



Vow Polling

**Software for conducting comprehensive
and flexible elections at a small scale.**


3 July 2024

Ilesh Kedharanath Thiada

The Study L'école Internationale

2024-2025

XII - A2A



1. Nomenclature

The word 'vote' in English originates from the Latin word *voveō*, meaning 'I vow'. When one votes for their preferred candidate(s), they vow to support them. 'Vow' is also a reference to the candidates' promises to the voters.

2. Requirements

This section is split into 'needs' and 'wants'. The former are absolutely necessary for polling to be conducted and results to be obtained, while the latter are nice-to-haves that make polling easier to conduct and more reliable.

2.1. Needs

- Polling should be electronic, no paper ballots or the like. The results should be instantly available.
- Votes should be recorded correctly, and you should be able to retrieve the voting data.

2.2. Wants

- Attendance should be marked and synced, so that voting twice is not possible.
- The terminal where voting takes place should be isolated, so that voting is anonymous.
- If multiple computers are involved, they should communicate through the internet so that LAN client-server communication does not need to be set up.
- Votes should be stored on the cloud, so that they can be tracked in real-time.

3. Architecture

Voting can be split into multiple rooms across a campus. Each room must have a reliable internet connection and computers with the necessary peripherals. Each room will have one computer to mark attendance, and multiple 'booth' computers for recording the votes.

The 'controller' computer is used to mark attendance, and it will be operated by a member of staff. They need to enter the ID of the voter, or search for them using their name. Afterwards, the voter is assigned to a booth for them to vote.

Each booth remains closed by default. Once the voter's attendance is marked, a random booth will open up for them to vote at. The assigned booth's number will be notified to the controller, who needs to direct the voter to the appropriate booth. At the booth, the candidate options will appear, and the voter should select their preferred candidate and cast their vote. Afterwards, more screens will appear if there are more positions to vote for.

After the voter casts their vote for all available positions, a thank-you screen will appear, and the booth will close itself after 5 seconds on this screen.

4. The Team

My friend Aadarsh Madan Kollan and I collaborated to create this project. We were guided by our Computer Science teacher, Mrs. Priyadarshini G, throughout the course of this project.

5. Implementation

The front-end is implemented using [Flutter](#), Google's framework for creating cross-platform apps. The computers available used Linux, so we used Flutter's ability to [create desktop apps](#). Although desktop OSs are not supported as well as Android and iOS, it is mostly feature complete using third-party libraries.

We chose Google's [Cloud Firestore](#) database for data storage and communication. It is part of the [Firebase](#) collection of services, and is extremely fast with the ability to stream data updates in real-time.

6. Development Process

6.1. Background

We were told about this opportunity when our school reopened after the summer break on the 12th of June. Our exams began immediately on the next Monday (June 17), so we prioritised them first. We began work on the next Thursday (June 20), during the exams, since the elections were planned to take place on the following Saturday (June 29). **This was a major hurdle as we had to finish everything and get it ready within 1 week.**

6.2. UI Design

We designed the UI in advance, during and before our exams, using Figma. This made it much easier to rearrange and try out new layouts and colour schemes, when compared to doing so in code. It was easier for the programmer as well, since I could pre-plan what widgets and layout tools to use to replicate the UI, and refer to predetermined colours.

6.3. The Linux Problem

The computers available to us used Linux. I knew that Flutter for desktop was quite mature at this point, and so I decided to create a native desktop app instead of a web-app.

Unfortunately, there was a major problem we had with Linux. The official Google [cloud_firestore](#) package only supported Android, iOS, macOS, Windows, and the Web. There was no support for Linux.

This was because the library delegates its functionality to the [native Firebase SDKs](#) of each platform. While Linux does have a native C++ SDK, it was not supported by the [cloud_firestore](#) package. I searched for alternatives, and found the [firedart](#) package. It is a **Dart native** package that does not delegate to the native SDKs, and thus it should work on any platform Dart supports, including Linux. However, this was an immature library that only supported a subset of the official SDK's Firestore API.

6.4. Voter Data

We used the school's student and employee database to download a list of the voters' details. For students, this included their student ID (TS number), name, class, and section. For employees, only the employee ID and name were available. We downloaded this data to a CSV file, processed it using Python, and uploaded it to a collection on our database. We had to manually filter out non-teaching staff.

6.5. Searching by Name

We intended to allow the controller to search for the voter by their name if they do not know their ID, which is often the case with staff.

Firestore is a NoSQL database, which has the advantages of being extremely fast, lightweight, and flexible. However, it loses some functionality compared to SQL. Namely, it is not possible to do substring matches as you would in SQL, i.e. `LIKE "%textfield%"` or `INSTR(name, text_field)`. It is also not possible to group data, as you would in SQL using a `GROUP BY` clause.

This posed a problem since it was not possible to query search directly to Firestore. Instead, we settled on fetching all voter details locally, keeping it up-to-date by streaming the data, and searching and sorting results on-device. This had the advantage of being ultra-fast of course, but it performed about 1,000 document reads every time the app started up. We realised this when our database went down, because we had exceeded the free-tier usage quota by repeatedly restarting the app and re-fetching data during development.

We resolved to using hard-coded stub data during development, and only streaming from the database in testing and production.

6.6. Controller-Booth Communication

As mentioned before, a nice-to-have is for the controller and booths to communicate over the internet, so that there is no need to set up a server and configure LAN switching. We also used Firestore for this communication, as it is possible to listen to changes by streaming documents.

The controller will listen to the booth statuses in a room, and assign a booth when one is available, i.e. closed to voters. Afterwards, it will display the assigned booth's number to the controller for them to guide the voter. The booth will be listening to the same document, and open itself when its flag is set. It will set its own flag to false once voting is complete, thereby closing itself.

Apart from opening and closing, there is no other communication between the controller and booth, maintaining the anonymity of the voting process.

6.7. Recording Votes

Votes are also recorded in Firestore to allow for remote monitoring. However, with multiple booths recording votes simultaneously, it is essential to implement vote recording in a way that avoids race conditions, in which case votes could be easily lost.

This was quite simple to fix. We stored all votes in one collection, and each booth had an individual document for itself. Each booth was only responsible for its own document, and since the booths themselves will not concurrently cast votes, the race condition is avoided.

In the app used for tracking the votes, we fetched all the booths' documents, consolidated the votes into a final count, and displayed it in pie charts.

7. Retrospect

Unfortunately, due to our tight schedule, we did not have the opportunity to test the app at scale outside our own limited functionality checks. Due to this, we encountered multiple minor issues during pre-production demonstrations, and go-live.

7.1. Drops in Internet Connectivity

When streaming data using `firedart`, any disruption to the internet connection unfortunately results in the app becoming unusable due to the lack of error handling and recovery. The app needs to be restarted even if internet connectivity is restored.

This issue was not discovered by us, since we had never run the app for a long period of time before. In production, we had one major outage lasting around 10 minutes. The internet connection seemed to have briefly cut out, after which all processes ground to a halt. Across the campus, all 4 controllers and 6 booths loaded perpetually even after the connection recovered. We had to manually restart the app on all computers to bring back functionality.

Even throughout the rest of the day, minor issues with connectivity resulted in the app failing to load in some isolated cases. These were solvable by restarting the app, however it resulted in some friction in the process for those managing the election.

7.2. Total Loss of Connectivity

While we were lucky to have a fast and mostly reliable internet connection at the polling venues, we did not have a contingency plan in the case where we went completely offline. As aforementioned, communication between controllers and booths relied on the internet, and vote casting was also fully internet-based. If a loss of internet connectivity occurred, the entire election would collapse.

In hindsight, both of these issues cropped up due to our use of the third-party `firedart` package. The official `cloud_firestore` package has mitigations for both of these issues. It will automatically recover the stream when connectivity is lost. It also has the ability to locally cache document writes when offline, and upload them when it's back online.

7.3. Data Integrity

While we had obtained student information from the school's database, there were a handful of students who had not been registered yet, and had to be manually added to the `voters` collection. When manually adding documents while the system is live, it is of the *utmost* importance to make sure that the data entered is correct. Otherwise, it is possible for the controller to temporarily go down when it receives invalid data.

It is crucial for controllers to check voter details when marking their attendance. In a few cases, people with similar names had their attendance marked incorrectly. Controllers should check the voter's class and section carefully before assigning them to a booth.

There was a minor bug where if the 'Assign Booth' button was accidentally clicked twice in quick succession, it could trigger 2 document writes. One of those would write invalid data to the database, causing other controllers to receive invalid data too, and temporarily crashing the system.

7.4. UX Guidance

During voting, many voters did not know how to use the voting UI. We had planned on posting up a sign with a guide on how to use the voting interface. However, we were not able to implement this.