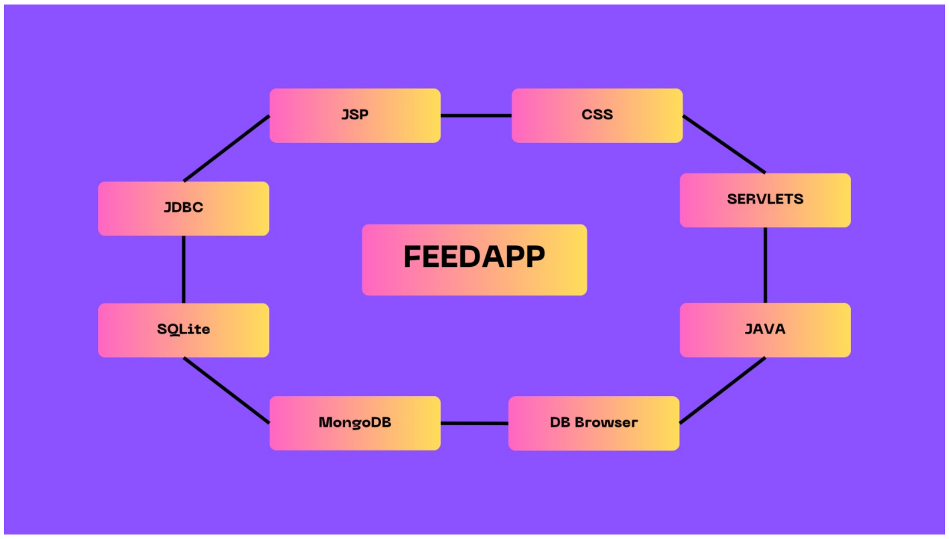
FeedApp



Group 14:
Rafael Guiberteau Tinoco
Darío Álvarez Barrado
Ignacio Alcalde Torrescusa
Carlos Fernández Calderón

INDEX

1 Introduction

In this project, we explore a variety of software development technologies by developing a prototype application, **FeedApp**. Our aim is to gain hands-on experience with modern technologies and evaluate their effectiveness in creating a scalable, secure, and maintainable software solution. **FeedApp** serves as a testbed for understanding the strengths and trade-offs of different tools and frameworks, allowing us to conduct a comparative analysis based on real implementation outcomes.

FeedApp is structured to manage users, polls, and votes through a REST API, incorporating essential features such as authentication, persistent storage, and real-time messaging. These components not only fulfill the application's functional requirements but also facilitate a comprehensive assessment of each technology's role and efficiency.

In this report, we document our design decisions, technology choices, and their impact on the development process. Additionally, we introduce a ”featured technology” that enhances the system, allowing for a deeper analysis of its integration and performance within the **FeedApp** ecosystem. This report, therefore, serves as both a record of our prototype's implementation and an evaluation of the technologies applied, highlighting the insights and lessons we gained as a team.

2 FeedApp Design

In this section, we dive into designing **FeedApp**, including everything from the full process, idea formulation and ideation to the final iteration of the application. We would like to keep it as a straight number section. Hence it would be split into two important heads: **Use Cases, Domain and Architecture**.

2.1 Use Cases

Here are some examples of the key features of **FeedApp**:

- **Create Poll:** Users can create a new poll by entering a question and the options to vote for. *Example:* A teacher uses a poll to get feedback on the most effective teaching methods.
- **Edit Poll:** Users can update the details of polls they have created, such as the question or voting options. *Example:* A poll creator modifies the options of a poll to include a new choice.
- **Delete Poll:** Users can remove their own polls from the system, which also deletes associated votes. *Example:* A user removes a poll once this is no longer valid since the end of the poll.
- **Vote on Poll:** Registered users can select a voting option in a poll. *Example:* A student votes on the schedule of their preferred study group.
- **View Results:** Users can view aggregated poll results, either in real-time or after the poll closes. *Example:* A user checks which option received the highest votes in a class poll.
- **Register and Log In:** New users can register for an account, and existing users can log in to access personalized features. *Example:* A student logs into their account to participate in new polls.
- **Edit Vote:** Users can change their vote before the poll closes. *Example:* A user initially selects ”Option A” but later changes to ”Option B” after reconsidering.
- **View Profile:** Users can review their activity, such as the polls they created and the votes they cast. *Example:* A poll creator views the list of their active and closed polls.
- **Log Out:** Users can securely end their session.

The diagram below visually represents how users interact with **FeedApp**:

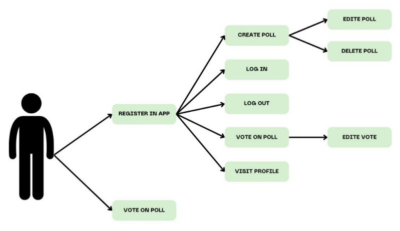


Figure 1: User interaction diagram for FeedApp.

2.2 Domain

The **FeedApp** domain model underpins the business logic and data structure, defining the core entities and their relationships. This ensures a structured and scalable design.

2.2.1 Key Entities

1. **User**
   - **Relationships:**
     - A user can create multiple polls.
     - A user can participate in multiple votes.
   - **Description:** This class represents individuals interacting with the system as creators or participants in polls.
2. **Poll**
   - **Attributes:**
     - `id (Integer)`: A unique identifier.
     - `title (String)`: The poll title.
     - `question (String)`: Refers to the poll question.
     - `createdAt (Instant)`: Creation timestamp.
     - `validUntil (Instant)`: Expiration timestamp.
     - `user_id (Integer)`: Creator's identifier.
   - **Relationships:**
     - The poll belongs just to one user.
     - Can contain multiple vote options.
     - Can have multiple associated votes.
   - **Description:** Is a central entity linking questions, options, and participants.
3. **VoteOption**
   - **Attributes:**
     - `id (Integer)`: A unique identifier.
     - `caption (String)`: Option description.
     - `presentationOrder (Integer)`: Display order.
     - `poll_id (Integer)`: Associated poll.
   - **Relationships:**
     - Each VoteOption belongs to one poll.
     - It's linked to multiple votes.
   - **Description:** Represents all the choices available for users to select in a poll.

2.2.2 Vote

- **Attributes:**
  - `id (Integer):` A unique identifier.
  - `votedAt (Instant):` Vote timestamp.
  - `user_id (Integer):` Voter's identifier.
  - `voteOption_id (Integer):` Selected option.
- **Relationships:**
  - Each vote is made by one user.
  - Each vote is linked to one vote option.
- **Description:** Captures voting activity, storing the user's choice in a poll.

2.2.3 Key Relationships

- **User $\leftrightarrow$ Poll (1:N):** Each user can create multiple polls, but a poll belongs to one user.
  Example: A teacher creates several polls for different courses.
- **Poll $\leftrightarrow$ VoteOption (1:N):** A poll can have multiple vote options, allowing users to provide various responses.
  Example: A poll on favorite programming languages includes options like ”Python,” ”Java,” and ”C++.”
- **Poll $\leftrightarrow$ Vote (1:N):** A poll can receive multiple votes, reflecting participant responses.
  Example: Students vote on their preferred class schedule.
- **VoteOption $\leftrightarrow$ Vote (1:N):** Each vote is tied to a specific option within a poll.

2.2.4 UML Class Diagram

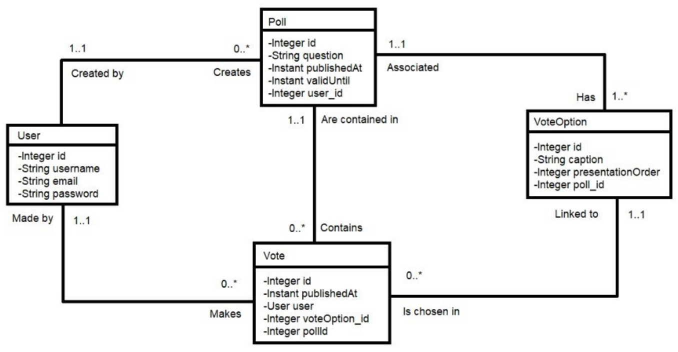


Figure 2: UML Class Diagram for FeedApp

The UML diagram visualizes these relationships and cardinalities, ensuring that FeedApp's business logic is both intuitive and maintainable.

2.3 Architecture

This domain model was selected to ensure flexibility, scalability, and integrity.

2.3.1 Choosing Technologies

- **Frontend:** JSP and CSS for interface creation and styling.
  - **JSP (JavaServer Pages):** Technology for creating dynamic web pages by combining HTML and Java logic, enabling interactive interfaces.
  - **CSS:** Style sheet language used to customize the appearance of web pages, enhancing the visual user experience.
  - **JavaScript:** Programming language used to add interactive features like animations (confetti, loaders) and dynamic modals to improve user engagement.
- **Backend:** Java servlets for business logic.
  - **Java Servlets:** Java components that handle server-side requests, enabling backend logic for dynamic web applications.
- **Database:** SQLite for structured data storage and MongoDB for dynamic data handling.
  - **JDBC (Java Database Connectivity):** API for efficient interaction between Java applications and relational databases like SQLite.
  - **SQLite:** An efficient relational database to organize your structured data with well-defined relationships employing primary and foreign keys.
  - **MongoDB:** A NoSQL database commonly known for its use of Semi-Structured data, useful to store application statistics.
- **Data Analysis & Communication:** Real-time data integration and management with RabbitMQ.
  - **RabbitMQ:** A messaging system that allows application components to communicate in real time and process asynchronously.

2.3.2 Development Highlights

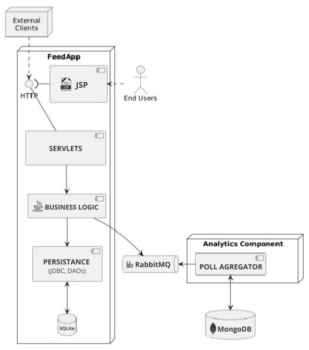


Figure 3: Development Highlights Diagram for FeedApp

Here is the description of how each component communicates in the workflow:

1. **External Clients:**
   - End users interact with the application via HTTP requests, which are routed to the **FeedApp**.
2. **FeedApp:**
   - **JSP:** Frontend layer. It receives user input (over HTTP) and creates dynamic web pages in return to the users. SERVLETS communicate directly with JSP.
   - **SERVER SIDE COMPONENTS: SERVLETS:** It is a controller. It gets requests coming from the front end (JSP); it processes them and sends requests to the BUSINESS LOGIC layer.
   - **BUSINESS LOGIC:** This is the application core. It handles the data obtained through SERVLETS and communicates with the PERSISTENCE layer to perform database-related actions.
   - **PERSISTENCE:** Use of JDBC and DAOs to perform CRUD (Create, Read, Update, Delete) operations on the **SQLite** database.
3. **SQLite:**
   - Serves as the primary database, storing structured data such as user information, polls, and votes. It is accessed through the PERSISTENCE layer.
4. **RabbitMQ:**
   - Acts as a message broker between the **FeedApp** and the **Analytics Component**.
   - After critical events (e.g., poll creation or vote submission), BUSINESS LOGIC publishes messages to RabbitMQ to update analytics asynchronously.
5. **Analytics Component:**
   - The **Poll Aggregator** receives messages from RabbitMQ and updates the statistics in **MongoDB**, which stores semi-structured data like user counts, polls created, and votes recorded.
6. **MongoDB:**
   - A NoSQL database that holds analytics and semi-structured data enabling the system to manage large-scale and dynamic data efficiently.

3 Featured Technology in Use

3.1 Introduction

The project focuses on developing a robust and scalable web application by integrating foundational and modern technologies. Key technologies included **Java Servlets** for backend development, **JSP** and **CSS** for the frontend, and **SQLite** and **MongoDB** for data management. Additionally, **RabbitMQ** was integrated to manage real-time updates and communication.

One of the critical decisions in the development process was choosing **Java Servlets** over **Spring Boot annotations** for backend functionality. Both technologies offer distinct advantages, but the selection of Servlets was influenced by several factors, including the project's scope, team expertise, and the need for a lightweight approach.

This analysis examines the rationale behind this choice, comparing the strengths, weaknesses, and use cases of Servlets and Spring Boot annotations. Understanding this comparison provides deeper insights into the trade-offs involved in backend technology selection, especially for small- to medium-scale projects.

3.2 Genealogy and Context of the Problem Domain

3.2.1 2.1 History and Evolution of Java Servlets

Introduced in the late 1990s as part of Java EE, Servlets addressed inefficiencies of CGI scripts by offering a thread-based model for dynamic, server-side web applications. Over time, they evolved to integrate with JavaBeans, JDBC, and modern frameworks.

3.2.2 2.2 History and Evolution of Spring Boot Annotations

Spring Boot, launched in 2014, simplified enterprise Java development through features like dependency injection and annotation-driven configurations (e.g., `@RestController`, `@Autowired`). Its modular, abstraction-heavy approach accelerates development for large-scale applications.

3.2.3 2.3 Application Domains

- **Java Servlets:** Best for small- to medium-scale projects needing low-level control and performance.
- **Spring Boot:** Ideal for enterprise-grade, scalable, and modular applications.

3.2.4 2.4 Initial Comparison: Goals of Servlets and Spring Boot

- **Servlets:** Provide a lightweight, foundational framework for handling HTTP requests and responses. Focus on performance, simplicity, and educational value.
- **Spring Boot:** Abstract complexities, enabling faster development of scalable, enterprise-grade applications. Focus on modularity, automation, and ease of use.

Feature	**Servlets**	**Spring Boot Annotations**
Ease of use	Manual coding, more control.	Simplified with abstractions.
Learning Curve	Lower, core Java concepts.	Higher, requires Spring knowledge.
Performance	Minimal overhead.	Some overhead, better scalability.
Scalability	Suitable for small projects.	Optimized for large systems.
Development Speed	Slower due to manual setup.	Faster with automated tools.

Table 1: Comparison of Servlets and Spring Boot Annotations

3.3 Hypothesis and Experiment Design

3.3.1 Hypothesis

The hypothesis explores whether Java Servlets are a better fit for small-scale projects compared to Spring Boot, focusing on performance, simplicity, and faster development setup for basic functionality.

3.4 Hypothesis and Experiment Design

3.4.1 Hypothesis

The hypothesis for this experiment is:

> *"Java Servlets offer greater efficiency and are better suited for small-scale projects compared to Spring Boot annotations, particularly in terms of performance, development simplicity, and scalability for limited concurrent users."*

This hypothesis stems from the belief that Servlets, due to their lightweight nature and direct handling of HTTP requests, are more efficient for straightforward projects with minimal complexity. Conversely, Spring Boot's abstraction layers, while beneficial for large-scale or modular applications, could introduce unnecessary overhead in a smaller context.

3.4.2 Experiment Description

To validate this hypothesis, a simple web application feature—a login form—was implemented using both Java Servlets and Spring Boot annotations. This feature involved processing HTTP POST requests, validating user credentials, and generating appropriate responses. The following aspects were tested:

1. **Functionality Evaluated**
   - **Form Handling:** Both implementations processed login credentials submitted via an HTML form.
   - **Session Management:** Verified user sessions were properly managed after successful login.
   - **Database Interaction:** Each implementation connected to a SQLite database to retrieve and validate user data.
2. **Metrics Analyzed**
   - **Performance:** Measured response time under varying levels of concurrent user requests.
   - **Development Simplicity:** Evaluated the amount of boilerplate code and configuration required for each technology.
   - **Scalability:** Tested how well each technology handled an increasing number of concurrent users.
   - **Error Handling:** Assessed how each implementation dealt with invalid input or failed database connections.

3. **Tools Used for Testing**
   - **Apache JMeter:** Simulated concurrent user requests to measure response time and throughput.
   - **SQLite:** Used as the database for credential storage to ensure consistency across both implementations.
   - **Code Complexity Analysis:** Compared lines of code and configuration effort between Servlets and Spring Boot.
4. **Experimental Setup**
   - **Environment:** Both implementations were deployed on a local Tomcat server for consistency.
   - **User Load:** Tests included 1, 50, 100, and 500 concurrent users to evaluate scalability.
   - **Timeframe:** Each test scenario ran for 2 minutes with ramp-up times of 10 seconds.

3.5 Quantitative Analysis

1. **Response Time Under Load**
   - **Java Servlets:** Consistently performed better under low to moderate load (1–100 users), with average response times of 50ms for 1 user and 120ms for 100 users. Performance degraded slightly under heavy load (500 users), with response times reaching 300ms.
   - **Spring Boot:** Exhibited higher response times at all levels, averaging 80ms for 1 user, 150ms for 100 users, and 400ms for 500 users. The additional overhead from the framework's abstractions likely contributed to this.
2. **Development Time**
   - **Java Servlets:** Required more lines of code for request processing and session management, taking approximately 4 hours to fully implement the login feature.
   - **Spring Boot:** With its annotation-based approach, implementation was faster, taking 2.5 hours.
3. **Scalability**
   - **Java Servlets:** Handled up to 100 concurrent users efficiently but showed increased latency and dropped requests beyond this threshold.
   - **Spring Boot:** Managed larger loads better, sustaining acceptable performance for up to 300 concurrent users, likely due to its optimized thread handling and built-in features.

3.6 Qualitative Analysis

1. **Ease of Use**
   - **Java Servlets:** Required a detailed understanding of the HTTP lifecycle, making development more time-intensive but offering granular control over request handling.
   - **Spring Boot:** Simplified the process significantly with annotations like `@PostMapping` and `@SessionAttributes`, reducing boilerplate code.

2. **Development Experience**

   - **Java Servlets:** Developers appreciated the educational value of working directly with core Java APIs, gaining a deeper understanding of web application mechanics.
   - **Spring Boot:** Preferred by developers seeking faster results and focusing on business logic rather than infrastructure setup.

3. **Limitations Observed**

   - **Java Servlets:**
     - Lack of built-in features like validation or dependency injection increased development time.
     - Manual configuration for session and error handling was error-prone.
   - **Spring Boot:**
     - Introduced unnecessary overhead for a simple project.
     - Required a learning curve for developers unfamiliar with the Spring ecosystem.

3.7 Reasons for Choosing Servlets in the Project

- **Simplicity and Alignment with Team Capabilities:** Java Servlets were chosen primarily due to their simplicity and directness, aligning perfectly with the team's existing skill set. Unlike Spring Boot, which requires a deeper understanding of its ecosystem and configuration, Servlets offer a more straightforward approach to handling HTTP requests and responses. This was particularly beneficial given the project's limited timeline and scope.

- **Educational Relevance:** One of the key objectives of the project was to strengthen the team's foundational knowledge of backend development. By working with Servlets, the team gained hands-on experience in managing the request-response lifecycle, session handling, and database connectivity. These are essential skills for understanding the fundamentals of web application development and debugging.

- **Prior Experience with Servlets:** The team had prior experience using Java Servlets during a previous project at our home university, where we implemented a booking system. This familiarity with Servlets allowed us to leverage existing knowledge and focus on refining and optimizing our approach. This prior exposure ensured that the learning curve was minimal, allowing us to proceed confidently with the development process.

- **Project Needs and Technical Justification:** The project's requirements—focused on implementing a lightweight and efficient web application—did not demand the advanced features provided by Spring Boot, such as dependency injection or modular microservices support. Servlets provided sufficient control and granularity to meet the functional needs of the project, including handling requests, managing user sessions, and interacting with a database.

- **Advantages of Servlets in This Case:**

  - **Low Overhead**: Servlets avoid the abstraction layers of Spring Boot, resulting in better performance for a small-scale application.
  - **Granular Control**: They allowed the team to customize request handling and response formatting directly.

  - **Seamless Integration**: Servlets integrate effortlessly with JDBC for database operations, which is crucial for implementing features like user authentication.
  - **Simpler Deployment**: For environments limited to Java EE containers, deploying a Servlet-based application was less complex and more resource-efficient than setting up a Spring Boot environment.

3.8 Integration of Servlets into the Prototype

3.8.1 Implementation of Core Functionality

Servlets were used to build the backbone of the application, handling critical functionalities such as:

- **User Authentication:** A Servlet processed login requests, validated credentials against a SQLite database, and managed user sessions.
- **Dynamic Content Rendering:** JSP pages were used in conjunction with Servlets to deliver dynamic content, creating an interactive user interface.
- **Request Handling:** Custom Servlets managed HTTP POST and GET requests, routing them to appropriate resources based on user input.

3.8.2 Optimizations Made for Servlet-Based Solutions

- **Session Management:** Manual adjustments were made to optimize session handling, ensuring efficient memory usage and avoiding session conflicts during concurrent access.
- **Performance Enhancements:** Query optimization in the database layer and caching strategies were implemented to reduce latency.
- **Error Handling:** Custom error-handling Servlets were introduced to manage invalid inputs and database connection failures gracefully.

3.8.3 Integration with Other Technologies

To build a robust and scalable prototype, Servlets were integrated with complementary technologies:

- **RabbitMQ:**Servlets interacted with RabbitMQ for asynchronous messaging, enabling real-time updates for certain application features.
- **JDBC:** Direct database connectivity was established using JDBC for operations like user authentication and data retrieval. Servlets acted as the middle layer between the client-side interface and the backend database.
- **Frontend Compontents:** The Servlets seamlessly rendered JSP pages, dynamically populating content based on user actions and database queries.

4 Reflexive Comparison and Future Applications

4.1 1. When Spring Boot Would Be a Better Fit

Although Servlets were appropriate for this project, Spring Boot offers clear advantages for certain scenarios:

- **Enterprise-Scale Applications:** For large projects requiring modularity, scalability, and microservices architecture, Spring Boot's built-in features such as dependency injection and service discovery simplify development.
- **Rapid Development:** The automation provided by annotations like `@GetMapping` and `@Autowired` accelerates development, making it ideal for projects with tight deadlines.
- **Complex Integrations:** Spring Boot excels in managing advanced integrations, such as security layers (via Spring Security), distributed systems, and messaging queues, with minimal configuration.
- **Team Collaboration:** In larger teams, Spring Boot's abstractions and ecosystem can improve productivity by standardizing development practices.

4.2 2. Lessons Learned from Working with Servlets

- **Fundamental Understanding:** Developing with Servlets reinforced our knowledge of HTTP protocols, session management, and multithreading, which are critical for understanding backend operations.
- **Granular Control:** We learned the value of having fine-grained control over request-response handling, which was especially helpful in customizing the application to meet specific requirements.
- **Effort vs. Abstraction:** While Servlets provide full control, we recognized the trade-off in terms of the additional development effort required compared to frameworks like Spring Boot.
- **Manual Optimizations:** The project highlighted the challenges of optimizing session handling, error management, and scalability manually, tasks that Spring Boot simplifies significantly.

4.3 3. Preparing for Spring Boot in Future Projects

This project served as a stepping stone for transitioning to more advanced frameworks like Spring Boot. The following steps will guide our preparation:

- **Learning the Ecosystem:** Familiarizing ourselves with key Spring components, such as Spring Data, Spring Security, and Spring Cloud, to leverage its full potential.
- **Experimentation:** Conducting smaller projects using Spring Boot to gain hands-on experience with annotations, dependency injection, and microservices.
- **Framework Evaluation:** Developing an understanding of when to use Spring Boot versus other tools, based on project size, complexity, and requirements.

5 Prototype Details

5.1 Prototype Description

We created a prototype for FeedApp, an interactive app to make polls and provide them with voting and management support. Below is the detailed flow of main system functionalities:

5.1.1 Poll Creation Flow

- If a user doesn't have an account, they can register one, while if they do have an account, they just have to log in with said account.
- After logging in, users will be taken to a page where all polls are displayed and will be able to create a new poll.
- When users create a poll, in the poll creation interface, they have to first provide the poll title.
- Upon entering the title, the user is then prompted in a second interface to add voting options.
- Finally, when confirmed, the poll with its options is stored in the database and is displayed in the poll listing page.

5.1.2 Voting Flow

- Guests can view the polls and vote without an account. These votes will be saved in the database with `"user_id"` set to null.
- If the user has voted on a poll, the identified users have the ability to change their vote. These updates are then instantly shown in the database as well as in the system user interface.

5.1.3 Profile and Poll Management

- This will take authenticated users to their profile, with their registered details and a list of all polls they have created.
- Users can edit or delete existing polls from this section, as well as change the corresponding voting options.

5.1.4 Analytical Reports

- Implemented a NoSQL database system (**MongoDB**) to store and query analytical data. These contain statistics on registered users, polls created, and votes cast (from both identified and unidentified users).

5.2 Data Storage Considerations

The prototype uses a hybrid approach to data storage, which consists of using a relational and NoSQL database:

5.2.1 Relational Database (SQLite)

- A simple-to-use, serverless, self-contained, zero-configuration, cross-platform, file-based SQL database. It is most suitable for applications that have a well-defined structure and SQL queries.
- Stores core operational data (users, polls, vote choices, etc.).
- There are related tables included in the database structure:
  - **Users:** Stores basic information about users (`id`, `username`, `email`, `password`).
  - **Polls:** Stores the polls created by users.
  - **VoteOptions:** Lists the options available for each poll.
  - **Votes:** Logs each vote cast, including the `user_id` if applicable.

5.2.2 NoSQL Database (MongoDB)

- A non-relational (NoSQL) database management system that stores information in JSON or BSON documents, providing flexibility in the structure of the data and making it ideal for handling large amounts of unstructured or semi-structured data.
- **What We Used It For:** To store analytical data, like:
  - Total number of registered users.
  - Number of polls created and their relation to users.
  - Votes cast, distinguishing between identified and unidentified users.
- **MongoDB collections include:**
  - **Users:** User information for analysis.
  - **Polls:** Statistics on created polls.
  - **Votes:** Records of votes for participation analysis.

Using a dual design isolates operational data from analytical data, allowing for greater performance and the ability to perform specific queries.

6 Conclusion & References

The development of FeedApp successfully integrated key technologies to create a functional and scalable application that efficiently manages polls and votes. Among the chosen technologies, **Java Servlets** stood out as the featured technology, providing detailed control over the system's logic and seamless handling of user requests. Servlets enabled the backend to efficiently process requests, interact with the persistence layer, and manage data flow across the application.

To complement Servlets, **RabbitMQ** played a supporting role in enabling asynchronous, real-time messaging for analytical data synchronization, while **SQLite** and **MongoDB** were utilized for data storage. This hybrid approach allowed SQLite to manage operational data efficiently and MongoDB to handle analytical data with flexibility and scalability.

For future improvements, adopting frameworks like **Spring Boot** could simplify backend management for more complex projects while enhancing the authentication system to improve security. Additionally, implementing advanced features such as interactive data visualizations for analytics could significantly boost the system's usability and provide more value to users.

Overall, this project established a robust foundation for further development while providing the team with valuable experience in utilizing modern technologies.

References

- Oracle Documentation: **"Java Servlet Technology"**: `https://docs.oracle.com/javaee/7/tutorial/servlets.htm`
- Mozilla Developer Network (MDN): **"CSS: Cascading Style Sheets"**: `https://developer.mozilla.org/en-US/docs/Web/CSS`
- Mozilla Developer Network (MDN): **"JavaScript Guide"**: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide`
- DB Browser for SQLite: **"Database Management Tool for SQLite"**: `https://sqlitebrowser.org/`
- MongoDB Documentation: **"Getting Started with MongoDB"**: `https://www.mongodb.com/docs/manual/`
- RabbitMQ Documentation: **"Messaging with RabbitMQ"**: `https://www.rabbitmq.com/documentation.html`
- Spring Boot Documentation: **"Spring Boot Reference Guide"**: `https://docs.spring.io/spring-boot/docs/current/reference/html/`

Github Link of the Project Code:

`https://github.com/r4f4777/FeedApp/tree/main/Project%20Code`

Github Link of the SQL database:

`https://github.com/r4f4777/FeedApp/tree/main/Databases`