Bern University of Applied Sciences (BFH)

Bachelor Thesis

Visualizing Geneva's Next Generation E-Voting System

Authored by Kevin Häni <kevin.haeni@gmail.com>
Yannick Denzer <yannick@denzer.ch>
Supervised by Prof. Dr. Rolf Haenni <rolf.haenni@bfh.ch>
Prof. Dr. Philipp Locher <philipp.locher@bfh.ch>
Expert Han van der Kleij

Bern, January 13, 2018

Contents

Management Summary

The following documentation describes the context, planning and implementation of an application intended to visualize a new e-voting protocol. The protocol our application is based on is described in the paper "CHVote System Specification" of Rolf Haenni, Reto E. Koenig, Philipp Locher and Eric Dubuis of the Research Institute for Security in the Information Society (RISIS) of the Bern University of Applied Sciences. Their specifications describe how to build a next generation e-voting protocol which satisfies the complex requirements set up by the Swiss government, such as allowing an e-voting system to include large electorates.

Previous attempts to establish e-voting platforms in Switzerland were limited to only a small percentage of the electorate, for example Swiss citizens living abroad, because they only met the basic requirements, set up by the government. There exist many concepts and e-voting protocols which satisfy basic requirements, such as the privacy of the voters. However, an e-voting platform that could be used on a nationwide scale needs to be both individually and universally verifiable. This means in essence that an external individual can verify that all, but only valid votes have been counted in the tally. Current systems were behaving more like a black box and were not transparent enough to allow that kind of verification.

Another challenge which e-voting is facing is the educational problem: it is difficult to understand such a complex protocol without sufficient knowledge of cryptography. This might result in mistrust towards e-voting.

The goal of this project is to develop an application that addresses the educational problem of e-voting by allowing users to get a hands-on experience with the whole voting process from the perspective of each participating actor of the protocol. This would allow users to gain a better understanding of the next generation e-voting platform. The system should also be able to display multiple perspectives on different screens which requires real-time synchronization of data.

This document will first introduce the context of e-voting and our project task. Next it will describe the planning aspects such as the requirements and the time schedule. A short summary of the most important aspects of the CHVote protocol should establish the terminology and background knowledge for better understanding of our work. The main goal of this paper is to document in detail the implementation of our application, such as the architecture and the challenges we were facing. The document concludes with a short summary and reflection.

1 Introduction

With the ongoing digital transformation it seems only a matter of time until the shift from paper based ballots to electronic voting (e-voting). Recent polls show that the majority of the Swiss population is interested in having the possibility to vote online [1]. However, e-voting is a very complex topic and designing a secure e-voting protocol is notoriously challenging in terms of IT-security and cryptography.

1.1 Electronic Voting

First trials with e-voting in Switzerland date back to 2003 [4]. Since 2015, it has been possible for Swiss citizens registered in the cantons of Geneva and Neuchâtel and living abroad to vote electronically [3]. These systems were available only for a limited electorate size since they did not yet meet the requirements in terms of security and transparency to be accepted as a secure e-voting platform on a nationwide scale.

An e-voting system must satisfy a large variety of security requirements set up by the government, including:

- Privacy: no one can find out information about a voter's candidate selection. This implies that a voter's ballot must be encrypted before it leaves the voter's client and until the election is tallied.
- Fairness: no one is able to learn the intermediate result or the outcome of an election before the result has been officially tallied and published to a public board.
- Authenticity: all voters must be authenticated as eligible voters in order to cast a vote.
- Soundness: only valid votes are being tallied. If a voter selects more candidates than he is allowed to or less than he is supposed to, the vote must not be counted.
- Robustness: an e-voting system detects cheating actors.
- Distribution of Trust: the security of an e-voting system must not rely on a single point of trust.

[2]

[1] https://www.swissinfo.ch/ger/umfrage_grosse-zustimmung-zu-e-voting-trotz-sicherheitsbedenken/42457426

There exist many different concepts and e-voting protocols that cover many of the basic requirements. However, existing solutions typically behave like a black-box and keep the internals hidden, such that a voter cannot be sure whether his vote has been recorded correctly and counted in the final result.

A common problem is the **insecure platform problem**: if a voter's computer is affected by malware, the vote casting process is no longer under the voter's control and the candidate selection could be possibly manipulated without the voter's notice. It must be **individually verifiable** to every voter that his intended vote has been recorded, while at the same time the voter's privacy must still be ensured at all times [1].

One of the requirements that is the hardest to achieve is the **universal verification**: a good e-voting system must be transparent and allow an external person to verify that every protocol participant has abided by the protocol and that all and only valid votes have been counted correctly.

Should an e-voting system be available to more than 30% of the respective cantonal electorate, it must be individually verifiable; if 50% - both individually as well as universally [2].

1.2 CHVote Protocol

Some of the mentioned requirements might sound as a paradox at first. However, they can be solved by advanced cryptography. A contract was formed between the state of Geneva and the Research Institute for Security in the Information Society (RISIS) of the Bern University of Applied Sciences to work out a new protocol which meets the complex requirements set up by the Swiss government. In 2017, Rolf Haenni, Reto E. Koenig, Philipp Locher and Eric Dubuis published the resulting specification document and a proof-of-concept has been successfully implemented by the State of Geneva. Geneva is currently developing a new version of their e-voting system with the name CHVote, based on these specifications. Other cantons, currently St. Gallen, Aargau, Bern and Lucerne have announced their interest in cooperating with Geneva and might use their system in future.

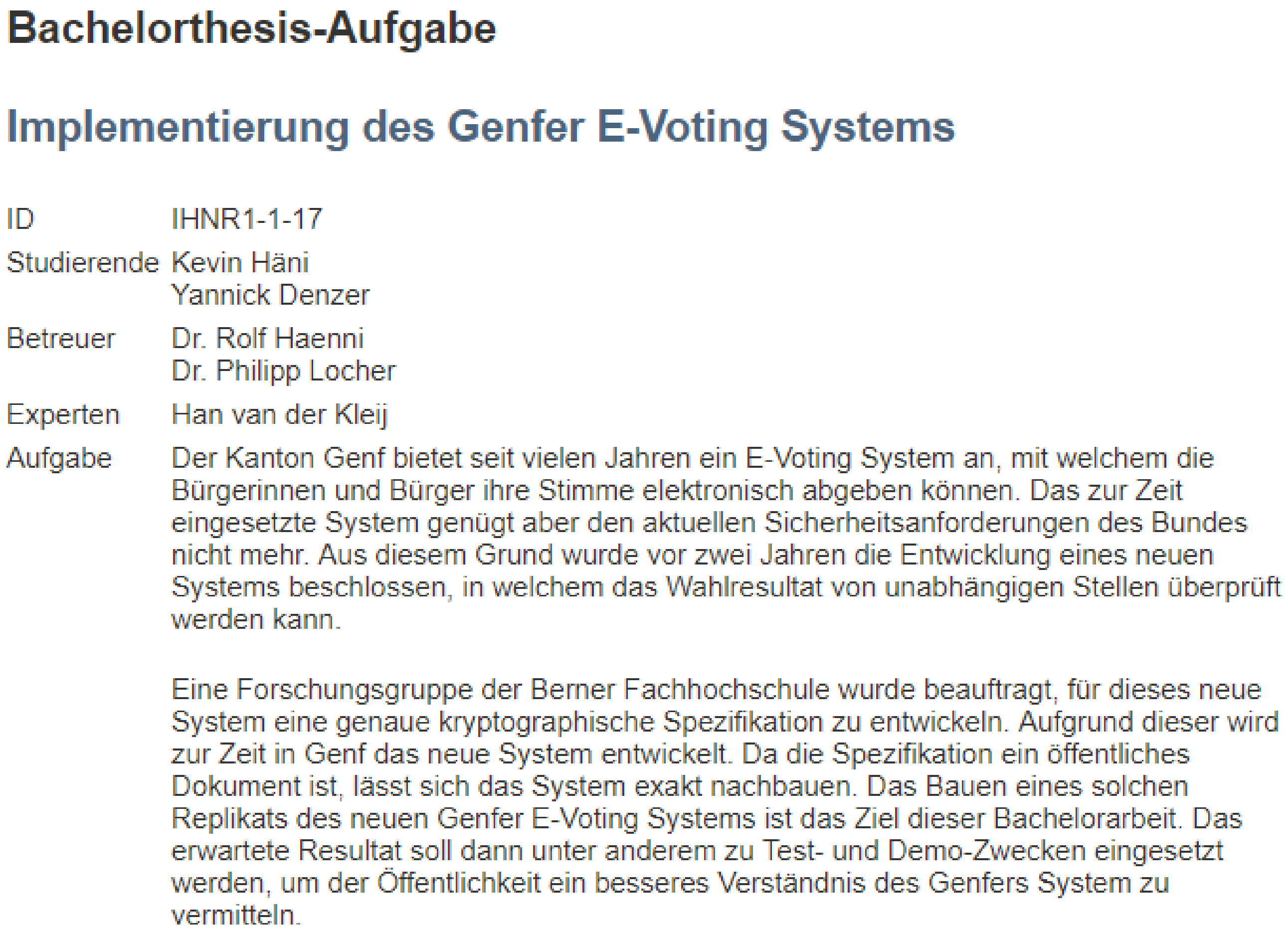
The CHVote specifications document is publicly available and describes not only the theoretical background but also provides pseudocode for the approximately 60 algorithms that are needed to implement their protocol.

Chapter 3 will summarize the most important aspects of the CHVote specifications for a better understanding of our work.

1.3 Project Task

An additional problem with which e-voting is confronted is that understanding such a complex protocol is not easy without good knowledge of cryptography and mathematics. This might be one of the reasons why many people still do not trust electronic voting systems.

In close consultation with the authors of the CHVote specifications, we were looking for an interesting project for our bachelor thesis in the thematic field of e-voting. We decided to study the CHVote specifications, implement the protocol and build an application on top of it.



Bachelorthesis-Aufgabe

Implementierung des Genfer E-Voting Systems

ID	IHNR1-1-17
Studierende	Kevin Häni Yannick Denzer
Betreuer	Dr. Rolf Haenni Dr. Philipp Locher
Experten	Han van der Kleij
Aufgabe	Der Kanton Genf bietet seit vielen Jahren ein E-Voting System an, mit welchem die Bürgerinnen und Bürger ihre Stimme elektronisch abgeben können. Das zur Zeit eingesetzte System genügt aber den aktuellen Sicherheitsanforderungen des Bundes nicht mehr. Aus diesem Grund wurde vor zwei Jahren die Entwicklung eines neuen Systems beschlossen, in welchem das Wahlresultat von unabhängigen Stellen überprüft werden kann. Eine Forschungsgruppe der Berner Fachhochschule wurde beauftragt, für dieses neue System eine genaue kryptographische Spezifikation zu entwickeln. Aufgrund dieser wird zur Zeit in Genf das neue System entwickelt. Da die Spezifikation ein öffentliches Dokument ist, lässt sich das System exakt nachbauen. Das Bauen eines solchen Replikats des neuen Genfer E-Voting Systems ist das Ziel dieser Bachelorarbeit. Das erwartete Resultat soll dann unter anderem zu Test- und Demo-Zwecken eingesetzt werden, um der Öffentlichkeit ein besseres Verständnis des Genfers System zu vermitteln.

Figure 1.1: The initial task description of the bachelor thesis left a number of open possibilities as to how exactly the project could be implemented. Specific tasks and requirements were discussed with our supervisors during the first few weeks of the project.

Based on the initial, rather general task description (see figure 1.1), there were several possibilities regarding the final product, such as a realistic prototype, a verifier software for the implementation which is being developed in Geneva, or an application that targets the educational problem and can be used to demonstrate the functionality of the new protocol.

Ultimately, we have decided to implement the last option: the goal of our project was to develop a web-based application which allows to visualize every step of a CHVote election event, from the pre-election tasks like generating the electorate data, casting and confirming ballots from a voter's point of view, to the post-election processes like mixing, decryption and tallying. The main goal of our project was not to implement a secure, "ready for production" e-voting system from an infrastructural perspective, but more the visualization and the user-interface. With the resulting application users would be allowed to get a hands-on experience with Geneva's new e-voting system and a preview of how the future of voting in Switzerland might possibly look like.

Studying of the specifications and the development of the approximately 60 algorithms that were defined as pseudocode in the specifications was done as preparatory work during the course "Project 2", which we had finished just before the start of the bachelor thesis. The resulting set of algorithms have been used as a library for implementing the protocol on which our application is built.

Outline of this document: in chapter 2 we will further discuss the goals and requirements that we have set for this project. Additionally, it covers the time planning and the project methodology.

For better understanding of our application, chapter 3 contains a short summary of the CHVote specifications document and establishes the terminology and theoretical background.

Chapter 4 will describe the final product and the concepts behind our application.

Chapter 5 covers the technical aspects regarding the implementation, such as the architecture, the language and technology decisions. Later sections contain detailed information about the internals of each component of our application and the challenges we were facing during the implementation phase.

Chapter 6 will conclude this document with a short summary and a reflection about the project.

2 Project Management

This chapter describes the several aspects of the project planning such as the project method, the requirements and use-cases as well as the time schedule.

Since this was not a project where the scope and goals were strictly defined and clear from the very beginning, it was important to elaborate the requirements in close collaboration with our supervisors early on. This is why we decided our project would be very suitable for an agile kind of project method. Because of the small team consisting of only two members, we did not choose a particular project method like SCRUM, as this would have probably caused more overhead than actual benefits.

We have therefore set up regular meetings (usually once every 2 weeks) with our supervisors to discuss our ideas and get feedback about the current progress.

2.1 Goals

As the first step, we have elaborated the goals and requirements for our application. We have structured the requirements into groups which correspond to the actors of the CHVote protocol and therefore the views our application is going to consist of.

Some requirements affect multiple or all actors and are therefore listed as "General Requirements". We have also added a priority and requirement type to simplify the planning and time management.

2.1.1 General Requirements

	Description	Type	Prio.	Phase	Status
R1	The CHVote protocol is implemented as specified in the latest specification document. The only exceptions are the algorithms for channel security.	Must	High	1	Done
R2	The application is web-based and shows updates within the same demo-election event in real-time.	Must	High	1	Done
R3	The system supports 1-out-of-3 type of elections (e.g. elect 1 of 3 possible candidates).	Must	High	1	Done
R4	The system supports multiple parallel election events.	Must	High	1	Done

R5	Users can create new elections.	Must	High	1	Done
R6	The system supports internationalization. Providing more than one language is not required.	Must	Med.	1	Done
R7	The system can handle k-out-of-n type of elections.	Can	Med.	1	Done

2.1.2 Election Overview

	Description	Type	Prio.	Phase	Status
R8	The overview shows which phase the election is currently in.	Must	High	2	Done
R9	A graphical scheme of the CHVote protocol gives an overview of all participating parties.	Must	Med.	2	Done

2.1.3 Election Administrator

	Description	Type	Prio.	Phase	Status
R10	An election can be set up by providing all required information such as the candidates, the number of parallel voters, the number of voters and the number of selections (simplified JSON input).	Must	High	1	Done
R11	The election can be set up without entering the parameters in JSON format and allows an easier set up of elections with multiple parallel election events.	Must	Low	2	Done
R12	The election administrator view allows to perform the tallying and displays the final result of an election in numbers and a pie chart.	Must	High	1	Done
R13	During election setup, the security parameters can be chosen from a set of predefined parameters.	Can	Low	2	Done

2.1.4 Printing Authority

	Description	Type	Prio.	Phase	Status
R14	Users can generate and display voting cards for an election.	Must	High	1	Done
R15	Voting cards hide sensitive information behind a scratch card.	Can	Med.	2	Done

2.1.5 Election Authority

	Description	Type	Prio.	Phase	Status

R16	The election authority view shows all information known to the election authority.	Must	High	1	Done
R17	After a voter has submitted a ballot, all election authorities can check and respond to the voter's submission.	Must	High	1	Done
R18	In the post-election phase, all election authorities can perform the mixing and decryption tasks.	Must	High	2	Done
R19	Each authority can optionally process all tasks automatically.	Can	High	2	Done

2.1.6 Voter

	Description	Type	Prio.	Phase	Status
R20	Users are able to go through the whole vote-casting process for every voter.	Must	High	1	Done
R21	The voting card of a voter is displayed on the screen. The voting and confirmation codes can be copied into the input textfields by double clicking.	Must	Med.	1	Done

2.1.7 Bulletin Board

	Description	Type	Prio.	Phase	Status
R22	The bulletin board view shows which information is publicly available.	Must	High	1	Done
R23	The bulletin board view is extended with verification functionality.	Can	Low	2	Done

2.1.8 Out-of-Scope

The following topics are considered out-of-scope for the duration of the project:

- The goal of the project is not to build a realistic prototype. Therefore, the whole backend will run on a single server while in a reality there would be components running on distributed servers. Another difference between our implementation and a productive implementation is that our application generates the ballots on the server. In a productive environment, the ballots would have to be generated on the client for security reasons. This, however, would require us to rewrite many of the already implemented algorithms in JavaScript.

- The protocol takes into account that not all voters might be eligible to vote in all elections of a given election event (eligibility matrix). For simplicity, we assume that all voters are eligible to vote in every election of our application.

- Message level encryption and signature based integrity protection are very important in a productive implementation of an e-voting system and are also described in the CHVote specifications. However, since our system is only used for demonstration purposes and we do not have a distributed infrastructure, there is no real need for channel security in the project.
- Providing more than one language is also out-of-scope. If there is enough time, a second language might be provided optionally.

2.2 Time Schedule & Implementation Phases

As the next step, we created a time schedule and structured our project into smaller units of work.

The actual implementation phase has been broken down into two phases:

- Phase 1 includes the implementation of all high priority must-requirements, which means basically developing an application that allows to visualize the whole CHVote election process. We have agreed that after phase 1 some parts of the user interface would still be in a rather primitive state (eg. user inputs and components used for data display will not yet be very user friendly and of more technical nature, for example in the JSON format).
- For phase 2 we planned implementing the can-requirements and all must-requirements with lower priority as well as improving the whole user experience.

This approach reduced the risk of hidden technical limitations of our architecture which could result in time-consuming architectural changes.

Project Tasks		38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
Documentation																		
Update documentation and journal	soll	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦 5
	ist	🟩	🟩	🟩			🟩		🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩 5
Concept																		
Working out goals / requirements	soll	🟦	🟦	🟦														
	ist	🟩	🟩	🟩	🟩	🟩												
Concept	soll		🟦	🟦														
	ist		🟩	🟩	🟩													
Implementation																		
Update CHVote cryptolibrary to the lastest specification	soll	🟦	🟦 1															
	ist	🟩	🟩 1															
Prototyping (proof of concept)	soll			🟦	🟦 2													
	ist			🟩	🟩 2													
Iteration 1																		
Implement backend API	soll					🟦	🟦	🟦	🟦	🟦	🟦							
	ist					🟩	🟩	🟩	🟩	🟩								
Implement frontend: Pre-election	soll							🟦	🟦									
	ist							🟩	🟩									
Implement frontend: Election	soll								🟦	🟦								
	ist								🟩	🟩								
Implement frontend: Post-election	soll									🟦	🟦 3							
	ist									🟩	🟩 3							
Iteration 2																		
Automation of election authority (R19)	soll											🟦						
	ist											🟩						
Voting Card layout, Scratchcard (R15)	soll											🟦	🟦					
	ist											🟩	🟩					
Election Overview (R8, R9)	soll												🟦	🟦				
	ist															🟩	🟩	🟩 4
k-out-of-n election types (R7)	soll													🟦				
	ist													🟩	🟩			
Improving the input forms and layout (R11)	soll														🟦			
	ist													🟩	🟩			
Flexible security level (R13)	soll															🟦		
	ist														🟩			
Verifier functionality (R23) and reserve	soll															🟦	🟦 4	
	ist														🟩	🟩		
Project completion																		
Poster, Final Day, Presentation	soll																🟦	🟦
	ist																🟩	🟩

Table 2.8: Project schedule

The following milestones have been defined based on the requirements:

- M1: Implementation of the CHVote crypto-library is finished
- M2: A proof-of-concept / prototype for our application has been drawn up.
- M3: Implementation phase 1 is finished (all must-criteria with a high priority are met).
- M4: Implementation phase 2 is finished (all requirements including the can-criteria are met).
- M5: The documentation is finished.

2.3 Use Cases

As the next step, we have converted the requirements into use cases, by grouping them according to the actors. One use case is shown as an example. Please refer to the appendix for a complete list of use cases.

Table 2.9: Use Case «Casting of a vote»

Use Case	**Casting of a vote**
Primary Actor	Voter
Description	The voter can cast a vote by selecting his favored candidate(s) and his voting code
Precondition	• The election has the status "Election Phase" • A voter is selected in the voter view • The voter has the status "Vote Casting Phase"
Postcondition	The first election authority receives a ballot-check task
Main path (M)	1. The voter visits the voter view and selects his voter object 2. The system demands a selection of the candidates and the voter's voting code 3. The voter clicks on "Cast ballot"

3 CHVote Protocol

Our project is based on the CHVote protocol specifications. We would like to point out that this protocol and the following ideas do not originate from us! The concept and specifications have been created by Rolf Haenni, Reto E. Koenig, Philipp Locher and Eric Dubuis of the Institute for Security in the Information Society (RISIS) of the Bern University of Applied Sciences. For this project, we have implemented the protocol according to their specifications. In this section we summarize the most important aspects of the protocol and establish the terminology for better understanding of this document and our application.

3.1 Actors

A **voter** is a person who is eligible to vote in his respective state. Every voter is assigned a **counting circle** which typically corresponds to the voter's municipal and is required for statistical purposes. For authentication purposes, every voter must possess a voting card which has been sent to him prior to an election event and which contains codes used to identify the voter during the vote casting process.

The **Election Administrator**, typically a person of the government, is responsible for setting up the election event by providing the required information such as the candidates or the voters and initiates the generation of the cryptographic electorate data. He is also responsible for tallying the election and publishing the final results.

The **election authorities** can be seen as some kind of independent election observers. In a CHVote election event there are always multiple election authorities in order to avoid having to trust a single authority. The authorities are involved in almost every step of the protocol, starting from the generation of their shares of the electorate data, checking and responding to new ballots, as well as in the mixing (a measure to ensure anonymity) and decryption phases. The public key that is used for encrypting the ballots has been jointly generated by all the election authorities. Therefore, in order to decrypt the ballots, all authorities must work together and provide their share of the private key.

The measure of multiple authorities participating in the whole e-voting process establishes a **distribution of trust** and ensures security of the whole election process as long as at least one election authority can be trusted.

The **Bulletin Board** acts as a central board over which most communication is done and where all the public data is stored. Publishing all data that is not secret onto a public board, including the list of encrypted ballots, and functionality to verify the correctness of these data prove that the protocol has made a big step towards the demanded transparency and the universal verification.

The **Printing Authority** is responsible for printing the voting cards for all voters. Since the printing authority needs to be in possession of all the secret voting codes, it is a very sensitive point in the protocol. The printed voting cards are handed over to a trusted mailing service for delivery.

3.2 Pre-Election & Voting Cards

Before the actual election phase, the election administrator sets up the election event by entering all required parameters, including the number of voters, the elections with their corresponding candidates and the number of candidates a voter can select in every election. All election authorities jointly generate the cryptographic data for the whole electorate from which the voting card information is derived. The printing authority combines the information from all the election authorities, prints the voting cards and delivers them to the voters by a trusted mailing service.

A voting card contains several codes, namely:

- a voting code;
- a confirmation code;
- a finalization code;
- one verification code for every candidate.

The **voting and confirmation codes** are authentication codes and are used to authenticate the voter twice during the vote casting process; the first time with the voting code when they cast their vote, and the second time for confirming their vote. The second authentication code is required because otherwise an attacker who could have infected a voter's computer with malware could just confirm the vote on behalf of the voter after he manipulated the candidate selection and could therefore skip the whole verification process.

3.3 Vote Casting with Oblivious Transfers

As mentioned earlier, one of the big challenges of an e-voting-protocol is dealing with the insecure platform problem: a voting platform that is infected with malware poses a risk that an attacker could manipulate the candidate selection on the vote-casting page after the voter has entered his voting code.

The CHVote specification suggests a "Cast-as-intended verification"-step to allow voters to detect this kind of manipulation: when a voter enters his voting code and the indices of his favored candidates, the voting client forms a **ballot** containing the voter's selection encrypted with the authorities' public key, his public voting credential derived from the voting code, and a **non-interactive zero knowledge proof** which proofs that the voter has formed the ballot according to the protocol and that he has been in knowledge of his voting code, without revealing any information about the voting code.

Every election authority has to check the voters public voting credential, the validity of the ballot proof and that the voter has not already cast a vote. The encrypted selection also serves as a query for an oblivious transfer. A **k-out-of-n oblivious transfer** allows a client to query a server for k messages, without the server knowing what messages the client requested and without the client learning anything about the other $n-k$ messages. Adapted to the CHVote protocol: the voting client queries the authorities for the corresponding verification codes of the selected candidates, without the authorities learning which candidates the voter has selected and without revealing any information about the other candidates.

The voter then checks if the returned verification codes match the codes of the candidates he has chosen on the printed voting card. If the selection was somehow manipulated by malware, the returned verification codes would not match the printed ones and the voter would have to abort the vote casting process. This way the integrity of the vote casting process can be guaranteed even in the presence of malware.

Another feature the protocol supports is that an election event can consist of t multiple parallel elections. In such cases, the voter has to submit a single ballot, which contains his candidate selection for all parallel elections. This raises the question of how the system can verify that a voter has chosen exactly the correct number of candidate in each election, and not for example one less in the first and one additional candidate in the second election.

The specification suggests the following trick: assuming an election consists of two parallel elections ($t = 2$) with 3 candidates each ($n_1 = 3, n_2 = 3$), and a voter can select one candidate in each election ($k_1 = k_2 = 1$). The verification codes are derived from $n = \sum_{j=1}^{t} n_j$ random points on t polynomials (one for every election event j) of degree $k_j - 1$, that each election authority has chosen randomly prior to the election. By learning exactly $k = \sum_{n=1}^{t} k_j$ points on these polynomials, the voting client is able to reproduce these polynomials and therefore is able to calculate a particular point with $x = 0$ on these polynomials. The corresponding y values are incorporated into the second credential from which the confirmation code is derived. As a result, only if a voter has been able to reconstruct these polynomials with the returned points by submitting a valid candidate selection, he will be able to confirm the vote that he has cast.

3.4 Anonymity with a Re-Encryption Mix-Net

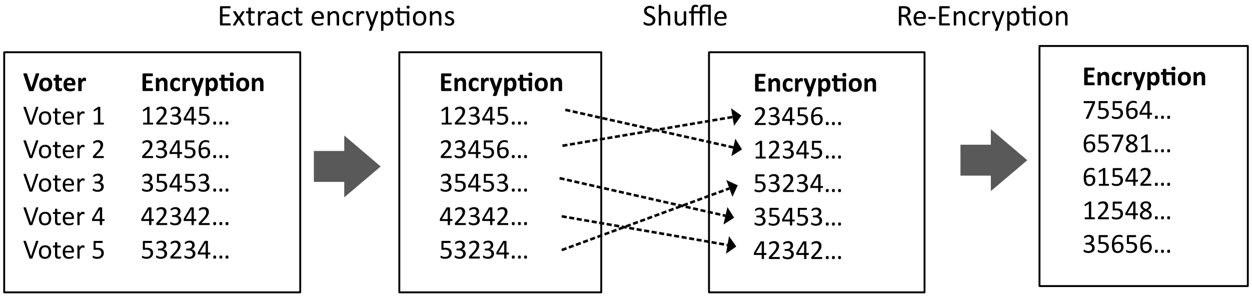
As mentioned earlier, both the election authorities as well as the bulletin board keep track of all the cast ballots, which contain the voter's candidate selection encrypted under the public key which has been jointly generated by all election authorities. Up to this point, the ballots are still linked to the voters, since the voter's identifier is required for the confirmation process and to avoid the voter casting multiple ballots. If the ballots were decrypted now, this would conflict with the anonymity and the privacy of the voters.

As the first step of the post-election phase, the first election authority extracts the encrypted candidate selections from the ballots. In order to anonymize this list of encryptions, the protocol suggests a "mixing-process", in which every authority performs a cryptographic shuffle with a random permutation. Additionally, all the encryptions are re-encrypted such that they receive a new ciphertext. This re-encryption is done by using the multiplicative homomorphic property of the ElGamal encryption scheme that is used for encrypting the ballots. A **multiplicative homomorphic encryption** scheme allows to perform multiplications on the ciphertext such that:

$$Enc(a) \cdot Enc(b) = Enc(a \cdot b)$$

The specification suggests multiplying the encrypted selection with the encryption of the neutral element 1 since this yields a new ciphertext for the exact same plaintext.

$$Enc(a) \cdot Enc(1) = Enc(a)$$



Voter	Encryption
Voter 1	12345...
Voter 2	23456...
Voter 3	35453...
Voter 4	42342...
Voter 5	53234...

Encryption
12345...
23456...
35453...
42342...
53234...

Encryption
23456...
12345...
53234...
35453...
42342...

Encryption
75564...
65781...
61542...
12548...
35656...

Figure 3.1: During the mixing phase, the encryptions are extracted from the list of ballots and then shuffled with a random permutation and re-encrypted by all election authorities. This measure ensures the anonymity of the votes.

While the extraction only needs to be done by the first election authority, every election authority is sequentially performing the shuffle and re-encryption of the mixed list of the previous election authority.

To prevent election authorities from cheating and not performing the mixing as specified, a proof is calculated which will be verified by all election authorities before decryption. If one of these **shuffle-proofs** is invalid, the election process will not proceed.

4 Application Description

This chapter describes the product of our bachelor thesis and the main component of our application - the visualizer web-application (front-end). We will focus mainly on the non-technical concepts as the next chapter will cover the technical aspects.

4.1 Application Overview

In essence, the functionality of our application revolves around visualizing an election event according to the CHVote protocol and guiding the users through its several phases. A typical CHVote election event can be broken down into phases shown in figure 4.1

To give the users an opportunity to experience and gain insight into the functionality of the protocol a concept was implemented which enables them to take the perspective of every actor of the protocol at any given time and during each phase of the election event. This is why our application is divided into separate **views**, one for each actor:

Every actor involved in a CHVote election event has its own set of data to be displayed and specific tasks and use cases it has to perform. Most use cases are only available during a particular phase.

- **Election Overview**: the election overview shows what phase the chosen election event is currently in. Additionally, a graphical schema shows how all the actors are connected and who is involved in the current phase.

- **Election Administrator View**: the view of the election administrator allows a user to set-up an election event by configuring the number of voters, the elections including the candidates and the number of selections per election. Instead of providing this information every time an election event is set up, previously defined election presets can be applied or the parameters can be generated randomly.

  The election administrator view is also the place where the elections can be tallied and the final result is determined during the post-election phase.

- **Printing Authority View**: in the printing authority view, the voting cards can be printed and displayed for every voter. Additionally, the voting cards can be sent to the voters.

- **Election Authority View**: the election authority view first lets a user choose one

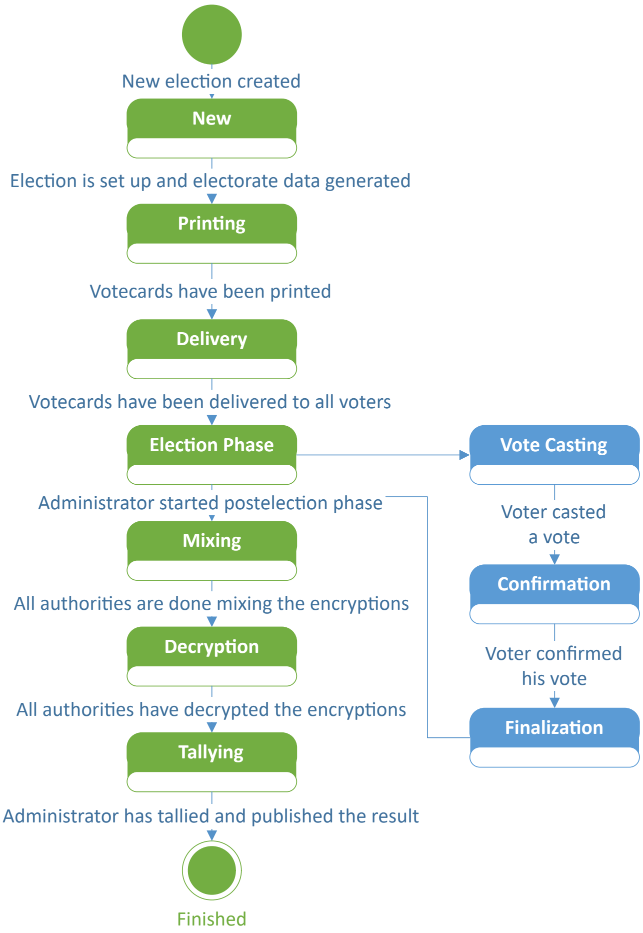

New election created
New
Election is set up and electorate data generated
Printing
Votecards have been printed
Delivery
Votecards have been delivered to all voters
Election Phase
Vote Casting
Voter casted a vote
Confirmation
Voter confirmed his vote
Finalization
Administrator started postelection phase
Mixing
All authorities are done mixing the encryptions
Decryption
All authorities have decrypted the encryptions
Tallying
Administrator has tallied and published the result
Finished

Figure 4.1: An election event consists of 7 different phases (green). During the election phase a voter can be in 3 different phases (blue)

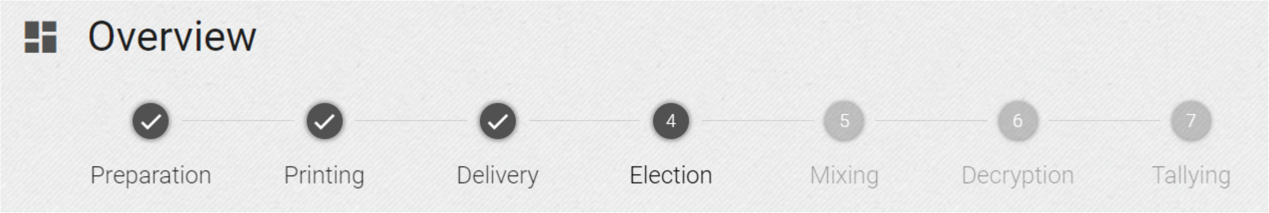


Figure 4.2: The election overview guides the user through the several phases of an election event and indicates which phases have been finished and what the next steps are.

of the three election authorities he wants to observe. On the top, new tasks will pop up whenever an election authority needs to perform a specific task, such as ballot or confirmation checking or mixing and decryption.

Additionally, the view shows the data that an election authority knows. This concept of dividing a view into tasks and data can be seen in figure 4.3 and has been used throughout the application in almost every view.

- **Voter View**: in the voter view, the vote casting process can be done for every voter that has been previously generated. The view displays both a voting form on the left and the voter's voting card on the right side. Two sensitive codes are hidden behind a scratchcard and can be copied to the voting card input by clicking on them after they have been revealed.
- **Bulletin Board View**: the bulletin board view always shows all the data that has been appended to the bulletin board by the other actors, such as the pre-election data, the ballots that the voters have cast, as well as all the proofs generated during the post-election phase.
- **Verifier View**: the view of the verifier becomes visible after the election result has been published to the bulletin board and the election event has reached its final stage. By clicking on the verify button, several checks are executed and the result is displayed on the page.

All the views are accessible from a tab view displayed on the top of the page which serves as the standard means of navigation, see figure 4.4.

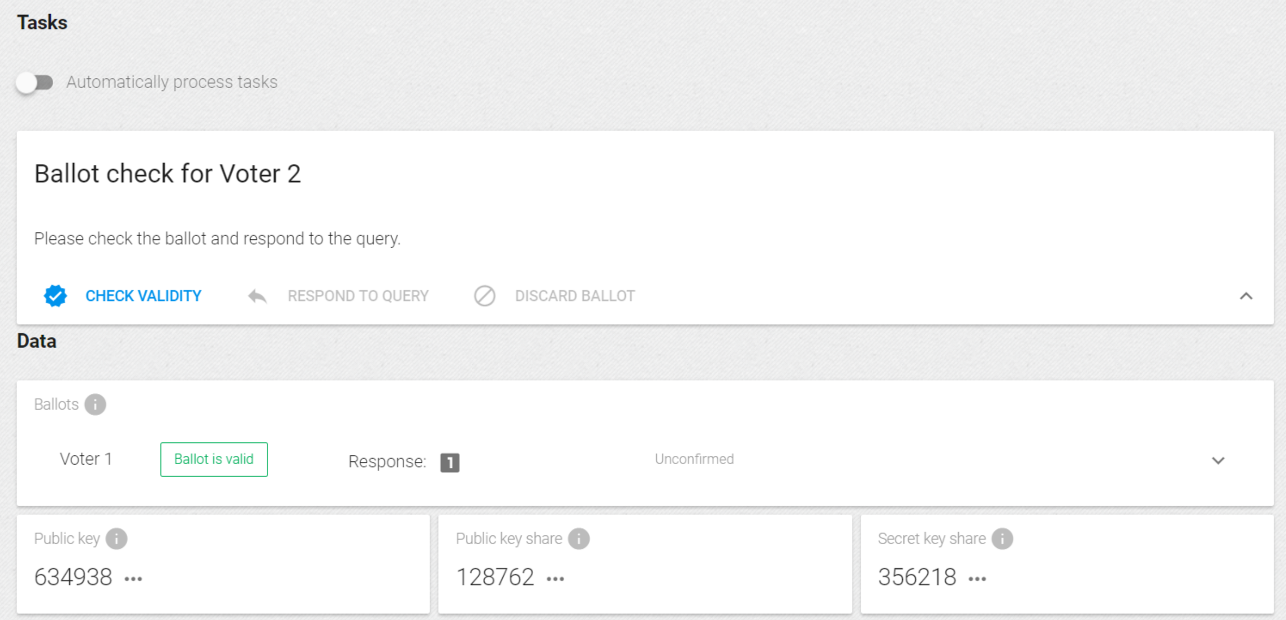
Based on the use cases, we tried to figure out all commonalities between the views: the views typically display the information known by the respective actor. Especially the bulletin board and election authorities could contain lots of information to be displayed. Most of the views have distinct tasks to be executed by the respective actor, such as casting a ballot in the voter's view or confirming ballots from an election authorities view.

The content that a view displays, or in other words, the functionality an actor has access to, depends on the phase the election event is currently in: during the pre-election phase the vote administrator needs to be able to set up the election, while in the tallying phase he must be able to tally and determine the final result.

4.2 Design

Given the rather large amount of complex data to be displayed, the main challenge of the project has been a well designed user interface which would allow to display all important information while maintaining a clear overview.

To achieve this goal we tried to keep our design very minimalistic and follow the Google material design guidelines as much as possible by choosing an appropriate UI component framework.



Tasks

Automatically process tasks

Ballot check for Voter 2

Please check the ballot and respond to the query.

Data

Ballots

Voter 1 Ballot is valid Response: 1 Unconfirmed

Public key 634938 ···

Public key share 128762 ···

Secret key share 356218 ···

Figure 4.3: The election authority view as an example of how the pages are generally structured: most views are divided into a tasks part which allows to perform the tasks of an actor, and a data part which shows what information about an election event the participant is in possession of.

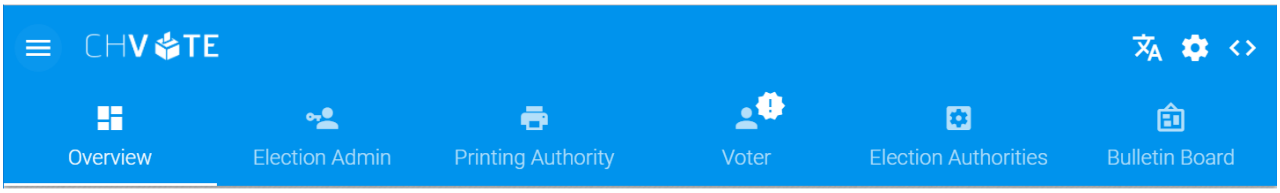


Figure 4.4: The tab beyond the page header is the main control for navigating through the different pages. It also indicates where some interaction is required next by displaying an exclamation mark icon in the respective actor's tab.

Even though mobile compatibility was not a requirement, it was nevertheless our goal to make the layout as responsive as possible such that it could at least be used from a mobile phone, even though it was clearly optimized for desktop resolution.

Throughout the application we tried to establish common concepts regarding the look and feel and on how to display our data. One popular layout-concept of the Google material guidelines is the card layout. Cards can be easily integrated in a responsive grid system, look modern and allow to visually group data. In addition, we used pushover menus, tool-tips and pop-ups as they made it possible to hide less relevant information by default and display it only on demand of the user.

Before starting with the implementation of our application, we created mock-ups for most of the views to discuss our ideas with our supervisors and incorporated their feedback as well. The following screenshots are an extract of the mock-ups in which we tried to visualize how we imagined the resulting application would look like during the conceptual phase.

Our conceptual work also involved developing a small prototype / proof of concept, in which we implemented one use case in a reduced extent with the envisaged frameworks and technologies to evaluate the technical feasibility. The next chapter covers in detail the languages and frameworks used in the project.

5 Technical Implementation

This chapter describes the application implementation. The first part gives an overview of the architecture from a high-level perspective with each component being a black-box. Later sections of this chapter describe the internals of every component in detail.

The application has been implemented following a "single page application (SPA)" architecture. Some reasons which led to this decision:

- Due to our personal interest in JavaScript and our intention to improve our knowledge of this language, we wanted a significant part of the development to be done client-side.
- We imagined the state handling to be easier with an SPA than having to pass around cookies and session data between every request.
- An SPA seemed to be the right tool for building modern looking, intuitive and responsive user interfaces.

From a high level perspective and following the SPA architecture, the application can be divided into 3 main components, as shown in figure 5.1:

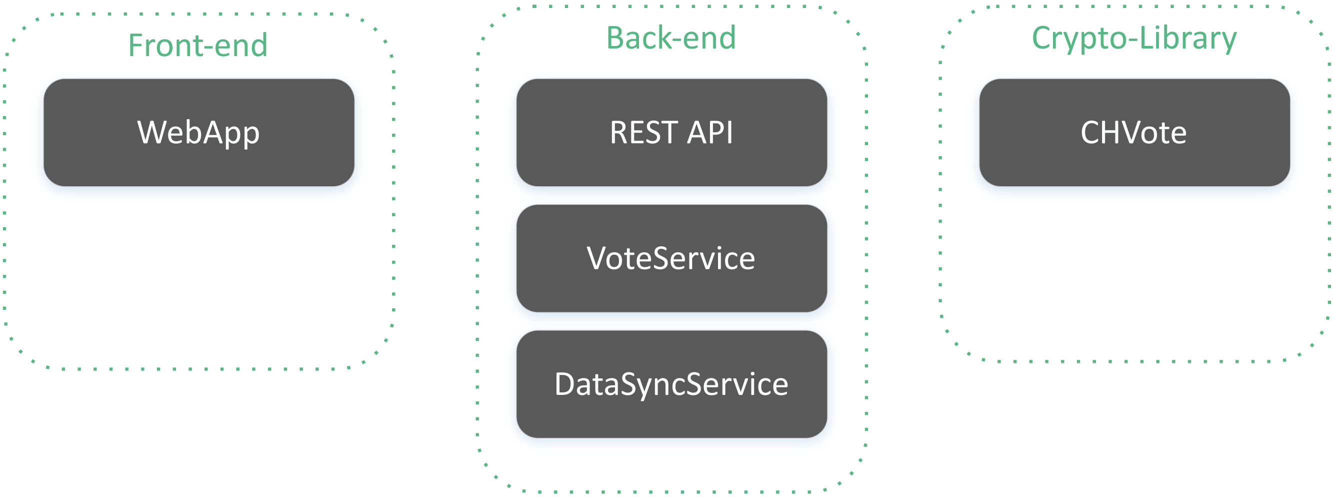


Figure 5.1: From a high level perspective, our application consists of 3 main components: the front-end (web-application), the back-end, and the crypto-library.

- The **front-end** is where all the functionality of the back-end is consumed and where a typical CHVote election event is visualized for the users. It is therefore the most important component of the application. The back-end is developed in response to the front-end requirements.

- The **back-end** consists of several components which use the CHVote crypto-library for building an actual e-voting ecosystem and providing an API to manipulate the data as well as a data-synchronization service to push the data from the back-end to the web-clients. All the sub-components of the backend run on a single server.
- The **CHVote crypto-library** is the result of our "Project 2" course project which we have finished before our bachelor thesis. This library contains all the algorithms specified as pseudocode in the specification document.

5.1 Technology & Language Decisions

During the CHVote crypto-library implementation we evaluated different programming languages and decided to use Python. Since Java has already been used by the team in Geneva, the use of a different language would additionally prove that the CHVote specification can be implemented regardless of the programming language. Python seemed like a rather suitable language for the project due to the following reasons:

- python is a mature language with a lot of libraries;
- python allows programs to be written in a compact and readable style, for example by supporting tuples;
- the protocol was not completely specified at that point and had still been undergoing some changes, we wanted to use a language in which we could adapt changes quickly and easily;
- native support for large integers (*BigInts*) and bindings for the GMP[1] library;
- supports a lot of platforms;
- many popular web development frameworks are available.

Throughout the project not all of the reasons above turned out to be true or ideal. The drawbacks that we have experienced during the implementation of this project will be discussed in section 5.6.2.

Since the project was implemented with the crypto-library in the back-end, Python was also the obvious language for the whole back-end. Python offers a wide variety of frameworks for building web-services. Since we planned on building a single-page-application for the client, we chose the lightweight micro-framework flask for building a restful web-service and the data synchronization service.

For the **front-end** web application we evaluated several single-page application frameworks. VueJS is a new, modern and lightweight SPA framework that in contrast to Angular has a much

[1] GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers, see `https://gmplib.org/`.

flatter learning curve but still offers all the functionality that we need. The VueJS add-on Vuex enabled us to establish a data-store pattern in our front-end, which made it possible to have a copy of the back-end data-store in our web application which was synchronized in real-time through web-sockets.

Socket.io simplifies the usage of web-sockets and offers fallback technologies such as long-polling in case web-socket is not supported by either the browser or the web-server. Both Flask as well as VueJS have plug-ins that support and integrate socket.io.

For persisting the state of an election, we decided to use mongoDB. The reason behind this choice will be described in more detail later on.

5.2 Architecture

The core of the application is the VoteService component in the back-end which implements the e-voting protocol by utilizing all algorithms of the CHVote crypto-library according to the CHVote specification. The VoteService component internally holds the state of a whole CHVote election event and exposes functions to manipulate this state at a granularity required by our web application to implement all use cases. For example: the VoteService contains a list of ballots and exposes functions to cast a new ballot, which will generate a new ballot according to the protocol, by calling the CHVote crypto-library, and then adds the ballot to the list.

On top of the VoteService we have implemented a REST service that acts as a facade to the VoteService component and makes its functionality available as an API to our web-clients. The REST service also initializes the VoteService by loading and persisting its state from and to the database between each API call, since each API call is stateless.

One of the requirements is that all clients must be informed in real-time about mutations of the election state made by other participants. To achieve this, a data sync service has been implemented which allows pushing the state of an election event to the web-clients by using the WebSocket protocol. This service is triggered by the REST service after every API call to push the delta between the old and the new state to the clients.

To establish a proper separation of concerns, the state of the VoteService is always sent to the client via the data sync service. The REST API only returns success or error codes or information which is required in response to some particular API call and never state objects. On the other hand, the data sync service never manipulates the state of the VoteService and is solely responsible for data synchronization.

From the client's point of view, the web-client contains a copy of the whole VoteService state in a local data-store. This store is initially populated when the user selects an election event. Whenever the state of this election event changes, the data sync service pushes the new data to the web-client. The local mutation handler within the web application handles those messages and writes the new data into the local data-store.

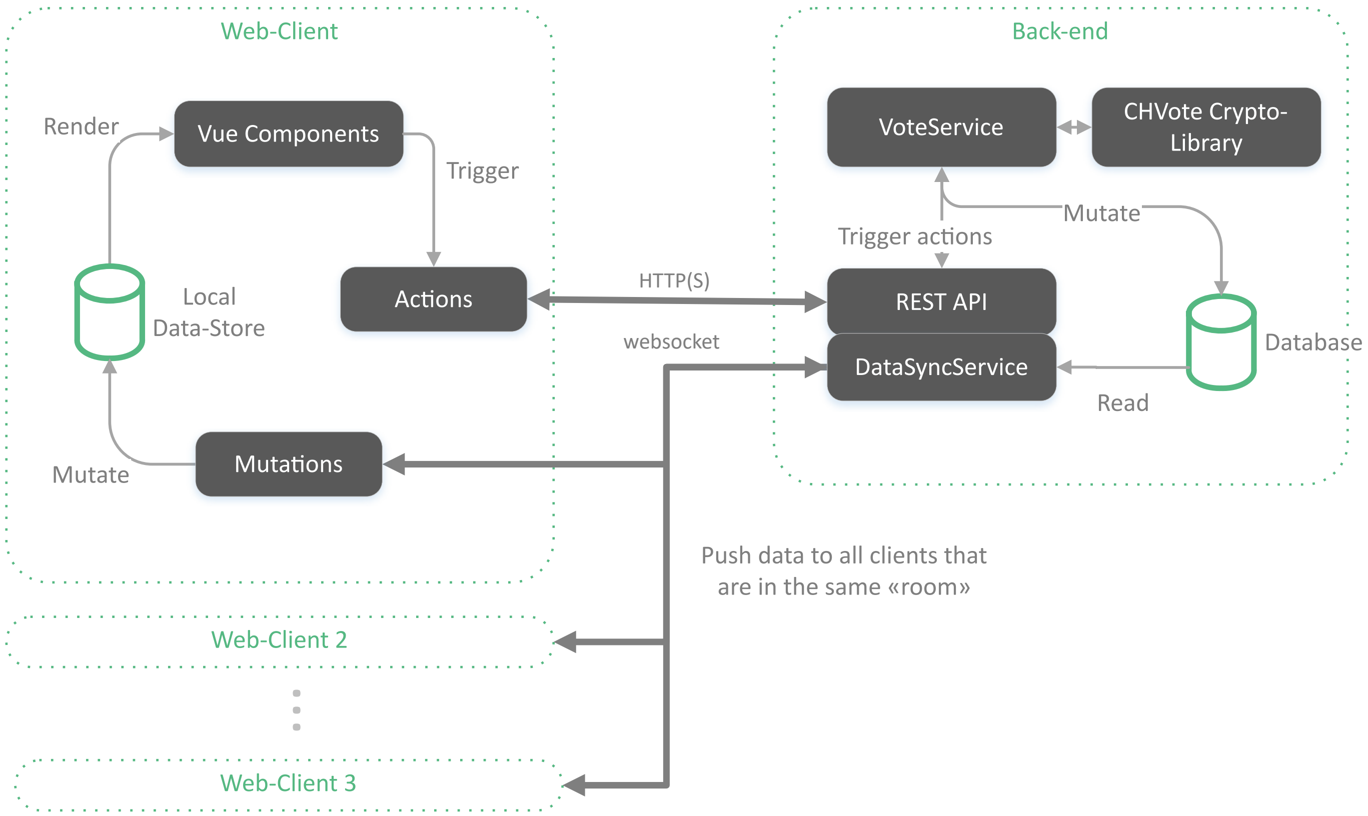


Figure 5.2: The architecture with the front- and the back-end. Both sides contain a database / data-store that stores the current state of an election event. Whenever some action is called on the server side by calling test REST API, the changes are written to the database and synchronized to all clients over web-sockets. The resulting manipulations on the local data-store are automatically updated in the user's view by binding the user controls to the local data-store.

Since the components of the web application pages are bound to the local data-store, all mutations are automatically displayed to the user. From those pages, the REST API can again be called, for example to cast a new ballot. The resulting state change would again be pushed to all clients. The client which performed the API request would additionally receive a success code or an error message in case of an error over HTTPS.

The architecture with all corresponding components is also shown in figure 5.2

5.3 Back-End

This section describes the internals of the back-end services.

5.3.1 VoteService

The VoteService builds a simplified e-voting ecosystem which provides functionality to conduct an election event by internally representing the state of a whole election event. In production, an e-voting system based on the CHVote specification would consist of multiple separate applications/services for each participant, such as a bulletin-board service for appending data to the public board or services for each election authority for performing their tasks. Additionally, several steps of the protocol would have to be executed on the client-side, such as generation of voter's ballots. In this application, it was reasonable to have the protocol functionality of all these parties combined within a single service.

We have decided on this VoteService centric approach mainly because we wanted to keep the whole protocol implementation within the central component and to avoid having protocol logic both on the client as well as on the back-end. The advantage of this approach is that in case the protocol undergoes any changes or if the application should be extended to support a different e-voting protocol in future, only the VoteService component (and of course its dependencies such as the crypto-library) will be affected or must be replaced. It also allowed us to implement the CHVote protocol almost identically as described in the specification. Since the actor's data and functions could be easily accessed from within our VoteService, passing data from one actor to another was as simple as setting an object property to some value.

Internally, we divided the VoteService into actors and states, as shown in figure 5.3: one actor for every protocol participant, providing the functionality a participant is responsible for, and a corresponding state-object for every actor, representing the participant's current state within a given election event.

The distinction between the actors and their states allowed us to easily serialize the state objects to JSON (a format that can be easily interpreted by the front-end) and made it easy to load and persist the state from and to the database. This measure was also necessary because of the way how our data synchronization between the clients and the back-end was implemented: By comparing and determining the differences between the state object before and after some VoteService actions, we can automatically find out the mutations that have been done to the state objects using the JSON Patch standard and generate operations to patch the client's local data-store in the same way. More about this technology can be found in the section 5.3.2.

The only common functionality between every state object is the ability to serialize the object to a JSON string. For this reason we had to write a custom transformer which tells the JSON parser how to serialize data-types such as mpz, byte arrays and custom classes. Luckily, python offers a way to easily serialize any custom object. By calling `object.__dict__` we can convert an object into a dictionary, as long as the transformer is able to serialize all properties of the object.

The following list shows all the state classes, figure 5.4 shows an UML representation:

- **BulletinBoardState**: holds all data that is publicly available on the bulletin board (the number of candidates, the tallied result).

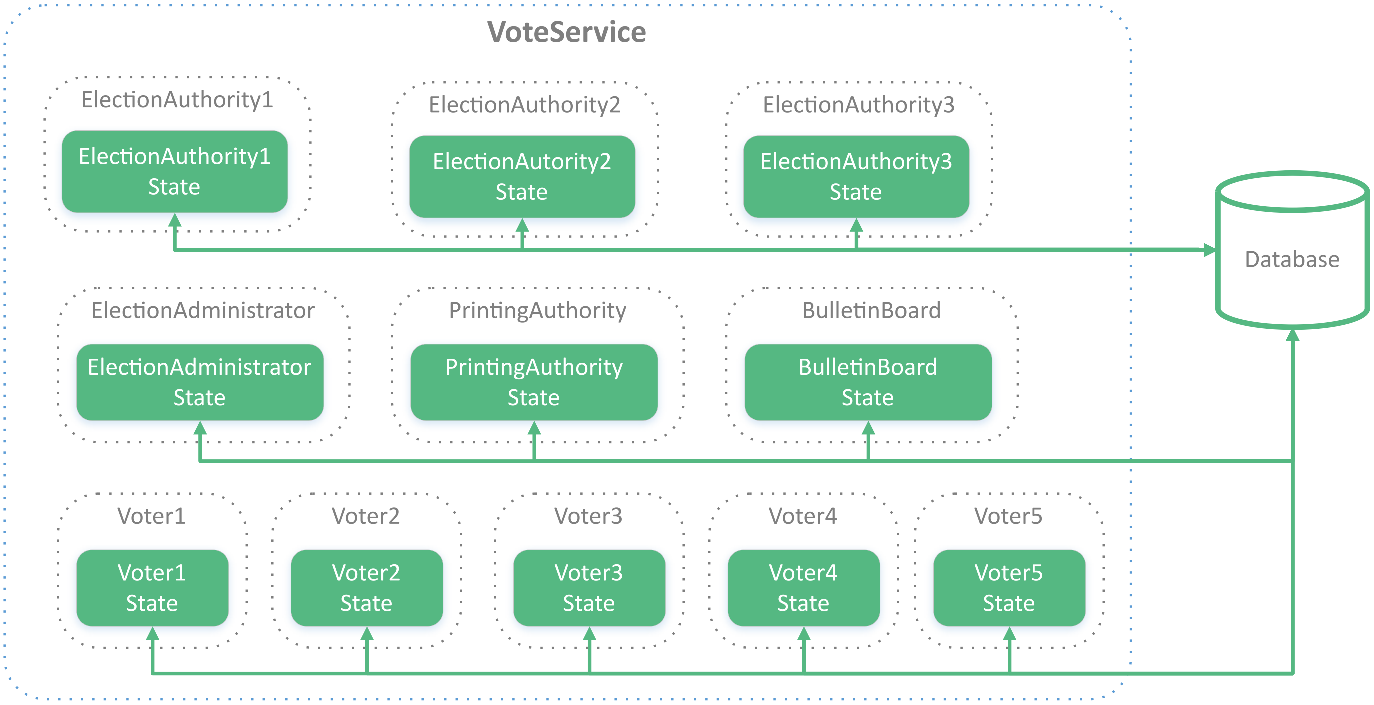


Figure 5.3: The internals of the VoteService: the functionality is divided into actor and state classes. The actors provide the functionality as described in the specification and by utilizing the crypto-library. The state is contained within the state classes that allow easy serialization both for persisting them to the database as well as to a JSON representation for the data synchronization.

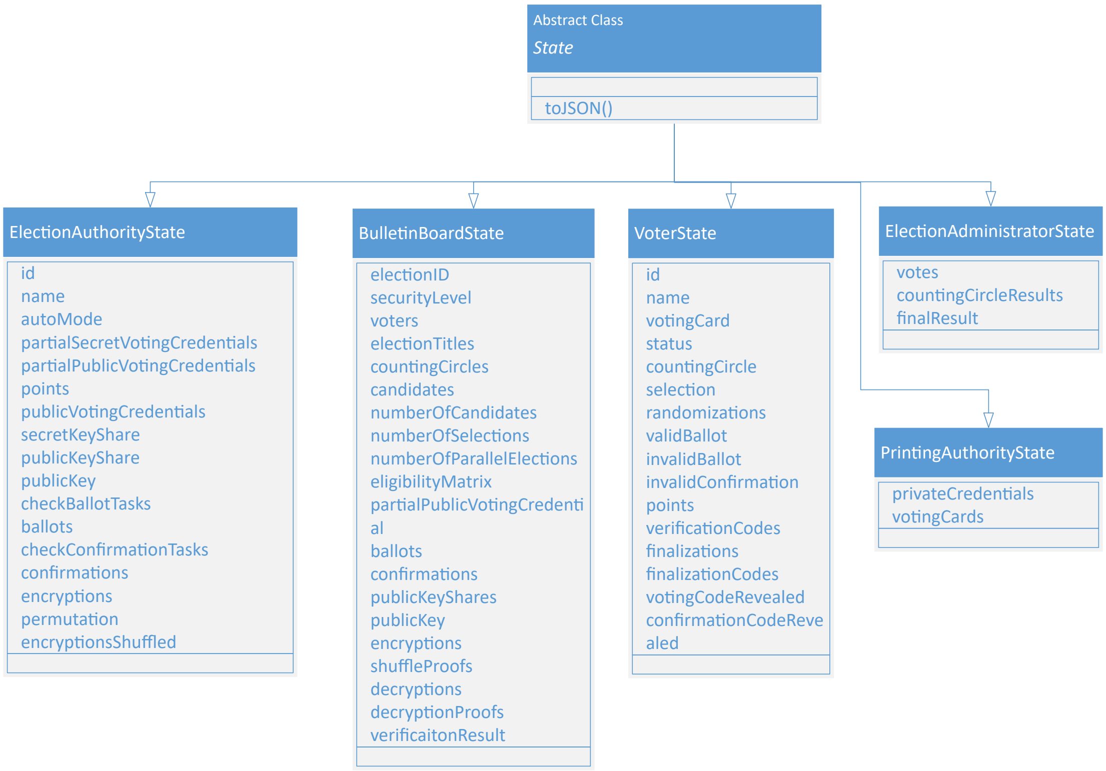
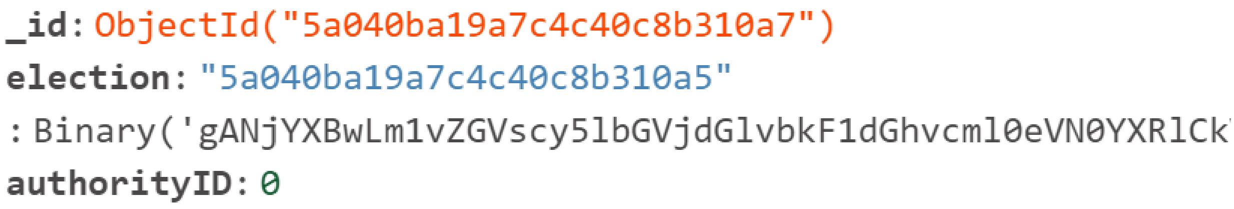


Figure 5.4: The state of an election event is broken down into separate state classes for every participating actor. The thing they have in common is that they are serializable to JSON.

- **ElectionAuthorityState**: holds all data that an election authority knows (e.g. the list of ballots, the secret key of an election authority).
- **VoterState**: since there is no distinction made between a voter and a voting client in the application, the VoterState contains the data of both the voter (e.g. the voting card) and the data typically known to the voting client (e.g. the points returned by the oblivious transfer).
- **PrintingAuthorityState**: holds the data known to the printing authority (e.g. the list of all voters private credentials and the voting cards).
- **ElectionAdministratorState**: holds all the data known to the election administrator.

Since our application supports multiple users working on different election events concurrently, the state of an election event cannot be kept in volatile memory, but needs to be persisted between every single request. For this reason different database systems and concepts were evaluated. We decided against a relational database system which would require a definition of a database schema since we wanted our state objects to be the only place where the schema would be defined. This "code-first" approach would make it easier to apply changes to the protocol in future.

For this purpose, MongoDB seemed like a good choice. Since there is no need to access and



```
_id: ObjectId("5a040ba19a7c4c40c8b310a7")
election: "5a040ba19a7c4c40c8b310a5"
: Binary('gANjYXBwLm1vZGVscy5lbGVjdGlvbkF1dGhvcml0eVN0YXRlCk
authorityID: 0
```

Figure 5.5: Example of a MongoDB collection (the equivalence of tables in other database systems). All states are stored as binary strings together with an identifier for the election-event as well as the election authority ID.

filter our data by arbitrary queries but only being able to save and load a state object of a particular election, the whole state is stored as a binary string in a MongoDB collection. The only additional attribute which is saved in the database alongside with the serialized state is the electionId which denotes which election event a particular state belongs to. An election contains multiple VoterStates and ElectionAuthorityStates. Therefore, these two states additionally require an electionAuthorityId and a voterId.

We described how the state classes are used to divide the data of the VoteService into smaller units. Similarly, the functionality of the VoteService is separated into classes, one for every actor of the protocol.

The common functionality, for example functions for loading the corresponding states from the database and one for persisting the states to the database, are contained in an abstract base class.

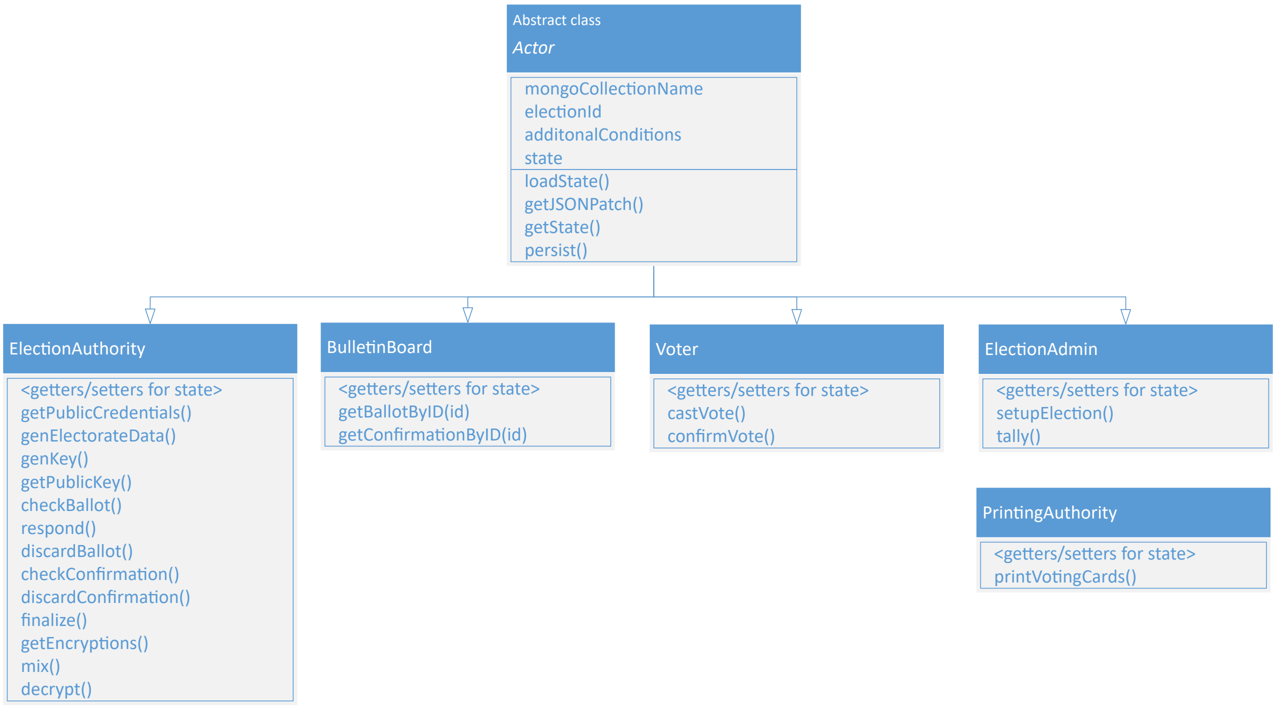


Figure 5.6: The functionality of the VoteService is divided into separate classes, one for each participating actor of the protocol.

5.3.2 Data-Sync Service

One of the big challenges of our application has been the synchronization of an election event's state between the back-end and the client's local data-store. As mentioned earlier, the web-application contains a local data-store, which is structured the same way as the states of the VoteService. As per our requirements, we wanted to achieve real-time data synchronization such that every web client observing a particular election event is informed of any changes of this election event's state. For this purpose we used web-sockets which - opposed to the HTTP protocol - make it possible to inform a web-client without having to rely on polling.

For the data synchronization we had to keep an eye on the performance of the data transfers since some state objects, especially the bulletin board and the election authority states, could grow big in size when they contained many ballots. We observed that the size of the whole state of an election event with 6 candidates and 10 voters, of which each had submitted at least one ballot and a confirmation, could easily reach 1 megabytes already. Admittedly, we did not notice any performance issues even with rather large election events. However, transferring the whole state of an election event after every single mutation did not seem like a proper solution.

When a client connects to the data sync service for the first time, it needs to get the full JSON representation of every state object of the VoteService. For this purpose we have implemented a "FullSync" method which populates the clients local data-store with the full state of an election.

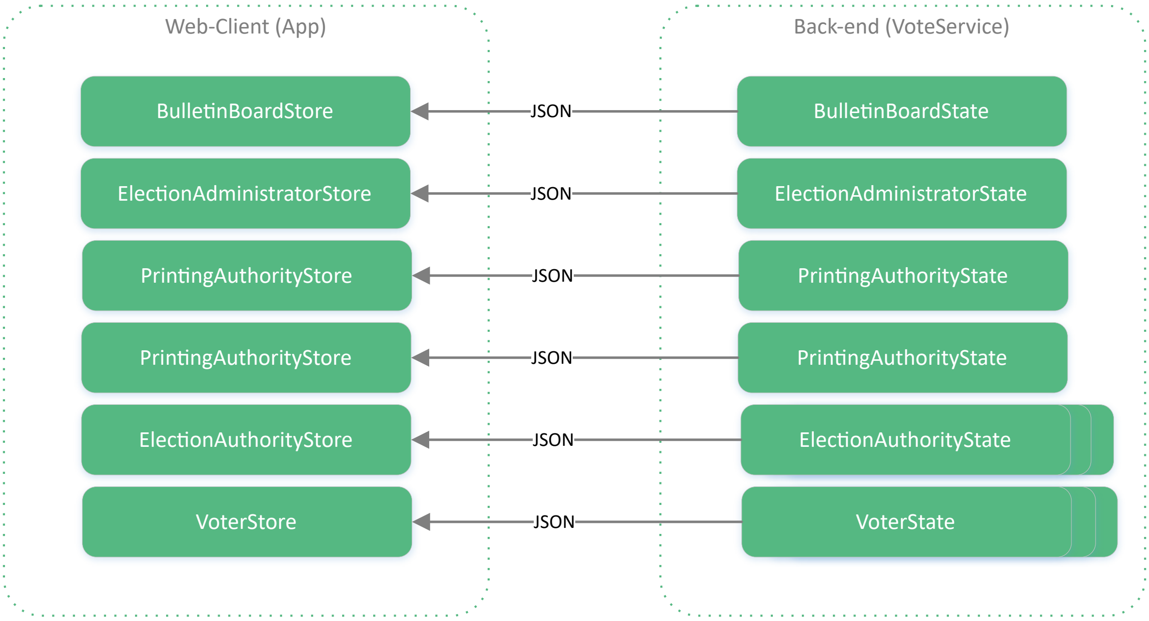


Figure 5.7: For every VoteService state there is a corresponding data-store on the client side. The client side data-store are initially populated with the whole dataset during the initial loading procedure of an election event.

After a client has populated its local data-store, future manipulations on the back-end are synchronized using the so called JSON Patch operations, which only contain the delta between the previous and the current state.

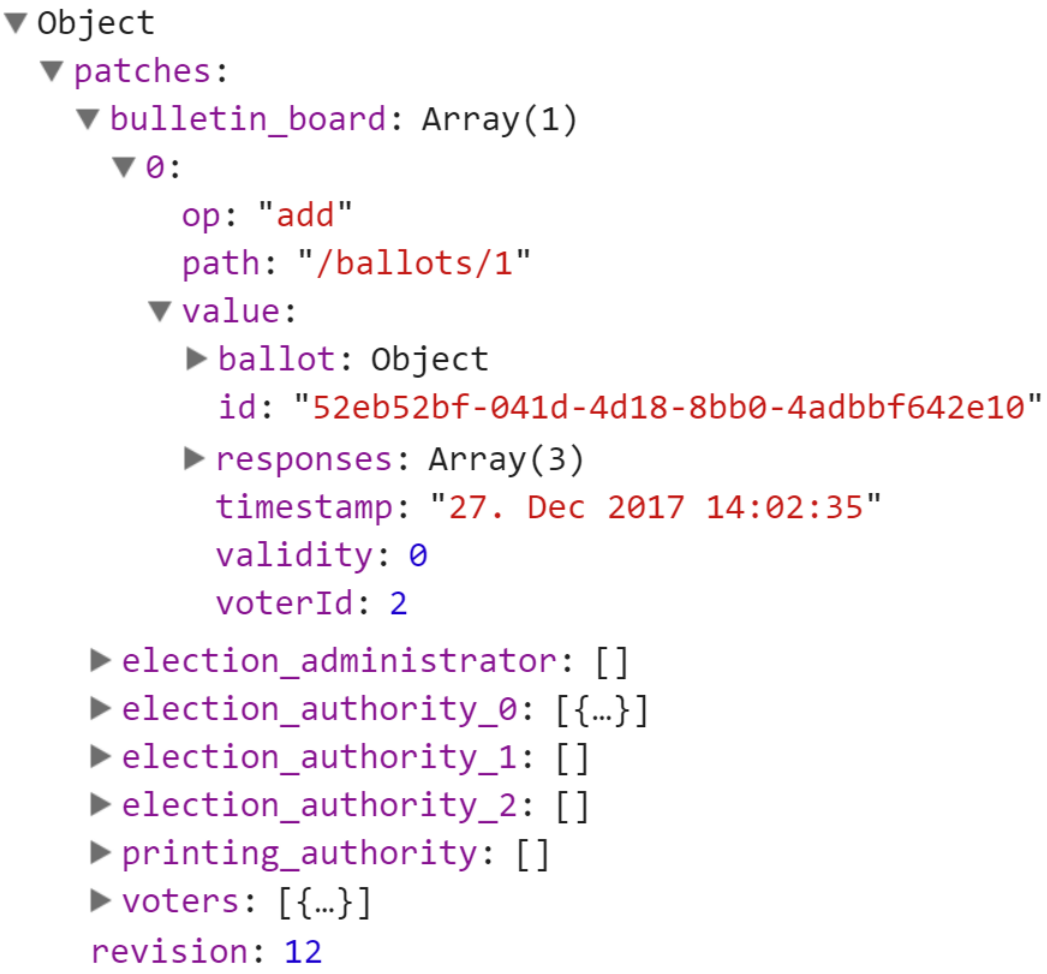
JSON Patch is a data structure for describing how to patch / modify a JSON object. The procedure is standardized and described in the RFC 6902 of the Internet Engineering Task Force (IETF). There exist JSON Patch implementations for many languages, including Python and JavaScript. We used JSON Patch to implement our incremental data synchronization as follows:

When the VoteService loads the state of its actor objects, it sets the `originalState` property of the actor to a deep copy of its state object. Mutations are always done only to the `state` property. Before calling the `persist()` method on an actor object, we use the Python JSON Patch library to determine the differences between the state and the `originalState` to find out if and what variables have changed within the states. As a result, we receive a set of JSON Patch operations which describe how the `originalState` could be patched, such that it becomes identical to the manipulated `state` object by calling

```
make_patch(json.loads(self.originalState.toJSON()), json.loads(self.state.toJSON()))
```

The result is an array of operations in JSON format that contains:

- the path of the manipulation;
- the type of operation (replace, add, remove, ...);
- the new value (if required).



```
▼ Object
  ▼ patches:
    ▼ bulletin_board: Array(1)
      ▼ 0:
          op: "add"
          path: "/ballots/1"
        ▼ value:
          ▶ ballot: Object
            id: "52eb52bf-041d-4d18-8bb0-4adbbf642e10"
          ▶ responses: Array(3)
            timestamp: "27. Dec 2017 14:02:35"
            validity: 0
            voterId: 2
    ▶ election_administrator: []
    ▶ election_authority_0: [{…}]
    ▶ election_authority_1: []
    ▶ election_authority_2: []
    ▶ printing_authority: []
    ▶ voters: [{…}]
    revision: 12
```

Figure 5.8: JSON Patch is an RFC standard that describes a procedure to incrementally patch a JSON object. By comparing two JSON objects, JSON Patch creates a set of operations that are needed to patch one object to match the other. We used JSON Patch to keep the client side data-stores in sync with the central database. The picture shows the resulting operations after submitting a new ballot.

For example: After casting a ballot, we might receive the following JSON Patch:

These JSON Patches are pushed to all the clients that need to receive the mutations and are applied to the local data-store which (under normal circumstances) contains the original state. After applying the JSON Patch, the data-store of all clients contains the same state of an election event as the back-end.

If for some reason a web-client does not receive a JSON Patch, its local data-store will no longer correspond to the back-end's version. This might happen because of network issues and an interrupted WebSocket connection or if applying the JSON Patch operations failed.

For this reason we have applied a data-store revision-number for every election event, which is incremented whenever some state is manipulated and persisted on the back-end. This revision number is sent alongside the JSON Patches to the clients and is also stored on the client side.

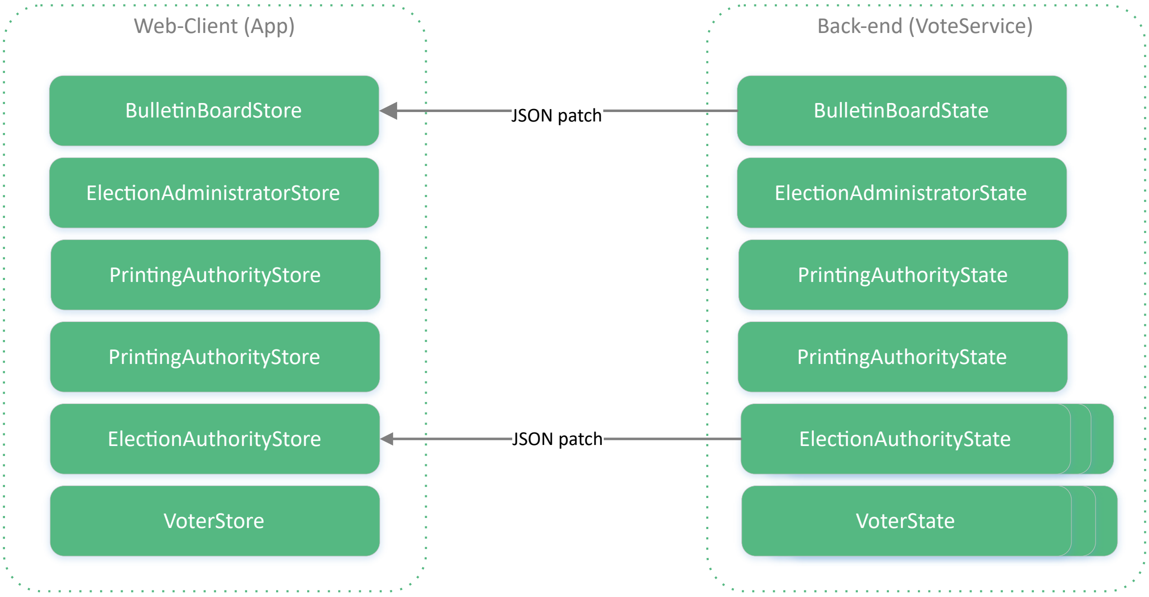


Figure 5.9: After the client side data-stores have been initially loaded, future mutations of the central database are synchronized to all clients by receiving and applying JSON Patches, which contain operations to patch the local data-stores to be in sync with the back-end.

Before applying the JSON Patches, the client checks if its local data-store's revision number is exactly 1 version behind the server's data-store revision, which normally will be the case. If the client detects that its revision number is 2 or more versions behind the server's, it will request a full-data synchronization over the DataSyncService to avoid out-dated, corrupted local data-stores.

During development we ran into an issue with the python JSON Patch library "python-json-patch v1.16" that we have been using. During some special cases the generation of JSON Patches failed and resulted in an exception when comparing objects where mutations were made to arrays within dictionaries - a combination which often occured in our data structures. After hours of debugging and analyzing the issue, we figured out a temporary workaround and reported the issue [2] to the developers of this library. A few days later a new version v1.20 of the library was released which fixed our issue.

5.3.3 REST API

The third component of the back-end is the REST API. Its responsibilities are to provide all the functionality of the VoteService to the web clients and trigger the data synchronization of the DataSyncService. Every function required by the front-end, such as `castVote()`, has a corresponding endpoint in the REST API service.

[2] https://github.com/stefankoegl/python-json-patch/issues/74

```
@main.route('/castVote', methods=['POST'])
@cross_origin(origin='*')
def castVote():
    data = request.json
    electionId = data["election"]
    selection = data["selection"]
    voterId = data["voterId"]
    votingCode = data["votingCode"]

    if len(selection) == 0:
        return make_error(400, "Empty selection")

    try:
        svc = VoteService(electionId) # prepare VoteService

        svc.castVote(voterId, selection, votingCode) # perform vote casting

        patches = svc.persist()  # persist modified state and retrieve JSON Patches

        syncPatches(electionId, SyncType.ROOM, patches)  # send the JSON Patches to all
         ↪   clients

    except Exception as ex:
        return make_error(500, str(ex))

    return json.dumps({'result': 'success'})
```

The API can be reached by sending a HTTP(S) POST request to our web server hosting the back-end services. The URL defines the function to be executed. for example: a POST request to https://<server>:5000/castVote/ would call the above function. The required parameters are provided in the POST body.

Step one: parameters are extracted from the POST request and validated if necessary. Step two: a VoteService object is instantiated by passing the electionId to the constructor. The VoteService will internally load the states of the corresponding election event from the database and instantiate the actor objects such as the election authorities.

Now the VoteService can execute the function which the user intended to call, for example "CastVote". By calling the function `persist()`, the new state is written to the database and the JSON Patches of all mutations caused by the "CastVote" function call are determined, returned and can be sent to all clients with the help of the DataSyncService.

The sequence diagram in figure 5.10 shows how the vote casting use case is implemented within the back-end and how all the components work together. The REST API instantiates a new VoteService object which itself instantiates the required actor objects by loading their states from the database. The "castVote" function then performs the required steps to create a new ballot and ballotCheckTask for this particular election authority. After persisting the changes to the database, the returned JSON Patch operations are transmitted to the clients by passing them to the DataSyncService.

The diagram is simplified and reduced to the election authority as the only actor; in reality the bulletin board and the voter are also involved in this use case but behave the exact same way as the election authority.

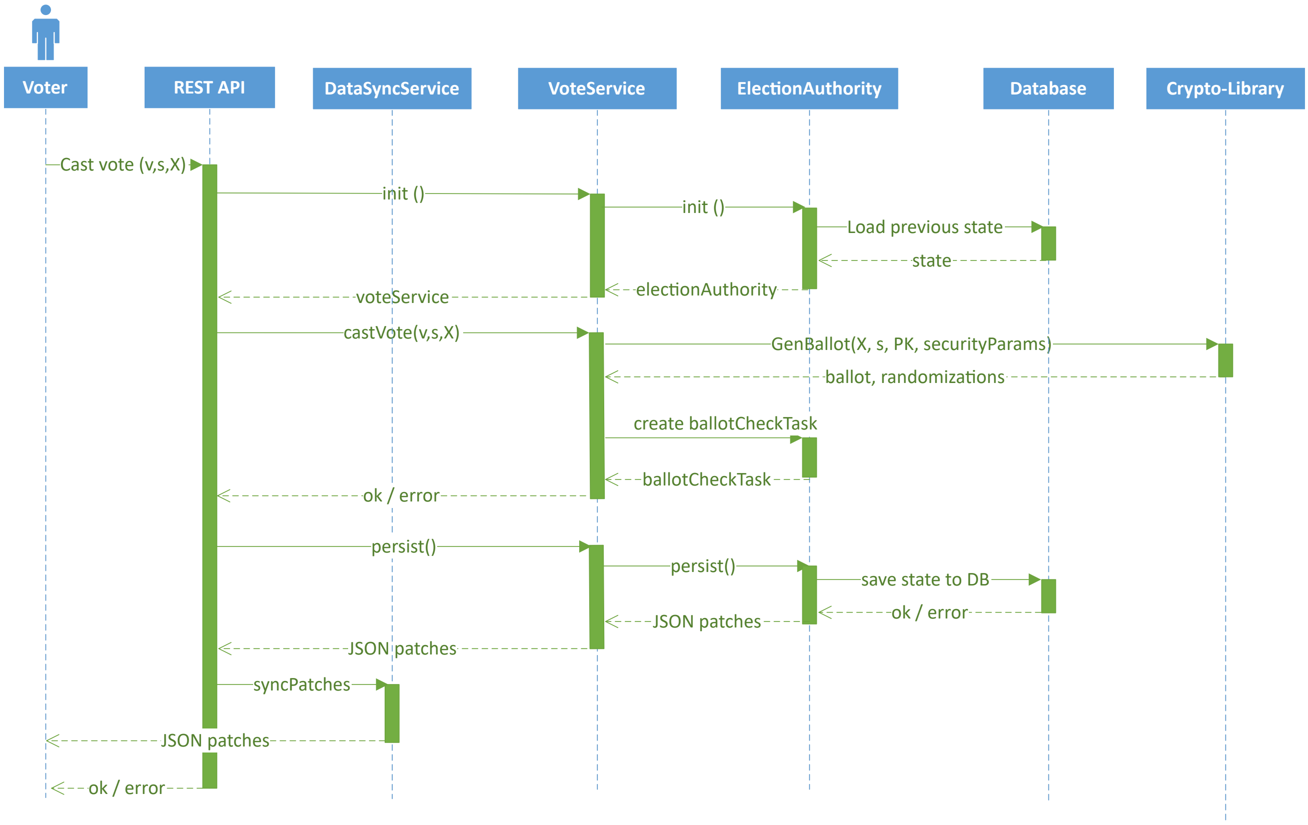


Figure 5.10: Vote casting sequence diagram, showing how all components of the application are working together during vote casting

5.4 Crypto-Library

5.4.1 File Structure

We decided to put every algorithm of the specification in its own file together with related unit tests. The files are structured according to the actors of the protocol, for example:

- **Common**: contains common cryptographic algorithms and the security parameters used by multiple algorithms.
- **ElectionAuthority**: contains all the algorithms used by the election authority.
- **PrintingAuthority**: contains all the algorithms used by the printing authority.
- **VotingClient**: contains all the algorithms used by the voting client.
- **ElectionAdministration**: contains all the algorithms used by the election administrator.

- **Utils**: contains helper classes and miscellaneous utility functions.

5.4.2 Public Parameters

There exist two types of public parameters:

The **security relevant parameters**, e.g:

- the order of the prime groups: p, ${}'p$, $\hat{p}$;
- the length of the voting, confirmation, verification and finalization codes;
- the number of authorities: s;

and **public election parameters**, e.g.:

- the size of the electorate: N_E;
- the number of candidates: n;
- the list of candidate descriptions: c.

The security parameters are typically used within the algorithms and remain unchanged for a longer time period, whereas the public election parameters are different for every election event.

The object `SecurityParams` holds all security relevant parameters and is injected as an additional function argument to all algorithms. Several different `SecurityParams` objects are created initially, which contain all the parameters according to the recommendations in the CHVote specification document ("level 0" for testing purposes and "level 1" through "level 3" for productive use). For simple unit and debugging purposes, we can inject the "level 0" object while in production level 1 - 3 are used.

The public election parameters, on the other hand, are directly passed to the algorithms by the calling party. If an algorithm needs to know certain election parameters (like the size of the electorate N_E), these values are typically derived from vectors that they have access to, so they do not require specific knowledge of these parameters.

5.4.3 Coding Style

The following source code sample shows a typical implementation of an algorithm (in this example, algorithm 7.18 according to the CHVote specification).

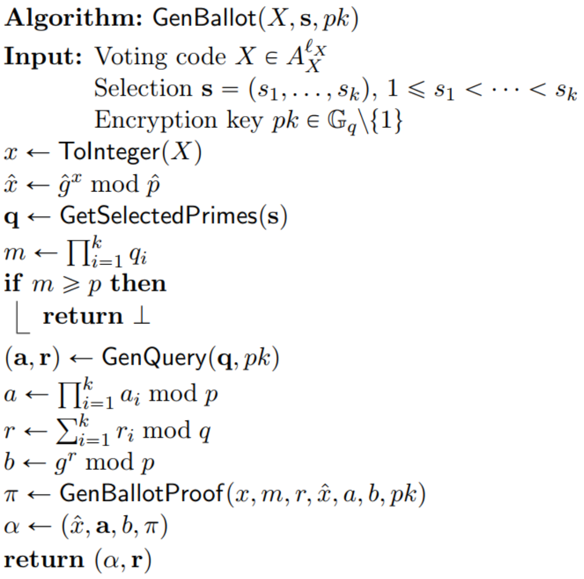
```
import gmpy2
from Utils.Utils                    import *
from Crypto.SecurityParams          import *
from VotingClient.GetSelectedPrimes import GetSelectedPrimes
from VotingClient.GenQuery          import GenQuery
from VotingClient.GenBallotProof    import GenBallotProof
from Types                          import Ballot

def GenBallot(X_bold, s, pk, secparams):
    """
    Algorithm 7.18: Generates a ballot based on the selection s and the voting code X.
 ↪   The
    ballot includes an OT query a and a proof pi. The algorithm also returns the random
    values used to generate the OT query. These random values are required in Alg. 7.27
    to derive the transferred messages from the OT response, generated by Alg. 7.25.

    Args:
        X_bold (str):                   Voting Code X ∈ A_X^l_X
        s (list of int):                Selection s = (s_1, ... , s_k)
        pk (mpz):                       ElGamal key pk ∈ G_p \ {1}
        secparams (SecurityParams):     Collection of public security parameters

    Returns:
        tuple:                          alpha = (r, Ballot) = (r, (x_hat, a, b, pi))
    """
    AssertMpz(pk)
    AssertList(s)
    AssertClass(secparams, SecurityParams)

    x = mpz(StringToInteger(X_bold, secparams.A_X))
    x_hat = gmpy2.powmod(secparams.g_hat, x, secparams.p_hat)

    q_bold = GetSelectedPrimes(s, secparams)                    # q = (q_1, ... , q_k)
    m = mpz(1)

    for i in range(len(q_bold)):
        m = m * q_bold[i]

    if m >= secparams.p:
        return None

    (a_bold, r_bold) = GenQuery(q_bold, pk, secparams)
    a = mpz(1)
    r = mpz(0)

    for i in range(len(a_bold)):
        a = (a * a_bold[i]) % secparams.p
        r = (r + r_bold[i]) % secparams.q

    b = gmpy2.powmod(secparams.g,r, secparams.p)
    pi = GenBallotProof(x,m,r,x_hat,a,b,pk, secparams)
    alpha = Ballot(x_hat,a_bold,b,pi)

    return (alpha, r_bold)

class GenBallotTest(unittest.TestCase):
    def testGenBallot(self):
        selection = [1,4]       # select candidates with indices 1,4
        (ballot, r) = GenBallot(unittestparams.X, selection, unittestparams.pk,
         ↪  secparams_l0)
        print(ballot)
        print(r)

if __name__ == '__main__':
    unittest.main()
```

All algorithms contain a short description, which was taken as is from the specification document, as well as a comment (Google-style documentation string), which can be used to automatically generate code documentation. The algorithm itself has been implemented as close to the specification as possible, using the same variable names and (as far as the language supports it) similar control structures:

- the suffix `_bold` for emphasized (bold) variables, e.g. `p_bold` for $\mathbf{p}$;
- the suffix `_hat` for variables with a hat, e.g. `a_hat` for $\hat{a}$;
- the suffix `_prime` for variables with a prime, e.g. `a_prime` for a';
- etc.

Each file also contains unit tests of the specific algorithm (if unit testing was considered useful for the particular algorithm).

The following example shows the similarities between the algorithm pseudo code and the actual implementation in Python:



```
Algorithm: GenBallot($X, \mathbf{s}, pk$)
Input: Voting code $X \in A_X^{\ell_X}$
       Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k$
       Encryption key $pk \in \mathbb{G}_q \backslash \{1\}$
$x \leftarrow \mathsf{ToInteger}(X)$
$\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$
$\mathbf{q} \leftarrow \mathsf{GetSelectedPrimes}(\mathbf{s})$
$m \leftarrow \prod_{i=1}^{k} q_i$
if $m \geqslant p$ then
    return $\perp$
$(\mathbf{a}, \mathbf{r}) \leftarrow \mathsf{GenQuery}(\mathbf{q}, pk)$
$a \leftarrow \prod_{i=1}^{k} a_i \bmod p$
$r \leftarrow \sum_{i=1}^{k} r_i \bmod q$
$b \leftarrow g^r \bmod p$
$\pi \leftarrow \mathsf{GenBallotProof}(x, m, r, \hat{x}, a, b, pk)$
$\alpha \leftarrow (\hat{x}, \mathbf{a}, b, \pi)$
return $(\alpha, \mathbf{r})$
```

```
x = mpz(StringToInteger(X_bold, secparams.A_X))
x_hat = gmpy2.powmod(secparams.g_hat, x, secparams.p_hat)
q_bold = GetSelectedPrimes(s, secparams)

m = mpz(1)
for i in range(len(q_bold)):
    m = m * q_bold[i]

if m >= secparams.p:
    return None

(a_bold, r_bold) = GenQuery(q_bold, pk, secparams)
a = mpz(1)
r = mpz(0)

for i in range(len(a_bold)):
    a = (a * a_bold[i]) % secparams.p
    r = (r + r_bold[i]) % secparams.q

b = gmpy2.powmod(secparams.g,r, secparams.p)
pi = GenBallotProof(x,m,r,x_hat,a,b,pk, secparams)
alpha = Ballot(x_hat,a_bold,b,pi)

return (alpha, r_bold)
```

5.4.4 Return Types

In most cases, when an algorithm returns more than a scalar datatype, tuples are used. Tuples allow returning multiple values from a function:

```
def foo():
   return (1, 2)

def main():
   a, b = foo()
```

This way large parts of the source code looked very similar to the pseudo code in the CHVote specification. For more complex data types or return values that were used more than once, named tuples were applied. The data type "namedtuple" is like a lightweight class and allows access to named properties.

```
Ballot = namedtuple("Ballot", "x_hat, a_bold, b, pi")

def main():
   Ballot b = getBallot()
   x_hat = b.x_hat
```

By following this approach we could avoid having lots of container classes only used to pass data structures between the algorithms.

5.5 Front-End

The front-end was the most important component of the project, since we put focus mostly on the visualization and less on the actual e-voting system. Displaying the rather large amount of voting specific data and large numbers ($>=$ 1024 bit) required a clean and well-structured layout and a modular component design. Luckily, the framework we had chosen, VueJS, did very well in supporting us to meet exactly those requirements. We tried to follow the design patterns and best practices proposed by the developers of VueJS wherever possible.

This section describes which concepts of VueJS we used and how we adapted them to our needs.

5.5.1 Components

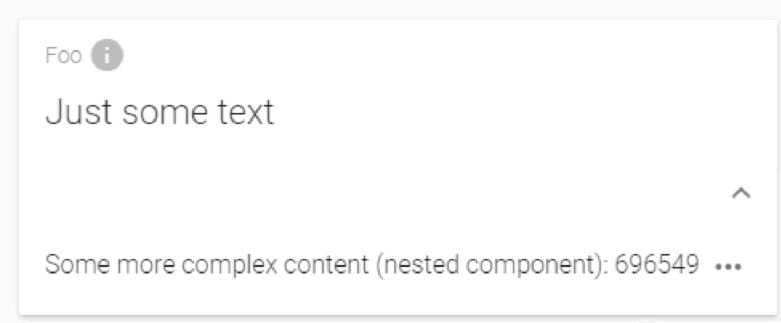
Components are the basic building blocks of the VueJS framework. The application itself is a component, every page of the application is a component and the pages typically contain lots of components, one for every object like form controls, buttons or custom controls such as the ballot-list, etc. The concept of components encourages to create reusable modules, provides a way to structure the application into smaller units and makes the resulting HTML template more expressive and easier to read.

We have created our own VueJS components for every control that we used more than once. For example, the ballot list that is shown in the bulletin board view as well as in the election authority view, the labels for displaying truncated large numbers or the cards used as our standard means for displaying data have all been turned into custom components. One of the most useful features of VueJS components is the concept of slots. By defining one or multiple slots within a component's template markup, it becomes possible from the parent of a component to embed content into different locations (slots) within the component's HTML template.

We have been using slots to create our card component, which can display information either as the main content of the card or within the expandable area on the bottom of the card:

```
<DataCard title="Foo" :expandable=true>
    Just some text
    <p slot="expandContent">More complex content <BigIntLabel
     ↪   :mpzValue="publicKey"></BigIntLabel>
    </p>
</DataCard>
```

The first line "Just some text", which gets inserted into the default unnamed slot, could as well be passed as a string parameter to the data card component. However, as things are getting more complicated, one might like to place arbitrary HTML or even another VueJS component inside the expandable content of our data-card. In such cases, component parameters will not work as they only accept primitive data types. Slots, on the other hand, allow arbitrary content to be injected. The following code shows how the data-card component is implemented:



Foo

Just some text

Some more complex content (nested component): 696549 •••

Figure 5.11: Our data-card component that allows content to be injected in different places within its DOM by using the slot-concept of VueJS.

```
<template>
    <v-card class="dataCard">
        <ConfidentialityChip v-if=showConfidentiality"' :type="confidentiality"
         ↪   class="confidentialityChip" />
        <v-card-title primary-title class="dataCardTitle">
            <div><span class="label grey--text">{{title}}
              <v-tooltip top>
                <v-icon v-if="!disableTooltip" color="grey lighten-1"
                 ↪   slot="activator">info</v-icon><span>Programmatic tooltip</span>
             </v-tooltip></span>
            </div>
        </v-card-title>
        <v-card-text class="dataCardContent">
                <slot></slot>
        </v-card-text>
        <v-card-actions v-show="expandable">
            <v-btn icon @click.native="showExpander = !showExpander">
                <v-icon>{{ !showExpander ? 'keyboard_arrow_down' : 'keyboard_arrow_up'
                 ↪   }}</v-icon>
            </v-btn>
        </v-card-actions>
        <v-slide-y-transition v-show="expandable">
            <v-card-text v-show="showExpander">
                <slot name="expandContent">
        </slot>
      </v-card-text>
    </v-slide-y-transition>
  </v-card>
</template>
<script>
  import { mapState } from 'vuex'

  export default {
   data: function () {
    return {
     showExpander: false
    }
   },
   computed: {
    ...mapState({
     showConfidentiality: state => state.showConfidentiality
    })
   },
   props: {
    title: {
     type: String,
     required: true,
     default: 'Title'
    },
    expandable: {
     type: Boolean,
     required: true,
     default: false
    },
    confidentiality: {
     type: String,
     required: true,
     default: 'public'
    },
    disableTooltip: {
     type: Boolean,
     required: false,
     default: false
    }

   },

   mounted () {
   }
  }
</script>
```

The first part within the <template> tag describes the HTML markup as well as the slots mentioned above.

The <script> tag contains the actual logic of the component, similarly to the controller in other SPA frameworks. The data object contains variables which are defined and valid only locally within the component. The computed object maps variables from the central data-store to a local variable which is reactively bound to the data-store. Whenever the value of the given variable changes in the data-store, the computed property is automatically updated. From the template, we can access both local data as well as computed properties. Computed properties can also be used whenever a local variable needs to be formatted or in some way manipulated before it is displayed in the component's template.

The third source of data are properties ('props'), which are passed as arguments from the parent component. They are typically used to define options for a component.

Additionally, components may contain `methods`, typically used for event-handlers like button clicks and event hooks like `mounted`, `beforeDestroy`, `beforeCreated` to influence the component's construction/destruction at different times during a component's lifecycle.

5.5.2 Centralized Data-Store & Flux Pattern

One of the big challenges regarding the front-end architecture was to decide how and where the data would be stored. Clearly, since we already had divided an election event's data into one state for every actor and given that every actor also had its corresponding view in our front-end, simply saving the data to the respective component was our first thought. Although a voter mainly needs to access his own data contained in the voters state, some data must be shared between multiple components, for example the information on the bulletin board.

Since we wanted to avoid keeping the data redundant in multiple components, we decided to use the Flux design pattern in our front-end. The basic idea of the flux pattern is to have a single, central data source where all the data is stored and is accessible to all components. This single data source is called a "store" by Flux terminology. VueJS has its own implementation of the Flux pattern called *Vuex*. Another important concept is that components can freely access the data in the store. However, they are not allowed to change the data, at least not directly. Instead, if a component wants to manipulate data in the store, this has to be done by calling **mutation functions**. Forbidding manipulation of the store makes it much easier to keep track of where mutations came from.

Our web application's data-store is divided into multiple modules, one data-store module for each corresponding state of the back-end. In reality, all these data-stores are part of one single data-store, but having multiple modules allows us to structure the mutation and getter-functions and help to avoid naming conflicts by having separate namespaces for every module.

The data-store can be accessed from any component by defining a computed property. If the computed properties have to perform some formatting, aggregation, filtering etc. on the state variables and are used from within multiple components, it is also possible and recommended to write getter-functions in the data-store to avoid redundancy.

5.5.3 Internationalization (i18n)

All text visible to the user on the front-end of the application is internationalized, i.e. the display language can be changed at any time by the user. The default language provided is English and translations for German have been added. The main language, English, and the manual translations for German are stored in the translations preset YAML file `frontend/src/translations.preset.yaml`. By using the translation script `frontend/src/translate.js`, those languages stored in the preset file can be translated automatically to predefined target languages. Those automatic translations (provided via Google Translate) are stored in the YAML file `frontend/src/translations.yaml` together with the main language and manual translations. This file is used to provide translations for the front-end application.

5.5.4 Development Environment

Webpack was chosen as a working environment for the development phase of the project. Webpack is a module bundler, i.e. a piece of software which generates static assets from JavaScript libraries including all dependencies, CSS files, images etc. Additionally, webpack features a development web server, which aids in fast development. While the development server is running, any changes in the application's source code are detected and assets are regenerated on the fly. This way, any changes in the application are directly visible in the browser.

5.5.5 Staging Environment

In order to deploy the application to an environment for staging purposes, we decided to use Docker together with Docker Compose. The Docker Compose file `docker-compose.yaml` stores the configuration for three distinct Docker containers: a MongoDB service, the back-end and the front-end. By running `docker-compose build` in the project's root directory, those three containers can be built. After building the containers, the services can be started by running `docker-compose up`. The application will then be available via `localhost:8080` (TCP).

5.6 Challenges

This chapter describes some of the challenges encountered during the application development.

5.6.1 WebSocket Subscription Concept

Since our application allows multiple election events to exist at the same time, the question arose how to let only those clients which are observing one specific election event receive websocket messages related to this particular election event in case an action has been triggered on the back-end.

There are some similarities between this problem and the one a typical chat with multiple chat-rooms has: only those users who have actually joined the chat-room should be notified of the new posts for this chat-room. We have adapted this "chat-room" concept to our case by defining a room to be equal to an election event.

We can assume that all the pages that actually show data of an election event require the election event's id to be passed as a part of the URL. For example: /BulletinBoard/1 is the route to reach the BulletinBoard view of the election with id 1.

Whenever a route is called that contains the argument 'electionId', we need to make sure that the client has joined the room of this election event. We have therefore set a global variable called `joinedElectionId` to match the id of the election event a user has joined. We have created a VueJS "mixin" (kind of a plug-in) that can be added to each election page, which makes sure that if the client has not yet joined the corresponding election event's room, it emits a "join" request to the socket.io server, passing along the electionId:

```
export default {
  created () {
    if (this.$store.getters.joinedElectionId !== this.$route.params.electionId) {
      this.$socket.emit('join', {election: this.$route.params['electionId']})
    }
  }
}
```

On the server side we have defined a socket.io listener called "join" which removes the calling client from every room before joining the room of the requested election event. There is only one exception: the client cannot leave the room that corresponds to the "sid" of the request, since this is basically the channel over which the "join" request is handled.

```
@socketio.on('join')
def on_join(data):
    electionID = data['election']
    for room in rooms():
        if room != request.sid:                                 # do not leave the room
         ↪  of the current connection
            leave_room(room)
    join_room(electionID)

    syncService.fullSync(electionID, SyncType.SENDER_ONLY)      # Perform a full
     ↪  data-sync

    emitToClient('joinAck', electionID, SyncType.SENDER_ONLY)    # send an
     ↪  acknowledgement/confirmation to the client
```

The `joinAck` handler in the web application will then set the `joinedElectionId` variable to the just joined election id, such that the join will only be called once or until the user chooses a different election.

5.6.2 Python Issues

During the project we experienced a few issues with Python as the programming language which we used for the crypto-library and the back-end. In particular, we observed the following issues:

- Performance issues due to Python being an interpreted language.

- Function overhead: function calls in Python seem to be very slow, especially when using recursions (such as recHash).
- Strongly dynamic typing vs. static typing: the Python interpreter needs to inspect every single object during run time (be it an integer or a more complex object).
- Surprisingly, the *BigInteger* library turned out to be not as fast as using directly the GMP library, and using the GMP bindings also meant having an overhead compared to the native BigIntegers.

As performance was not the main focus of our application, these issues were mostly ignored.

The following issues were more problematic for the project:

- Because of the dynamic typing, the code becomes much more error-prone, as there is no standard checking of function argument types etc. We have tried to overcome this issue by using asserts to check the types of input parameters wherever it was useful.
- There are little to no standards regarding the project structure when using our "crypto-library" approach. Most solutions depend on paths set as environment variables or absolute imports, which we wanted to avoid. We have chosen to use relative imports and define the crypto-library as a module, which explains why there are many empty "‘__init__.py"’ files in our solution, which are required for this module-approach.

For detailed information regarding the performance issues that we have experienced see [5] and [6]. Based on the reasons above we would not recommend using Python for similar or larger projects. Python is indeed a very handy language to write quick prototypes and proof of concepts, but issues become more frequent in larger projects.

5.7 Automatic Task Processing for Election Authorities

Every election authority normally has to manually process all incoming tasks such as ballot checking, confirmation checking, mixing and decrypting. As it can become cumbersome repeating the same steps multiple times during a short presentation, it would seem reasonable to perform these tasks manually only for the first election authority. Thus, we have implemented an automatic-mode in which an election authority automatically performs all tasks.

There were several different ways to implement this feature. One possibility would be building a service which runs in the background and regularly checks for new tasks and processes them. Since each task requires a preceding action (for example: a Ballot-Check-Task requires a voter casting a vote), and since it was reasonable for our use-cases to assume that the authorities perform the tasks sequentially, we chose the simplest approach.

When a user casts a ballot and a ballot-check-task is created, we check if the first authority is set to automatic. If yes, the function for checking this ballot is automatically called for the first election authority. This function checks again if the next election authority is set to automatic and recursively calls itself if that is the case.

This means that if the first authority is in manual mode and the other two are set to automatic, they both wait with their execution until the first election authority has manually started executing the task. This strict sequential order is only required by the protocol for the mixing task, all other tasks could be called in any order. If desired, this behavior could also be implemented with our approach by simply executing the function for every authority with auto-mode set to true.

5.8 Testing

We have applied different testing concepts for the several components of our application.

The front-end has been tested using manual test-cases. One of the test-cases is shown as follows, the others can be found in the appendix. We decided not to use automatic end-to-end or unit testing on our front-end because this would have generated a lot of work both for learning and integrating the testing frameworks like karma and selenium, and for writing the actual tests.

For our back-end, we have used automatic unit testing wherever it made sense, especially for the CHVote algorithms. Most basic algorithms which were used within other algorithms, such as hashing, prime number generation etc., have unit-tests provided within the same file. More complex algorithms are not tested with unit-tests, especially when they require input generated by other algorithms. Algorithms like genShuffleProof and checkShuffleProof are tested by our voteSimulation test. If a shuffleProof can be generated and is recognized as a valid proof by checkShuffleProof and passes all internal asserts, we assume that it is working properly.

Additionally, we have used asserts a lot in our backend to compensate for the missing of static typing of the Python programming language and to make sure that both the input as well as the output of the algorithms are correct in terms of boundaries, dimensions, etc.

Table 5.1: Test Case «Pre-Election»

Test-Case	1. Pre-Election
Description	This test covers all the pre-election steps, including the creation of a new election, setting it up from the election administration view and the printing- and delivery of the voting cards
Precondition	
Postcondition	• The election event is in the status "Election" • The voters have received a voting card
Steps	1. Start the application 2. Choose "Election Events" from the main menu 3. Click on "Create new election event" 4. Enter a name for the election event, choose a security level and click on "create" 5. The "Election Admin" tab should now have an interaction notification 6. Visit the "Election Admin" view 7. Enter at least 3 for the number of voters, at least 3 different candidates and 1 for the number of selection and click on "Setup Election Event" 8. The "Printing Authority" tab should now have an interaction notification 9. Visit the "Printing Authority" view 10. Click on "Print Voting Cards" 11. The voting cards for all voters should now be displayed 12. Click on "Deliver Voting Cards To Voters" 13. The voting cards should now disappear and the "Voters" tab should now have an interaction notification 14. Visit the Voters view, select a voter and check that the voting card is displayed correctly

6 Conclusion

The new CHVote specifications seem like a real breakthrough in e-voting. Many of the technical limitations which prevented the current systems like the one of Geneva from being used as a large scale e-voting platform, can now be solved. However, e-voting must ultimately be approved both in politics as well as by the Swiss citizens. The complexity and cryptographic nature of e-voting makes it difficult for ordinary citizens to fully understand why and how this protocol works. The missing of knowledge might even result in mistrust towards e-voting. Even though the protocol is very complicated, our application allows users to gain a better understanding of e-voting by visualizing the internals of Geneva's next generation e-voting protocol. An application like ours might not be enough to reach vast majority of the population and change their opinion about e-voting. However, the authors of the CHVote specifications intend to use our application for future presentations of their protocol, which might positively influence the attitude of their audience.

Finally, we would like to reflect on some aspects of our bachelor thesis: especially in the beginning of the project, we have underestimated the amount of work required for working out goals and a concept for the architecture and user interface. Regular meetings with our supervisors and the agile project methodology using prototyping and mockups helped a lot to create a common understanding and a concept for our application.

With all the envisaged technologies and frameworks being relatively new and without prior knowledge and experience using them, we were unsure whether we chose the right tools and if we would succeed with our concept. However, especially Vue.js turned out to be the perfect framework for developing the front-end of the application. It allowed us to rapidly develop an intuitive user-interface. Even though the framework is relatively new and lightweight, we haven't been missing any functionalities or libraries. As for the back-end, we still believe that Python isn't the best language for implementing cryptographic protocols for the reasons we mentioned in section 5.6.2. However, these problems did not hinder us from meeting all the requirements, including the optional can-criteria.

Especially during the first implementation phase, we were able to follow the time schedule as planned. In the second phase, we have changed the order of some of the planned features because some features required less, and some more time than we expected.

Implementing the e-voting protocol turned out to be a very enriching experience through discovering new programming languages, building up know-how in cryptography as well as deep-diving into the technology behind the e-voting protocol. This project enabled us to get a better insight into how electronic voting could look like in a few years and even contribute a small part to its future development.

Statement of Authorship

We hereby declare that we made the submitted thesis independently and without use of others than the indicated sources.

Ort, Datum:	Bern,	
Namen Vornamen:	Kevin Häni	Yannick Denzer
Unterschriften:	....	....

Bibliography

[1] "CHVote System Specification", by Rolf Haenni, Reto E. Koenig, Philipp Locher and Eric Dubuis, October 31, 2017, https://eprint.iacr.org/2017/325

[2] "Anforderungskatalog für eidgenössische Volksabstimmungen mit der elektronischen Stimmabgabe", by Bundeskanzlei BK, June 07.2014.

[3] "ASO Factsheet E-Voting", by Auslandschweizer-Organisation, http://aso.ch/files/webcontent/direction/Factsheets/ASO_Factsheet_E-Voting.pdf

[4] "Geneva mounts e-voting charm offensive", by Swissinfo, https://www.swissinfo.ch/eng/politics/pitching-for-partners_geneva-mounts-e-voting-charm-offensive/42439582

[5] "Why Python is Slow: Looking Under the Hood", by Jake VanderPlas, see http://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

[6] "Python speed: performance tips", from the official Python wiki, see https://wiki.python.org/moin/PythonSpeed/PerformanceTips

7 Appendix

7.1 Sourcecode

The source code of this project can be found on github: `https://github.com/nextgenevoting`

7.2 Use Cases

Table 7.1: Use Case «Create new election events»

Use Case	**Create new election events**
Primary Actor	User
Description	The system allows to create new election events
Precondition	The system shows the available election events in a list
Main path (M)	1. User clicks on "'create election event"' 2. System demands a name for the election event 3. User is redirected to the election overview page

Table 7.2: Use Case «Set-up an election event»

Use Case	**Set-up an election event**
Primary Actor	Election Administrator
Description	The election administrator can set up an election. This involves generation of the cryptographic electorate data in the back-end
Precondition	A new election has been created
Postcondition	The election has the status "Printing"
Main path (M)	1. Election Administrator visits the "Election Administrator"-view of a new election. 2. The system demands the following information: • Number of parallel elections • Candidates per election • Number of possible selections per election event • Number of voters • Counting circles of the voters 3. User clicks on "Generate"

Table 7.3: Use Case «Printing of voting cards»

Use Case	**Printing of voting cards**
Primary Actor	Printing Authority
Description	The printing authority generates voting cards
Precondition	The election has the status "Printing"
Postcondition	The election has the status "Delivery"
Main path (M)	1. The election administrator visits the "Printing Authority"-view of an election. 2. The election administrator clicks on "Print Voting Cards" 3. A list of all voters is displayed 4. The election administrator can select a voter to see his voting card

Table 7.4: Use Case «Delivery of voting cards»

Use Case	**Delivery of voting cards**
Primary Actor	Printing Authority
Description	The printing authority can send the voting cards to the voters
Precondition	The election has the status "Delivery"
Postcondition	The election has the status "Election Phase"
Main path (M)	1. The election administrator visits the "Printing Authority"-view of an election. 2. The election administrator clicks on "Deliver Voting Cards" 3. The voting card shows up for every voter within the Voters view.

Table 7.5: Use Case «Casting of a vote»

Use Case	**Casting of a vote**
Primary Actor	Voter
Description	The voter can cast a vote by selecting his favored candidate(s) and his voting code
Precondition	• The election has the status "Election Phase" • A voter is selected in the voter view • The voter has the status "Vote Casting Phase"
Postcondition	The first election authority receives a ballot-check task
Main path (M)	1. The voter visits the voter view and selects his voter object 2. The system demands a selection of the candidates and the voter's voting code 3. The voter clicks on "Cast ballot"

Table 7.6: Use Case «Confirmation of a vote»

Use Case	**Confirmation of a vote**
Primary Actor	Voter
Description	The voter can confirm his vote by verifying the verification codes and entering his confirmation code
Precondition	• The election has the status "Election Phase" • A voter is selected in the "Voter"-view • The voter has the status "Confirmation Phase"
Postcondition	The first election authority receives a "Check-confirmation task"
Main path (M)	1. The voter visits the "Voter"-view and selects the corresponding voter from a list 2. The system displays the verification codes of the selected candidates 3. The voter must manually verify that the displayed codes match the verification codes of the selected candidates on his voting card 4. The system demands the confirmation code 5. The voter clicks on "Confirm vote"

Table 7.7: Use Case «Checking a ballot»

Use Case	**Checking a ballot**
Primary Actor	Election Authority
Description	The election authority can verify the validity of a ballot and respond to the voters query
Precondition	• The election has the status "Election Phase" • The currently selected election authority has a new "Check ballot task"
Postcondition	• The next election authority receives a "Check ballot task" • If this election authority was the last one, and the ballot was valid, the voter now has the status "Confirmation Phase"
Main path (M)	1. The user visits the "Election Authority"-view and selects one of the available election authorities that has new "Check ballot task" 2. The system displays the query, the ballot proof and the voting credential of the voter 3. The user clicks on "Check validity" 4. The system displays the result of the validity check 5. The user clicks on "Respond"

Table 7.8: Use Case «Checking a confirmation»

Use Case	**Checking a confirmation**
Primary Actor	Election Authority
Description	The election authority can verify the validity of a confirmation and respond to the voters query
Precondition	• The election has the status "Election Phase" • The currently selected election authority has a new "Check ballot task"
Postcondition	• The next election authority receives a "Check confirmation task" • If this election authority was the last one, and the confirmation was valid, the voter now has the status "Finalization Phase"
Main path (M)	1. The user visits the "Election Authority"-view and selects one of the available election authorities that has new "Check confirmation task" 2. The system displays information about the confirmation 3. The user click on "Check validity" 4. The system displays the result of the validity check 5. The user clicks on "Finalize"

Table 7.9: Use Case «Mixing»

Use Case	**Mixing**
Primary Actor	Election Authority
Description	Every election authority can perform the mixing on the extracted list of encryptions
Precondition	• The election has the status "Mixing" • The previous election authority has already performed the mixing
Postcondition	• The next election authority is able to mix
Main path (M)	1. The user visits the "Election Authority"-view and selects one of the available election authorities that has not mixed before 2. The system displays the list of encryptions of the previous election authority (or the first one in case the first election authority is selected) 3. The user clicks on "Mix" 4. The new, mixed list of encryptions is added to the known data of this election authority

Table 7.10: Use Case «Decryption»

Use Case	**Decryption**
Primary Actor	Election Authority
Description	Every election authority can perform the (partial) decryption
Precondition	• The election has the status "Decryption" • The previous election authority has already performed the decryption
Postcondition	• The next election authority is able to decrypt
Main path (M)	1. The user visits the "Election Authority"-view and selects one of the available election authorities that has not decrypted before 2. The system displays the list of encryptions 3. The user clicks on "Decrypt" 4. The list of partial decryptions is added to the known data of this election authority

Table 7.11: Use Case «Tallying»

Use Case	**Tallying**
Primary Actor	Election Administrator
Description	The election administrator can perform the tallying and view the final result
Precondition	The election has the status "Tallying"
Postcondition	The election has the status "Finished"
Main path (M)	1. The user visits the "Election Administrator"-view 2. The user clicks on "Tally" 3. The final result is added to the known data of the election administrator

7.3 Test-Cases

Table 7.12: Test Case «Pre-Election»

Description	This test covers all the pre-election steps, including the creation of a new election, setting it up from the election administration view and the printing- and delivery of the voting cards
Precondition	
Postcondition	- The election event is in the status "Election" - The voters have received a voting card
Steps	1. Start our application 2. Choose ¨Election Events" from the main menu 3. Click on "Create new election event" 4. Enter a name for the election event and choose a security level and click on "create" 5. The "Election Admin" tab should now have an interaction notification 6. Visit the "Election Admin" view 7. Enter at least 3 for the number of voters, at least 3 different candidates and 1 for the number of selection and click on "Setup Election Event" 8. The "Printing Authority" tab should now have an interaction notification 9. Visit the "Printing Authority" view 10. Click on "Print Voting Cards" 11. The voting cards for all voters should now be displayed 12. Click on "Deliver Voting Cards To Voters" 13. The voting cards should now disappear and the "Voters" tab should now have an interaction notification 14. Visit the Voters view, select a voter and check that the voting card is displayed correctly

Table 7.13: Test Case «Election»

Test-Case	2. Election
Description	This test covers the election phase in which a voter casts and confirms a ballot and the election authorities checks the ballots and confirmations and responds to the voter.
Precondition	• The election event has been set up appropriately as described in test-case 1. • The second and third election authorities should have automatic task processing enabled
Postcondition	• The election event is in the status "Election" • There are at least 3 confirmed ballots
Steps	1. Visit the "Voters" view and select voter 1 2. Check the box for candidate 1 in the vote casting form 3. Scratch open the voting code on your voting card and click on the revealed code. The code should be copied into the "voting code" input field 4. Click on "Cast vote" 5. Visit the "Election Authority" view and click on the first election authority 6. There should be a check-ballot task for voter 1. Click on "Check Validity" to check the ballot. The ballot should be valid. Click on "Respond". 7. The ballot should be added to the ballot list of this election authority. Verify that the ballot is also contained in the bulletin boards ballot list! 8. Return to the voters view. The voter should be prompted to verify that the returned verification codes match. If so, reveal your confirmation code and click it to copy it to the input field. Click on "Confirm Vote" 9. Visit the "Election Authority" view and click on the first election authority 10. There should be a check-confirmation task for voter 1. Click on "Check Validity" to check the confirmation. The confirmation should be valid. Click on **Finalize**. 11. The ballot list should now display the ballot as "‘confirmed"’. The confirmation should also show up in the confirmation list if you expand the ballot. 12. Return to the voters view. The voter should now see the returned finalization code that should match the code on the voting card 13. Repeat this test-case for voter 2 and voter 3, choosing the second candidate for voter 2 and the third candidate for voter 3

Table 7.14: Test Case «Post-Election»

Test-Case	3. Post-Election
Description	This test covers the post-election phases: mixing, decryption, tallying and verification
Precondition	• The election event has been set up appropriately as described in test-case 1. • 3 voters have casted their vote according to test-case 2.
Postcondition	• The election event is in the status "Finished" • The verification has succeeded and the correct election result is published on the bulletin board
Steps	1. Visit the **Election Administrator** view 2. Click on **End Election Phase and Start Mixing** 3. Visit the **Election Authority** view and choose the election authority 1 4. Click on **Mix** 5. You should see the list of encryptions being shuffled and re-encrypted. The result should be added to the data. The same should automatically have happened for the other election authorities. 6. Visit the **Election Administrator** view and click on **Start Decryption** 7. Visit the **Election Authority** view and choose the election authority 1 8. Click on **Decrypt** 9. The partially decrypted data should appear in the election authority data. The same should have happened for the other election authorities. 10. Visit the **Election Administrator** view and click on **Tally** 11. Under "Data", you should see the decrypted votes and the final result in a text- and chart representation. 12. Click on **Publish Result** and visit the **Bulletin Board** view 13. The bulletin board should now also contain the result of the election event. 14. Visit the **Verifier** view and click on **Verify Election** 15. All checks should show a green icon and a success message.