DAT250 Project Report: A Full-Stack Polling Application

Group 21: Anne Madelen Fjeldstad, Anna Ø. Folmann, Georg Risøy
Magnus Sletten

28 November 2025

1 Introduction

The project aims to design and implement a prototype FeedApp, which is a polling application that allows users to create , vote on and view polls through a secure and scalable web platform. It is built as a full-stack system with a RESTful API and an interactive web interface, demonstrating modern web development practices such as authentication, data persistence, caching and asynchronous messaging.

The core domain consists of Users, Polls, and Votes, managed through a domain model and business logic layer. The backend is implemented using Spring Web MVC with token-based authentication and role-based authorization. Data is stored in an H2 relational database via JPA, while Redis serves as a caching layer for frequently accessed data like vote counts. This hybrid storage approach balances performance and consistency. The frontend is built as a React Single Page Application that communicates with the backend over secure HTTPS through Caddy. Each voting action also triggers a message published to a RabbitMQ queue, enabling real-time event distribution and future scalability.

As part of the featured technology study, the project also investigates reverse proxies and load balancing by experimentally comparing Nginx with the newer Caddy server. Using controlled CPU-bound workloads and two load-balancing strategies (least connections and random), the experiment tests the hypothesis that Caddy can match Nginx in terms of stability and performance. The results show no par with Nginx in this setup and can be considered a viable alternative in modern full-stack architecture.

Overall, the project showcases the integration of backend services, secure APIs, interactive frontend design, caching mechanisms, and message-based communication into a cohesive prototype. The prototype with the experimental study offer both a practical and analytical perspective on modern full-stack development. It showcases how thoughtful architectural choices and empirical evaluation can support the creation of robust, efficient and scalable web systems.

The full source code for the FeedApp prototype is available at the following repository: `https://github.com/MagnusSletten/dat250-pollapp`.

2 Feed App Design

2.1 Use Cases

The FeedApp system is designed to support the creation, voting and viewing of online polls in a secure and user-friendly environment. The application has three primary roles including Guest, Registered User and Administrator. Each role interacts with the system in different ways depending on varying levels of access and responsibility as illustrated in the UML diagram below.

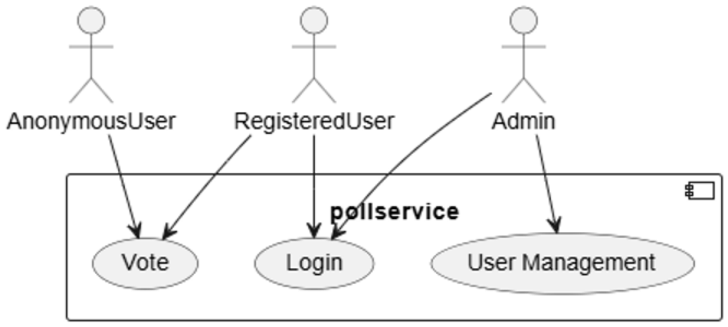

Figure 1: Use case diagram showing system interactions for Anonymous Users, Registered Users, and Admins.

Guest users represent individuals who access the platform without authentication. Their interactions are limited to ensure data integrity while still allowing engagement with the system. Guests can view publicly available polls and vote. They are also given the option to register an account in order to gain access to all features. Registered users are authenticated participants who login with username and password. They can create new polls by defining questions and possible response options, cast votes in available polls and view results. Each vote is tied to the user's identity. Registered users have control over the content they created, allowing them to edit or delete their own polls.

The Administrator role provides oversight and the ability to moderate polls. This role is important for maintaining a secure platform and serves as a safeguard against misuse. In addition to user-driven interactions, several system-level use cases operate in the background to support reliability and performance.

Distinct user roles within the FeedApp serve both technical and conceptual purposes. From a security perspective, role separation upholds the principle of least privilege, meaning each user has access to the resources and operations necessary for their role. For example, guest users cannot modify data, which prevents unauthorized actions, while registered users are restricted from editing others' polls to maintain ownership boundaries. Administrators, on the contrary, are granted permissions to address issues that could affect the systems standards.

From a design standpoint, different roles contribute to scalability and maintainability. They make it easier to introduce new features without restructuring the security framework. For instance, future roles such as moderator or analyst could be added by extending existing authorization rules rather than rewriting them. Additionally, this structure aligns with real-world application design patterns where role-based access control (RBAC) is essential for security and efficiency[9].

These use cases show how users and system components can deliver a secure polling experience. The roles provide a foundation for the design and illustrate the balance between accessibility and data integrity that is important for the FeedApp architecture.

2.2 Domain

The FeedApp Domain is centered around User, Poll and Vote. These three entities are supported by related concepts such as Option and Role. They define the logical structure of the system and how data interacts within it.

A User represents an individual using the application, either as a guest, registered participant or administrator. A Poll encapsulates a question created by a user along with a set of options that represent possible answers. Polls are timestamped and linked to their creator, ensuring clear ownership and traceability. Vote records a user's selection for a given poll option. Each vote is associated with both a user and a poll. The system ensures that a user can only vote once per poll, preserving data integrity and fairness.

2.3 Architecture and Information Flow

Introducing Caddy to the polling architecture allows for easy HTTPS integration where the information to and from the user is more secure. Caddy becomes the first point of contact with our polling service through HTTPS then takes care of routing traffic to different internal components through HTTP on the server. Each of our main components are containerized using Docker and deployed using Docker Compose.

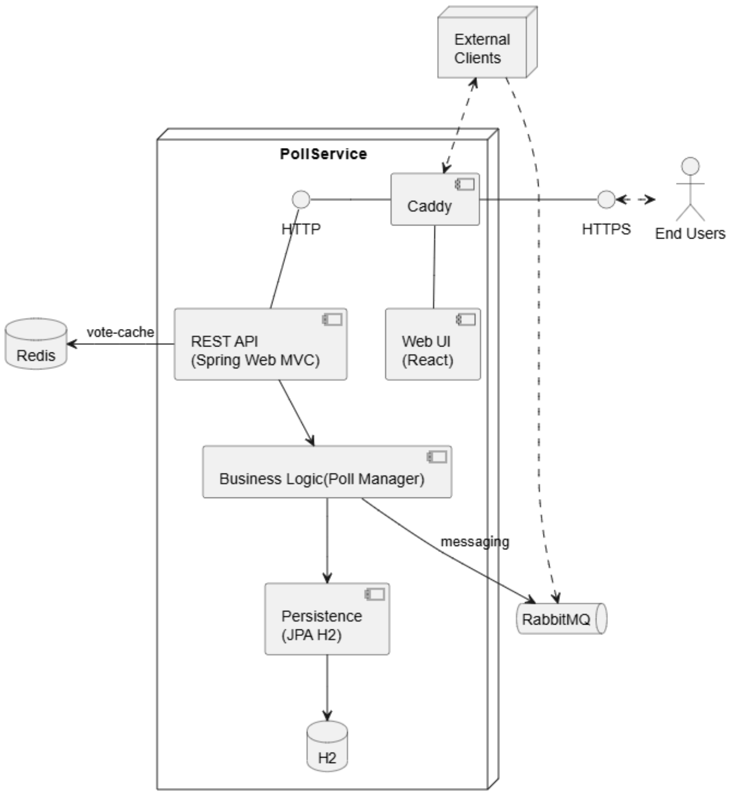

Figure 2: The diagram shows how user requests flow through Caddy into the PollService and its connected components

The first container in the information flow is then Caddy and this container also includes the minimal frontend files which are served directly here. Any interaction with other backend elements happens through our main backend container: Poll-Service. The poll service container which includes the spring framework includes all modelling, business logic and also handles sending information to and from other components like Redis, RabbitMQ and our persistence component of choice: the H2 database.

As part of the spring framework we use Jackson deserialisation to allow users to send information which is used to construct java objects, for instance a new User or a new Vote object. This works by mapping JSON-information to Java classes through classes with empty constructors and methods to assign the class variables. This happens within the Rest-controller annotated classes which define HTTP endpoints of which there are four different ones each handling its own domain: Votes, Polls, Users and one for CSRF tokens.

The constructed java objects can then be used within business logic in our Poll-

Manager class which handles our use cases, for instance what happens with votes from unregistered users vs registered users.

Prior to deserialisation, several layers of security provided by the Spring framework are applied. At the browser level, CORS ensures that only specific approved origins—corresponding to our frontend can access the backend. This prevents scripts from unauthorized sites from reading information[2]. CSRF tokens add another layer of protection by requiring a valid token in the request headers for any operation that modifies important data, safeguarding users against malicious websites that might attempt to send requests using automatically forwarded cookies such as session IDs. Although we provide an endpoint for retrieving CSRF tokens, browser restrictions prevent external sites from accessing and injecting them, maintaining this layer of defence. In addition to browser-level protections, role-based authentication ensures that users can modify only their own data by validating the session ID associated with each request, while administrative actions affecting all users are restricted exclusively to admin accounts [3].

After deserialisation and business logic information may be stored within our relational H2 database handled via Spring Repositories. These store information about the classes assisted by annotations within the classes, some noting relationships. For example, in memory a User object may contain direct references to its associated Vote objects. In the database this relationship is represented differently: the User class defines a one-to-many relationship with Vote, meaning each Vote row contains a foreign key referencing the User who created it. So while the objects in memory are linked through references, the relational database links them through foreign keys managed automatically by JPA.

In order to speed retrieval of information which requires costly database queries we also have a Redis non-relational database acting as a cache for vote-counts. One poll being fetched multiple times is then a lot more efficient. To make this cache consistent with our database the votecounts are removed from the cache if new votes are counted, then re-cached upon next retrieval.

To enable loosely coupled expansion of functionality, the system includes a messaging service, RabbitMQ; that publishes events whenever new votes are added to the database. Other systems can subscribe to these events and process them independently of the core polling application, allowing for external integrations or asynchronous workflows without modifying existing components. RabbitMQ uses a different application-level protocol than HTTP, the AMQP protocol. It allows for creating patterns of messaging routing through exchanges such as topic, direct and fanout. With a topic-exchange different topics can be registered by publishers and consumers of events can listen to events from specific topics or use wildcard patterns targeting patterns of topics [1].

We use the latter actively in this application where there is one topic-exchange called poll but with sub routes associated with each individual poll.

3 Featured Technology

3.1 Technology Domain Genealogy and Problem Domain Habitat

In the initial design of the FeedApp, we selected NGINX as the primary reverse proxy and public entry point into the system. Its responsibilities included terminating HTTPS connections, routing incoming traffic to backend services and serving the static assets for the React frontned. This choice is aligned with common industry practice, since NGINX is widely deployed, highly stable and known for its strong performance under heavy load [6].

As the project progressed, we evaluated whether a more recent and developer-friendly alternative could simplify configuration and reduce operational overhead. This led us to experiment with Caddy, which fulfills a similar architectural role but has a different design philosophy. Both proxies belong to the same lineage of high-performance, event-driven web servers, yet caddy represents a newer branch that emphasizes automation, usability, and secure-by-default settings [5]. Understanding how these technologies relate helped frame our motivation to compare them in a controlled performance study.

NGINX was originally developed to overcome concurrency limitations in older web servers such as Apache [7]. its event-driven, asynchronous architecture allows it to manage large numbers of concurrent connections with low resource consumption, making it well suited for systems like FeedApp. It offers flexible routing, static file hosting, standards-compliant HTTPS support and seamless integration with containerized environments.

However, this robustness comes with certain trade-offs. The configuration syntax can be lengthy, TLS handling typically requires manual setup or external tooling and small changes often involve coordinating multiple configuration files or reloading the service [8]. for a fast-evolving prototype, these factors introduce friction that motivated us to investigate a more streamlined alternative.

Caddy was introduced more than a decade after NGINX and builds on similar architectural principles, but with a modern implementation focused on simplicity and automation [5]. Its most distinctive feature is automatic HTTPS, where certificate issuance, renewal and management occur without manual intervention [4]. Combined with a concise and readable configuration language (the Caddyfile), secure defaults and a Go-based codebase, Caddy reduces both setup time and the risk for incorrect setup.

These qualities made caddy an attractive candidate for our architecture. By lowering the operational complexity of the reverse-proxy configuration, it allowed us to allocate more time to other parts of the feedapp prototype, rather than administrative details.

3.2 Hypothesis Formulation and Experiment Design

This study investigates whether Caddy can match the load-balancing performance of the more established NGINX when exposed to strictly CPU-bound workloads. Although NGINX has a long operational history and extensive community support, Caddy represents a newer generation of web servers. The experiment therefore examines whether the differences in maturity and philosophy translate into measurable performance differences.

Hypothesis 1: Load-Balancing Performance

The hypotheses are defined as follows:

- **Null hypothesis (H_0):** The mean load-balancing performance of Caddy is equal to that of NGINX. Mathematically, $H_0 : \mu_{Caddy} = \mu_{NGINX}$.
- **Alternative hypothesis (H_1):** The mean load-balancing performance of Caddy differs from that of NGINX. Mathematically, $H_1 : \mu_{Caddy} \neq \mu_{NGINX}$.

The experiment compares how each proxy distributes CPU-heavy tasks across multiple backend workers using two load-balancing strategies: **Least Connections (LC)** and **Random (RR)**. The goal is to determine whether Caddy matches the stability and throughput characteristics of NGINX when executing tasks of varying computational weight.

3.2.1 Workload

Each incoming request triggers a synthetic CPU-bound task consisting of a fixes number of hash operations as shown in table 1.

Table 1: Workload Model

Job	Description	Relative Cost (Iterations)
A	Heavy CPU workload	80M
B	Medium workload	40M
C	Small workload	20M

These job classes allow us to assess how well each proxy handles mixed workloads where task durations differ significantly.

3.2.2 Load Generation Method

Two types of load tests were performed:

1. **Continuous load**
    - Fixed concurrency of 5 requests

    - Ran for 60 seconds per policy (LC and RR)
    - We record the median and 95th-percentile processing times

2. **Batch stress test**
    - Repeated bursts of 50 parallel requests
    - We measure total completion time per batch
    - Captures variance and stability under intermittent high load

All tests were executed against identical backend workers to isolate the effect of the reverse proxy.

3.2.3 Test Environment

Both proxies and all backend workers ran inside Docker containers. The network path was local and negligible, and backend workers were implemented using identical Gunicorn-based services configured to perform CPU-bound operations. Docker resource limits ensures consistent backend performance across runs. No caching, buffing, or auxiliary optimization was enabled in either proxy, which ensured that observed differences could be attributed to the load-balancing implementations rather than external forces.

3.2.4 Measured Quantities

For each job type (A, B, C) and each balancing policy (LC, RR):

- Median processing time
- 95th-percentile processing time

For continuous load tests:

- Total completed requests

For batch tests:

- Distribution of batch completion time

3.3 Experiment Results and Demonstrator Study

3.3.1 Least Connections vs Random Load Balancing

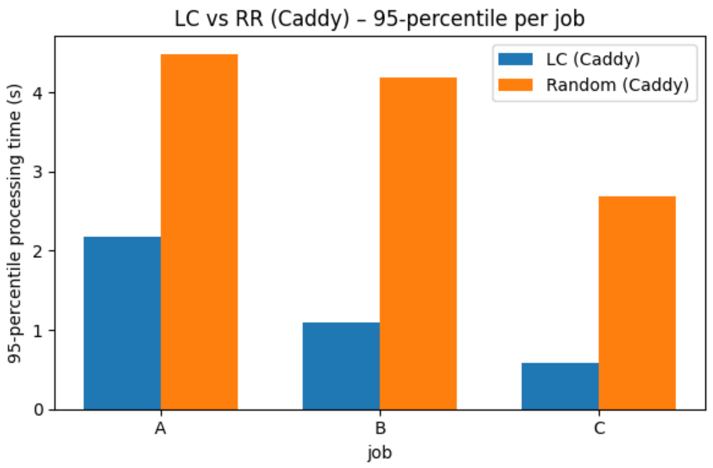

Figure 3: 95-percentile processing time per job for LC vs RR using Caddy. Data and scripts from [10].

Under the **Least Connections** strategy, both proxies behave as expected. The medium workload (Job B) consistently completed in about half the time of the heavy workload (Job A), while the light workload (Job C) completed in roughly half the time of Job B. This proportional relationship indicates that LC distributes requests effectively and prevents small tasks from being delayed behind longer ones.

In contrast, the **Random** Strategy produced noticeably less efficient behavior. Although Job C has only a quarter of the computational cost of Job A, it completed at roughly 60% of A´s processing time. This is an inherent limitation of random assignment: small requests often queue behind heavy ones, inflating their completion time. LC avoids this issue by always routing new requests to the least-loaded worker.

3.3.2 Caddy vs NGINX

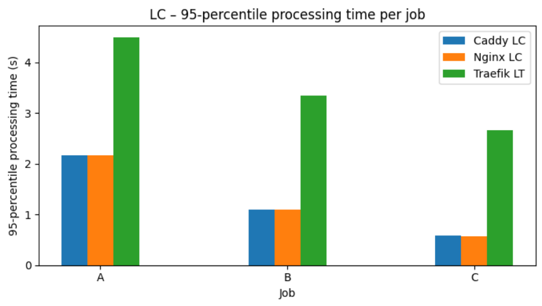

Figure 4: 95th-percentile processing time per job using LC in Caddy and Nginx. Both proxies show similar tail-latency performance. Data: [10].

Across all job types, load-balancing strategies and test methods, no statistically significant performance difference was observed between Caddy and NGINX. Their median latencies, tail latencies, and throughput were nearly identical, with neither proxy showing a consistent advantage.

Note that Traefik performs significantly worse in this benchmark, which is suspected to be due to a bug in its least-time algorithm. Small changes to the benchmark itself can circumvent this bug and would yield very different results. This bug is currently under investigation by the Traefik contributors.

3.3.3 Batch Stess Test

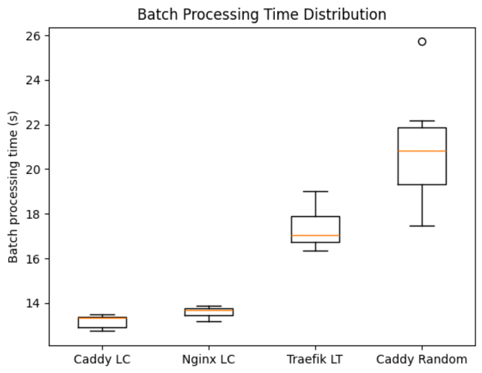

Figure 5: Batch processing time distribution for Caddy LC, Nginx LC, and Caddy RR. Caddy RR shows higher variance and longer batch times. Data: [10].

The batch tests revealed similar trends. Both proxies demonstrated low variance in batch completion times under LC, while RR produced much greater spread. Caddy performed slightly better than NGINX in some runs, but the difference was minor and not statistically meaningful. Overall, the data shows that the choice of load-balancing strategy has far greater impact than the choice of proxy.

3.3.4 Conclusion of Experiment

Based on all measurements, we do not reject the null hypothesis. Caddy's load-balancing performance is statistically equivalent to NGINX under CPU-bound workloads in this controlled environment. Both proxies handle heterogeneous tasks consistently, and neither displays a systematic advantage. Within the scope of this demonstration experiment, Caddy is a fully viable alternative to NGINX from a performance perspective.

4 Prototype details

The prototype was implemented to verify that the chosen architecture can be realized in code and that its components behave correctly when integrated. Rather than recreating the full system, the prototype focuses on the execution of core logic, backend–frontend interaction, state management, and security. The goal is to build a complete and working example that runs through the entire stack.

4.0.1 Backend Implementation

The backend is built using Spring Boot, with Spring Web MVC providing the controller layer and Spring Security handling authentication and authorization. Most application logic is isolated in dedicated service classes to keep controllers thin and maintainable.

4.0.2 REST API Structure

Each domain area, including Users, Polls, Votes, and CSRF tokens, is handled by a dedicated controller class. Controllers expose clear, resource-oriented endpoints:

```
POST /api/users            registers a new user
POST /api/login            initiates a session
GET  /api/polls            retrieves all polls
POST /api/polls/{id}/vote  submits a vote
GET  /api/csrf             provides CSRF tokens to the frontend
```

Controllers perform minimal work such as parameter validation, session checks, and the conversion between HTTP requests and internal objects. All domain decisions (e.g., who can vote, who can edit a poll) are delegated to service-layer components.

4.0.3 Domain Logic and Services

The central application logic is implemented in service classes such as `PollManager`. These classes encode the behavior rules of the system.
This mapping defines the relationships between a vote, the user who cast it, the poll it belongs to, and the selected option.

4.0.4 Persistence and Data Modeling

Entity relationships are defined using JPA annotations. For instance, a `Poll` stores its options as a `@OneToMany` relation, while a `Vote` references both the `User` and the `Poll` through `@ManyToOne` mappings. The persistence layer automatically manages cascading operations and foreign keys, providing predictable SQL behavior through the H2 database used in the prototype.

An example of the entity mapping for the `Vote` domain class is shown below:

```
public class Vote {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long voteId;

    Instant publishedAt = Instant.now();

    @JsonIdentityReference(alwaysAsId = true)
    @ManyToOne
    User voter;

    @JsonIdentityReference(alwaysAsId = true)
    @ManyToOne
    Poll poll;

    @ManyToOne
    @JsonIdentityReference(alwaysAsId = true)
    VoteOption votesOn;
}
```

4.0.5 Caching Logic

The prototype uses Redis to cache poll result summaries. This reduces repeated summation over `Vote` records, which would otherwise become expensive under load.
Caching is managed through a utility function in the service layer:

```
@Transactional
public Map<Integer, Integer> getVoteResults(Integer pollID) {
    Poll poll = getPoll(pollID).get();
    if (voteCache.isCached(poll)) {
        System.out.println("Poll is already cached. returning
        ↪  from cache");
        return voteCache.getVoteResults(poll);
    } else {
        voteCache.setVotes(poll);
        System.out.println("Poll not cached, querying DB and
        ↪  caching result");
        return pollRepo.findById(poll.getId())
                       .get()
                       .countVotesByPresentationOrder();
    }
}
```

Whenever a new vote is submitted, the cache is explicitly invalidated to ensure consistency.

4.0.6 Vote Submission Flow

Votes are submitted through the `VoteController`:

```
@PostMapping("/{pollID}/votes")
public ResponseEntity<VoteDTO> addVote(@RequestBody VoteDTO
↪  voteRequest) throws Exception {
    try {
        pollManager.addVote(voteRequest);
        return ResponseEntity.ok(voteRequest);
    } catch (Exception e) {
        return ResponseEntity.internalServerError().build();
    }
}
```

The controller delegates all vote-handling logic to `PollManager`:

```
@Transactional
public Vote addVote(VoteDTO request) {
    if (request.hasUsername()) {
        return addVoteWithUser(request);
    } else {
        return addVoteAnonymous(request);
    }
}
```

When an authenticated user submits a vote, the service verifies that the poll is open, invalidates any cached results, and either creates a new vote or updates the user's previous vote. While the full implementation is extensive, the essential steps are:

- Validate poll time window
- Invalidate stale cache entries
- Create or update the user's vote
- Update poll and user associations
- Publish the vote event to RabbitMQ

This ensures correct behavior for repeated voting, closed polls, and real-time result updates without embedding overly verbose code into the document.

4.0.7 Event Publishing

Every submitted vote triggers an event sent to RabbitMQ:

```
public void sendVote(Integer pollId, VoteDTO vote) throws
↪  IOException, TimeoutException {
    try (Connection connection = factory.newConnection();
         Channel channel = connection.createChannel()) {

        channel.exchangeDeclare(EXCHANGE_NAME,
        ↪  BuiltinExchangeType.TOPIC, true);

        String routingKey = "poll." + pollId + ".vote.cast";
        String message    = vote.toJson();
        channel.basicPublish(
                EXCHANGE_NAME,
                routingKey,
                null,
                message.getBytes(StandardCharsets.UTF_8)
        );

        System.out.println("[PollBroker] sent '" + routingKey
        ↪  + "':'" + message + "'");
    }
}
```

Although the prototype does not consume these events, the mechanism validates that asynchronous extensions such as analytics or dashboards can easily be added.

4.0.8 Frontend Implementation

The frontend is implemented as a React SPA, served as static files by Caddy. The prototype focuses on browser-based authentication, CSRF handling, and session persistence.

4.0.9 Request Flow and Security

When the SPA loads, it retrieves a CSRF token:

```
const fetchCsrf = async () => {
  const res = await fetch("/api/csrf", { credentials: "include" });
  setCsrfToken(res.headers.get("X-CSRF-TOKEN"));
};
```

This token is included in all state-changing requests:

```
await fetch("/api/polls/3/vote", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "X-CSRF-TOKEN": csrfToken,
  },
  body: JSON.stringify({ optionId }),
  credentials: "include",
});
```

4.0.10 UI Structure

The frontend contains simple views:

- Login and registration forms
- Poll list and poll detail pages
- A voting interface with available options
- A minimal admin panel

4.0.11 Deployment and Runtime

The system is deployed using Docker Compose, with each service (frontend, backend, Caddy, Redis, RabbitMQ) running in its own container.

4.0.12 Prototype Limitations

The prototype deliberately excludes non-essential features:

- Minimal UI
- No RabbitMQ consumer
- Limited validation
- Basic admin functionality

These exclusions help focus the prototype on verifying integration, backend logic, and security mechanisms.

5 Conclusion

The project successfully demonstrates the design, implementation, and evaluation of a secure and scalable polling application. Through the FeedApp prototype, we validated the architectural choices, integrating a full-stack backend, interactive frontend, caching, and asynchronous messaging into a cohesive system. The prototype confirmed that the architecture effectively supports core user workflows, maintains data integrity and enables real-time event distribution.

The experimental comparison between NGINX and Caddy highlighted important considerations for choosing reverse proxies. While both performed similarly under CPU-bound workloads, Caddy offered a simpler configuration model and automated HTTPS, demonstrating how modern tooling can reduce operational complexity without compromising performance.

Security and reliability were central considerations throughout the design. Role-based access control, CSRF protection, and consistent data management ensured that user actions were constrained, while caching and messaging improved performance. Containerized deployment with Docker Compose showed that the system could be deployed and maintained in a realistic environment.

Overall, the project illustrates the benefits of combining architectural design with empirical evaluation. It showcases how modern development practices can be applied to build robust web systems. As a practical prototype and case study, it highlights the challenges and solutions of full-stack development, proving that thoughtful design makes a difference.

References

[1] Amqp 0-9-1 model explained. `https://www.rabbitmq.com/tutorials/amqp-concepts`, 2025. Accessed: 22 November 2025.

[2] Spring framework documentation: Cors support in spring web mvc. `https://docs.spring.io/spring-framework/reference/web/webmvc-cors.html`, 2025. Accessed: 25 November 2025.

[3] Spring security reference: Protection against cross-site request forgery (csrf) attacks. `https://docs.spring.io/spring-security/reference/servlet/exploits/csrf.html`, 2025. Accessed: 25 November 2025.

[4] Caddy contributors. Caddy web server features: Automatic https and more. caddyserver.com. Accessed: 27 November 2025.

[5] Wikipedia contributors. Caddy (web server). `https://en.wikipedia.org/wiki/Caddy_(web_server)`. Accessed: 27 November 2025.

[6] Wikipedia contributors. Nginx. `https://en.wikipedia.org/wiki/Nginx`. Accessed: 27 November 2025.

[7] Owen Garrett. Inside nginx: How we designed for performance scale. NGINX blog, June 2015. Accessed: 27 November 2025.

[8] TechAI Insights. Caddy vs nginx: A complete comparison of web servers. TechAIInsights.in, August 2025. Accessed: 27 November 2025.

[9] Gregg Lindemulder and Matt Kosinski. What is role-based access control (rbac)? `https://www.ibm.com/think/topics/rbac`, 2025. Accessed: 22 November 2025.

[10] Magnus Sletten. Load balancer analysis – experiment repository. `https://github.com/MagnusSletten/load-balancer-analysis`, 2025. Accessed: 27 November 2025.