Bachelor Thesis

Aliaksei Khomchanka

Design and implementation of an electronic voting system for student elections

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Aliaksei Khomchanka

Design and implementation of an electronic voting system for student elections

Bachelor Thesis based on the examination and study regulations for the Bachelor of Engineering degree programme Information Engineering at the Department of Information and Electrical Engineering of the Faculty of Engineering and Computer Science of the University of Applied Sciences Hamburg

Supervising examiner : Prof. Dr. Heike Neumann
Second examiner : Prof. Dr.-Ing. Luty Leutelt

Day of delivery November 4, 2019

Aliaksei Khomchanka

Title of the Bachelor Thesis

Design and implementation of an electronic voting system for student elections

Keywords

Electronic voting, cloud technology, public cloud, Amazon Web Services

Abstract

As e-voting becomes more widespread in recent decades, there are multiple attempts to implement e-voting in elections of various scales. One of such use cases is university elections for student organizations. A good class of platforms for implementing such voting systems is public clouds, such as Amazon Web Servers. This report shows the motivation behind the implementation of such a system, its technical requirements and methods of making such a system serverless.

Aliaksei Khomchanka

Thema der Bachelorarbeit

Design and implementation of an electronic voting system for student elections

Stichworte

Elektronische Abstimmung, Cloud-Technologie, öffentliche Cloud, Amazon Web Services

Kurzzusammenfassung

Da E-Voting in den letzten Jahrzehnten immer weiter verbreitet ist, gibt es mehrere Versuche, E-Voting bei Wahlen in verschiedenen Größenordnungen umzusetzen. Ein solcher Anwendungsfall sind Hochschulwahlen für Studentenorganisationen. Eine gute Plattform für die Implementierung solcher Abstimmungssysteme sind öffentliche Clouds wie Amazon Web Server. Dieser Bericht zeigt die Motivation hinter der Implementierung eines solchen Systems, seine technischen Anforderungen und Methoden, um ein solches System serverlos zu machen.

Table of Contents

1. Introduction and motivation

Voting has historically been the most convenient way to make a large-scale decision that could take into account opinions and preferences of large groups of people. Voting can be applied in a multitude of situations, from simple decision-making in a study group to electing government officials. Despite advances in information technology in recent decades, voting worldwide, when it comes to various types of elections, is still typically done in the same manner as before: via voting booths. Such a method introduces lots of potential vulnerabilities due to the human factor. Unauthorized votes may be added or legitimate votes may be removed by people on site or people responsible for counting all the votes after the voting. The higher the scale of the election - the higher the need for proper observability. While it is, to some extent, possible to assign multiple independent observers to as many voting centers as possible (which is normally done in presidential elections worldwide) so that they could make sure the data integrity is not compromised, the amount of management and funding required grows together with the scale of the voting process.

The motivation behind this thesis is to design a cloud-based voting platform which would allow to significantly reduce the number of human actors (as in voting booth supervisors and independent observers) involved in the process and as a result - the risk of voting fraud or errors due to the human factor. Instead of having a system where observers need to constantly monitor every step of the process for possible interventions, it is, perhaps, easier to provide an online voting platform with an open design where the entire software environment can be isolated (isolation itself can also be confirmed by observers) and not require further monitoring. Modern cloud platforms provide all necessary tools to build such a system.

2. Modern e-voting systems and possibilities

As soon as Internet started getting more and more widespread, the matter of moving voting for elections online started arising. Rapid development of cloud technologies only increased interest to this matter. In recent decades, not only have such online voting platforms been suggested, but were even implemented on several occasions for official government elections on various scales, such as municipal elections in Estonia in 2005 [1] [2] (pioneers in state-wide e-voting at the time) or city-wide elections in Geneva, Switzerland in 2004, performed for testing e-voting mechanics on a smaller scale before integration country-wide [3] (other cities followed later).

Nowadays, such systems become more and more widespread, new technologies become involved, such as cellphones that make the voting process even more convenient and blockchains that aim to make the voting process more transparent and fraud-resistant.

Another interesting piece of technology that could prove useful for such systems is public clouds, implemented by modern IT giants, such as:

- **AWS** by Amazon
- **Azure** by Microsoft
- **Google** Compute Cloud by Google
- **IBM Cloud** by IBM
- and a multitude of other offers from other companies

An important feature of such public cloud providers is that they allow users and companies to delegate all the tedious infrastructure-related tasks (or at least most of them) to the provider and concentrate on what they actually want to implement. A testament to that is the concept of microservices that allows developers to implement cloud-based applications that don't even need explicit dedicated servers, while still providing all necessary functionality, flexibility and security.

Usage of microservices and serverless concepts in may cases also allows for more simple and easy-to-manage applications.

Considering usefulness and convenience of such systems, it seems to be a logical step to investigate their possible use in the development of voting platforms for elections on different

[1] **E-Voting in Estonia 2005. The first practice of country-wide binding Internet voting in the world**, Madise, Martens, 2nd International Workshop, Bregenz, Austria, 2006
[2] **Security Analysis of the Estonian Internet Voting System**, Springall, Finkenauer, Durumeric, Kitcat, Hursti, MacAlpine, Halderman, University of Michigan, USA
[3] **Three Case Studies from Switzerland: E-Voting**, Gerlach, Gasser, Berkman Center Research Publication No. 2009-03.1

scales. Particularly, this thesis attempts to implement a platform for voting in elections for various student organizations. This use case is chosen due to several reasons.

It is small-scale

Student elections are typically faculty or university-wide, giving us a reasonable amount of voters to deal with. At the same time, all formal elements of an actual election are present, such as a set of candidates, election programs, organized voting, the need for proper management of personal data, ensuring confidentiality of votes and preventing vote fraud. Thus it is possible to design all systems for real-life conditions and if they prove themselves usable and useful - one can always scale up (which is also a convenient consequence of using public clouds and microservices).

It is not as strict security-wise

While it is certainly necessary to maintain data security during student election, the price of a mistake can be considered lower as the parties involved are far less influential than, for example, at presidential elections. There is also the obvious question of delegating user data to a public cloud provider for storage. Providers themselves claim that all data is stored in encrypted state in remote heavily guarded locations[4] (sometimes, according to Amazon, actual underground bunkers) and no one can have unauthorized access to it, one can't exactly check it due to the exact fact of those locations being unavailable for public audit.

That being said, looking at the problem from a purely financial point of view, it can be assumed that it would be disadvantageous for any public cloud provider to get involved with user data in any unlawful way due to the amount of money at stake. They simply wouldn't risk losing their customer base (which they are already fighting for on a competitive market of public cloud providers), especially to interfere in some unimportant student elections. Thus, one can assume security of data as long as:

- It is contained within the public cloud provider datacenter
- Our access credentials are properly kept private

It is important to note that this project doesn't concern itself with developing a front-end application with a GUI for casting votes. The primary concern here is the backend, the actual logic behind the voting process itself and ensuring data integrity.

[4] https://aws.amazon.com/compliance/data-center/data-centers/

3. System requirements

3.1 A voting process

A voting process, in general, would have the following actors on the technical side (meaning not taking into account “social” elements like reporting on elections in media, election campaigns etc.):

- Voters
- Candidates
- System administrators for the voting platform
- Independent observers

Voters have the intention of casting their vote for their intended candidate and are interested in their vote not being altered. In certain cases, they may also be interested in abusing the system in favor of certain candidates, therefore it is necessary to not only give them the required functionality to cast a vote fairly, but also to prevent them from possible cheating.

Candidates are interested in having all votes intended for them to be, indeed, counted in their favor. Some candidates may also be interested in cheating and exploiting the system to get more votes than were actually intended for them, so precautions must be taken.

System administrators are tasked with maintaining the voting platform and ensuring that all its elements are operational when needed. They are also tasked with ensuring that all internal logic is set according to predefined rules of the fair voting process. Administrators, as other actors, may also be interested in abusing the system and need to be held accountable for their actions.

Independent observers are interested in ensuring that all system components work as intended and no fraud is taking place at any step of the process. Once again, seemingly “independent” observers may be biased and potentially malicious, so there need to be countermeasures for possible fraud attempts.

As can be seen, the most challenging part is not ensuring that all “fair” actions can be performed (that part is quite straightforward and easy to implement) but ensuring that potential malicious actors have no power to perform unauthorized actions to alter voting results. Each type of actor needs a set of countermeasures:

Voters

Ensuring that voters can only cast one vote for one candidate is ensured by the use of single-use credentials for authenticating votes. Said credentials are generated automatically inside the isolated system only once when the voter is registered in the database. To prevent “fake” voters in the database, independent observers are involved.

Candidates

Candidates can't directly influence the voting process, relying on accomplices among voters, administrators and observers, so by taking care of these actor categories one would take care of candidates as well.

Administrators

Administrators are by their nature the people with most power as they have all access to the internal logic, databases and credentials. Taking that into account, all their actions must be monitored by independent observers. When monitoring is not possible (such as during actual voting to preserve confidentiality of voter-vote pairs), the entire system must be locked to administrators as well. If administrators attempt to access system internals at unauthorized time - observers must be automatically and immediately informed.

Observers

This is the "weakest link" in the entire system as here the human factor really comes into play. However, resolving the potential issue with biased observers is more of a policy matter rather than a technical one, so a general advice would be to have as many observers from as many (preferably differently aligned) organizations to minimize possible bias.

3.2 Requirements

Intended system must have the following basic elements:

- An isolated remote environment for the infrastructure
- A control panel that can give us an overview of the entire infrastructure
- A way to interact with external users (for example, to cast votes via a web API)
- A way to securely lock the system from unauthorized access or inform us about it

All these elements in combination must be able to perform the following actions:

1. Add voters, candidates and other necessary data and offer long-term storage for it
2. Communicate with voters via emails and HTTPS requests
3. Allow users to remotely cast votes via HTTPS requests
4. Automatically count votes and compose an election summary
5. Be able to receive, register and count votes while being locked away from outside access, meaning being reliable enough to operate without observation or maintenance during the election itself

6. Track access to resources and be able to differentiate between its own legitimate usage of said resources and unauthorized access from outside

These are the crucial elements that absolutely MUST be present. Some in-place improvements for the sake of convenience would be good to have, but not crucial.

With that in mind, requirements can be stated as the following:

Requirement	**Functional of non-functional**
A web API endpoint for receiving remote commands	Functional
API commands for: - Adding new voters/candidates - Casting votes for candidates - Fetching information about available candidates - Locking the system and starting voting	Functional
Autonomous internal program(s) for handling incoming votes	Functional
Autonomous internal program(s) for handling vote counting and informing voters about results	Functional
Autonomous internal program(s) for handling possible security breaches	Functional
Methods of authentication for incoming votes and administrator commands	Functional
A database for storing voter data	Functional
A database for storing candidate data	Functional
A database for storing votes	Functional
A database for storing internal system parameters, such as: - Lock state (locked/unlocked) - Voting state (ongoing or not) - Authentication credentials	Functional
All components are serverless	Non-functional
A single control interface for the entire system	Non-functional

Aside from functional requirements, two features would be very convenient. One of them is making the whole system serverless, meaning that there is no need to manage internal routing or firewalls. Aside from removing potential mismanagement (and as a result introducing vulnerabilities into the system), it also allows for less components and as a result - easier monitoring by independent observers.

A single control interface is also a useful feature when it comes to easier monitoring, ensuring higher confidence in system’s integrity.

4. Technology used

The system is implemented using various services provided by AWS. AWS is a public cloud platform provided by Amazon. It's been chosen for this project due to it being the most well-documented and generally more well-known to most developers (with AWS holding a much larger share of the market compared to its direct competitors, Azure by microsoft and GCC by Google), which would make post-deployment support and modification easier. Using AWS also allows us to delegate various management responsibilities to Amazon's backend (more on that below). But most importantly, AWS allows us to use different technologies while managing all of them from the same control panel and interact with them using the same API.

Out of various services provided by AWS, several are of particular interest for this project. The detailed explanation of the reasoning behind choosing certain services over others is given in Section 5.

4.1 AWS Lambda

Perhaps one of the most well-known and widely used AWS services, Lambda[5] provides means to run code in a serverless manner, meaning that you just need to upload your code, set various parameters (maximum run time before forcibly terminating, allocated memory etc.), environment variables, access control - and then trigger the resulting program on demand using one of the long list of possible trigger types provided by AWS. There is no need to provision any servers or take care of server replication and scaling, all this is done "behind the scenes" by AWS backend. Lambda provides several managed runtimes for user code, such as:

- Node.js
- Python
- Ruby
- Java
- Go
- .NET

[5] https://aws.amazon.com/lambda/

Each runtime category provides different possible versions (such as Python 2 and Python 3 and their sub-version, for example). There is also a possibility to define own runtime to run functions in any language of choice. For this particular project Python 3.6 was used as a runtime.

In the context of the intended system, Lambda can be used for handling all internal logic, such as vote processing and counting or handling security breaches.

4.2 EC2

Amazon
EC2

EC2[6] or Elastic Compute Cloud is a service that provides on-demand servers of various types. Servers can be provisioned either manually via the AWS Management Console or programmatically via AWS SDK. Possible instance types include:

- General Purpose
- Compute Optimized (used for media transcoding, machine learning inference etc.)
- Memory Optimized (for large in-memory data sets)
- Accelerated Computing (instances with several powerful GPUs)
- Storage Optimized (optimized for large amounts of I/O operations)

In addition to providing servers themselves, EC2 also offers tools for scaling, cross-region replication and automatic management of server groups. After being configured, it can automatically maintain the desired amount and type of servers and adjust it accordingly depending on provided metrics, for example, launching more servers in an auto-scaling group when existing servers are saturated with requests.

EC2 also provides various cost-optimization possibilities, such as reserved instances (for 1 to 3 years at a significant discount) or spot instances (instances bought in certain regions when the demand for them in that region is low, resulting in discounts up to 90% of their original price).

[6] https://aws.amazon.com/ec2/

In the context of the intended system, EC2 can be used, similarly to Lambda, for handling all internal logic via processes running on provisioned servers.

4.3 IAM

AWS IAM

This is, perhaps, the most fundamental service in AWS that manages all credentials[7]. There are two fundamental types of credentials in AWS:

- Management console login credentials
- API access keys

These two types cover different means of access to AWS resources. Ordinary login credentials provide access to the AWS Management Console (the web GUI), but you can’t sign API requests with those. API keys are the opposite, they are used to authenticate API requests, but can’t be used to log into the Management Console.

Each set of credentials has its set of permissions that specify access to types of AWS resources and are set to deny access to everything by default. You can specify access in detail, up to allowing access to specified operations on some instances of a service and forbidding them (or allowing different ones) on a different instance.

Credentials can be permanent or temporary. Regarding temporary credentials, the ones of particular interest to us are so-called IAM Roles. IAM Roles are credential rules that are assumed by services to perform some predefined actions. When you set up a service - you assign a role that this service will have access to. A role has a permissions set like any normal AWS credentials would. An example of using roles would be the following:

- An AWS Lambda function is invoked by an external API call.
- Lambda function assumes a role assigned to it.
- The role generates a pair of API keys for that particular run of that particular function so that it can interact with other services via AWS SDK.

[7] https://aws.amazon.com/iam/

- Function finishes execution.
- Temporary credentials for that function are erased.

Thus, with roles, no unchecked credentials remain, improving security.

In the context of the intended system, IAM is used for storing credentials for administrator access to the Management Console and providing credentials for all other AWS services. Using it is not actually a choice but a necessity, as it is integrated with ALL services in AWS.

4.4 Amazon RDS

Amazon
RDS

Amazon RDS[8] provides means to quickly provision and manage relational database servers, such as PostgreSQL, MySQL and others. Load balancing, data replication, updates, security and other tedious tasks are managed by the AWS backend, leaving you free to divert your attention to other important matters.

Database servers provided by this service are launched on EC2 instances and can be managed as such.

In the context of the intended system, RDS can be used for storing voter data, candidate data, vote records and global system parameters.

[8] https://aws.amazon.com/rds/

4.5 DynamoDB

Amazon DynamoDB

DynamoDB[9] is an AWS-specific non-SQL serverless database platform, allowing you to define desired tables and interact with them via AWS SDK without worrying about managing actual servers. As with other AWS services, load balancing, data replication and other such issues are automatically managed by AWS backend. In many cases, using DynamoDB is easier than a more classic RDS database server. Being a platform developed specifically for AWS as a built-in service, it has more intuitive integration with AWS SDK compared to Amazon RDS, which requires external SDKs to interact with specific types of database platforms.

In the context of the intended system, DynamoDB, like RDS, can be used for storing voter data, candidate data, vote records and global system parameters.

4.6 SNS

Amazon
SNS

Amazon Simple Notification Service[10] (or SNS for short) is a service used to automatically send notifications of various types to subscribers. You can define different topics and assign a list of subscribers to each of them. After that, you trigger the topic via AWS SDK with a desired payload, making it send your message to all recipients subscribed for the topic. Possible notification methods include:

[9] https://aws.amazon.com/dynamodb/
[10] https://aws.amazon.com/sns/

- Email messages
- HTTP/HTTPS requests
- Amazon SQS messages
- AWS Lambda (this one can trigger Lambda functions with SNS messages)

SNS, in the context of the intended system, can be used to inform observers of security breaches. This may also be done with EC2, but that requires additional setup, while SNS is designed for this sole purpose.

4.7 SES

Amazon SES

Amazon Simple Email Service[11] allows you to send personalized emails to specified recipients. This is useful when you can't use SNS, for example, when you need to notify all users of your system, but each of them must receive a different personal token or something of that nature.
A limitation of this service is that it is only available in some regions, so depending on your primary region, an additional parameter in your API calls may be needed.

SES can be used for informing voters about election results. While this is also possible via email servers provisioned with EC2, once again, similarly to SNS, SES is built for a specific purpose and is easier to use in this context.

[11] https://aws.amazon.com/ses/

4.8 CloudWatch

CloudWatch[12] is a set of tools for monitoring your AWS resources within your entire account. It covers log streams for your services, various metrics (such as resource utilization for your servers, usage of storage services etc.), scheduled events, metric alarms and offers many other capabilities.

CloudWatch Logs.

CloudWatch Logs offer a structured storage for all AWS service logs. As an example, your AWS Lambda functions will automatically post some of their execution logs to corresponding streams, which will assist you in monitoring and troubleshooting. You can also define your own log groups and streams manually or via the AWS SDK. it is also possible to post events from external sources (as in outside your AWS account) to CloudWatch Log streams, provided that proper API keys are present.

CloudWatch Metrics.

CloudWatch Metrics monitor various user-defined metrics within your AWS account, providing means of visualizing them in the Management Console of exporting them via the SDK. These metrics can also supply their data to CloudWatch Alarms.

CloudWatch Alarms.

CloudWatch Alarms allow you to define a set of rules that can trigger an internal alarm based on the behavior of certain CloudWatch metrics. For example, you can set an alarm that will react to the increased number of requests to a certain server and trigger a Lambda function when a certain threshold is reached.

[12] https://aws.amazon.com/cloudwatch/

CloudWatch Events.
CloudWatch Events can trigger other AWS services on a user-defined schedule provided in CRON format. A sample application of CloudWatch Events can be automated scheduled maintenance of your infrastructure with Events launching some Lambda functions.

CloudWatch is, similarly to IAM, not exactly a choice but rather a necessity in this case as some crucial other services, such as CloutTrail, route their logs through CloudWatch. Additionally, CloudWatch events are the most reliable way to schedule events in AWS. EC2 servers may crash and disrupt the timing, but CloudWatch Events are always available and consistent.

4.9 CloudTrail

Amazon
CloudTrail

CloudTrail[13] is one of the internal monitoring tools that lets you see all account activities at a glance. Every API request that happens within your AWS account is listed as an event in CloudTrail, from logging in to editing resources to logging out. CloudTrail is on of the most essential services for security audits and post-incident investigations.

CloudTrail is absolutely crucial for any AWS-based system that assumes any degree of security as it is the service that monitors all internal activities and is the first to “know” about them.

[13] https://aws.amazon.com/cloudtrail/

4.10 Amazon Managed Blockchain

Amazon Managed
Blockchain

This service[14] provides a managed platform for running distributed ledgers based on popular blockchain runtimes such as Hyperledger Fabric and Ethereum. AWS backend takes care of certificate storage, scaling the network and other tasks that in a classic setup would take a lot of time to set up and would provide a lot of opportunities to mismanage some of its elements.

This service can be used to store all data related to voting (voters, candidates and vote records) on a ledger, ensuring integrity, consistency and transparency to voters.

4.11 API Gateway

API Gateway

API Gateway[15] is a service that allows users to define their own RESTful APIs with a detailed resource and method tree-like structure, schema checks, custom endpoint names and pipelining incoming requests to other AWS services, such as, for example, AWS Lambda.

In the context of the intended system, API Gateway allows to set up an easily managed API endpoint with all necessary tools for integration with other services. Setting up a similar data pipeline with other services (such as, for example, receiving API commands with EC2 servers and launching Lambda functions) would take much more work and provide opportunities to introduce vulnerabilities.

[14] https://aws.amazon.com/managed-blockchain/
[15] https://aws.amazon.com/api-gateway/

5. System design

This section explains the general design decisions, particularly the choice of AWS services, as for the purpose of this system some may provide more advantages than others.

In the course of designing the system, one fundamental design principle has been used to make sure the system is optimized as much as possible:

Automate as much as possible, delegate backend management to the cloud provider wherever possible.

One convenient way to ease end user's responsibilities in managing the system is making as many components as possible serverless to remove the need to manage (and possibly MISmanage) components such as routing tables, internet gateways, firewalls, VPNs etc., all of which one would need to set up when using servers. Another detail is taking advantage of existing management tools inside AWS as much as possible, such as the AWS Management Console. Implementing an inferior (and less polished) management interface wouldn't be rational when there is a superior solution already available.

Elements of the end system can be put into four categories:

- Code runners that govern the actual logic of the voting process and interact with other resources
- Database storage for all voting-related data
- Triggers to launch code runners
- Notification services that send emails to voters with generated tokens, voting results, registration confirmation etc.

5.1 Code runners

For code runners two most obvious choices would be either dedicated servers continuously running applications that control the voting process (implemented using the **EC2** service) or **Lambda** functions that are completely serverless and are only running actual code when started by some external triggers. After evaluation, considering functional requirements of the intended system, it's been concluded that there is no functionality offered by **EC2** that couldn't be implemented using **Lambda** functions. Lambda functions also remove the need to set up networking and can be managed from the default AWS graphic interface, so it adheres to our design approach perfectly. It also allows to integrate other AWS services as triggers for functions, again, via the standard GUI.

5.2 Database storage

At first, the possibility of using blockchains for this system has been considered. Ideas of implementing blockchain-based voting applications are becoming more popular with each passing year[16], so their usefulness for this project has been investigated. The purpose of using blockchains in voting applications is to ensure that data about voting process, stored in the distributed ledger, is consistent across all voters, meaning that no unauthorized changes are possible. However, that would mean that one would need to run a node for each voter. Considering that, there were two possibilities:

1. Make voters run nodes on their computers/phones

   This idea doesn't seem realistic, considering that most people either wouldn't be keeping their computers on all the time or wouldn't be willing to waste their phones' battery charge on computations used in consensus algorithms to add new votes to the ledger. With that in mind, a distributed ledger with a consensus algorithm wouldn't really make much sense in a network where an absolute majority of nodes is offline.

2. Run nodes in **Amazon Managed Blockchain** service

   AWS provides a service for running blockchain networks based on popular platforms such as **Hyperledger Fabric** or **Ethereum**. This, however, raises the question: if users can't access nodes inside the AWS account (and providing them such access would compromise the entire security of the system) - how would they confirm that the entire process is, indeed, consistent? In the end, to an outside observer, it makes no difference to use a blockchain or not if everything is contained within a closed AWS infrastructure.

Taking all this into account, as well as the relatively low scale and importance of the intended voting process (compared to elections of government officials, for which blockahais are often suggested) - it is, perhaps, a wiser decision to concentrate more on making our AWS infrastructure more secure to breaches.

Thus, the **Amazon Managed Blockchain** service was ultimately not used.

The second matter regarding database storage was whether to use the **Amazon RDS** service which would allow me to use familiar relational SQL-compatible databases or use an AWS-managed solution, such as **DynamoDB**. While the general design principle is to make as much of the system serverless as possible, it still needs to be usable by an average person who may not be familiar with AWS-specific database types (and **DynamoDB** is AWS-specific with a

[16] **A Conceptual Secure Blockchain-based Electronic Voting System**, Ahmed Ben Ayed, International Journal of Network Security & Its Applications, May 2017

proprietary API), so there may have been a need for a trade-off between managing the infrastructure and using a familiar/unfamiliar API.

Initially, the idea was to use **Amazon RDS** to provide a **PostgreSQL** server to store data for voters, candidates, votes and system parameters (some of the remaining code is available in the **VoteManager** function code for comparison). While the database server was working normally and it was absolutely possible to use it, the amount of work to set it up and the need to use third-party libraries to connect to the database from **Lambda** functions (such as **psycopg2** library for **Python**) meant that there were lots of opportunities to mismanage something in way that would introduce vulnerabilities to my server, as there was a need to set all routing, firewalls etc. manually. Another matter is making it easier to manage for the end user. Taking that into consideration, it's been decided to scrap everything there was at the time (which was most of the code) and redesign it for use with **DynamoDB** instead. **DynamoDB** not only provided a serverless platform for the database, but also allowed to interact with stored data using default **AWS SDK** libraries. While **DynamoDB** required some additional study (considering no prior experience with it) - it ultimately proved itself to be much easier to use when developing and managing system components. Regarding the initial concern of making users work with an unfamiliar AWS-specific API, the solution to that was just making a separate library in **Lambda** functions that provided simple wrapper methods for all the API calls. Thus, in 99% of cases, the end user would not ever need to interact with the API at all, except for cases when someone wants to modify the system, but that assumes some level of knowledge of the API to begin with, so that isn't really a problem as well.

5.3 System isolation and breach detection considerations

A strong way to lock our AWS infrastructure to outside access during the election process would be to temporarily delete login credentials and automatically re-create them at the election deadline. However, that has not been actually attempted (and not event confirmed whether that was technically possible) as the risk of mismanaging something and getting locked out of the account permanently is not desirable (especially considering that some paid services may still be running and wasting money).

In this case, one always has a set of admin credentials that can potentially be leaked (at the very least due to the human factor), so it's not possible to lock the account with a 100% reliability. However, while one can't lock the account, one can detect unauthorized login attempts and inform administrators and observers about them. Thus some solutions were considered.

The trickiest part was implementing a way to trigger the **BreachHandler** function quickly enough to make it impossible to turn it off before it sends the notification. It's been decided decided to try online search first to see how others have solved similar problems before. Surprisingly, there were no instances of implementing quick notifications for logging into the **Management**

Console. Online search, however, revealed two relatively similar problems dealing with login events in general. They provided two approaches to the problem at hand.

5.3.1 Set CloudTrail to put logs into S3 buckets to trigger a Lambda function

S3, or **Simple Storage Service** is the most used AWS service for file storage. **CloudTrail**, which tracks all login events, can be configured to put its logs to so-called “buckets” in **S3**. When a new item is added to a bucket - it can trigger a Lambda function. Therefore, a design have been implemented in which **CloudTrail** was tracking all account events and putting them to **S3**, **S3** was triggering the **BreachHandler**, which was filtering events to search for the ones with the “ConsoleLogin” tag and acted accordingly when such an event was found.
While such a design was ultimately sending the alert notification, it could easily take up to a minute or two to do so due to the large amount of log files being processed by **S3** and **Lambda**. Clearly, such a design was not acceptable as long waiting times completely defeated the purpose.

5.3.2 Using CloudWatch Alarms as triggers

Another suggested idea was to use **CloudTrail** to, once again, put logs to **CloudWatch**, then filter them and send them to **CloudWatch Metrics**, which would track the count of login event records repeatedly and trigger the **BreachHandler** if the metric went above 0.
This design worked at first glance, however, further testing revealed that the waiting time was very inconsistent, from near instantaneous to several minutes. Further investigation revealed that metrics that were triggering the alarm were not exactly updating in real time. While CloudTrail was immediately putting records for log events into CloudWatch Logs, metrics were not immediately seeing them. Their behavior was nor based on interrupts, but on polling instead. Thus, it was completely up to luck whether the alert notification would get sent immediately or after a minute or two. Obviously, that was also not acceptable.

5.3.3 The solution

The problem was solved in a rather simple way by routing **CloudWatch Logs** directly to **Lambda** with the filter for login events applied. Since **CloudTrail** puts login events into logs immediately, they can also immediately trigger the **Lambda** function. Testing proved this method to be consistent in terms of waiting time.

6. Intended system structure

6.1 Services involved

After careful consideration, a set of AWS services has been chosen. Said services and their relationship is depicted in the following diagram:

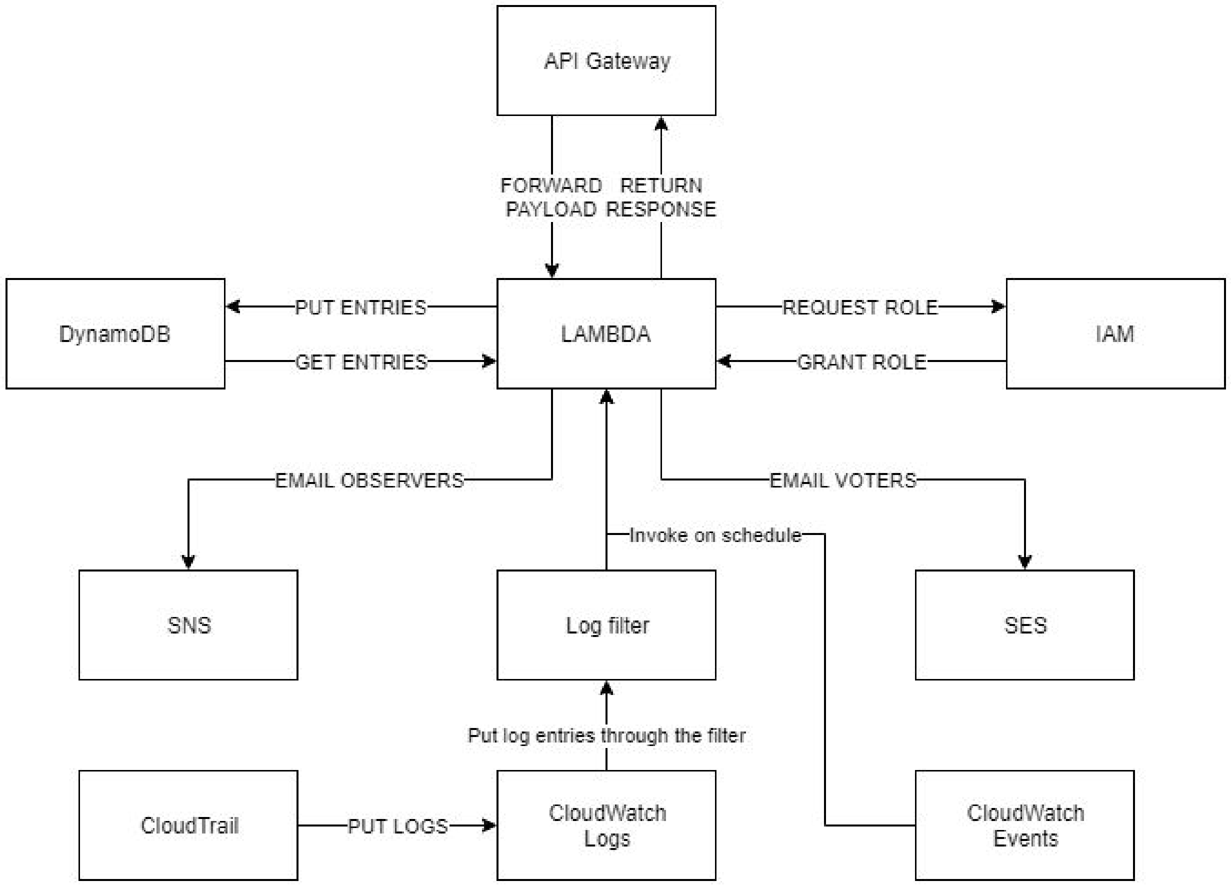


Fig. 6.1 Used services and their relationship

The first service to deal with is **API Gateway**. When HTTPS requests reach the specified API endpoint (which developers define themselves) - the API Gateway forwards the corresponding JSON payload and triggers a **Lambda** function with the payload as an input parameter.
Lambda service contains all the functions that are needed, such as :

1. **Voting Manager** - the function that contains the logic for the voting process itself, receives commands, checks all necessary credentials, voting eligibility and interacts with the database, creating new vote records.
2. **Deadline Handler** - the function that is triggered at the specified time and finishes the election process, also calculating results and distributing them among voters.

3. **Breach Handler** - the function that reacts to Management Console login attempts and if someone tries to log in while the election is in progress - informs independent observers that someone breached the account and voting results may be compromised.

Each **Lambda** function has its own assigned **IAM Role** that gives it permissions to certain AWS services.

1. **Voting Manager** has access to:
    - **DynamoDB** to work with database entries for voter and candidate data, vote records and general system parameters
    - **SES**, to distribute automatically generated voter tokens
2. **Deadline Handler** has access to:
    - **DynamoDB** to get access to system parameters table and voting results
    - **SES** to distribute voting results
3. **Breach Handler** has access to:
    - **DynamoDB** to get access to system parameters table
    - **SNS** to inform all observers about the breach

The only permanent set of credentials in the AWS account is the single login/password pair for Management Console admin access. There are no permanent API key pairs so no one can access AWS resources from outside the AWS environment itself. And if the admin login/password pair is somehow leaked - the login attempt will be intercepted by the **Breach Handler** and all observers will be alerted.

All Lambda functions interact with **DynamoDB**, which has four dedicated tables:

1. **Voter Data** - records for each voter, with the following fields:
    - Student ID (supplied from the outside)
    - E-Mail address (supplied from the outside)
    - Personal token (generated automatically on registration)
    - “Voted” flag (set to false automatically on registration)

2. **Candidate Data** - records for each candidate with the following fields:
    - Candidate full name
    - Candidate EMail address
    - A link to Candidate’s web page with election program

3. **Vote Records** - records for each vote, with the following fields:
    - Event ID (generated by AWS automatically)
    - Candidate, for which the vote is cast

4. **System Parameters** - various internal parameters, such as:
    - Admin token, needed for authentication of all setup commands passing through API Gateway
    - E-Mail address from which voting results will be distributed after the deadline
    - Locked flag, turning on the **Breach Handler**
    - Started flag, indicating that the voting process is in progress

When the **Breach Handler** function is triggered - it interacts with the **System Parameters** table in **DynamoDB** to see whether the system is supposed to be locked and the **Observer Topic** in **SNS** to send actual emails with a breach warning to observers.

When the Deadline handler is triggered - it interacts with the **Candidate Data** table in **DynamoDB** to get the list of all candidate names and with the **Vote Records** table to get records and count actual votes. It then uses **SES** to send the summary to all voters, whose EMail addresses it retrieves from the **Voter Data** table.

CloudWatch Events have a single rule that is set to trigger **Deadline Handler** at a time specified in the rule in CRON format.

CloudTrail keeps track of all API requests within the AWS account and writes their info to **CloudWatch Logs**. Then, those log entries are filtered with a user-defined filter for “ConsoleLogin” events. When such an event appears - it triggers the **Breach Handler** function in **Lambda**.

6.2 Lambda function workflows

This section explains the internal logic of three Lambda functions that oversee the election process.

6.2.1 Voting Manager

This function processes incoming API commands and launches a corresponding handler for that particular command.

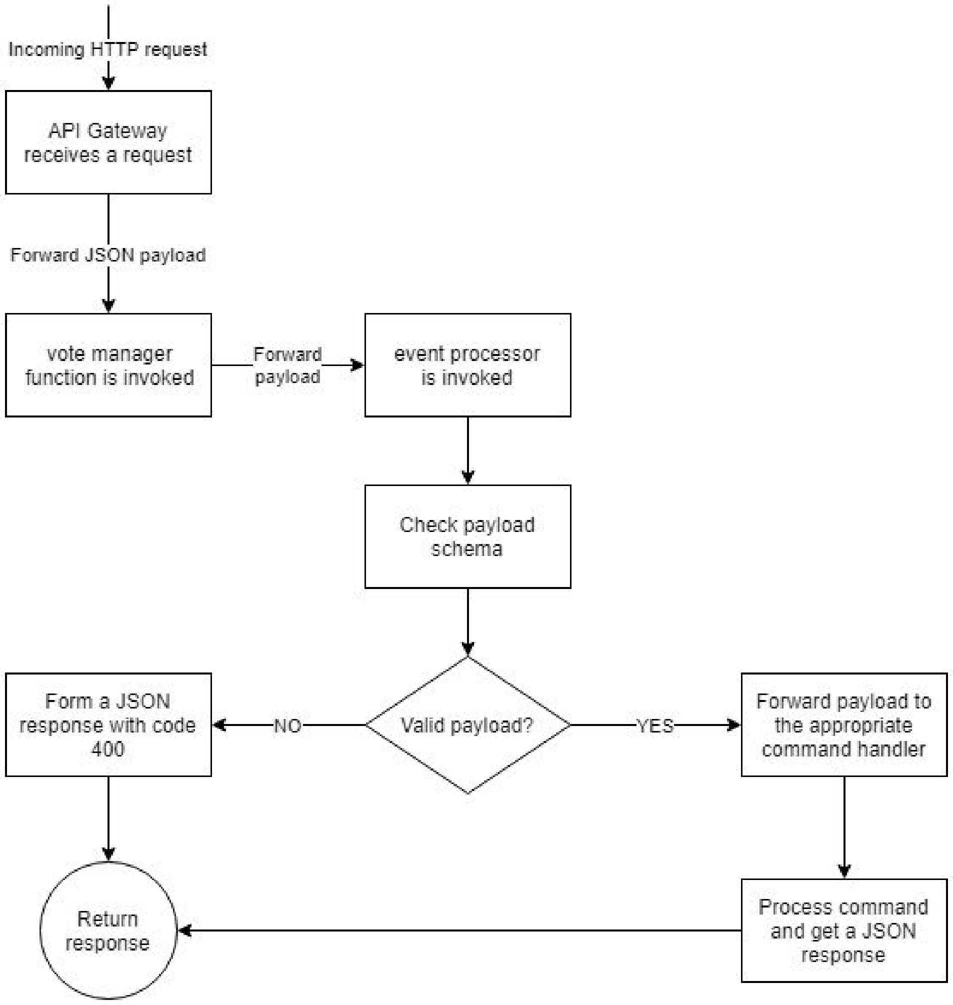

Incoming HTTP request
API Gateway receives a request
Forward JSON payload
vote manager function is invoked
Forward payload
event processor is invoked
Check payload schema
Valid payload?
NO
YES
Form a JSON response with code 400
Forward payload to the appropriate command handler
Process command and get a JSON response
Return response

Fig. 6.2 HTTPS request processing in Voting Manager

This Lambda function is triggered by the **API Gateway** when it receives a request, forwarding the JSON file as an input. The **Voting Manager** creates an Event Processor object and makes it validate the JSON schema of the incoming payload to make sure it fits any of the defined commands. This is a strict requirement, no other fields and/or data types aside from the ones defined in the schema are accepted. If the payload is invalid, a 403 code is returned and the command is rejected. If the payload is valid - it is forwarded to the handler method for that command.

There are six possible commands in total:

Command	Action	Required token	Requirements
add_voters	Adds voters from the provided list to the database	Admin	STARTED flag FALSE
add_candidates	Adds candidates from the provided list to the database	Admin	STARTED flag FALSE
cast_vote	Adds the record for a vote to the table	Voter	STARTED flag TRUE
get_candidates	Gets a list of data for all candidates	None	STARTED flag TRUE
lock	Sets the LOCK flag to true	Admin	LOCKED flag FALSE
start	Sets the STARTED flag to true	Admin	LOCK flag TRUE START flag FALSE

Note that each command can be only used at certain system states. For example, you won't be able to add more voters or candidates if voting is already started.

JSON payload examples for commands:

(all numbers such as voter/administrator tokens shown are examples)

add_voters:

```
“command” : “add_voters”,
“administrator_token” : “123456”,
“voters_to_add”: [
                        {“student_id”: “256”,
                         “Student_email”: “email@address.com”},
                        {“student_id”: “300”,
                         “Student_email”: “another_email@address.com”}
                       ]
```

add_candidates:

```
“command” : “add_candidates”,
“administrator_token” : “123456”,
“candidates_to_add”: [
                              {“candidate_full_name”: “First Candidate”,
                               “candidate_email”: “first@address.com”,
                               “candidate_page_link”: “https://first.com”},
                              {“candidate_full_name”: “Second Candidate”,
                               “candidate_email”: “second@address.com”,
                               “candidate_page_link”: “https://second.com”},
                             ]
```

cast_vote:

```
“command” : “cast_vote”,
“voter_token” : “23469576143958619576 45”,
“voter_id” : “256”,
“candidate_full_name”: “First Candidate”
```

get_candidates:

```
command” : “get_candidates”
```

lock:

```
“command” : “lock”,
“administrator_token” : “123456”
```

start:

```
“command” : “start”,
“administrator_token” : “123456”
```

Command handler workflows:

add_voters:

This is the handler for the command that attempts to add new voters to the database.

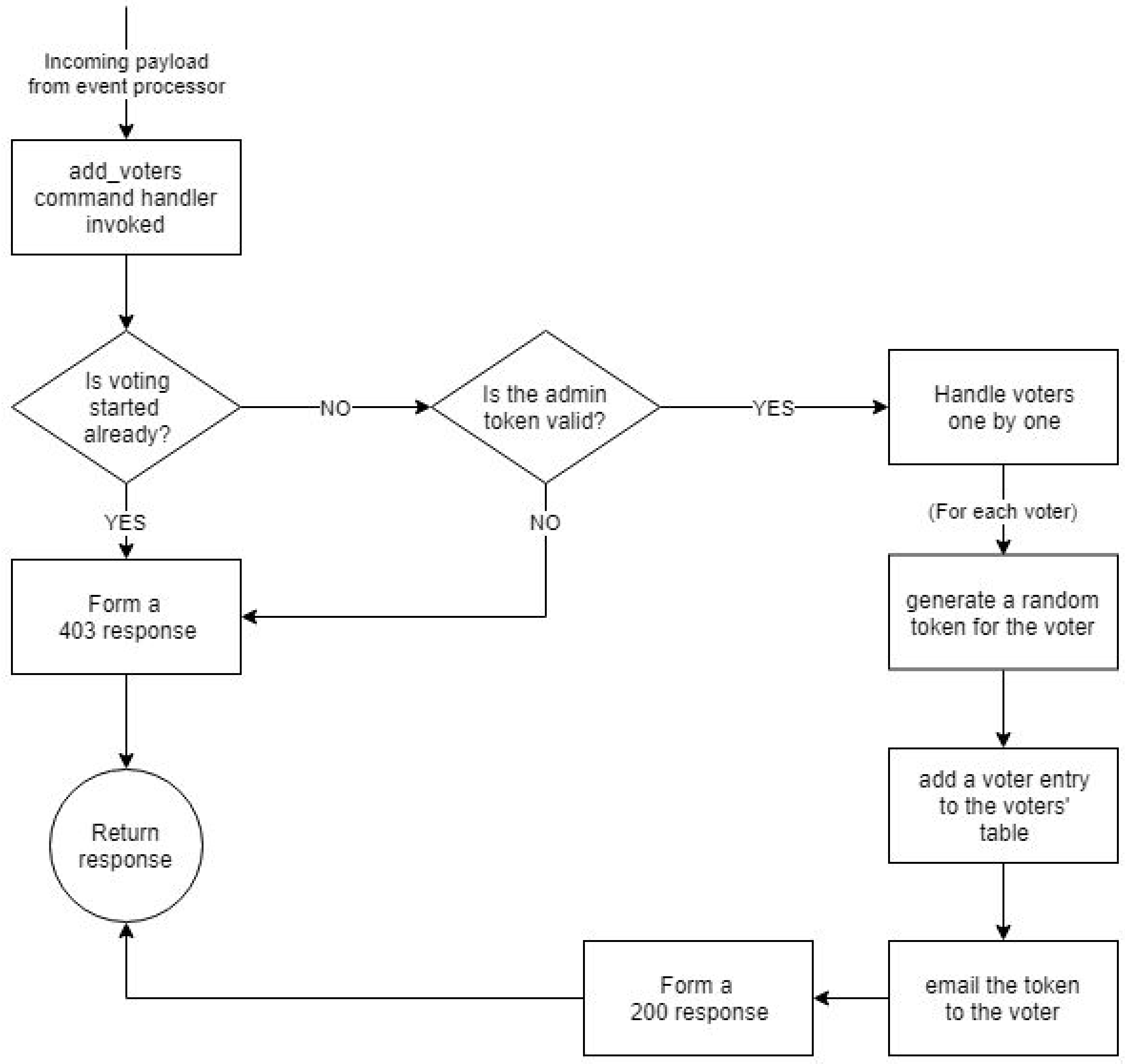

Incoming payload from event processor
add_voters command handler invoked
Is voting started already?
NO
YES
Is the admin token valid?
YES
NO
Handle voters one by one
(For each voter)
generate a random token for the voter
add a voter entry to the voters' table
email the token to the voter
Form a 403 response
Form a 200 response
Return response

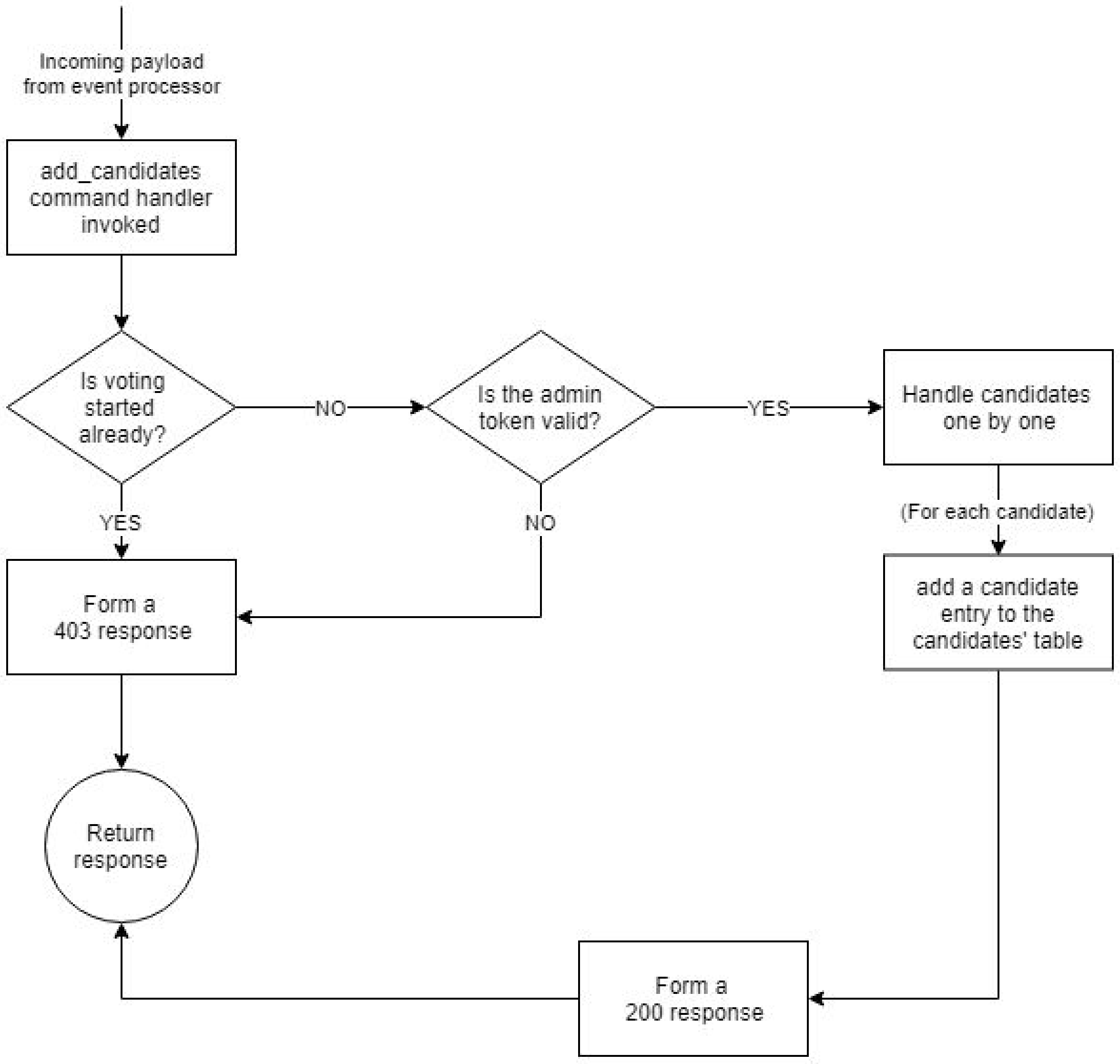
Fig. 6.3 Function flow

First, the system checks whether the voting is already going on. If yes, then one can’t allow to add new candidates, so the function returns a 403 response. If voting hasn’t started yet, then it is acceptable to add new voters at this stage. That being said, the system can only accept the command payload from a valid admin token holder, so the function checks for that. If the token is invalid - return a 403 response. If valid - proceed to add each voter from the list into the database. The system also automatically generates a token for each voter and emails it to them. When finished, return a 200 response.

Note that it is also possible to populate the voters table directly via the Management Console, but as the number of voters is reasonably high, it is easier to add them in batches via the API endpoint.

add_candidates:

This is the handler for the command that attempts to add new candidates to the database.



Incoming payload from event processor
add_candidates command handler invoked
Is voting started already?
NO
Is the admin token valid?
YES
Handle candidates one by one
YES
NO
(For each candidate)
Form a 403 response
add a candidate entry to the candidates' table
Return response
Form a 200 response

Fig. 6.4 Function flow

First, the system checks whether the voting is already going on. If yes, then it's not acceptable to allow to add new candidates, so the function returns a 403 response. If voting hasn't started yet, then it is acceptable to add new candidates at this stage. That being said, the system can only accept the command payload from a valid admin token holder, so the function check for that. If the token is invalid - return a 403 response. If valid - proceed to add each candidate from the list into the database. When finished, return a 200 response.

Note that it is also possible to populate the candidates table directly via the Management Console, the API command is there for consistency with adding voters, it is not necessary to use it.

cast_vote:

This is the handler for the command that attempts to cast a vote for a candidate.

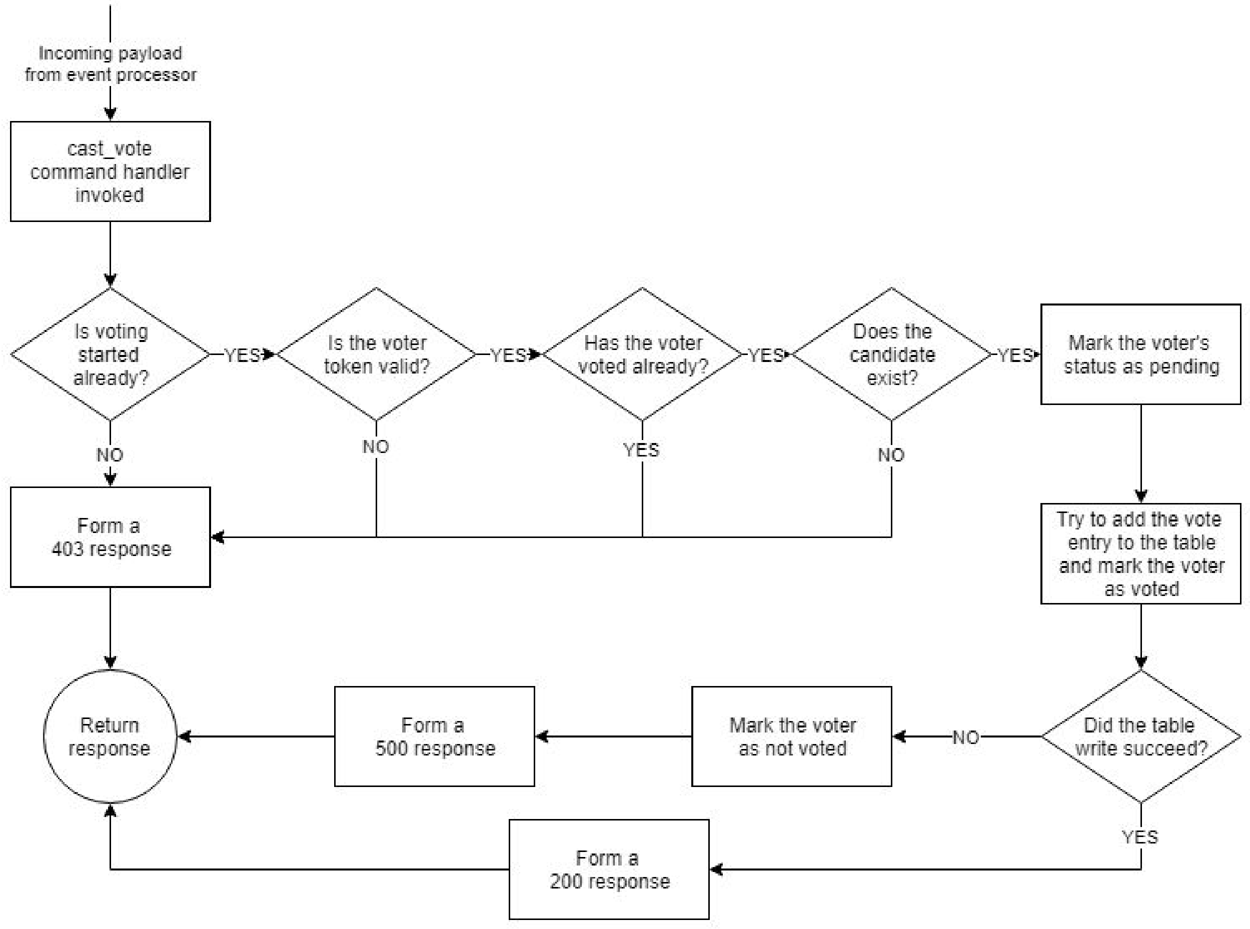

Incoming payload from event processor
cast_vote command handler invoked
Is voting started already?
YES
NO
Is the voter token valid?
YES
NO
Has the voter voted already?
YES
YES
Does the candidate exist?
YES
NO
Mark the voter's status as pending
Form a 403 response
Try to add the vote entry to the table and mark the voter as voted
Return response
Form a 500 response
Mark the voter as not voted
NO
Did the table write succeed?
YES
Form a 200 response

Fig. 6.5 Function flow

This command has the most checks of all. First, the function checks whether the voting process has started. If not - reject the command and return a 403. Then it checks the voter token and only proceed if it’s valid. After that, check whether this voter has already voted and if yes - do not allow a second vote, returning a 403. Then, obviously, it is necessary to check whether they are voting for an actual existing candidate and proceed if the candidate is valid.

Before adding a vote entry, the function sets the “voted” flag for this particular voter to “pending” so that a possible parallel execution of Voting Manager with the same API payload couldn’t cast several votes at the same time. This way, parallel calls will get rejected at one of the previous checks.

Then, switch the “voted” flag to true, try to add a vote entry to the database and check whether it actually succeeded, returning a 500 and reverting the “voted” flag back to false if not. This is an important step as otherwise one could mark the voter as voted but not actually count their vote. If succeeded - return a 200.

get_candidates:

This is the handler for the command that attempts to fetch all candidate data from the database.

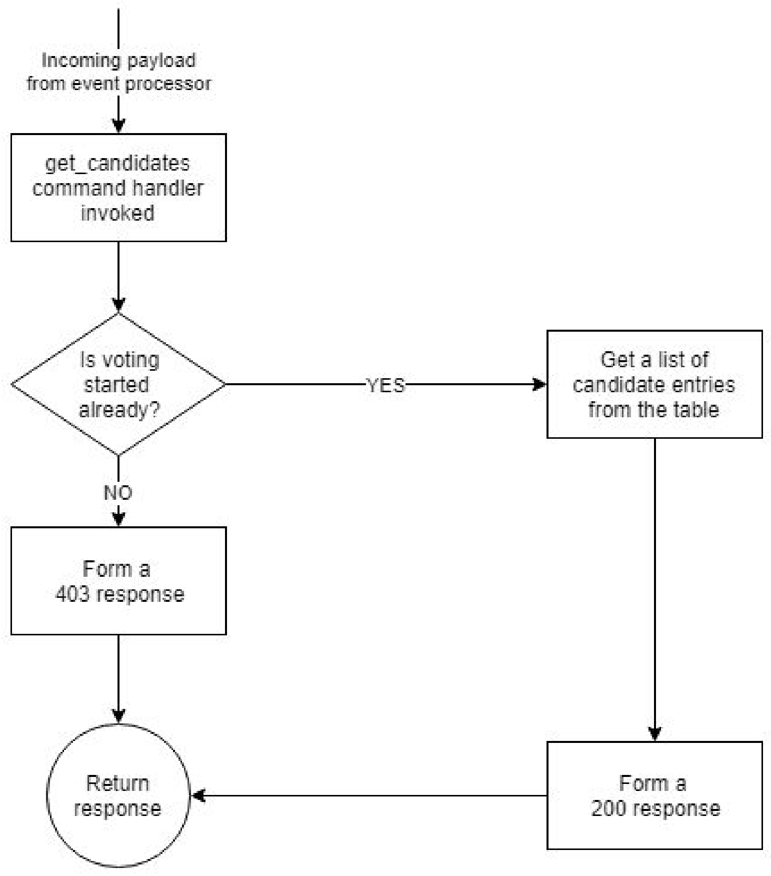

Incoming payload from event processor
get_candidates command handler invoked
Is voting started already?
YES
NO
Get a list of candidate entries from the table
Form a 403 response
Return response
Form a 200 response

Fig. 6.6 Function flow

First of all, check whether the voting process has already started. If not - don’t allow to fetch candidates’ list, as it may not be final. If yes - fetch the list and return it with the 200 code.

lock:

This is the handler for the command that attempts to lock the system from further login attempts.

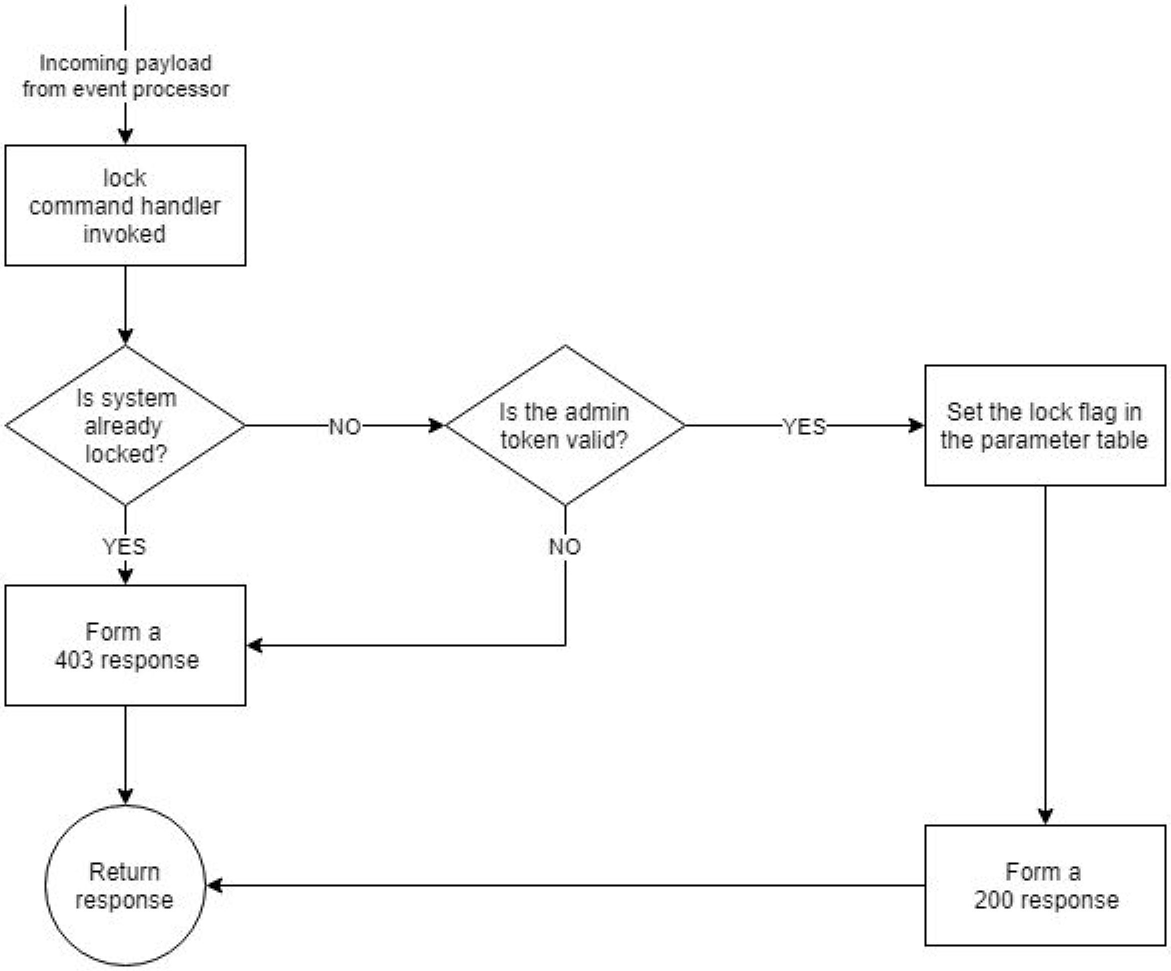

Incoming payload from event processor
lock command handler invoked
Is system already locked?
NO
Is the admin token valid?
YES
Set the lock flag in the parameter table
YES
NO
Form a 403 response
Return response
Form a 200 response

Fig. 6.7 Function flow

If the system is already locked - reject the command. Same if the admin token is invalid. Otherwise set the lock flag to true.

Note that it can also be done directly via the Management Console.

start:

This is the handler for the command that attempts to start the voting process (enabling voting-related commands).

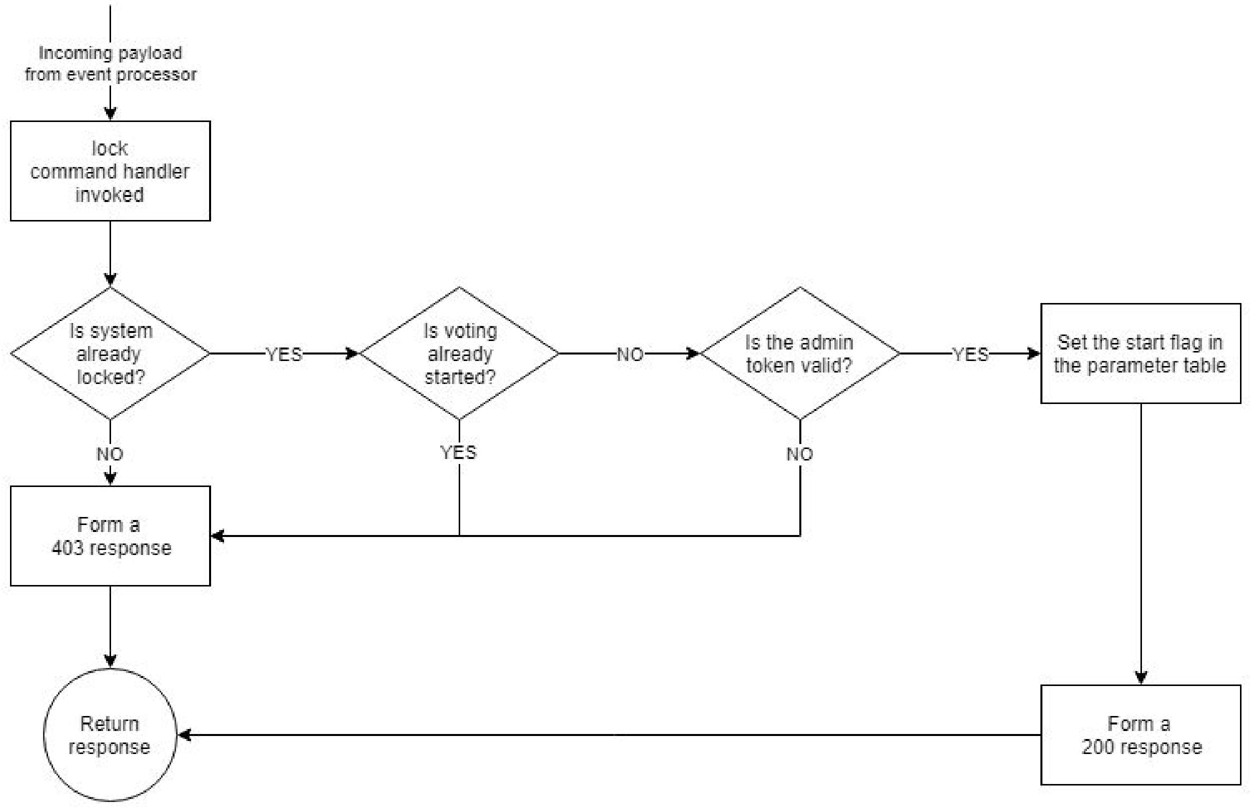

Incoming payload from event processor
lock command handler invoked
Is system already locked?
YES
NO
Is voting already started?
NO
YES
Is the admin token valid?
YES
NO
Set the start flag in the parameter table
Form a 403 response
Return response
Form a 200 response

Fig. 6.8 Function flow

If the system is not locked or already started - reject the command. Same if the admin token is invalid. Otherwise set the start flag to true.

Note that it can also be done directly via the Management Console.

6.2.2 Deadline Handler

This function is responsible for all actions that need to be performed after the voting deadline, such as vote counting and sending results to voters.

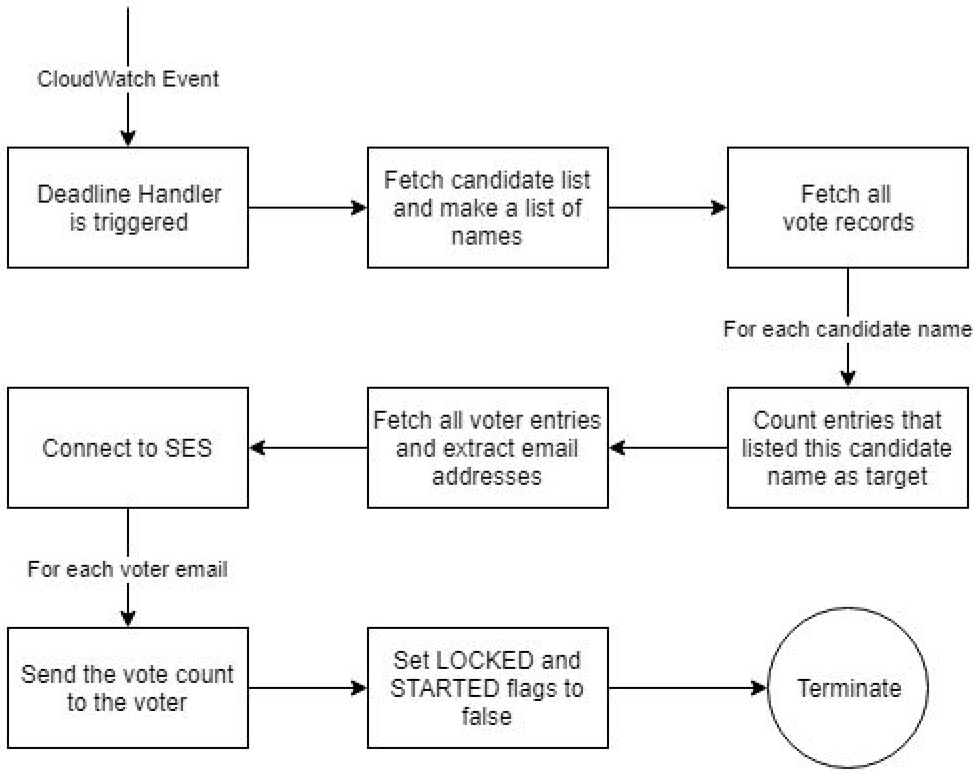

CloudWatch Event
Deadline Handler is triggered
Fetch candidate list and make a list of names
Fetch all vote records
For each candidate name
Count entries that listed this candidate name as target
Fetch all voter entries and extract email addresses
Connect to SES
For each voter email
Send the vote count to the voter
Set LOCKED and STARTED flags to false
Terminate

Fig. 6.9 Deadline Handler workflow

Deadline Handler is very straightforward. It fetches the candidate list, the vote list and the voter list, then counts votes for each candidate and emails the summary to each voter via SES.

6.2.3 Breach Handler

This function is triggered on login attempts into the locked system. It is responsible for informing observers about a security breach.

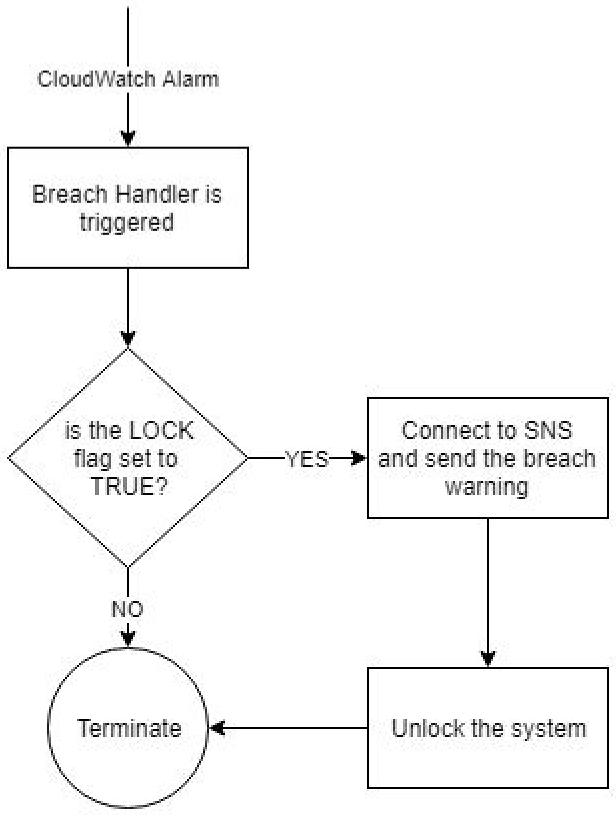

CloudWatch Alarm
Breach Handler is triggered
is the LOCK flag set to TRUE?
YES
Connect to SNS and send the breach warning
NO
Terminate
Unlock the system

Fig. 6.10 Breach Handler workflow

Breach handler checks the LOCKED flag and if it is set to false - then it's just a normal login, which can be ignored as the system is still being set up. If the system is supposed to be locked - the login is unauthorized and a breach is taking place, so the function informs all observers via the corresponding SNS topic and unlock the system for further investigation.

6.3 Interaction between Lambda and other AWS services

All interaction with AWS services outside of the Management Console is done via AWS SDK. In this particular case, the official **boto3** library for Python is used that provides all AWS API methods. By importing this library in our Lambda function code, it is now possible to interact with any other service. All necessary credentials are automatically retrieved from environment variables (which will be set by AWS when the lambda function assumes an IAM role).

7. System setup and administrator actions

This section describes how to set the entire system up and run an entire election lifecycle on it.

7.1 System components

To get a working system, the following is needed:

1. Four **DynamoDB** tables, for corresponding data:
   - Voters' data
   - Candidates' data
   - Vote records
   - System parameters
2. An **SNS** topic to inform observers about breaches
3. Three **Lambda** functions (code for which is provided with this paper):
   - **Voting Manager**
   - **Deadline Handler**
   - **Breach Handler**
4. Three **IAM** roles for three **Lambda** functions with different access permissions:
   - **Voting Manager**: access to **DynamoDB** and **SES**
   - **Deadline Handler**: access to **DynamoDB** and **SES**
   - **Breach Handler**: access to **DynamoDB** and **SNS**
5. An **SES** service to send information to voters
6. An **API gateway** and an API set as a trigger for the **Voting Manager** function
7. A **CloudWatch Event** rule that will trigger **Deadline Handler** at the specified time
8. A **CloudTrail** trail that will automatically send account events to **CloudWatch Logs**
9. A **CloudWatch Logs** subscription that will trigger the **Breach Handler**.

7.2 Setup process

7.2.1 Setting up DynamoDB

This part is straightforward as it is an AWS-managed service so it is much easier to set up than, for example, **Amazon RDS**, which, while partially managed by AWS, still deals with third-party software.

First, go to **DynamoDB** service, then to the "Tables" tab and press the "Create table" button. There, it is necessary to set up the primary partition key for each table. In this particular setup, tables and their keys are the following (case-sensitive):

Table	Primary partition key
VotersTable	student_id (string)
Candidates	full_name (string)
Votes	event_id (string)
VoteManagerParameters	parameter (string)

Then, navigate to the **VoteManagerParameters** table and add the following items:

parameter	value (data type)
admin_token	%WHATEVER VALUE YOU USE% (string)
sender_email	%WHATEVER EMAIL YOU WILL USE% (string)
locked	false (string)
started	false (string)

Here **sender_email** is the email address that will be used to send information to voters.
That is it for the DynamoDB setup.

7.2.2 Setting up an SNS topic

It is necessary to set up an **SNS** topic that will be used to inform observers about security breaches. First, navigate to the **SNS** service page, then to the topics page. Create a topic, give it the desired name and leave everything else at default values. After creating the topic, go into its settings and take note of its **ARN (Amazon Resource Number)**, it will be needed later.

This it it for **SNS** setup.

7.2.3 Setting up AWS Lambda functions

Most of the infrastructure work is setting up Lambda functions.
Firstly, navigate to **AWS Lambda** page and create three functions:

- VoteManager
- DeadlineHandler
- BreachHandler

The following must be repeated for each function:
In the parameters, it is necessary to set the runtime to Python 3.6. In the Permissions section, check the “Create a new role with basic Lambda permissions” mark to automatically get an individual **IAM Role** assigned to each function on creation.

Wait until functions are created, then navigate to each of them and set them up.

The following must be repeated for each function:
Find the Function code window, then the “Code entry type” menu, in which one must choose the option to upload the .zip archive with the function code (archives for all functions are provided with this paper).
Make sure that the “Runtime” field in the “Function code” window is set to Python 3.6 and the “Handler” field is set to the “lambda_function.lambda_handler” value, this one defines the entry point for the function.
Then, navigate to the “Environment variables” section and add necessary parameters. Those are, for each function:

VoteManager

CANDIDATES_TABLE	Name of the candidates table
PARAMETER_TABLE	Name of the parameters table
VOTERS_TABLE	Name of the voters table
VOTES_TABLE	Name of the vote records table

DeadlineHandler

CANDIDATES_TABLE	Name of the candidates table
PARAMETER_TABLE	Name of the parameters table
VOTERS_TABLE	Name of the voters table
VOTES_TABLE	Name of the vote records table

BreachHandler

OBSERVER_ALERT_TOPIC	ARN of the SNS topic for breach alerts
PARAMETER_TABLE_NAME	Name of the parameters table

Now it is necessary to set resource permissions for each function. For that, one needs to edit each function’s **IAM Role**.

In the “Execution role” window, find the link to the role for a given function. By clicking it, one is transported to the page where the list of attached policies is displayed.

Now it’s necessary to attach additional policies to each of our functions’ roles:

VoteManager
- AmazonDynamoDBFullAccess
- AmazonSESFullAccess

DeadlineHandler
- AmazonDynamoDBFullAccess
- AmazonSESFullAccess

BreachHandler
- AmazonDynamoDBFullAccess
- AmazonSNSFullAccess

Side note: normally, according to the AWS Well-Architected Framework, one would give minimal possible permissions to roles. However, considering that there is still room for improvement, it’s easier for a developer to have full permissions to a resource.

Finally, navigate to the “Basic settings” back on the Lambda function’s page. There, set the timeout to a higher value than the default one as testing showed that the default value is sometimes too low. To be on the safer side set it to a minute or two.

This is it for **AWS Lambda** setup.

7.2.4 Setting up SES

SES is a little bit different from other services used in that it is only available in certain regions. In this particular case, the **EU (Ireland)** region is used to set up our **SES** config.
On the **SES** page, find the “Email Addresses” tab and verify the email address from which the system is going to send mails to voters.

Important note:
By default, AWS can only send emails from and to emails that were verified in **SES**. There are two ways around this issue:
- One can implement a verification request via an API call from one of our **Lambda** functions
- One can request a free-of-charge extension of **SES** functionality from Amazon by sending them an email explaining why it is needed

This is it for **SES** setup.

7.2.5 Setting up API Gateway

API Gateway is the service that will actually receive our external HTTPS requests and forward them to the **VoteManager** function.
First, go to the API Gateway page. There, create a new API via the corresponding button. Leave all check marks at their default values (giving us a new REST API), only setting the name.

After our API is created, open its settings. There, via the "Actions" menu, click the "Create method" button and add a POST method. In the settings of the newly created method choose the Lambda integration, check the "Use Lambda Proxy Integration" mark and type the name of our **VoteManager** function in the corresponding field. AWS will ask for a confirmation and upon our agreement will automatically create a trigger linked from our newly created API to the **VoteManager** function.

This is it for **API Gateway** setup.

7.2.6 Setting up a CloudWatch Event

CloudWatch Events will allow us to trigger **DeadlineHandler** function at a specified time.
First, navigate to the **CloudWatch**, then to the "Rules" tab. There, create a new rule, check the "Schedule" mark and provide a CRON expression for when to trigger the **DeadlineHandler** function. It is also necessary to add the **DeadlineHandler** to the trigger targets in the Targets section. When confirming, do not enable this rule by default so that it doesn't accidentally misfire if something is mismanaged before the start of the voting process.

This is it for **CloudWatch Events** setup.

7.2.7 Setting up CloudTrail

CloudTrail is the most crucial element of security in this system. It allows us to track login events and send them further for analysis and alerts.

First of all, go to the **CloudTrail** page and "Trails" tab. There, create a new trail (if there are none so far). Leave all parameters except the trail name as is, they concern integration with the **S3** service, which is not used in this implementation. After the trail is created, a corresponding log group will automatically be available in **CloudWatch Logs**.

This is it for **CloudTrail** setup.

7.2.8 Setting up a trigger for BreachHandler

First, navigate to the **CloudWatch Logs** page, find the log group that was created by **CloudTrail**, select it and via the “Actions” menu add a metric filter to it. In the filter config, set the following filter pattern:

```
{($.eventName = ConsoleLogin)}
```

Then, navigate to the **BreachHandler Lambda** function page and add a trigger to it, choosing **CloudWatch Logs** from the list. There, choose the desired log group and the log filter (which has been set up just now).

Now the **BreachHandler** function will be automatically triggered on new ConsoleLogin events.

This is it for the trigger setup.

At this point everything is ready for running an entire voting lifecycle.

8. Voting lifecycle

This section describes actions that need to be taken by administrators during the voting process to run the process. The user guide assumes that users read through the setup page and can navigate through the pages of different AWS services.

8.1 Independent observers

It is preferred that independent observers are present in person during the initial setup to confirm every action taken by administrators (except for the setup of parameters table as that will expose the administrator token, so you can set that table up in advance).

8.2 Security of credentials

Needless to say, it is crucially important to take care of Management Console login/password pair to avoid a credentials leak. As this is not a technical matter but rather a matter of policy, decision on exact methods of achieving security here is left to users. On the technical side, no other permanent credentials exist, so as long as Management Console login/password pair is confidential - there is no way to access resources in the AWS account, as there are no API keys for that. All API keys are created inside of AWS itself and exposed only to AWS services. Thus, there are only two ways to interact with the vote database during the election:

- Logging into the Management Console (which will fire an alert if the system is locked)
- Using API commands (which require proper admin/voter tokens)

8.3 Observer registration

First of all, admins need to register all independent observers so that they could receive notification in case of a system breach. This is done via navigating to **SNS** page and adding observer email addresses to the list of subscribers.

8.4 Database setup

Before the election process starts, the database needs to be set up properly.
Make sure that tables for voter data, candidate data and vote records are all empty or clear them if they are not. As for the parameters table, parameters need to be set as following:

- **admin_token** must be set to the desired value. It is not generated automatically, you need to come up with the value yourself.
- **sender_email** must be set to the email address from which the system will be sending information to voters
- **locked** and **started** flags must be set to “false”

8.5 Set the election deadline

Admins need to navigate to the **CloudWatch Event** set up earlier for triggering the **DeadlineHandler** function. There, set the execution timestamp in CRON format if it is not set yet and mark the rule as active.

8.6 Add candidates

Admins must add candidate data to the corresponding table either manually or via the API command, whichever preferred.

8.7 Add voters

At this stage, voters are added to the corresponding table. While it is possible to do that manually, using the corresponding API command of the **VoteManager** function would be more convenient due to a large number of voters.

8.8 Lock the system

At this stage, it is necessary to set the **lock** parameter to true to prevent further login attempts to the Management Console. This parameter may be set either manually, or via the API command through the **VoteManager** function.

8.9 Make sure no one else is logged into the Management Console

Admins need to navigate to **CloudWatch Metrics** and check the login statistics to see if there are more than one active connection at the present time. If there are two or more connections - credentials may be leaked (or admins are logged in on several machines) and the matter needs to be investigated.

8.10 Start the voting process

Finally, admins need to set the **started** parameter to true and announce that the voting process is now in progress. As with the **lock** parameter, it is possible both to set it manually and via the API command.

8.11 Log out (or not)

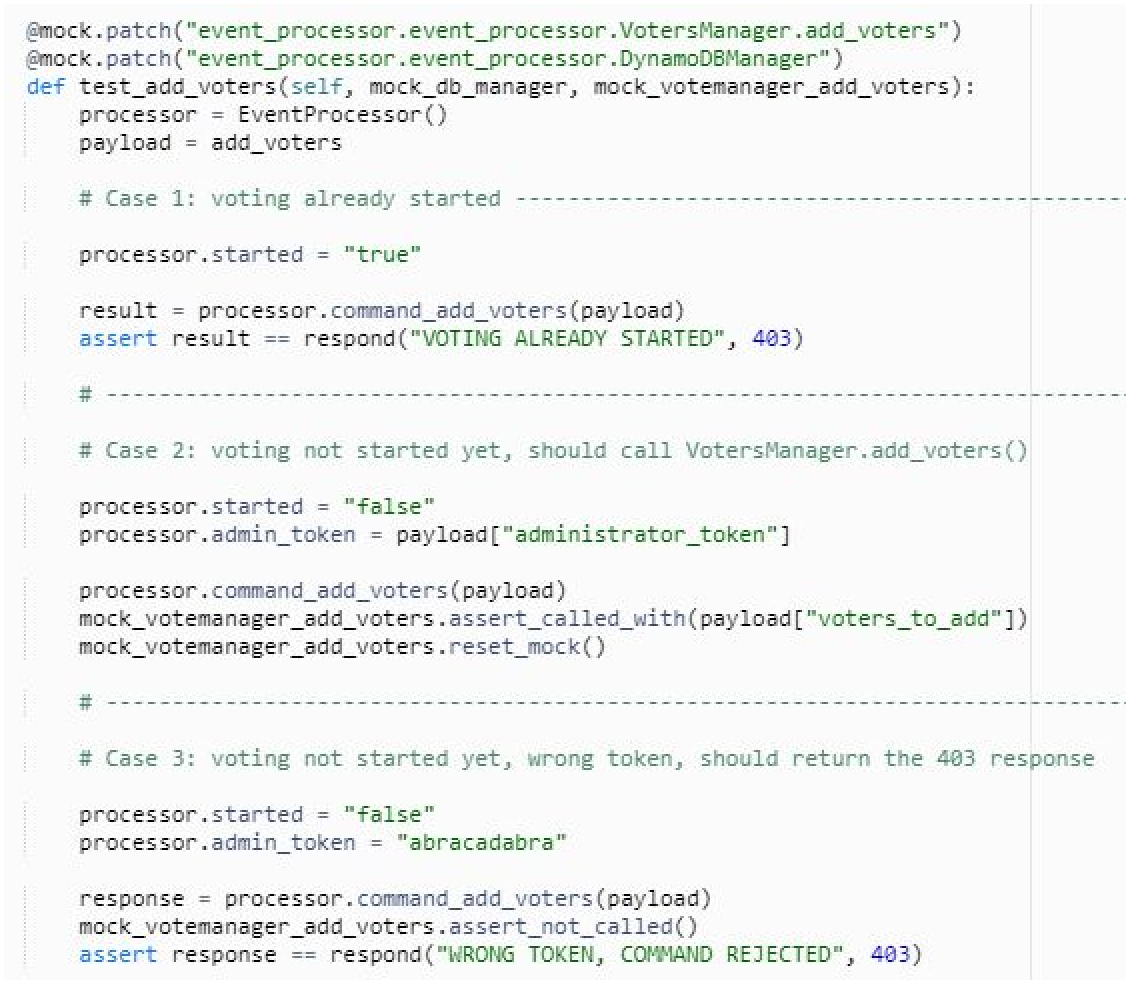
At this stage, admins either log out if they're running an actual election or stay logged in if they are live-testing the system and want to monitor its resources and behavior. After the **DeadlineHandler** or **BreachHandler** is triggered - the system unlocks automatically.

9. Testing the system

As any program, this system needs to be tested before being put into production. In this project, individual components of the system are tested (unit tests) and live-tests are performed in an actual cloud environment.

9.1 Unit tests

VoteManager function in **AWS Lambda** has two test suites - one for testing API request validation, delegation to appropriate handlers and internal handler logic and the second one for interacting with the **DynamoDB** service. Both test suites are implemented with the **unittest** library included by default with all **Python** distributions. All tests contain in-code documentation and are generally designed to be as clear as possible.



```
@mock.patch("event_processor.event_processor.VotersManager.add_voters")
@mock.patch("event_processor.event_processor.DynamoDBManager")
def test_add_voters(self, mock_db_manager, mock_votemanager_add_voters):
    processor = EventProcessor()
    payload = add_voters

    # Case 1: voting already started --------------------------------------------

    processor.started = "true"

    result = processor.command_add_voters(payload)
    assert result == respond("VOTING ALREADY STARTED", 403)

    # ---------------------------------------------------------------------------

    # Case 2: voting not started yet, should call VotersManager.add_voters()

    processor.started = "false"
    processor.admin_token = payload["administrator_token"]

    processor.command_add_voters(payload)
    mock_votemanager_add_voters.assert_called_with(payload["voters_to_add"])
    mock_votemanager_add_voters.reset_mock()

    # ---------------------------------------------------------------------------

    # Case 3: voting not started yet, wrong token, should return the 403 response

    processor.started = "false"
    processor.admin_token = "abracadabra"

    response = processor.command_add_voters(payload)
    mock_votemanager_add_voters.assert_not_called()
    assert response == respond("WRONG TOKEN, COMMAND REJECTED", 403)
```

Fig. 9.1 Unit test example for event handler logic

DeadlineHandler and BreachHandler unit tests are deemed unnecessary due to the simplicity of these single-module functions.

9.2 Live testing

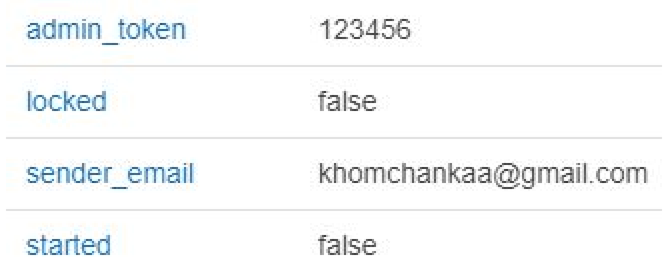
To make sure the system works correctly, an election lifecycle has been run with prepared data to see whether the system behaved as expected. In the course of testing, the following operations have been performed:

9.2.1 Prepare DynamoDB tables

Manually reset all DynamoDB tables and set the following parameters:

- **admin_token**: “123456”
- **sender_email**: “khomchankaa@gmail.com”
- **locked**: “false”
- **started**: “false”

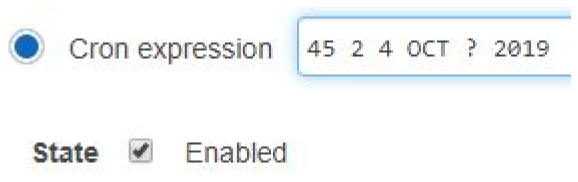
Result:



admin_token	123456
locked	false
sender_email	khomchankaa@gmail.com
started	false

9.2.2 Set the deadline

Set the **CloudWatch Event** for **DeadlineHandler** to active state and set a deadline in CRON format.



Cron expression: `45 2 4 OCT ? 2019`

State ✓ Enabled

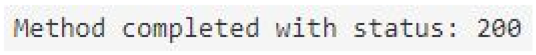
After the intended date and time is reached - it should run the **DeadlineHandler**.

9.2.3 Add voters

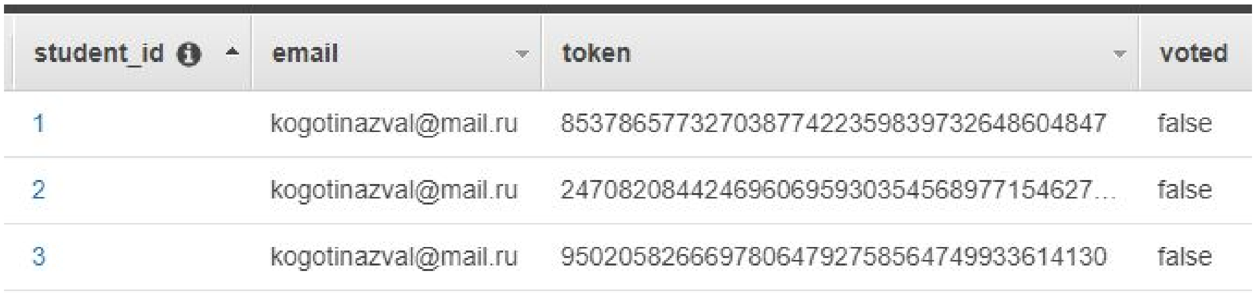
Next, a test payload for adding voters via the API command has been prepared:

```
{  "command": "add_voters",
            "administrator_token": "123456",
            "voters_to_add": [
                                    {"student_id": "1",
                                     "student_email": "kogotinazval@mail.ru"},
                                    {"student_id": "2",
                                     "student_email": "kogotinazval@mail.ru"},
                                    {"student_id": "3",
                                     "student_email": "kogotinazval@mail.ru"}
                                 ]
}
```

After feeding it into my API Gateway, obtain the result:

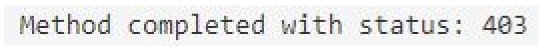


```
Method completed with status: 200
```



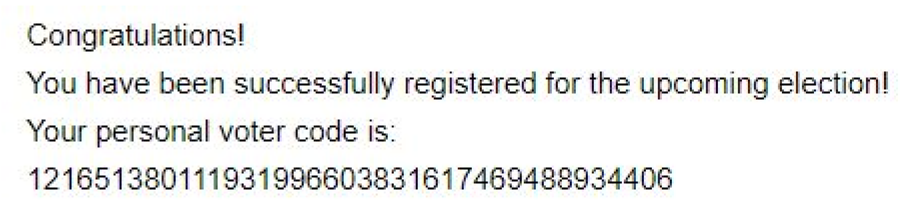
student_id	email	token	voted
1	kogotinazval@mail.ru	853786577327038774223598397326486O4847	false
2	kogotinazval@mail.ru	2470820844246960695930354568977154627...	false
3	kogotinazval@mail.ru	95020582666978064792758564749933614130	false

Voters were added successfully. Trying to add voters via a similar payload, but with a wrong admin token returned a 403 FORBIDDEN response, as intended:



```
Method completed with status: 403
```

All “voters” also got an email with the personal token:



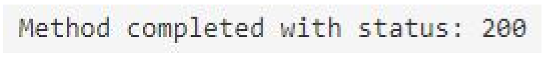
Congratulations!
You have been successfully registered for the upcoming election!
Your personal voter code is:
121651380111931996603831617469488934406

9.2.4 Add candidates

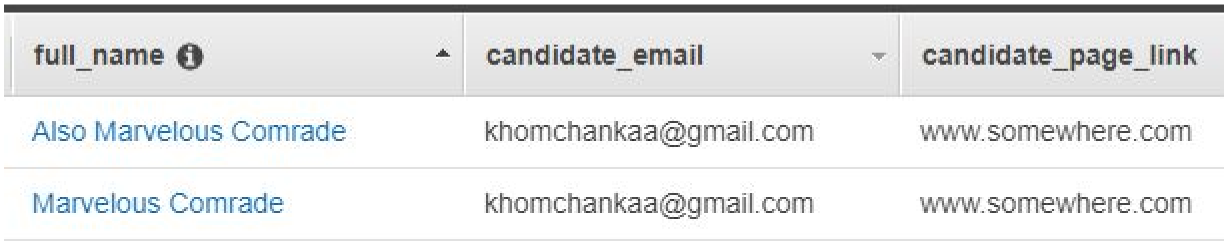
Payload:

```
add_candidates = {  "command": "add_candidates",
                  "administrator_token": "123456",
                  "candidates_to_add":
                     [
                        {"candidate_full_name": "Marvelous Comrade",
                         "candidate_email": "khomchankaa@gmail.com",
                         "candidate_page_link": "www.somewhere.com"
                        },
                        {"candidate_full_name": "Also Marvelous Comrade",
                         "candidate_email": "khomchankaa@gmail.com",
                         "candidate_page_link": "www.somewhere.com"
                        }
                     ]
}
```

API call result:



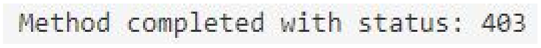
```
Method completed with status: 200
```



full_name	candidate_email	candidate_page_link
Also Marvelous Comrade	khomchankaa@gmail.com	www.somewhere.com
Marvelous Comrade	khomchankaa@gmail.com	www.somewhere.com

Candidates added as intended.

Result from a similar payload, but with a wrong token:



```
Method completed with status: 403
```

9.2.5 Locking and starting the system

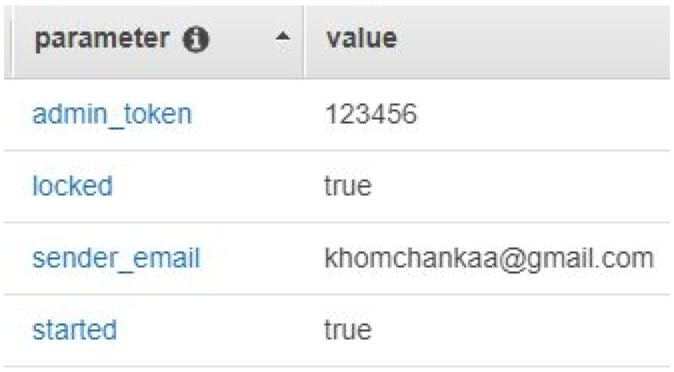
Payloads:

{ "command": "lock",
"administrator_token": "123456"
}

{ "command": "start",
"administrator_token": "123456"
}

Result for both:

```
Method completed with status: 200
```



parameter	value
admin_token	123456
locked	true
sender_email	khomchankaa@gmail.com
started	true

Trying to run both commands again:

```
Method completed with status: 403
```

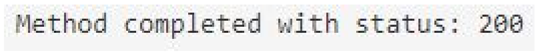
API returns a 403, as expected.

9.2.6 Casting votes

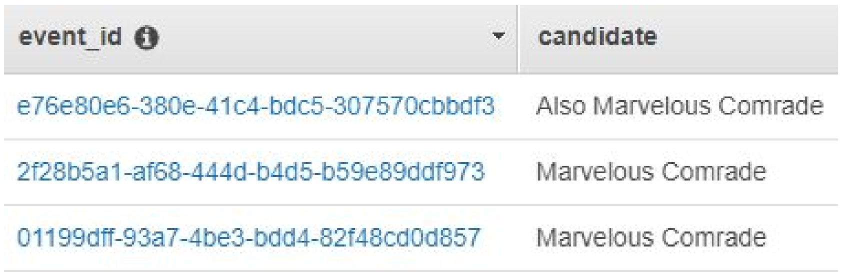
Three payloads for three voters (with added auto-generated tokens):

```
{  "command": "cast_vote",
   "voter_token": "361072638536616714980778382706978236 0",
   "voter_id": "1",
   "candidate_full_name": "Marvelous Comrade"
}

{  "command": "cast_vote",
   "voter_token": "328577712875495374698266069133357368 10",
   "voter_id": "2",
   "candidate_full_name": "Marvelous Comrade"
}

{  "command": "cast_vote",
   "voter_token": "302066180645471304022344990191567280979",
   "voter_id": "3",
   "candidate_full_name": "Also Marvelous Comrade"
}
```

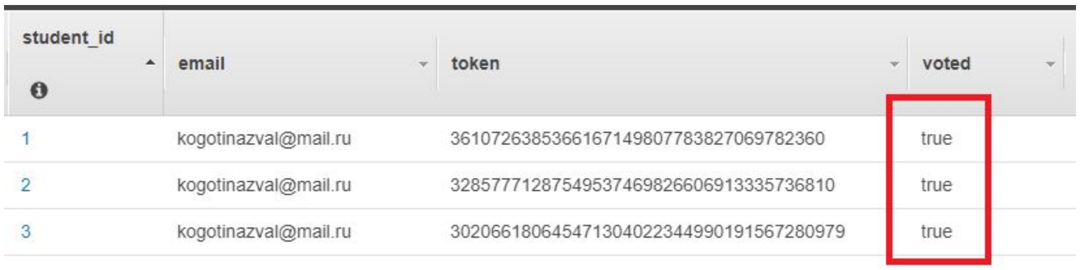
Result:



```
Method completed with status: 200
```

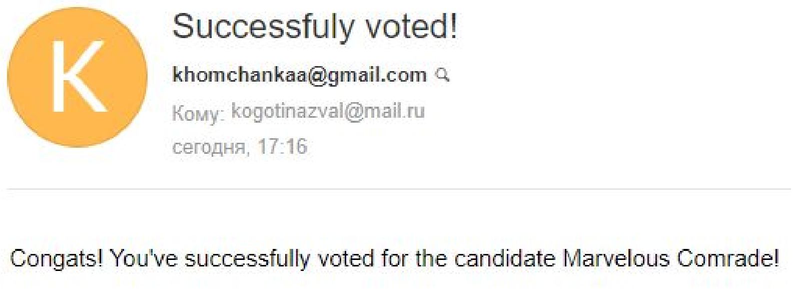


event_id	candidate
e76e80e6-380e-41c4-bdc5-307570cbbdf3	Also Marvelous Comrade
2f28b5a1-af68-444d-b4d5-b59e89ddf973	Marvelous Comrade
01199dff-93a7-4be3-bdd4-82f48cd0d857	Marvelous Comrade



student_id	email	token	voted
1	kogotinazval@mail.ru	3610726385366167149807783827069782360	true
2	kogotinazval@mail.ru	32857771287549537469826606913335736810	true
3	kogotinazval@mail.ru	302066180645471304022344990191567280979	true

Note that voters are now marked as voted.
Voters are also successfully informed about their vote going through:


Successfuly voted!

khomchankaa@gmail.com

Кому: kogotinazval@mail.ru

сегодня, 17:16

Congats! You've successfully voted for the candidate Marvelous Comrade!

Trying to vote again or vote before voting is started results in command rejection:

```
Method completed with status: 403
```

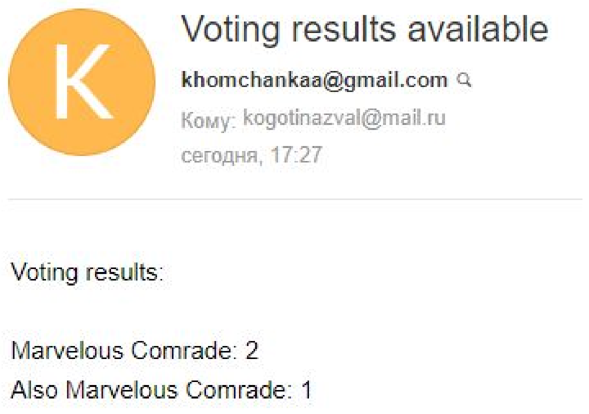
9.2.7 Trying to add voters and candidates during the election

Attempting to add new voters and candidates during the election results in a 403 FORBIDDEN, as intended:

```
Method completed with status: 403
```

9.2.8 DeadlineHandler being triggered

After the DeadlineHandler is triggered, voters receive emails with voting results:


Voting results available

khomchankaa@gmail.com

Кому: kogotinazval@mail.ru

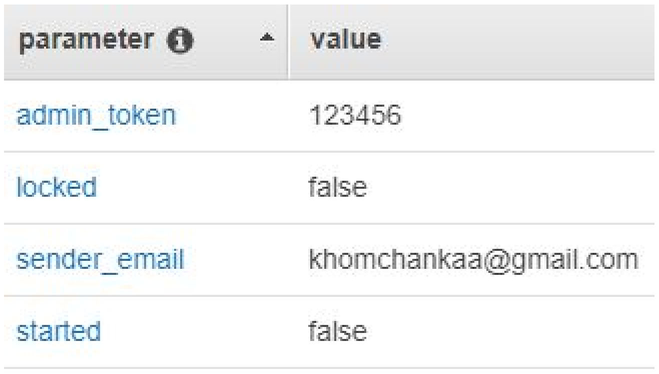
сегодня, 17:27

Voting results:

Marvelous Comrade: 2

Also Marvelous Comrade: 1

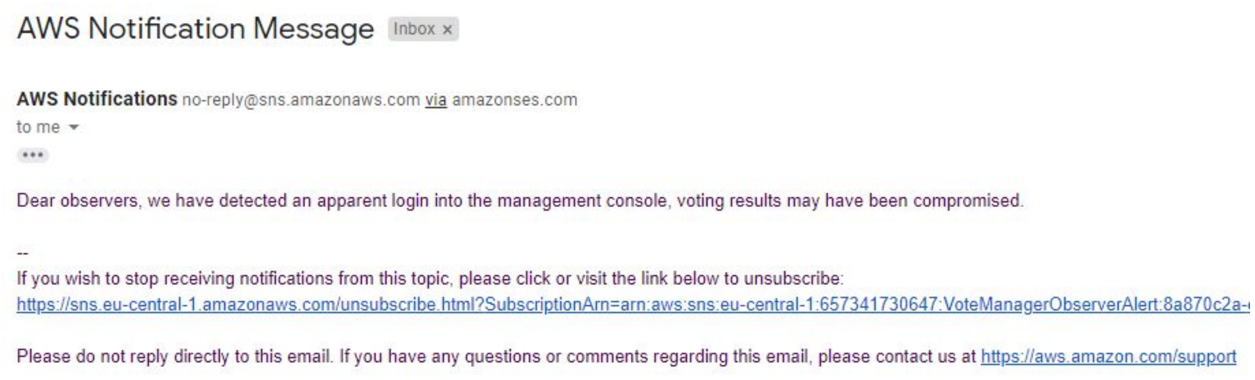
System is unlocked as well:



parameter	value
admin_token	123456
locked	false
sender_email	khomchankaa@gmail.com
started	false

9.2.9 Trying to log into a locked account (triggering BreachHandler)

When the lock parameter is set to “true” - a login attempt results in the alert email:



AWS Notification Message Inbox ×

AWS Notifications no-reply@sns.amazonaws.com via amazonses.com
to me

Dear observers, we have detected an apparent login into the management console, voting results may have been compromised.

--
If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
https://sns.eu-central-1.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:eu-central-1:657341730647:VoteManagerObserverAlert:8a870c2a-

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at https://aws.amazon.com/support

This is intended behaviour and what is even more reassuring, this happens at a split second, meaning there’s no way for someone to log in and “cut the line” before the message is sent.

10. Security analysis

Main points where security needs to be ensured, are:

- AWS credentials must only be available to administrators and have a single login-password pair. No API access from outside AWS backend must be possible.
- Administrator tokens must be stored securely inside the AWS infrastructure and only be available within said infrastructure.
- All voting-related actions must be properly authorized with corresponding access keys.
- All API commands (not AWS API, but voting platform API) must have a strictly defined schema and be safe from code injections.
- The AWS account must be securely locked with an active alarm during the voting process.

Credentials-wise, the only login-password pair explicitly present in IAM is administrator credentials for the Management Console. This key pair can not be removed as there still needs to be access to the account outside of the voting process timeline. This key pair can not be used to authenticate API calls, so no one can alter system's AWS resources in any way from outside the infrastructure using this set of credentials. All services in the implemented platform use IAM roles. As a result, no permanent API access key pairs exist, they are automatically created inside AWS backend specifically for AWS services on-demand and are immediately deleted after functions terminate. Thus, it can be concluded, that no external API access is possible from outside the AWS infrastructure.

Administrator token is stored in a DynamoDB database, which, being a service inside the AWS infrastructure, can only be accessed by other AWS resources via IAM roles and not outside actors due to the aforementioned fact that no external API keys exist.

All voting API commands (aside from the get_candidates command, which can be allowed for anyone) require authorization using either an administrator token (for adding voters/candidates and locking/starting the system), or a voter token (for actual voting). If an incorrect token is used - the command is immediately rejected before making any actual changes to AWS resources.

Each voting API command has a predefined JSON schema and is rejected if there are any deviations from said schema. Code injections also are not possible due to the strict schema and the way Python libraries processes data in JSON format. Since there are no attempts in the code to directly execute some string variables as code (such as using eval() method) - there exists no possibility to inject code via incoming JSON payloads. As there are no other incoming messages aside from said payloads - no code injection is possible at all.

Using CloudTrail allows for immediate detection of login attempts in Management Console and informing observers about security breaches. Live testing, described in Chapter 9.2.9, confirms that the timing of the entire process is such that it would not be possible to log into the Management Console and disable the BreachHandler function or the SES service before it sends a notification to observers. By the time the Management Console even loads, notifications would already be on their way. It would be possible to disable alarms from outside using external API keys, but as mentioned before, none exist.

Other security measures, such as firewalls, which would normally be present in cloud-based systems, don’t need to be taken care of in this case due to all utilized AWS components being serverless. All network security-related work is done automatically by AWS backend.

11. Requirement analysis

This chapter describes requirement analysis of the implemented system and whether it adheres by said requirements.

A web API endpoint for receiving remote commands

This requirement is fulfilled by the API gateway service, providing an endpoint for all commands.

Requirement fulfilled.

API commands for:

Adding new voters/candidates
Casting votes for candidates
Fetching information about available candidates
Locking the system and starting voting

Implemented commands:
- add_voters
- add_candidates
- get_candidates
- cast_vote
- lock
- start

provide all necessary functionality.

Requirement fulfilled.

Autonomous internal program(s) for handling incoming votes

Internal logic for incoming votes is handled by accepting API commands via the API Gateway, then forwarding them to the VotingManager Lambda function, which, in turn, is integrated with all necessary AWS services, such as DynamoDB, SNS and SES.

Requirement fulfilled.

Autonomous internal program(s) for handling vote counting and informing voters about results

Internal logic for incoming votes is handled by accepting API commands via the API Gateway, then forwarding them to the DeadlineHandler Lambda function, which, in turn, is integrated with all necessary AWS services, such as DynamoDB and SES.

Requirement fulfilled.

Autonomous internal program(s) for handling possible security breaches

Security breaches are handled by CloudTrail that logs login attempts and invokes the BreachHandler Lambda function that uses SNS to send actual notifications.

Requirement fulfilled.

Methods of authentication for incoming votes and administrator commands

Authentication of incoming votes and administrator command is done via administrator tokens and auto-generated voter tokens, all of which are stored in DynamoDB tables. Authentication logic itself is implemented in the VotingManager lambda function.

Requirement fulfilled.

A database for storing voter data
A database for storing candidate data
A database for storing votes
A database for storing internal system parameters

All data is stored in separate DynamoDB tables.

Requirement fulfilled.

All components are serverless

All components have indeed been made serverless by using specialized services like Lambda, SNS, SES and others. Said services allow all necessary functionality without the need to explicitly provision servers.

Requirement fulfilled.

A single control interface for the entire system

All system components can be managed via the default AWS Management Console, there is no need to develop any additional tools as AWS already provides all instruments for editing and monitoring resources.

Requirement fulfilled.

As can be seen, all requirements have been fulfilled, including non-functional, which, while not extending functionality itself, allow for easier maintenance and monitoring of resources, delegating some security responsibilities to the cloud provider and making the job of independent observers during the voting process easier.

12. Conclusions

12.1 Development results

The result of the project is a functioning cloud-based voting platform capable of all necessary actions that would be required of such a system. Setup instructions and operating notes have also been written to explain the structure and help possible future users.

The implemented system is performing exactly as was intended by the initial functional requirements and even simplifies various elements by using serverless approach and relying on existing vendor-provided management tools for our cloud infrastructure. Several approaches have been tried/considered and rejected with an explanation of why they may not be the best fit for this system.

12.2 Possible improvements

While the system does perform all necessary basic functions of a voting platform, there are several improvements that can be made in the future to make it more convenient and reliable:

12.2.1 A default front-end application

A simple application (perhaps even a mobile one) could be added to be able to demonstrate the system to possible users in a more convenient manner.

12.2.2 A CloudFormation stack

The whole infrastructure can be packaged into a deployment package for **AWS CloudFormation**. CloudFormation is a particularly useful service that allows to deploy entire infrastructures with multiple microservices (including all credential relations) in very little time. This was attempted with an automated **CloudFormer** tool provided by AWS, but there were some difficulties with running it (possibly because of that tool still officially being in beta status). Writing a CloudFormation stack, while producing an extremely useful end result, also requires a more extensive study of CloudFormation and orchestration in AWS , which goes beyond the scope of this thesis, although outside of it is definitely a wise idea to implement such a stack.

References

1. **E-Voting in Estonia 2005. The first practice of country-wide binding Internet voting in the world**, Madise, Martens, 2nd International Workshop, Bregenz, Austria, 2006
https://subs.emis.de/LNI/Proceedings/Proceedings86/GI-Proceedings-86-1.pdf
DATE OF ACCESS: 20.10.2019

2. **Security Analysis of the Estonian Internet Voting System**, Springall, Finkenauer, Durumeric, Kitcat, Hursti, MacAlpine, Halderman, University of Michigan, USA
https://jhalderm.com/pub/papers/ivoting-ccs14.pdf
DATE OF ACCESS: 20.10.2019

3. **Three Case Studies from Switzerland: E-Voting**, Gerlach, Gasser, Berkman Center Research Publication No. 2009-03.1
https://cyber.harvard.edu/sites/cyber.law.harvard.edu/files/Gerlach-Gasser_SwissCases_Evoting.pdf
DATE OF ACCESS: 20.10.2019

4. **Amazon Data Centers page**
https://aws.amazon.com/compliance/data-center/data-centers/
DATE OF ACCESS: 20.10.2019

5. **AWS Lambda service page**
https://aws.amazon.com/lambda/
DATE OF ACCESS: 20.10.2019

6. **EC2 service page**
https://aws.amazon.com/ec2/
DATE OF ACCESS: 20.10.2019

7. **IAM service page**
https://aws.amazon.com/iam/
DATE OF ACCESS: 20.10.2019

8. **Amazon RDS service page**
https://aws.amazon.com/rds/
DATE OF ACCESS: 20.10.2019

9. **DynamoDB service page**
https://aws.amazon.com/dynamodb/
DATE OF ACCESS: 20.10.2019

10. **SNS service page**
https://aws.amazon.com/sns/
DATE OF ACCESS: 20.10.2019

11. **SES service page**
 https://aws.amazon.com/ses/
 DATE OF ACCESS: 20.10.2019

12. **CloudWatch service page**
 https://aws.amazon.com/cloudwatch/
 DATE OF ACCESS: 20.10.2019

13. **CloudTrail service page**
 https://aws.amazon.com/cloudtrail/
 DATE OF ACCESS: 20.10.2019

14. **Amazon Managed Blockchain service page**
 https://aws.amazon.com/managed-blockchain/
 DATE OF ACCESS: 20.10.2019

15. **API Gateway service page**
 https://aws.amazon.com/api-gateway/
 DATE OF ACCESS: 20.10.2019

16. **A Conceptual Secure Blockchain-based Electronic Voting System,** Ahmed Ben Ayed, International Journal of Network Security & Its Applications, May 2017
 http://aircconline.com/ijnsa/V9N3/9317ijnsa01.pdf
 DATE OF ACCESS: 20.10.2019

Declaration

I declare within the meaning of part 16(5) of the General Examination and Study Regulations for Bachelor and Master Study Degree Programmes at the Faculty of Engineering and Computer Science and the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

City, Date   Signature