Electronic Voting

Architecture Requirements Specification

Andreas du Preez 12207871

Azhar Mohungoo 12239799

Gift Sefako 12231097

Stakeholders

Epi-Use Advance Roelof Nuade

Contents

1 Architectural Requirements

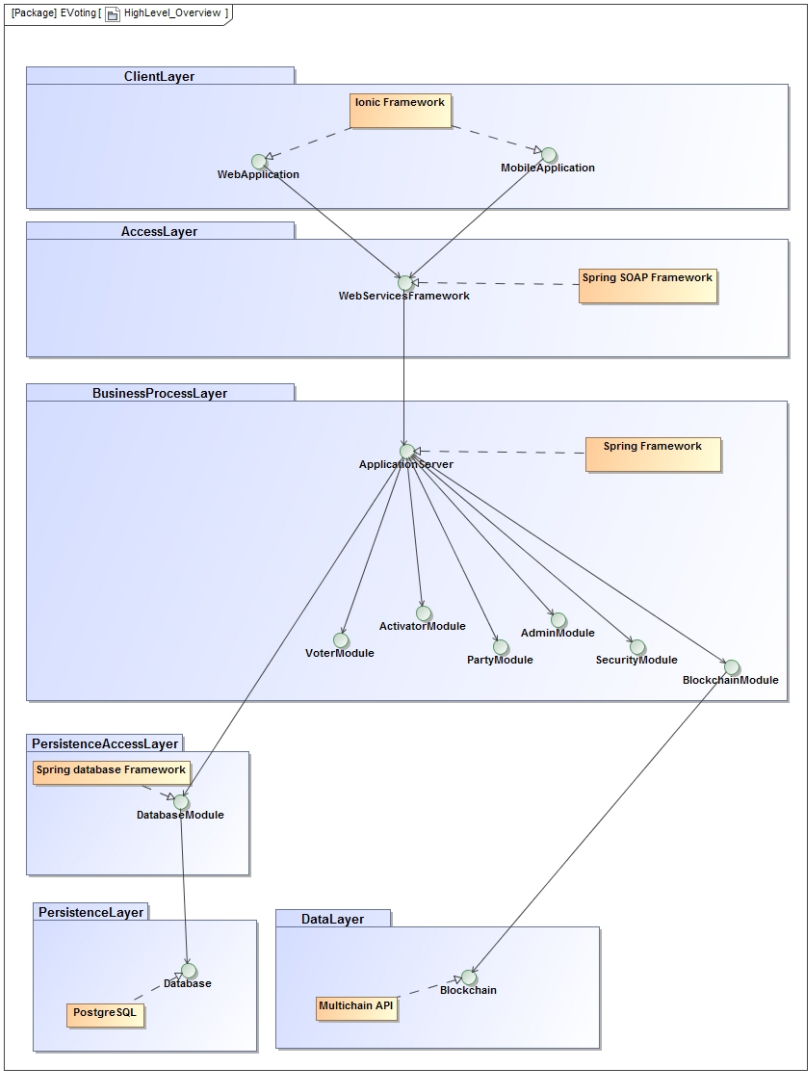

Figure 1: High Level System Diagram

1.1 Architectural Scope

1. Supply a persistence framework which supports integration with databases and database technologies, and provides secure access to persistent data
2. Support system flexibility in adding or removing existing modules without necessitating system shutdown
3. Provide a reporting infrastructure, which is: distributed, consistent, accurate, timely, and reliable
4. Supports system integration with front-end clients, the bare minimum of which should be a desktop web client and an Android mobile client
5. Cater for concurrent stateless access to services to at least one hundred clients
6. Provide functionality benchmarking infrastructure
7. Provide integration with authentication frameworks

1.2 Quality Requirements

1. **Convenience:** The system shall allow the voters to cast their votes quickly, in one session, and should not require many special skills or intimidate the voter.
2. **User-Interface:** The system shall provide an easy-to-use user-interface. Also, it shall not disadvantage any candidate while displaying the choices.
3. **Transparency:** Voters should be able to possess a general knowledge and understanding of the voting process.
4. **Accuracy:** The system shall record and count all the votes and shall do so correctly.
5. **Eligibility:** Only authorized voters, who are activated, should be able to vote.
6. **Uniqueness:** No voter should be able to vote more than once for the same poll.
7. **Auditability:** It should be possible to verify that all votes have been correctly accounted for in the final election tally, and there should be reliable and demonstrably authentic election records, in terms of physical, permanent audit trail, which should not reveal the users identity in any manner.
8. **Confirmation:** The voter shall be able to confirm clearly how his vote is being cast, and shall be given a chance to modify his vote before he commits it.
9. **No Over-voting:** The voter shall be prevented from choosing more than one party.
10. **Documentation and Assurance:** The design, implementation, and testing procedures must be well documented so that the voter-confidence in the election process is ensured.

11. **Cost-effectiveness:** Election systems should be affordable and efficient.

12. **Authenticity:** Ensure that the voter must identify himself (with respect to the registration database) to be entitled to vote.

13. **Anonymity:** Ensure that votes must not be associated with voter identity.

14. **System Integrity:** Ensure that the system cannot be re-configured during operation.

15. **Data Integrity:** Ensure that each vote is recorded as intended and cannot be tampered with in any manner, once recorded. Votes should not be modified, forged or deleted without detection.

16. **Privacy:** No one should be able to determine how any individual voted.

17. **Reliability:** Election systems should work robustly, without loss of any votes, even in the face of numerous failures, including failures of voting machines and total loss of network communication. The system shall be developed in a manner that ensures there is no malicious code or bugs.

18. **Availability:** Ensure that system is protected against accidental and malicious denial of service attacks.

19. **Simplicity:** The system shall be designed to be extremely simple, as complexity is the enemy of security.

20. **System Accountability:** Ensure that system operations are logged and audited.

21. **Authentication and Control:** Ensure that those operating and administering the system are authenticated and have strictly controlled functional access on the system.

22. **Distribution of Authority:** The administrative authority shall not rest with a single entity. The authority shall be distributed among multiple administrators, who are known not to collude among themselves.

1.3 Integration and Access Channel Requirements

- The aim of the Electronic Voting system is to allow users to participate in elections remotely, i.e. that they do not have to go into a voting station to cast their vote. The easiest way to achieve this is by providing both a web interface as well as an Android interface (seeing as Android is currently the most widely used operating system for mobiles in South Africa).

- The ultimate goal is that users will be able to easily download and use the Android application from the Google Play Store. The web interface will obviously be accessed from the users preferred browser so we will, whilst developing the system test it currently with multiple web browsers too to ensure that users are never disadvantaged according to their preferred browser.

- The Electronic Voting system requires the use of Blockchain to maintain elections, votes, voters and candidates.

  Blockchain Explained:

  > The Blockchain is a shared transparent ledger on which the entire Bitcoin network relies. All confirmed transactions are included in the block chain.
  > Once a transaction is entered in the Blockchain, it can never be erased or modified. Blockchain also allows us to ensure that voters cannot vote more times than allowed. So once a vote has been cast, a voter is sure that their vote was counted for the right candidate.

- The backbone of our Blockchain servers will be the Linux based MultiChain implementation which will allow us to create and manage nodes of the which we will use to implement our own local Blockchain.

- The backend of the system will be implemented using Java since it integrates easily with the Spring Framework.

- PostgreSQL will be used for the database as it provides several external authentication tools and it provides support for the use of JSON(this is also how MultiChain communicates with browsers in the MultiChain Explorer that provides a visual representation of the local blockchain), which will assist with the integration of the server-side communication.

- RESTful

  1. The front end will communicate with the backend using a RESTful webservice. REST (REpresentational State Transfer) is an architectural style, and an approach to communications that is often used in the development of Web services.
  2. REST uses a smaller message format than SOAP, which uses XML for all messages, which makes the message size much larger, and thus less efficient. This means REST provides better performance, as well as lowers costs over time. Moreover, there is no intensive processing required, thus its much faster than traditional SOAP.

1.4 Architectural Constraints

1. Blockchain
   (a) We're using Multichain to represent the Blockchain component of the Electronic Voting system. The Multchain allows us to manage transactions that happen when a user casts a vote, instantiation and management of all the types of nodes in the chain as well retrieving the balances of all the Party nodes when required.
   (b) The constraint which has been imposed with regards to the Multichain servers is the fact that there is only Linux support for the servers.
2. Ionic
   (a) The Ionic framework allows for develop hybrid mobile applications. The biggest drawback with Ionic since it is not native (Android or iOS) development is that it must support multiple platforms with a single codebase. In the greater scheme of things this is a minor issue since most web browsers are webkit based.
   (b) Hybrid development is best when there wont be intense functionality such as 3D graphics on the application/website itself. So the architectural constraint with regards to the front end is basically that interfaces of the system should be basically be used for simple message passing.
3. Postgre
   (a) Postgre databases have an inherent speed constraint because of its representation of database records as fully instantiated objects in code, this might slow down pre-processing of the system, more especially when it is compared to MySQL.
   (b) Does not support the entire ANSI SQL 92' standard, much less the ANSI SQL 99' standard
4. Android
   (a) The front end of the system will be a website component as well as an android component. The Android component will only be supported by Android 4.3 to more recent releases.

1.5 Architectural Patterns and Styles

Electronic Voting will make use of a combination between the Client-Server and Component-Based architecture styles. The system will be divided into a 'client side' and a 'server side', with a RESTful web service running on the server side as a communication medium between the two. Component-Based architecture style will be used on the server side, where each component contains everything necessary to execute only one desired aspect of the system. We will be using Component-Based architecture to separate all of the functionality for Voter, Activator, Party, Admin, Security, Blockchain and Database into their own respective modules. Reasons for Client-Server style:

1. Accessibility:
   (a) eVoting can be accessed form various platforms on a local network or over the Internet.
   (b) Our server can be accessed remotely by administrators or support teams.
2. Adaptability and Scalability:
   (a) Performance changes on the server can be easily made by upgrading the server.
   (b) New resources can be added to the server.
3. Pluggability:
   (a) We are using a RESTful web service on the server end.
   (b) Because of this, it allows us add any number of client platform types (eg iOS, Windows apps, or even other web applications) to integrate with our system if they support our communication medium.

Reasons for using a Component-Based architecture style:

1. Separation of concern (modularization):
   (a) Re-use of the business logic across the platform.
   (b) Multiple modules can be developed without affecting the other modules.
2. Developer specialisation and focus:
   (a) A developer can focus exclusively on one part of the business logic while another developer can focus on other business logic.
   (b) A developer that has more experience and better skill levels in a certain module, can give better input.
   (c) If there is a bug in a module, only that module needs attention.
3. Parallel development by separate teams:
   (a) Business logic developers can build the classes, while the UI developer(s) can involve in designing UI screens simultaneously.

(b) UI updates can be made without slowing down the business logic process.

4. Multiple view support:

   (a) Due to the separation of the model from the view, the user interface can display multiple views of the same data at the same time.

5. Testing:

   (a) Different business logic modules can be tested independently of other modules by using integration and unit testing.

   (b) Each module can expect that the other modules have a good level of correctness, thus expecting a pre-defined output.

1.6 Architectural Tactics and Strategies

1. **Authentication**
   All of our web service methods requires that the requesting object contains an ID number and a password. We will be authenticating the user to see if he/she is in fact registered, and if he/she has access to our methods' functionality and will be denied otherwise.

2. **Encryption**
   To ensure a safe and secure system, we will be encrypting the users' password on the client side before sending it to the server side. The encrypted passwords will directly be stored in our database as a string. As soon as a user is to be authenticated, the encrypted password strings will be matched to see if the password is the same.
   Encryption will also be used to encrypt the voting node's address and party node's address. This we will be decrypted on the server side.

3. **Database Indexing**
   All of our database tables will have primary keys, and also foreign keys linking to other tables.

4. **Database Integrity**
   To ensure database data fault prevention, each database base will have its respective audit table. Inside each audit table, there will be columns for: the type of modification (insert/update/delete); a date column when the modification was made; a column specifying who did the modification (Admin, Node1, Node2 etc); and the rest of the columns will all of the original table's columns. Its important to note that no one, not even the Administrator, will be able to delete rows from the audit table.

1.7 Reference Architectures and Frameworks

1. **Spring Boot** is one of the features of Spring Framework version 4.0 which gives us a way to improve Java applications quickly and simply through an embedded server this means we can expose components such as REST services independently. This framework will prove useful when deploying the Electronic Voting system.

2. **Spring Data** aims to significantly improve the implementation of data access layers by reducing the effort to the amount that's actually needed.

3. **Ionic (mobile app framework)** provides tools for developing hybrid mobile apps using web technologies such as HTML5 and CSS. It is an open-source framework, which helps to build hybrid apps using HTML5, and makes use of AngularJS.

4. **AngularJS** is a Javascript framework that be added to a HTML web page. AngularJS extends HTML attributes and binds data to HTML. It will be used to build highly interactive web pages for the Electronic Voting system.

5. **Bootstrap** is a HTML, CSS, and Javascript framework for developing responsive mobile-first web sites. We will use when writing the front-end of the web site and Android interface of the Electronic Voting system.

1.8 Technologies

This section will give a brief overview and description of the technologies that will be implemented in the system to achieve the desired outcome and allow not only successful completion of the project but doing it in the most effective and efficient manner.

1. **PostgreSQL:**

   PostgreSQL is an object-relational database management system (ORDBMS) with an emphasis on extensibility and standards-compliance. As a database server, its primary function is to store data securely, supporting best practices, and to allow for retrieval at the request of other software applications.

2. **Maven:**

   Maven is a project management tool, which provides developers with a complete build lifecycle framework. Development teams can automate the project's build infrastructure in almost no time as Maven uses a standard directory layout and a default build lifecycle.

   What this means is that the process for building and distributing a particular artifact (project) is clearly defined; it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.

3. **Blockchain:**

   MultiChain is a free software which people can download and run to design, deploy, and operate distributed ledgers (blockchains) that track ownership of digital tokens or assets.

4. **Bower:**

   Keeping track of all these packages and making sure they are up to date (or set to the specific versions you need) is tricky. It manages components that contain HTML, CSS, JavaScript, fonts or even image files. Bower doesnt concatenate or "minify" code or do anything else - it just installs the right versions of the packages you need and their dependencies. It's optimized for the front-end. If multiple packages depend on a package - jQuery for example - Bower will download jQuery just once. This is known as a flat dependency graph and it helps reduce page load.

5. **Grunt:**

   A task-based command line build tool for JavaScript projects. Here's the idea: when working on a JavaScript project, there are a bunch of things you'll want to do regularly. Like what, you ask? Well, like concatenating given files, running JSHint on your code, running tests, or minifying your scripts.

6. **Npm:**

   A NodeJS package manager. As its name would imply, you can use it to install node programs. Also, if you use it in development, it makes it easier to specify and link dependencies.

7. **Cordova:**

   Formerly called as Phone Gap is a platform to build Native Mobile Applications using HTML5, CSS and Java Script.

8. **LiquiBase:**

   An open source database-independent library for tracking, managing and applying database schema changes. It was started in 2006 to allow easier tracking of database changes, especially in an agile software development environment.