensweise kann nun auch auf alle anderen Funktionen der einzelnen Verträge übertragen werden. Da sich dies allerdings immer ähnlich gestaltet, wird auf eine vollständige Betrachtung aller Tests verzichtet.

6.2 Testabdeckung

Nach der Implementierung der einzelnen Tests wird im nächsten Schritt eine Metrik benötigt, mit der bewertet werden kann, ob alle Anforderungen erfolgreich überprüft wurden. Hierfür dient die sogenannte Testabdeckung, die den prozentualen Anteil aller durchlaufenen Programmzeilen und Verzweigungen der Smart Contracts angibt. Letzteres soll dem Entwickler eine Hilfestellung zur Verfügung stellen, um alle möglichen Testfälle abzudecken.

Für die eigentliche Berechnung der Testabdeckung kann das Paket *solidity-coverage*[2] verwendet werden, das über npm dem Truffle-Projekt hinzugefügt wird. Als Ergebnis entsteht dabei eine HTML-Ausgabe, die sich mit einem Browser öffnen und analysieren lässt. In Abbildung 6.1 ist ein beispielhaftes Ergebnis aus dem vorliegenden Anwendungsfall dargestellt.



File		Statements		Branches		Functions		Lines	
Election.sol		95.83%	23/24	100%	16/16	87.5%	7/8	95.83%	23/24
ElectionFactory.sol		100%	25/25	70%	7/10	100%	6/6	100%	26/26
VoteToken.sol		100%	19/19	60%	6/10	100%	8/8	95.45%	21/22

Abbildung 6.1: Ergebnis der Testabdeckung

Aus diesem Diagramm lässt sich direkt ablesen, dass ein großer Prozentsatz der Programmzeilen und Verzweigungen mit den entwickelten Tests bereits abgedeckt werden konnte. Darüber hinaus besteht mit dieser Analyse die Möglichkeit, die Stellen im Quellcode der Smart Contracts zu bestimmen, die von den Tests noch nicht durchlaufen wurden.

Zusammenfassend stellt die Bestimmung der Testabdeckung dem Entwickler somit ein hilfreiches Werkzeug zur Verfügung, um iterativ alle möglichen Testfälle zu bestimmen und anschließend mithilfe von JavaScript und Mocha zu implementieren. Des Weiteren kann dadurch in Kombination mit einem erfolgreichen Testdurchlauf die Gewährleistung gegeben werden, dass die Schnittstelle gemäß der definierten Anforderungen funktionsfähig ist.

[2] https://www.npmjs.com/package/solidity-coverage

7 Frontend

Neben den Smart Contracts soll den Benutzern eine ansprechende Oberfläche zur Verfügung gestellt werden. Grundsätzlich lassen sich alle Funktionen der Anwendung nutzen, indem man die Smart Contracts direkt - etwa über JSON RPC - anspricht. Alle Funktionalitäten der Applikation werden also über die Smart Contracts bereitgestellt. Da es sich um eine Wahlplattform handelt und die meisten potentiellen Anwender tendenziell wenig bis keine technischen Kenntnisse von Smart Contracts besitzen, stellt eine entsprechende Oberfläche eine essenzielle Anforderung dar.

Damit die Anwendung von möglichst vielen Personen auf verschiedensten Endgeräten genutzt werden kann, liegt die Entscheidung für eine Webanwendung nahe. Für die Erstellung moderner Webanwendungen existieren drei bekannte Javascript-Bibliotheken beziehungsweise Frameworks. Dabei handelt es sich um *Angular*, *React* und *Vue.js*.

Da innerhalb der Gruppe bereits Erfahrungen mit Vue.js vorhanden waren und das Framework eine flache Lernkurve aufweist, wurde die Webanwendung mithilfe von Vue.js geschrieben.

Im Folgenden wird auf die Funktionen, die Architektur und die Kommunikation zwischen Frontend und Smart Contracts eingegangen.

7.1 Funktionen

7.1.1 Administration

Der Administrator der `ElectionFactory` soll in der Lage sein, neue Wahlen zu erstellen. Dies soll über einen Administrationsbereich im Frontend umgesetzt werden. Jeder Besitzer eines `Election` Contracts soll außerdem die erstellte Wahl bequem verwalten können. Neben dem Hinzufügen von Kandidaten gehört dazu das Starten und das Beenden der Wahl.

7.1.2 Stimme beantragen

Die Wahlberechtigung wurde in diesem Projekt über einen eigenen `ERC-1155` Token umgesetzt. Jeder Benutzer, der im Besitz mindestens eines `VoteToken` ist, hat die Möglichkeit seine Stimme

abzugeben. Die Umsetzung mithilfe eines eigenen Tokens hat einen großen Vorteil: Er ermöglicht die Entkopplung von Stimmverwaltung und Stimmabgabe. Eine triviale Möglichkeit ist es den Smart Contract so zu implementieren, dass jede Adresse nur ein einziges mal abstimmen darf. Dazu merkt sich der Smart Contract `Election` die Adressen, die bereits abgestimmt haben. Dies stellt aber eine harte Kopplung zwischen der Stimmabgabe und dem Wahlvertrag dar. Möchte man eine Wahl abbilden, bei der ein Nutzer gegebenenfalls mehr als eine Stimme hat, so wird ein neuer Smart Contract für die Wahl benötigt, welcher diese Anforderung abbildet. Die Umsetzung mithilfe des `VoteToken` erlaubt eine flexible Umsetzung. Besitzt ein Benutzer einen Token, so darf jeder lediglich eine Stimme abgeben. Besitzt ein Wähler mehr als eine Stimme, so kann er auch mehr als eine Stimme abgeben. Die Distribution der `VoteToken` kann dann durch eine dritte Autorität wie etwa das zuständige Ministerium erfolgen.

Da es sich bei dieser Umsetzung lediglich um eine Demo-Anwendung handelt, werden die `VoteToken` nicht durch eine separate Partei ausgegeben. Stattdessen hat jeder Benutzer die Möglichkeit, genau ein Token pro Wahl zu beantragen.

7.1.3 Stimme abgeben

Die wichtigste Funktionalität der Frontend-Anwendung ist die Abgabe der Stimme. Das zuvor beantragte `VoteToken` kann dafür genutzt werden, an der zugehörigen Wahl teilzunehmen. Dabei ist es möglich, entweder direkt für einen Kandidaten zu stimmen oder seine Stimme alternativ zu delegieren. Delegiert man seine Stimme an eine andere Adresse, so kann der zugehörige Wähler entsprechend mehr als eine Stimme abgeben. Der Status der Wahl ist dabei zu jeder Zeit einsehbar. Es lässt sich also während der Wahl nachvollziehen, welcher Kandidat aktuell wie viele Stimmen hat. Je nach Stand der Wahl kann dies kurzfristig weitere Wähler motivieren, ihre Stimme noch abzugeben, um den Wahlausgang zu beeinflussen.

7.2 Architektur

Im Folgenden wird die Architektur der Frontend Anwendung beschrieben. Vue.js erlaubt die Aufteilung einer Website in unterschiedliche Komponenten. So lässt sich eine Website aus mehreren Modulen aufbauen, welche wiederverwendbar sind. Die grundsätzliche Architektur ist in Abbildung 7.1 dargestellt.

Die gesamte Website wird in drei Bereiche eingeteilt. Die Beantragung einer Stimme, die Wahladministration und die Stimmabgabe. Jeder dieser Bereiche stellt eine eigene *View* der Website dar, welche über eine eigene Route verfügt. Die drei Views nutzen wiederum die `Election` Komponente, welche mehrfach wiederverwendet werden kann. Diese Komponente dient zur Darstellung essentieller Informationen wie Name und verfügbare Stimmen der Wahl.

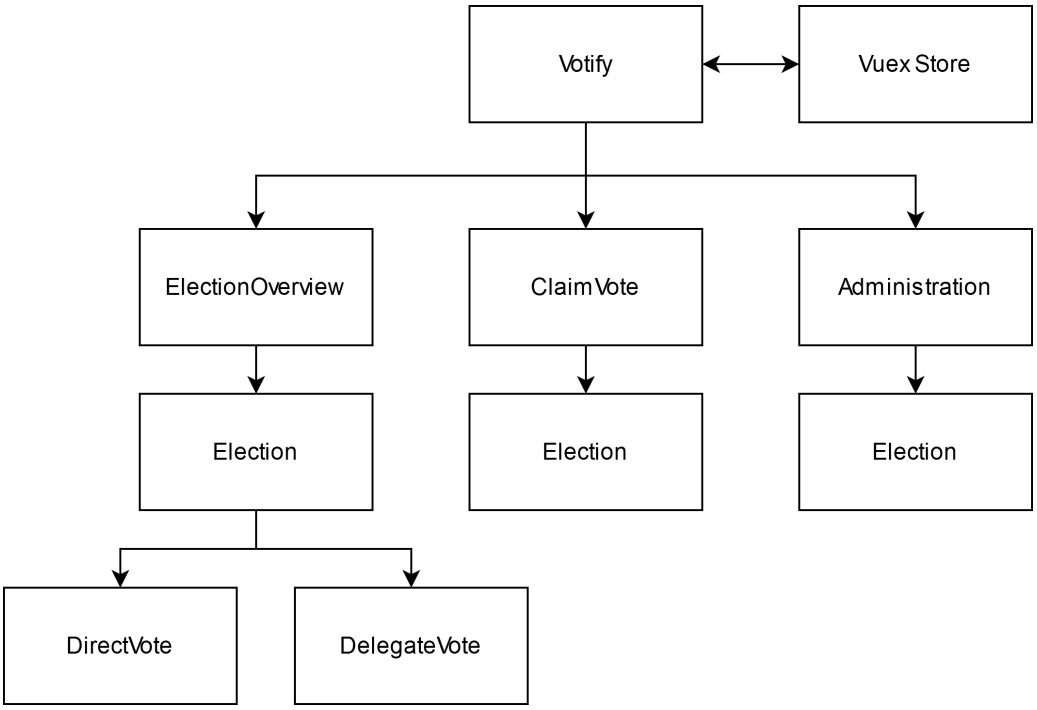


Abbildung 7.1: Architektur des Frontends

Ein wichtiger Punkt jeder modernen Website ist das State-Management. Generell besitzt jede Komponente ihre eigenen Daten. Der Zugriff auf Daten anderer Komponenten ist nicht ohne Weiteres möglich. Da es bei fast jeder Website aber Daten gibt, welche von mehreren Komponenten benötigt werden, existieren in allen bekannten Javascript-Frameworks Bibliotheken für das State-Management. Vue.js wird bereits mit der State-Management Bibliothek `Vuex`[1] ausgeliefert. Diese dient zur Verwaltung einer geteilten Datenbasis. Man spricht auch von einem *Store*, auf welchen alle Komponenten zugreifen können.

Zentraler Bestandteil der Anwendung sind die Wahlen. Eine Übersicht der existierenden Wahlen wird in allen drei Views `ElectionOverview`, `ClaimVote` und `Administration` benötigt. Aus diesem Grund wird etwa die Liste der Wahlen in Vuex gespeichert. Somit haben alle Komponenten Zugriff auf zentrale Informationen, ohne diese bei jedem Aufruf selbst abfragen zu müssen. Diese Architektur verringert auch die Anzahl der Abfragen, die gegen die Blockchain gestellt werden müssen. Weitere zentral gespeicherte Informationen sind etwa die Adressen der Smart Contracts und der Verbindungsstatus zur Blockchain.

Neben dem Javascript-Framework Vue.js wird für ein einheitliches Styling die CSS-Bibliothek *Vuetify*[2] verwendet. Aus der Funktionalität des Wählens (Voting) und der CSS-Bibliothek (Vuetify) ergibt sich auch der Name der Anwendung - *Votify*. Die Bibliothek ermöglicht ein einheitliches Design nach Googles Material Design Standard.

[1] `https://vuex.vuejs.org`
[2] `https://vuetifyjs.com`

7.3 Blockchain-Verbindung

Um mit der Ethereum-Blockchain zu kommunizieren, wird ein *Provider* benötigt. Dabei handelt es sich um eine Abstraktion, die ein Interface bereitstellt, um mit einer Ethereum-Node zu kommunizieren. Es gibt mehrere Wege einen Provider zu nutzen. Typischerweise werden dafür *IPC*, *Websockets* oder *HTTP* genutzt. Ein Provider nimmt JSON-RPC Anfragen entgegen und gibt eine Antwort zurück. [Ethereum Foundation, 2021]

Für die Nutzung der Website wird das Plugin *MetaMask*[3] vorausgesetzt. Dies erlaubt die Verbindung der Website zur Ethereum-Blockchain. Außerdem stellt MetaMask eine Wallet dar, mit der sich Transaktionen signieren lassen. MetaMask injiziert beim Aufruf der Website automatisch einen Provider, mit welchem sich Anfragen an die Blockchain stellen lassen. Es existieren allerdings weitere Bibliotheken, um die Kommunikation mit der Blockchain weiter zu vereinfachen. Die zwei bekanntesten Vertreter sind *web3.js*[4] und *ethers.js*[5].

Während web3.js länger existiert und weit verbreitet ist, erfreut sich ethers.js steigender Popularität. Wenngleich beide Bibliotheken ausreichend für die umzusetzenden Anforderungen sind, hat ethers.js den Vorteil, dass es deutlich kleiner ist. Dies kann die Ladezeit einer Website deutlich verkürzen. Aus diesem Grund wird für diese Website die Bibliothek ethers.js verwendet. Ethers.js nutzt den bestehenden Provider, der von MetaMask bereitgestellt wird, und stellt darauf basierend eine einfache Schnittstelle für die Kommunikation mit der Ethereum-Chain zur Verfügung.

Da für die Nutzung der Website MetaMask benötigt wird, wird das Vorhandensein der Erweiterung beim initialen Aufruf der Website überprüft. In periodischen Abständen werden die wichtigsten Verbindungsdaten abgefragt und im Vuex-Store gespeichert.

```
const state = () => ({
  connected: false,
  error: null,
  address: "",
  network: "",
  balance: "",
});
```

Listing 7.1: Im Store gespeicherte Verbindungsdaten

Listing 7.1 zeigt die zentral gespeicherten Verbindungsinformationen, auf die jede Komponente Zugriff hat. Dazu gehört etwa die Information, ob eine Verbindung mit MetaMask besteht,

[3] https://metamask.io
[4] https://web3js.readthedocs.io
[5] https://docs.ethers.io

die Adresse des verbundenen Accounts und das Netzwerk mit welchem MetaMask verbunden ist.

7.4 Smart Contract Interaktion

Die Kommunikation mit den Smart Contracts wird durch ethers.js stark vereinfacht. Da die Vorgehensweise dabei immer gleich ist, sollen im Folgenden lediglich zwei Aufrufe exemplarisch betrachtet werden.

7.4.1 Lesender Zugriff

Ein zentraler Bestandteil der Anwendung ist das Auslesen aller verfügbaren Wahlen. Damit auch neu angelegte Wahlen zeitnah angezeigt werden, werden alle Wahlen in periodischen Abständen aus dem `ElectionFactory` Vertrag abgefragt. Für die Interaktion mit Smart Contracts bietet ethers.js die `ethers.Contract`[6] Abstraktion. Für die Erstellung eines `ethers.Contract` Objekts existiert folgender Konstruktor:

```
new ethers.Contract(address, abi, signerOrProvider)
```

Listing 7.2: Erstellung einer ethers.Contract Instanz

Für die Angabe der ersten beiden Parameter können die Ausgabedateien des Truffle-Deployments genutzt werden. Kompiliert und verteilt man die Smart Contracts mit Truffle, so wird für jeden Vertrag eine JSON-Datei angelegt. Diese enthält alle Informationen zu den Smart Contracts. Darunter Name, Metadaten und die Adresse des Vertrages auf der Blockchain eines jeweiligen Netzwerks. Da die Adressen der Verträge an weiteren Stellen genutzt werden, werden diese im zentralen Store gespeichert. Auch das *application binary interface* (ABI) ist Teil der von Truffle generierten JSON-Datei. Das Interface beschreibt die Funktionen und Parameter des Vertrags. Der letzte Parameter - `signerOrProvider` - ist entweder ein Provider, oder ein Signer. Möchte man lediglich lesend auf einen Smart Contract zugreifen, so ist die Angabe eines Providers ausreichend. Sollen auch Funktionen eines Vertrages aufgerufen werden, welche etwa den Status des Vertrages ändern können, so ist eine Transaktion notwendig. Eine solche Transaktion muss vorab signiert werden, sodass in diesem Fall die Angabe eines Signers notwendig ist. Da in diesem Projekt auch Funktionen aufgerufen werden und mit MetaMask eine Wallet vorhanden ist, wird für die Aufrufe stets der vorhandene Signer genutzt.

Konkret lässt sich ein Smart Contract Objekt damit wie folgt erstellen:

```
import ElectionFactory from "@/contracts/ElectionFactory.json";
import * as connect from "@/store/ethers/ethersConnect";
```

[6] `https://docs.ethers.io/v5/api/contract/contract/`

```
function getElectionFactoryContract(electionFactoryAddress) {
  return new ethers.Contract(
    electionFactoryAddress,
    ElectionFactory["abi"],
    connect.getWallet()
  );
}
```

Listing 7.3: Erstellung einer ethers.Contract Instanz

Über die Instanz der `ethers.Contract` Klasse lassen sich dann alle Funktionen des Vertrags aufrufen. Konkret lassen sich alle Wahlen in einem zweistufigen Prozess auslesen:

```
async function getElections(electionFactoryAddress) {
  const efContract = getElectionFactoryContract(electionFactoryAddress);

  var electionCount = (await efContract.getElectionCount()).toNumber();
  var electionAddresses = [];
  var i;
  for (i = 0; i < electionCount; i++) {
    electionAddresses.push(await efContract.electionAdresses(i));
  }

  const elections = [];
  for (i = 0; i < electionAddresses.length; i++) {
    var election = await efContract.elections(electionAddresses[i]);
    elections.push({
      address: electionAddresses[i],
      name: ethers.utils.parseBytes32String(election[0]),
    });
  }
  return elections;
}
```

Listing 7.4: Abfrage aller Wahlen mithilfe von ethers.js

Der `ElectionFactory` Vertrag ist für das Deployment einer Wahl verantwortlich. Erstellt er eine neue Wahl, so wird die zugehörige Adresse in einem Array `electionAdresses` gespeichert. Die Informationen der zugehörigen Wahl werden in einem Mapping `elections` gespeichert. Um alle Wahlen auszulesen, wird zunächst die Länge des Arrays ausgelesen, welches alle Adressen

enthält. Mit der Länge, lässt sich dann über das Array iterieren. Die Adressen aller `Election` Smart Contracts werden dabei ausgelesen und gespeichert. In einem zweiten Schritt wird anschließend der Name einer jeden Wahl ausgelesen, indem das Mapping mit jeder Adresse abgefragt wird.

Bei den Rückgabewerten der `ethers.Contract` Aufrufe handelt es sich um *Promises*. Ein Promise stellt den finalen Rückgabewert einer asynchronen Operation dar. Um den finalen Wert zu erhalten wird deshalb jeweils auf die Auflösung der Promises gewartet.

7.4.2 Schreibende Aufrufe

Möchte man echte Funktionen des Vertrages aufrufen, so ist das Signieren einer Transaktion erforderlich. Dafür ist auch eine Interaktion des Benutzers mit MetaMask nötig. An der Vorgehensweise bei der Smart Contract Interation über ethers.js ändert sich hingegen wenig.

Zentraler Bestandteil von Votify ist das Abgeben einer Stimme. Die Berechtigung der Stimmabgabe wird durch das `VoteToken` repräsentiert. Möchte man eine Stimme abgeben, so sendet man ein `VoteToken` zusammen mit dem Kandidaten an den zugehörigen `Election` Contract einer Wahl. Dazu kann die `safeTransferFrom`-Funktion[7] des ERC-1155 Interfaces genutzt werden.

Die Abgabe einer einzelnen Stimme lässt sich damit wie folgt umsetzen:

```
async function vote(electionAddress, voteTokenAddress, candidateName) {
  const voteTokenContract = getVoteTokenContract(voteTokenAddress);
  const electionID = await voteTokenContract.getID(electionAddress);

  return voteTokenContract.safeTransferFrom(
    connect.getWalletAddress(),
    electionAddress,
    electionID,
    1,
    ethers.utils.formatBytes32String(candidateName)
  );
}
```

Listing 7.5: Abfrage aller Wahlen mithilfe von ethers.js

Für jede Wahl gibt es eine Reihe von zulässigen `VoteToken`. Es existiert also eine Verknüpfung zwischen einer konkreten Wahl und einer Menge von zugehörigen `VoteToken`. Somit lässt sich

[7] `https://docs.openzeppelin.com/contracts/3.x/api/token/erc1155#ERC1155-safeTransferFrom-address-address-uint256-uint256-bytes-`

nicht jedes `VoteToken` für jede Wahl nutzen. Möchte man entsprechend bei einer Wahl seine Stimme abgeben, so muss zunächst ermittelt werden, welche Art von `VoteToken` für diese Wahl erforderlich ist. Nur wenn der Benutzer im Besitz eines passenden Tokens für die konkrete Wahl ist, kann er teilnehmen. Aus diesem Grund wird zunächst die ID abgefragt, die der Wahl zugeordnet ist.

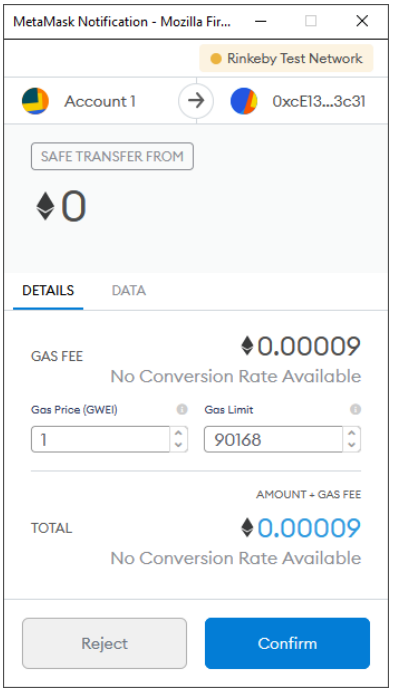


Abbildung 7.2: Bestätitung des `safeTransferFrom` Funktionsaufrufes

Anschließend wird genau ein ERC-1155 Token mit der entsprechenden ID an den Wahlvertrag gesendet. Dabei wird auch der Name des Kandidaten mitgegeben. Nur wenn der Benutzer mindestens einen Token der zugehörigen ID besitzt, kann die Transaktion erfolgreich durchgeführt werden. Da bei diesem Funktionsaufruf auch Gas verbraucht wird, muss dieser freigegeben werden. Entsprechend öffnet sich automatisch eine MetaMask Benachrichtigung, welche das Signieren der Transaktion erfordert (vgl. Abbildung 7.2).

7.4.3 Events

Wird aus dem Frontend eine Funktion des Smart Contracts aufgerufen, so kann der Rückgabewert nicht direkt abgefragt werden. Dies liegt daran, dass lediglich eine Transaktion eingestellt wird, die zeitverzögert gemined wird. Bei einem Funktionsaufruf wird deshalb nur der Hash der Transaktion an das Frontend zurückgegeben. Eine Möglichkeit, Informationen aus Funktionsaufrufen zu erhalten, wenn diese geminded werden, ist die Nutzung von Events. So emittiert der `Election` Smart Contract beispielsweise ein Event, wenn ein Stimme abgegeben wurde. Das Event erhält auch die Information, für welchen Kandidaten gestimmt wurde.

Ethers.js bietet die Möglichkeit sich für bestimmte Events benachrichtigen zu lassen. Diese Funktionalität wird für das Aktualisierungen der Stimmen einer Wahl genutzt. Umsetzen lässt sich dies über die `on`-Funktion[8] einer `ethers.Contract` Instanz.

Konkret lässt sich damit wie folgt ein Listener registrieren:

```
var electionContract = getElectionContract(this.address);
electionContract.on("voteAdded", (candidate) => addVote(candidate));
```

Listing 7.6: Registrierung eines Event-Listeners

Dieses Vorgehen hat auch den Vorteil, dass das Frontend nicht periodisch die Stimmen aller Kandidaten abfragen muss. Stattdessen müssen diese nur ein einziges mal abgefragt werden. Alle weiteren Stimmabgaben werden automatisch über die erhaltenen Events erfasst.

7.4.4 Fazit

Mit Hilfe von Vue.js konnte eine modulare Architektur aufgebaut werden, die sich leicht erweitern lässt. Vorteil eines Frameworks wie Vue.js ist auch die Reaktivität. Änderungen an den Daten werden automatisch erkannt und wirken sich somit unmittelbar auf die UI aus, ohne dass diese manuell aktualisiert werden muss.

Weiter wird für die Kommunikation mit der Ethereum-Blockchain die Bibliothek ethers.js verwendet. Um die Funktionsweise der Bibliothek zu veranschaulichen, wurde an exemplarisch ausgewählten Funktionen gezeigt, wie sich die `ethers.Contract` Abstraktion nutzen lässt, um mit den Smart Contracts zu interagieren. Da sich die Interaktion immer gleich gestaltet, soll auf eine vollständige Betrachtung aller Smart Contract Aufrufe verzichtet werden.

Insgesamt konnten alle geforderten Funktionen (vgl. Abschnitt 7.1) durch die gewählten Technologien Vue.js, ethers.js und MetaMask umgesetzt werden.

[8] `https://docs.ethers.io/v5/api/contract/contract/#Contract-on`

8 Code Reviews

Neben den wöchentlichen internen Projekttreffen wurden auch Code-Reviews mit einer weiteren Gruppe durchgeführt. Da ein zweites Projektteam ebenfalls eine dezentrale Wahlapplikation erstellt hat, bot es sich an, die Code-Reviews mit dieser Gruppe durchzuführen. Da beide Gruppen mit der gleichen Thematik vertraut waren, konnte somit sowohl gutes Feedback gegeben als auch eingeholt werden. Die regelmäßigen Code-Reviews haben zu folgenden Änderungen und Erweiterungen von *Votify* geführt:

VoteToken als Nachweis der Wahlberechtigung

Die ursprüngliche Idee war es, dass das `VoteToken` eine Stimme darstellt, die einem Kandidaten gegeben wird. Wird eine Stimme für einen Kandidaten abgegeben, so wird das `VoteToken` an diesen Kandidaten gesendet. Eine Wahl gewinnt dann der Kandidat, der zum Wahlende im Besitz der meisten Token ist.

Während eine solche Umsetzung durchaus denkbar ist, bringt sie allerdings einen Nachteil mit sich. Es lässt sich nicht zwischen der Delegation und der finalen Abgabe der Stimme unterscheiden. Eine abgegebene Stimme für einen Kandidaten, kann durch diesen theoretisch einfach weiterversendet werden, auch wenn dies nicht im Sinne des Wählers ist.

Auch vor Wahlbeginn und nach Wahlende können die standardisierten Token theoretisch versendet werden. Um den Wahlausgang zu beurteilen, muss entsprechend immer ein historischer Stand der Blockchain betrachtet werden.

Auf Grund dieser Problematik setzt die aktuelle Implementierung den `VoteToken` als Nachweis der Wahlberechtigung ein. Die Abgabe einer Stimme läuft über den `Election` Vertrag, der die Token nachweislich nicht weiterversendet. Die Abgabe einer Stimme stellt somit - analog zur Realität - eine finale Handlung dar. Die aktuelle Umsetzung erlaubt es auch Prüfungen bei der Stimmabgabe vorzunehmen und Events zu emittieren.

Verwendung eines ERC-1155 Tokens

Da die Anwendung mehrere Wahlen unterstützen soll, aber nicht gewollt ist, dass jedes `VoteToken` für jede Wahl verwendet werden kann, sollte ein *Non-fungible Token* verwendet werden. Ein

bekannter Token Standard ist der ERC-721 Standard, der auch etwa vom Projekt *CryptoKitties*[1] verwendet wird. Jedes einzelne Token besitzt eine eigene ID und ist somit nicht durch ein beliebiges anderes Token austauschbar, wie es etwa bei dem ERC-20 Token der Fall ist.

Der Nachteil eines ERC-721 Tokens ist, dass sich mehrere `VoteToken` auch dann voneinander unterscheiden, wenn sie der gleichen Wahl zugeordnet sind. Prinzipiell sollten Token, die zu der gleichen Wahl gehören aber austauschbar sein und somit die Eigenschaften eines *ERC-20* Token erfüllen.

Um diese beiden Eigenschaften zu vereinen, ist die Entscheidung auf einen ERC-1155 Token gefallen. Dieser erlaubt es Gruppen von eindeutigen Tokens zu erzeugen. Somit lassen sich mehrere `VoteToken` erzeugen, die jeweils nur für eine bestimmte Wahl eingesetzt werden können. Zu einer Wahl existieren dann mehrere Token mit der gleichen ID, die prinzipiell austauschbar sind.

Zentrale Speicherung von Informationen

Die Anwendung besteht aus mehreren einzelnen Komponenten. Viele dieser Komponenten greifen auf die gleichen Daten zu. So wird für Stimmabgabe, Stimmbeantragung und Wahladministration jeweils die Liste aller Wahlen benötigt. Damit diese Information nicht von jeder Komponente abgefragt wird, was besonders beim Wechsel zwischen den Komponenten für einen erhöhten Aufwand sorgt, werden zentrale Informationen im Vuex-Store gespeichert. Diese Änderung sorgt dafür, dass geteilte Informationen durch alle Komponenten genutzt werden können. Die lesenden Zugriffe auf die Ethereum-Blockchain konnten dadurch um ein Vielfaches reduziert werden.

Nutzung von Events für die Aktualisierung der Stimmen

Während der Wahl können die aktuell abgegeben Stimmen für einen Kandidaten bereits eingesehen werden (vgl. Abbildung 8.1).

Damit die Stimmen möglichst in Echtzeit aktualisiert werden, wurde in der initialen Implementierung die Blockchain in periodischen Abständen abgefragt. Dabei handelt es sich um mehrere Zugriffe, da die Anzahl der Stimmen für jeden Kandidaten einzeln abgefragt werden müssen.

Um eine effizientere Lösung zu schaffen, wurde im Laufe des Projektes die eventbasierte Vorgehensweise eingeführt. Sobald eine Stimme für einen Kandidaten abgegeben wurde, emittiert der `Election`-Contract ein Event, welches den Kandidaten enthält für den gestimmt wurde.

Das Frontend muss somit nur noch beim initialen Aufbau der Seite die aktuelle Stimmenanzahl

[1] `https://www.cryptokitties.co/`

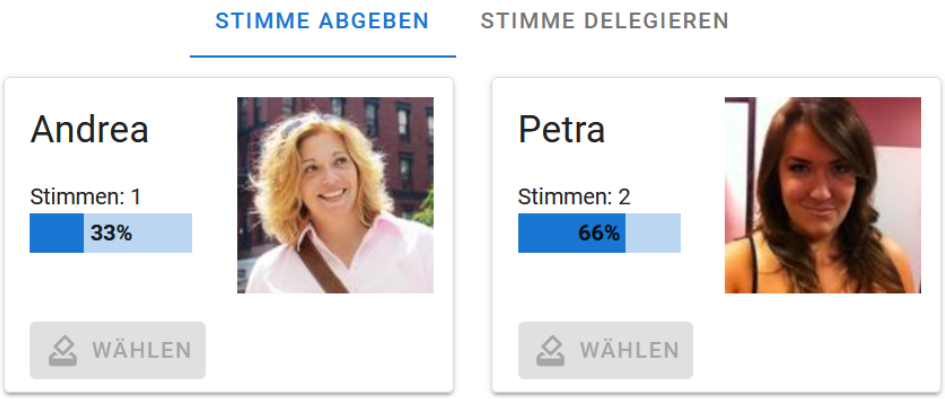


Abbildung 8.1: Anzeige der aktuellen Stimmen während einer Wahl

aller Kandidaten abfragen. Über alle danach abgegeben Stimmen wird es über ein Event benachrichtigt. Dies stellt eine effiziente Lösung dar, durch die ein periodisches Abfragen verhindert werden kann.

9 Fazit und Ausblick

Ziel dieses Projektes war die Erstellung einer dezentraler Wahl-Applikation. Zu diesem Zweck wurden mehrere Smart Contracts für die Ethereum-Blockchain geschrieben.

Der Fokus liegt dabei auf einer variablen Lösung. So soll es möglich sein, flexibel mehrere Wahlen zu erstellen. Eine neue Wahl soll das manuelle Deployment oder das Schreiben eines angepassten Smart Contracts nicht erforderlich machen. Dieses Ziel konnte durch die `ElectionFactory` erreicht werden. Mit Hilfe dieses Smart Contracts können flexibel neue Wahlen erstellt werden, indem die Factory jeweils einen neuen Smart Contract vom Typ `Election` erstellt und die Referenz aller Wahlen speichert. Die Umsetzung des `Election` Vertrages erlaubt es, dynamisch Kandidaten hinzuzufügen, solange die Wahl noch nicht gestartet wurde.

Außerdem wurde die Stimmabgabe von der Stimmberechtigung entkoppelt. Statt einer starken Kopplung innerhalb des Vertrages, in der man etwa festlegt, dass jede Adresse genau ein einziges mal abstimmen kann, wurde die Wahlberechtigung mit Hilfe eines extra Token umgesetzt. Jede Person, die im Besitz eines `VoteToken` für eine Wahl ist, kann diesen nutzen, um an der entsprechenden Wahl teilzunehmen. Dabei ist zu beachten, dass eine Person auch mehrfach abstimmen kann, wenn diese im Besitz von mehreren Token ist. Diese Umsetzung erlaubt entsprechend eine lose Kopplung zwischen Wahlberechtigung und Stimmabgabe. Wer wie oft bei welcher Wahl abstimmen darf, kann somit durch eine dritte Instanz, welche die `VoteToken` ausgibt, kontrolliert werden.

Nicht alle Funktionen der Smart Contracts wurden dabei selbst geschrieben. Bei dem `VoteToken` etwa handelt es sich um einen standardisierten ERC-1155[1] Vertrag. Viele Standardfunktionalitäten wurden in der OpenZeppelin[2] Bibiliothek bereits umgsetzt. Mit Hilfe von Vererbung lassen sich entsprechend viele Funktionalitäten übernehmen, ohne diese selbst neu implementieren zu müssen. Dabei ist das ERC-1155 Interface nicht die einzige Funktionalität, die von OpenZeppelin übernommen werden konnte. Auch einige Modifier und arithmetische Funktionen konnten von OpenZeppelin importiert werden.

Während die Implementierung durch OpenZeppelin beschleunigt werden konnte, konnte für das Deployment und Testing der Smart Contracts Truffle erfolgreich eingesetzt werden. Truffle ist ein

[1] `https://github.com/ethereum/eips/issues/1155`
[2] `https://openzeppelin.com/`

Entwicklungsframework, welches das Kompilieren, das Testen und das Deployment von Smart Contracts vereinfacht. Durch verschiedene Konfigurationen ist es somit möglich, die Verträge sowohl auf der lokalen Blockchain *Ganache*, als auch auf einem beliebigen anderen Netzwerk zu verteilen. Die umgesetzten Verträge wurden mit Hilfe von Truffle auf dem Rinkeby-Testnet verteilt. Damit ein Eindruck von der umgesetzten Anwendung und deren Interaktion mit den Smart Contracts auf dem Rinkeby Netzwerk gewonnen werden kann, wurde auch das Frontend öffentlich zur Verfügung gestellt:

`https://votify-dapp.netlify.app`

Neben den erreichten Anforderungen lässt sich aber auch über Erweiterungen der Anwendung nachdenken. Eine Erweiterung ist die Anpassung von Wahlbeginn und Wahlende. Die Wahl muss aktuell sowohl manuell eröffnet als auch manuell beendet werden. Denkt man echte Wahlen wie etwa die Bundestagswahl, so steht vor der Wahl bereits ein fester Zeitraum fest, in dem gewählt werden kann. Eine mögliche Erweiterung ist es also, den Zeitraum der Wahl bereits bei Erstellung der Wahl mit anzugeben. Der Smart Contract prüft dann automatisch, ob eine Stimmabgabe im zugelassenden Zeitraum liegt und lehnt die Stimme gegebenenfalls ab.

Weiter ist denkbar, die Wahlplattform generischer zu gestalten. Aktuell liegt der Fokus auf Kandidaten, also Personen, die gewählt werden können. Grundsätzlich lässt sich aber in vielen Bereichen über bestimmte Sachverhalte abstimmen. Beispiele dafür sind etwa Bürgerentscheide oder auch Abstimmungen im Freundes- oder Bekanntenkreis. Gestaltet man *Votify* generischer, so lässt sich diese Anwendung grundsätzlich für dezentrale Abstimmungen aller Art nutzen und ist nicht auf reine Personenwahlen beschränkt.

Außerdem lässt sich die Vertraulichkeit weiter verbessern. Theoretisch ist es aktuell möglich, anhand der Funktionsaufrufe der Smart Contracts nachzuvollziehen, welche Adresse für welchen Kandidaten abgestimmt hat. Möchte man also sicherstellen, dass die eigene Stimme geheim bleibt, so gilt es sicherzustellen, dass keine andere Person die eigene Adresse kennt. Während eine vollständig dezentrale und geheime Wahl eine große Herausforderung darstellt, so gibt es dennoch erste Protokolle, um dies umzusetzen [McCorry et al., 2017]. Bei dieser Umsetzung handelt es sich aber um einen mehrstufigen Prozess, was gegebenenfalls erhöhten Aufwand für die Wähler mit sich bringt.

Für eine reale Wahl - etwa eine Bundestagswahl - sollte die Ausgabe der `VoteToken` außerdem ausgelagert werden. Gegen einen Identitätsnachweis kann dann allen berechtigten Personen ein oder mehrere `VoteToken` ausgegeben werden.

Literatur

Canessane, Aroul et al. (2019). „Decentralised Applications Using Ethereum Blockchain“. In: *Fifth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*.

Ethereum Foundation (2021). *Providers*. URL: `https://web3py.readthedocs.io/en/stable/providers.html` (besucht am 09. 03. 2021).

Lifanova, Crescenzi (2020). *How to build a Contract Factory that Creates Contract Clones*. URL: `https://medium.com/upstate-interactive/how-to-build-a-contract-factory-that-creates-contract-clones-efcc9619be0b`.

McCorry, Patrick, Siamak F. Shahandashti und Feng Hao (2017). „A smart contract for boardroom voting with maximum voter privacy“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.

Radomski (2018). *EIP-1155: ERC-1155 Multi Token Standard*. URL: `https://eips.ethereum.org/EIPS/eip-1155`.