

DAT250 FeedApp Report

Erlend André Høntorp, Jonas Heen Opsahl and Thomas Tolo Jensen

November 12, 2025

Abstract

FeedApp is a website where people can vote in live polls and discuss them in comment threads. The goal is to let a user create a poll (private or public), either open or invite participants (for private polls), and receive votes and feedback that update instantly across all clients. The system consists of a Spring Boot (Java) backend and a React (TypeScript) frontend, packaged with Docker for repeatable local deployment. Data is stored in an in-memory H2 database during development, Redis is used for caching, and Kafka works like a messaging service that separates data storage from live updates and background processing. WebSocket notifications share vote and comment changes to the UI without page reloads. The architecture follows an event-driven style, this allows the system to stay organized and makes it easy to grow in the future. The main results are a fast and responsive user interface, an easy-to-expand modular design, and a container setup that can quickly switch to full production services with little effort. The main things we have learned are about keeping data in sync between REST and WebSocket, handling cache updates correctly, and seeing how Kafka helps improve real-time features.

Contents

1	Introduction	3
1.1	Project Overview	3
1.2	Technology Stack Summary	3
1.3	Project Outcome	3
1.4	Report Organization	4
2	Design	4
2.1	Use Cases	4
2.2	Domain Model	4
2.3	Architecture (including applied technologies)	5
2.4	Summary	5
3	Technology Assessment	5
3.1	Descriptive Modeling	7
3.2	Experiment Design	7
3.3	Experiment Evaluation	7
4	Prototype Implementation	7
4.1	Setup and Deployment	7
4.2	Backend Implementation	8
4.2.1	Authentication and Security	8
4.2.2	Core APIs and Real-Time Updates	9
4.3	Frontend Implementation	9
4.4	Extensibility and Future Work	10
5	Conclusions	10

1 Introduction

1.1 Project Overview

FeedApp makes it easy to gather quick feedback and interaction between users. They can create polls, vote, and discuss through comments. The app solves the problem of gathering input in real time, which often means manual counting or refreshing page for each new vote or comment. With live server updates, everyone instantly sees new votes and comments as they appear, creating a smooth and interactive experience.

The system is divided into a backend and a frontend. The backend provides REST endpoints for creating, reading, updating, and deleting polls, options, votes, and comments, and it also sends out events. The frontend shows the current state, listens to WebSocket updates, and keeps the user interface in sync with the server. This app is built to test key user interactions, explore event-driven design, and demonstrate a system that can easily scale as more users join.

1.2 Technology Stack Summary

The backend is built with Spring Boot, Spring Security, and Java, providing REST APIs, data validation, and integration with messaging and caching systems. The frontend is made with React and TypeScript, focusing on reusable components and real-time state updates. An H2 in-memory database is used during development to speed up testing and iteration. Redis works as a cache and a simple coordination layer for frequently accessed data and precomputed views. Kafka acts as the messaging backbone, sending and receiving domain events to separate data writes from notifications and later analytics. Docker and Docker Compose manage all services for a consistent and repeatable setup. This technology stack was chosen for its scalability (through Kafka and horizontal growth), modularity (clear separation between frontend, backend, cache, and messaging), and maintainability (familiar frameworks and container-based tools).

1.3 Project Outcome

The prototype supports creating and managing polls and options, voting with real-time result updates, and a threaded comment system with replies and edits. WebSocket broadcasting makes sure that votes and comments show up instantly for all users without needing to refresh. The event-driven setup makes it easy to add new features or services later, while Redis helps speed up repeated data access. The whole project is containerized for simple local development and easy migration to production databases and clusters.

1.4 Report Organization

This report is organized as follows: Section 2 presents the overall design and architecture of FeedApp, including key use cases, the domain model, and system components. Section 3 focuses on the featured technology experiment with Kafka, covering its background, hypothesis, and evaluation results, along with supporting technologies such as Redis and Spring Boot with Spring Security. Section 4 describes the prototype implementation, including backend and frontend details, deployment setup, and integration of Kafka-based event handling. Finally, Section 5 summarizes what was learned, discusses the results from Kafka experiment, and possible future improvements.

2 Design

Around 5 pages about functional aspects of the FeedApp application.

2.1 Use Cases

Describe:

- Main functionality from the user's perspective,
- Actors: Guest User, Registered User, System,
- Core use cases: Register/Login, Create Poll, Vote, Comment and Managing Polls,
- Brief explanation of real-time updates and visibility (public/private polls)

Optional:

- Use Case Diagram (Figure: Use case overview)

2.2 Domain Model

Describe:

- Core domain entities (User, Poll, Vote, VoteOption, Comment(?)),
- Relationships between them (one-to-many, many-to-one),
- Persistence through JPA/Hibernate

Optional:

- UML Class Diagram (Figure: Domain Model Overview)
Show main entities, attributes, and associations

2.3 Architecture (including applied technologies)

Describe:

- High-level architecture (frontend–backend–infrastructure),
- Backend layers: Controller, Service, Repository,
- Key technologies: Spring Boot, H2, Redis, Kafka, WebSocket, JWT,
- Frontend technologies: React, TypeScript, REST API communication,
- Interaction between components (data flow from frontend to backend),

Optional:

- Architecture Diagram (Figure: System Architecture)
Show containers/components (Frontend, Backend, Redis, Kafka, DB, WebSocket)

2.4 Summary

Summarize:

- How the design supports scalability, modularity, and real-time updates,
- The rationale behind chosen technologies,
- How the design connects to later implementation and deployment steps

3 Technology Assessment

(Introduce in sufficient depth the key concepts and architecture of the chosen software technology. As part of this, you may consider using a running example to introduce the technology.)

At its heart, Apache Kafka is a distributed event streaming platform. Think of it as a highly scalable, fault-tolerant, and durable "log file" that different parts of an application can write to and read from simultaneously. The FeedApp uses these core concepts to decouple tasks. When a user votes, the system doesn't immediately try to update every other user's screen. Instead it publishes an "event" to Kafka, and other services (like the WebSocket broadcaster) react to that event.

Event:

An event is the most basic unit of data, which represents the fact that something happened. In the FeedApp implementation, this event is represented as a map of string and object that gets serialized into JSON. For example, when a poll is created, the PollEventListener creates an event with the pollId, question and event type. The consumer listens for vote change events, which it receives as a map containing the pollId and optionOrder.

Producer:

A producer is a client application that writes (publishes) events to a Kafka topic. The ProducerService in the FeedApp implementation is a classic producer implementation. It uses Spring's Kafka Template to send events to a specific topic. This service is used after PollEventListener to send a message after a new poll is successfully saved to the database.

Topic:

Consumer:

This part and other parts of the report probably need to refer to figures. Figure 1 from [1] just illustrates how figure can be included in the report.

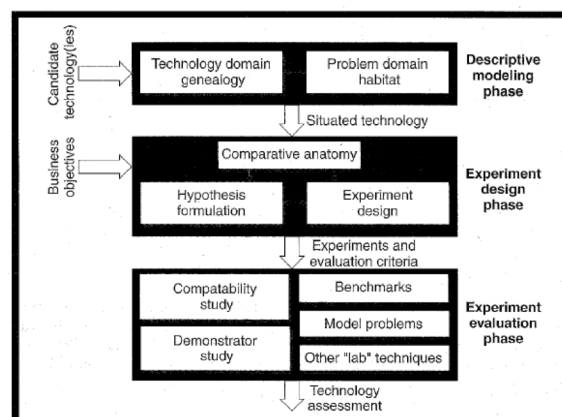


Figure 1: Software technology evaluation framework.

3.1 Descriptive Modeling

write where the technology comes from, its history, its context and what problem it solves. Consider drawing a graph like in [1].

3.2 Experiment Design

Write your hypotheses about what benefits the technology brings and how you can support or reject them via experiments.

3.3 Experiment Evaluation

Write about the results of your experiments, either via personal experience reports, quantitative benchmarks, a demonstrator case study or a combination of multiple approaches.

For some reports you may have to include a table with experimental results or other kinds of tables that for instance compare technologies. Table 1 gives an example of how to create a table.

Config	Property	States	Edges	Peak	E-Time	C-Time	T-Time
22-2	A	7,944	22,419	6.6 %	7 ms	42.9%	485.7%
22-2	A	7,944	22,419	6.6 %	7 ms	42.9%	471.4%
30-2	B	14,672	41,611	4.9 %	14 ms	42.9%	464.3%
30-2	C	14,672	41,611	4.9 %	15 ms	40.0%	420.0%
10-3	D	24,052	98,671	19.8 %	35 ms	31.4%	285.7%
10-3	E	24,052	98,671	19.8 %	35 ms	34.3%	308.6%

Table 1: Selected experimental results on the communication protocol example.

4 Prototype Implementation

This section summarizes how the FeedApp prototype has been implemented, focusing on the core technical components and how developers can run or extend the system.

4.1 Setup and Deployment

The FeedApp prototype runs as a multi-container application orchestrated by Docker Compose with four services: backend (Spring Boot, port 8080), frontend (React/Vite, port 5173), Redis (cache,

port 6379), and Kafka (event bus, ports 9092/29092). The backend reads infrastructure dependencies from environment variables: `SPRING_KAFKA_BOOTSTRAP_SERVERS=kafka:29092` and `SPRING_REDIS_HOST=redis`.

To run locally, developers execute `docker compose up -d` and access the frontend at `http://localhost:5173`. The system can be stopped with `docker compose down`.

```
1 backend:
2   ports: ["8080:8080"]
3   environment:
4     - SPRING_KAFKA_BOOTSTRAP_SERVERS=kafka:29092
5     - SPRING_REDIS_HOST=redis
6 frontend: { ports: ["5173:5173"], depends_on: [backend] }
```

The frontend communicates with the backend via REST APIs and maintains a WebSocket connection for real-time push notifications.

4.2 Backend Implementation

The backend uses Spring Boot with a layered architecture: Controllers handle HTTP mapping, Services contain business logic, and Repositories manage persistence. The system supports two persistence strategies via Spring profiles: `HibernatePollService` (JPA/Hibernate for production) and `InMemoryPollService` (for testing). A `CachingPollService` decorator adds Redis-based caching with Jedis, improving performance for frequently accessed data like poll results.

4.2.1 Authentication and Security

The application implements stateless JWT authentication with BCrypt password hashing. The `/api/auth/login` endpoint authenticates users and returns a JWT token, while `/api/auth/register` creates new accounts. The `JwtAuthFilter` validates tokens on each request, making the authenticated user available via `@AuthenticationPrincipal`.

```
1 // SecurityConfig (rules excerpt)
2 http.csrf(AbstractHttpConfigurer::disable)
3   .sessionManagement(s -> s.sessionCreationPolicy(STATELESS))
4   .authorizeHttpRequests(a -> a
5     .requestMatchers("/api/auth/**", "/rawws").permitAll()
6     .requestMatchers(HttpMethod.GET, "/api/polls/**").permitAll()
7     .requestMatchers(HttpMethod.POST, "/api/polls/*/vote").permitAll()
8     .anyRequest().authenticated());
```

This configuration allows public access to viewing polls and voting while protecting write operations. CORS is configured for localhost development.

4.2.2 Core APIs and Real-Time Updates

The REST API provides two main resource groups: `/api/polls` (CRUD operations, voting, comments) and `/api/users` (profile management with authorization checks).

```
1 // PollController (excerpt)
2 @PostMapping("/{id}/vote")
3 public boolean castVote(@PathVariable Integer id,
4                         @RequestParam Integer presentationOrder,
5                         @AuthenticationPrincipal User user) {
6     return pollService.castVote(id,
7                                 user == null ? Optional.empty()
8                                   : Optional.of(user.getUserId()),
9                                 presentationOrder);
10 }
```

When data changes occur, the backend emits Kafka messages and broadcasts lightweight WebSocket notifications to connected clients:

```
1 // on create/update/delete
2 RawWebSocketServer.broadcast("pollsUpdated");
```

This ensures all clients stay synchronized without polling.

4.3 Frontend Implementation

The frontend uses React with TypeScript, bundled via Vite. Key components include `CreatePoll`, `VoteOnPoll`, `ManagePolls`, `CommentSection`, and `Login`. The application uses a hybrid communication strategy: REST for data operations and a single WebSocket for real-time events.

```
1 // ws.ts
2 export const onWs = (f:(e:unknown)=>void) =>
3   (listeners.add(f), () => listeners.delete(f));
4
5 export function connectWs() {
6   const u = `${location.protocol}===https:"?"wss":"ws"}://
7             ${location.hostname}:8080/rawws`;
8   const ws = new WebSocket(u);
```

```
9     ws.onmessage = ev => {  
10         try { emit(JSON.parse(ev.data)); }  
11         catch { emit(ev.data); }  
12     };  
13 }
```

The App component listens to WebSocket events and selectively refreshes data:

```
1 // App.tsx  
2 useEffect(() => onWs(m => {  
3     if (m === "pollsUpdated") refreshPolls();  
4     if (m === "commentsUpdated") bumpCommentsVersion(activePollId);  
5 }, []);
```

When a poll is created, the UI automatically switches to the "Vote" tab and refreshes, allowing immediate verification. A WebSocketMessages component displays a rolling log of events for development debugging.

4.4 Extensibility and Future Work

The layered architecture and Spring profiles make the system easily extensible. For example creating an app for Ios or Android would primarily involve building a new frontend that consumes the existing REST and WebSocket APIs. The modular design ensures incremental feature additions without disrupting existing functionality. Security improvements could be made by implementing refresh tokens and stricter CORS policies for production.

5 Conclusions

Concludes on the project, including the technology, its maturity, learning curve, and quality of the documentation.

The references used throughout the report should constitute a well chosen set of references, suitable for someone interesting in learning about the technology.

References

- [1] A.W. Brown and K. C. Wallnau. A framework for evaluating software technology. *IEEE Software*, 13(5):39–49, September 1996.