Consensus E-Voting System

Software Architecture and Design Assignment

Kieran Drewett

February 2026

Evaluation Report

Architecture

The voting system I have created (named Consensus) utilises a layered architecture with repository patterns sitting between the core services and underlying database singleton. In this app, I have four main layers:

- Controllers/EJS views - responsible for managing incoming requests to the server, and EJS is the markup language that renders the web page
- Services (i.e. VotingService, VoterService, ElectionService) - handle deeper logic that relate to managing core entities and performing essential business logic
- Repositories (BallotRepository, VoterRepository, etc.) - responsible for connecting with the SQLite database and performing SQL queries against typed objects
- Domain/Entities (Voter, Admin, Election) - include the underlying essential objects and basic primitive types

It trickles down from the very top, which I've demonstrated in my architecture diagram, it shows the initial incoming request from the web browser, going through the controllers, calling methods on services, fetch data from the database with repositories, transforming database objects into type-safe entities/objects, passed back to the controller for rendering via EJS.

Since errors mostly stay confined to their layer: debugging, development and testing is easy as each layer can be controlled independently with mock repositories for testing or functionality can be swapped out without requiring too much refactoring. Moreover, code flows cleanly throughout the application, so tracing error stacks is painless as each layer is clearly visible in the path name.

Next, for scalability, I made a conscious decision to implement a shared interface for a lot of the repositories like IVoterRepository, IElectionRepository and IBallotRepository. This meant I could easily write unit or E2E tests that "mock" the top-level repository, bypassing the connection to the database entirely, especially when you want your testing suite to be as fast as humanly possible.

On the topic of the database, I am using SQLite rather than a more managed solution like Postgres or MySQL/MariaDB. This is primarily because SQLite allowed rapid protyping and development in the app and I could easily inspect the data graphically as the database is written to a file. In a production app, this probably wouldn't cut it, so I'd likely transition to Postgres, since the two share a lot of syntax. Thanks to the layering architecture, I can also just swap out the underlying database implementation and it *should* in theory just work.

I think where this architecture shined the most though, would have to be for maintainability, I've worked on apps that try to use something like this, with clear separations of concerns, but they usually fall short on the "developer experience" because of mismanaged code, poor documentation or messy organisation.

Language/OOP features

TypeScript is a very powerful language built on top of JavaScript, adding support for a nice types system on top. All the features you know and love from other object oriented languages like C# are almost 100% transferrable. Personally, I am more comfortable using TypeScript rather than C#, and I think I made the right call given the scope and level of code I actually ended up writing.

To start, encapsulation is a key concept I adopted in the domain entities. For instance, Voter, stores a lot of its fields as private (in TypeScript, common convention is to underscore fields as well as applying the `private` keyword), the actual values then become accessible to consumers via getters/setters. This way read-only data is strictly read-only to consumers, and data can optionally be writable if we allow people to do so.

Next, polymorphism shows up a lot in the voting strategies implementations. The `VotingService` stores a agnostic `IVotingStrategy` interface. All of the actual top-level voting strategy logic will implement the methods/fields in this interface, keeping it consistent between them all. Because of this, adding the first strategies `FPTPStrategy` meant I could easily define a clean and understandable API that could be implemented for the other strategies.

Finally, dependency inversion is a common pattern used across the service layer. Since services act as the brains of the whole app, they do a lot of heavy lifting and act as the intermediary between web browser <-> server <-> database. For instance, the `VotingService` utilises the agnostic versions of repositories, i.e. IBallotRepository, IVoterEligibilityRepository, etc. rather than the concrete repository classes.

Design Patterns

In the app, I created a factory method called `BallotFactory`, which serves to solve the problem of differing election types requiring different ballot structures. BallotFactory.createBallot() takes an ElectionType and either a candidateID or a preferences array, validates the input matches what that election type expects and returns the right Ballot.

Next, I built an adapter called `AnonymousBallotAdapter`, which enforces the privacy requirement within the brief that ballots cannot be traceable back to the voter. This adapter takes in a `VoteInput` object, which contains information about the election being voted in, the voter themselves and their choices, via the `adapt()` method, which when splits the input into two separate unlinked objects: `Ballot` and `VoteConfirmation`.

As mentioned briefly, I have also created a strategy pattern called `IVotingStrategy`, which serves to handle the different implementations of voting systems for elections. The interface expoes two methods `validateBallot()` and `calculateResults()`.

Software Reuse and Trade-offs

The use of interfaces and patterns made adding new features relatively quick. The `AVStrategy` was implemented last and only required a small amount of new code.

The only real downside with a layered architecture approach is “getting used to it”, to understand what happens when a vote is cast, a reader has to follow the flow through controllers, services, factories, adapters, strategies, and observers.

The area I wish I had spent more time on is candidates: my implementation treated them as metadata rather than full users with their own dashboards/login, which I realise did not fully meet the brief. I also didn’t anticipate the effort required to keep diagrams and documentation, like the wireframes and class diagram, in sync as the design of the system evolved.

Conclusion

I think overall, the layered architecture and adoption of those key patterns mentioned worked well together, and it resulted in a really clean and accessible project. The system behaves as expected and all tests pass.

AI Transparency Declaration

AITS Level: 2 (AI Assisted)

AI was used in the following limited ways during this assessment:

1. Seed data generation: AI generated the sample data for voters, candidates, elections, and ballots in `src/seed.ts` so the application would have realistic demo data to work with. **The code in `src/seed.ts` is my own work, the data like names, emails, passwords, addresses etc is NOT.**
2. Test data generation: AI generated mock objects and test fixtures used in unit tests and end-to-end tests. **The test logic and assertions themselves are my own.**

3. Design advice: I used AI to discuss architectural approaches for e-voting systems and to get suggestions on which design patterns might fit the requirements.

All application code, architecture implementation, pattern code, business logic, UI templates, test logic, and documentation was written by me.