# DAT250 FeedApp Report

Erlend André Høntorp, Jonas Heen Opsahl and Thomas Tolo Jensen

November 11, 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 is keeping data in sync between REST and WebSocket, handling cache updates correctly, and seeing how Kafka helps improve real-time features.

## Contents

# 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.

Concretely, you shall write about

- the *use cases*,
- the *domain model*, and
- the *architecture* (including applied technologies)

Each part shall ideally be accompanied with a graphical representation (diagram).

You may have a look at the Examples on GitHub.

# 3 Technology Assessment

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

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

## 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].

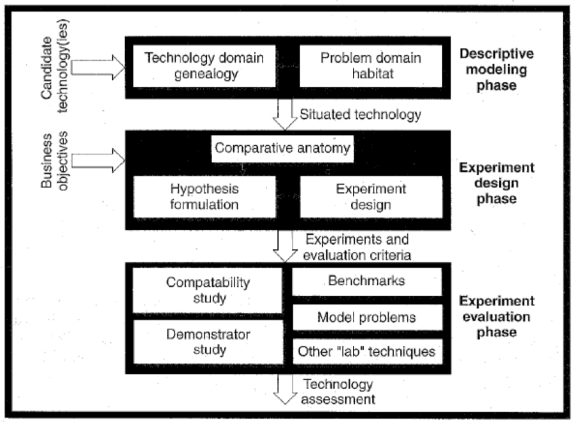

Figure 1: Software technology evaluation framework.

### 3.2 Experiment Design

Write you hypotheses about what benefits the technology bring 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 demostrator case study or a combination of multiple approaches.

For some reports you may have to include a table with experimental results are other kinds of tables that for instance compares 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 gives a short overview of how the prototype is put together and how others can run and extend it.

## 4.1 Setup and Deployment

Services: Spring Boot backend, React (Vite) frontend, Redis, and Kafka. All run with Docker Compose.

How to run

- `docker compose up -d`
- Frontend: `http://localhost:5173`
- Backend: `http://localhost:8080`

Compose

```yaml
backend:
  ports: ["8080:8080"]
  environment:
    - SPRING_KAFKA_BOOTSTRAP_SERVERS=kafka:29092
    - SPRING_REDIS_HOST=redis
frontend: { ports: ["5173:5173"], depends_on: [backend] }
```

## 4.2 Backend Implementation

Layering: Controllers map HTTP to Service methods; Services hold domain logic; Repositories persist entities. Small events are pushed to clients over a raw WebSocket to keep UIs in sync.

Controller example (vote)

```java
// PollController (excerpt)
@PostMapping("/{id}/vote")
public boolean castVote(@PathVariable Integer id,
                        @RequestParam Integer presentationOrder,
                        @AuthenticationPrincipal User user) {
  return pollService.castVote(id, user == null ? Optional.empty() :
      Optional.of(user.getUserId()),
                              presentationOrder);
}
```

Notify clients after changes

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

## 4.3 Frontend Implementation

The React app fetches data via REST and listens to a single WebSocket for deltas (new polls, comments, votes).

WebSocket client

```
// ws.ts
export const onWs = (f:(e:unknown)=>void) => (listeners.add(f), () =>
    listeners.delete(f));
export function connectWs() {
  const u =
      `${location.protocol==="https:"?"wss":"ws"}://${location.hostname}:8080/rawws`;
  const ws = new WebSocket(u);
  ws.onmessage = ev => { try { emit(JSON.parse(ev.data)); } catch { emit(ev.data);
      } };
}
```

Integrating updates (App)

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

Calling the API (vote)

```
await fetch(`/api/polls/${id}/vote?presentationOrder=${order}`, { method: "POST" });
```

## 4.4 Security

Stateless JWT protects write operations; selected GET routes are public; CORS allows localhost during development.

```
// SecurityConfig (rules excerpt)
http.csrf(AbstractHttpConfigurer::disable)
    .sessionManagement(s -> s.sessionCreationPolicy(STATELESS))
    .authorizeHttpRequests(a -> a
      .requestMatchers("/api/auth/**", "/rawws").permitAll()
      .requestMatchers(HttpMethod.GET, "/api/polls/**").permitAll()
      .requestMatchers(HttpMethod.POST, "/api/polls/*/vote").permitAll()
```

```
        .anyRequest().authenticated());
```

## 5 Conclusions

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

The references used throughput 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.