DevOps, Software Evolution and Software Maintenance, BSc (Spring 2024)

Course code: BSDSESM1KU

May 12, 2024

Exam Assignment by:

Student	Email
Daria Damian	dard@itu.dk
Hallgrímur Jónas Jensson	hajj@itu.dk
Mathias E. L. Rasmussen	memr@itu.dk
Max-Emil Smith Thorius	maxt@itu.dk
Fujie Mei	fume@itu.dk

Contents

1 System's Perspective

1.1 Design and architecture of the Minitwit system

ITU-MiniTwit is designed as a microservices architecture, leveraging containerization to ensure isolation, ease of deployment, and scalability. The system's architecture facilitates independent development and service deployment, enhancing maintenance and testing capabilities.

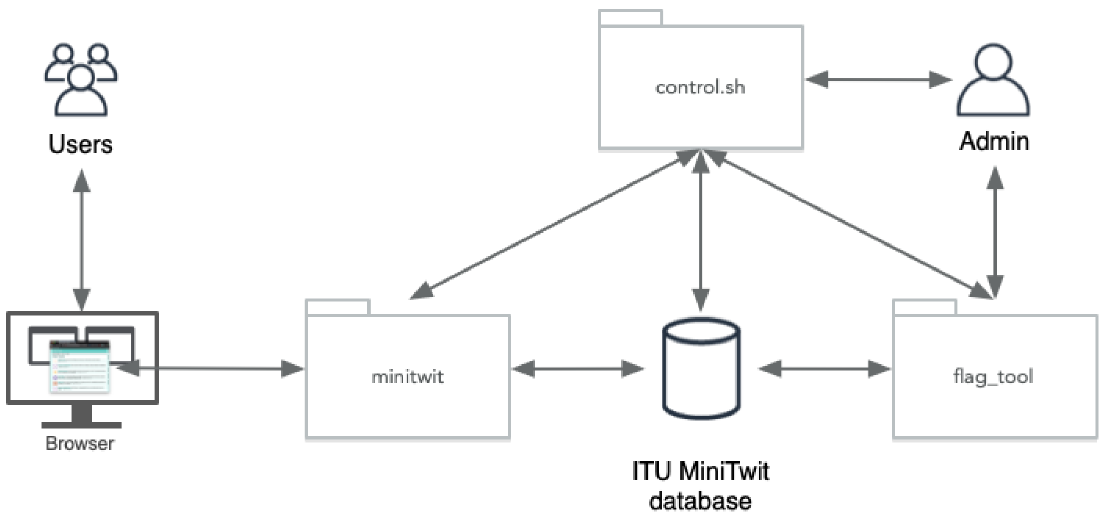


Figure 1.1: High-level ITU-MiniTwit architexture

Detailed Architecture:

- API Service:

  Function: Handles all client-server interactions, processing client requests and sending responses back to clients.

  Implementation: Implemented using Go, which is suitable for creating performant and scalable backend services.

  Files: Located within the API directory, which contains all the routing and logic for handling HTTP requests.

- Database Service:

  Function: Manages data storage and retrieval, serving as the persistent storage layer for the application.

  Implementation: Utilizes SQL, making use of relational database management systems.

  Files: The database directory contains scripts and configurations for database setup and management.

- Front-end:

  Function: Provides the user interface through which users interact with the ITU-MiniTwit application.

  Implementation: Usage of HTML, CSS, and JavaScript files served by the Python service.

  Files: included in the root directory that contains HTML templates and JavaScript files.

- Containerization and Orchestration:

  Utilizes Docker to containerize the application components, ensuring that each component runs in an isolated environment with its dependencies.

  Docker Compose helps in defining and running multi-container Docker applications, where services such as the API and database are orchestrated to work together.

1.2 Dependencies of the Minitwit system

This section will detail all the technologies, tools, and external services our system depends on, structured into categories for clarity.

1. Development Dependencies

- Programming Languages:

  Python: Used for writing backend logic, potentially for web serving and scripting tasks. Dependencies are managed via requirements.txt, which includes libraries for web frameworks, database integration, etc.

  Go: Employed for building efficient, compiled services, possibly for high-performance backend components.

- Version Control:

  Git: Essential for source code management, allowing version control and collaboration among developers.

  GitHub: Hosts the repository and provides additional tools for code reviews, issue tracking, and GitHub Actions for CI/CD.

- Local Development Environment:

  Vagrant: Used to configure and manage virtual development environments that mirror the production environment, ensuring consistency across different machines.

  Docker: Provides a containerized environment to run services isolated from the host system, facilitating easier setup and dependency management.

- External APIs and Services:

  Graphana?

  Prometheus: Employed for monitoring the application's performance metrics, helping in proactive management and optimization of the application performance.

2. Build and Deployment Dependencies Tools and technologies used to build, test, and deploy the ITU-MiniTwit application.

- Build Tools such as Makefile: Automates the compilation and building process, allowing developers to run predefined scripts for building the application, running tests, and other tasks with simple commands.
- Continuous Integration/Continuous Deployment (CI/CD) such as GitHub Actions: Automates the workflow of code integration, testing, and deployment whenever the repository is updated, ensuring that new code changes integrate smoothly and deploy automatically.
- Container Orchestration such as Docker Compose: Manages multi-container Docker applications, used in development and production to configure and orchestrate multiple services (API, database, etc.) to work together seamlessly.

3. Runtime Dependencies

- Server Environment:

  Docker

  Linux/Unix: The underlying operating system for servers running Docker containers, providing robustness and stability.
- Database Management: SQL Database: Our application uses an SQL database for data persistence, and for database operations and management.

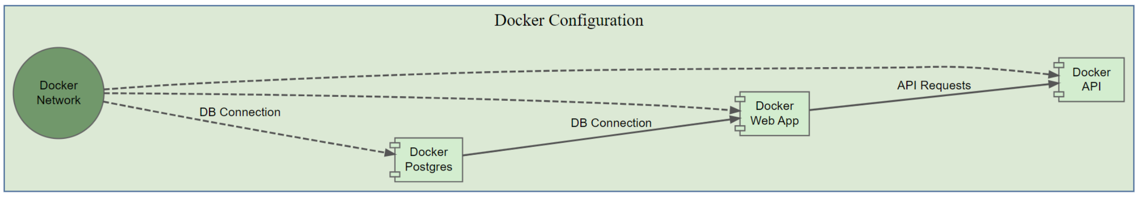


Figure 1.2: Docker Configuration

1.3 Important interactions of subsystems

Information Flow in UML ITU-MiniTwit:

link for the UML diagrams: link1,link2

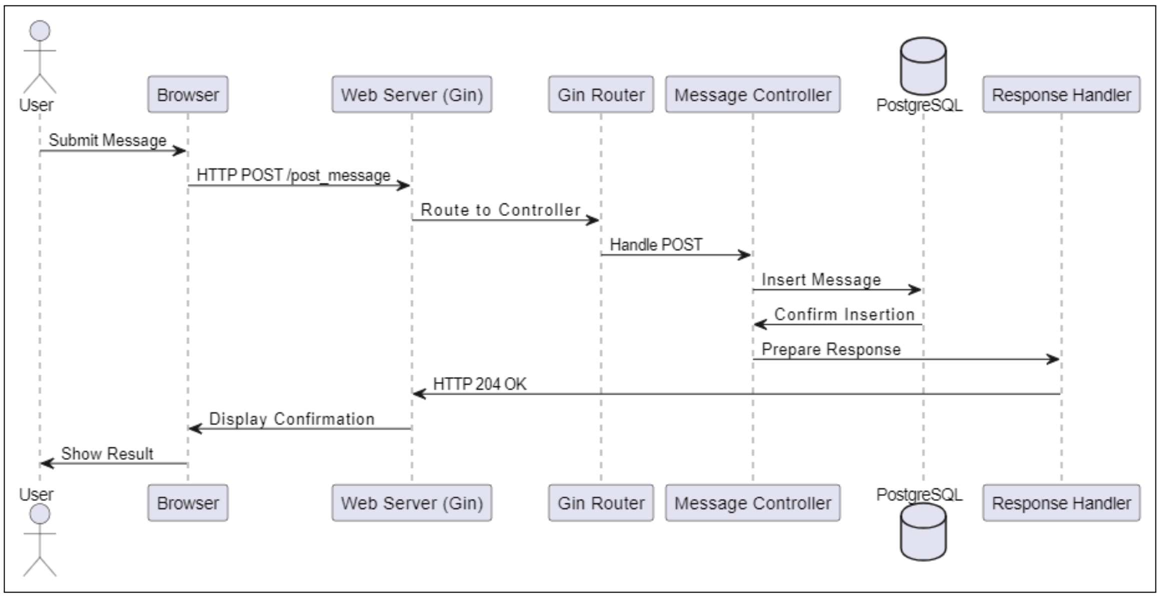


Figure 1.3: Information Flow - UML Sequence diagram

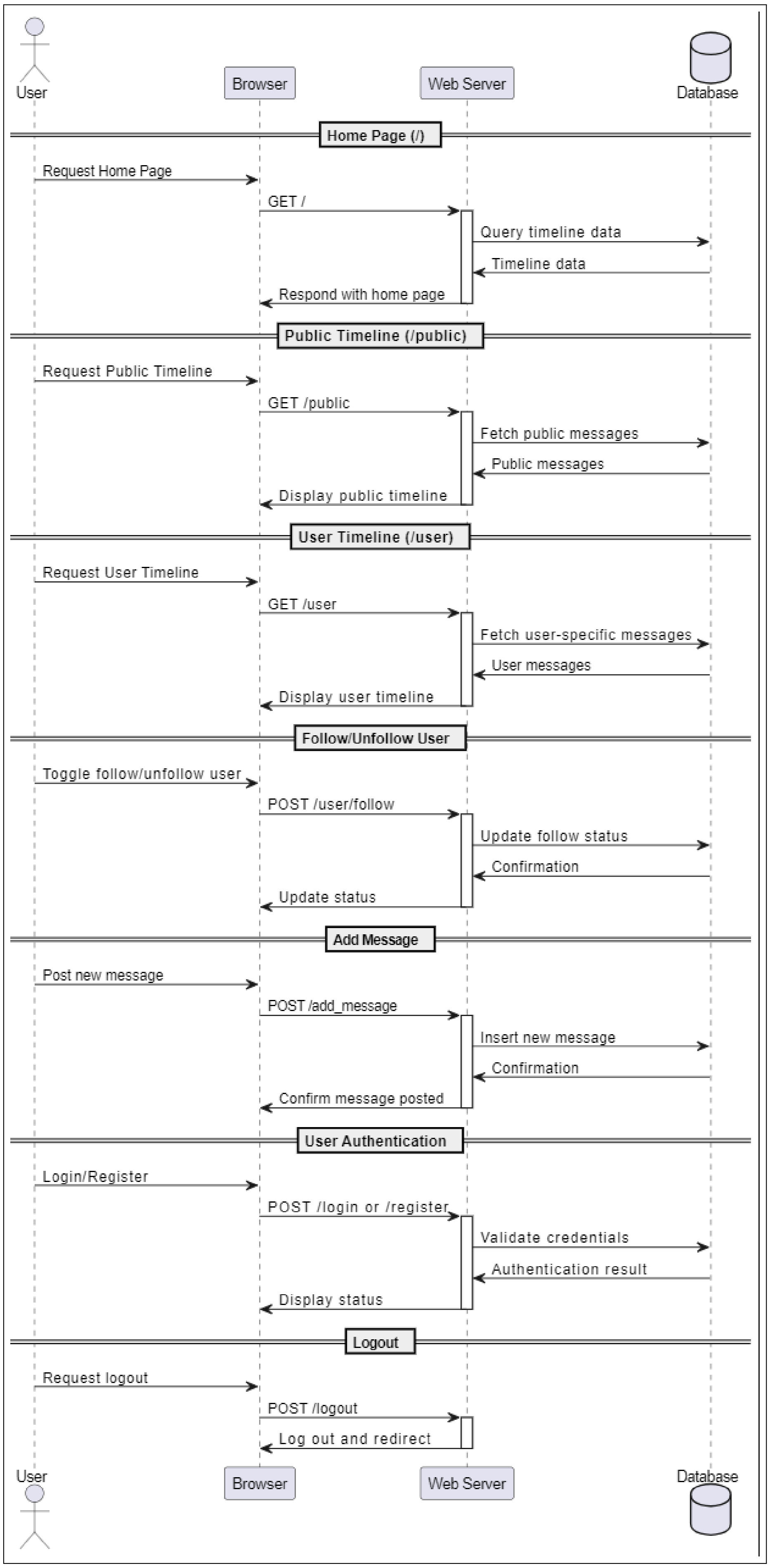


Figure 1.4: How requests from the simulator traverse your system

1.4 Current state of the system

Ideas:

- Static Analysis:
- Present results from static analysis tools to discuss code quality, potential bugs, and security issues.
- Mention any significant findings and how they impact the overall system.
- Quality Assessment:
- Discuss methods used for assessing the quality of the application (e.g., code reviews, testing coverage).
- Summarize the current quality metrics and any goals or benchmarks your team is aiming to achieve.
- Using Prometheus, document current performance metrics of the system, analyzing aspects such as response times, system throughput, and error rates.

1. Static Analysis Results: Static analysis tools examine your code without executing it to find potential issues. Here's what to look at:

- Code Quality Metrics:
- Complexity: Measures how complex the logic in your codebase is. Lower complexity often correlates with easier maintenance.
- Code Style Compliance: Checks adherence to coding standards and conventions which help in maintaining code consistency.
- Potential Bugs: Identification of code patterns that could lead to errors.
- Code Smells: Highlight suboptimal code practices that might not cause bugs but could reduce code quality or increase complexity.
- Security Vulnerabilities:
- Known Vulnerabilities: List of known vulnerabilities in your dependencies or your code that could be exploited.

- Security Hotspots: Areas in the code that require a manual review to ensure they are secure.

2. Quality Assessments: Quality assessments go beyond static code analysis by incorporating other factors such as unit test coverage and integration test results.

- Test Coverage:
- Unit Test Coverage: Percentage of your codebase covered by unit tests. High coverage can reduce the likelihood of bugs making it to production.
- Integration Test Results: These tests cover interactions between modules or services and help identify issues in the way components integrate.
- Performance Metrics:
- Response Times: Average time it takes for your system to respond to user actions. Resource Usage: Usage statistics for CPU, memory, and other resources. High usage might indicate a need for optimization.
- Dependency Health:
- Outdated Libraries: Using outdated libraries can expose your system to security vulnerabilities and compatibility issues. License Compliance: Ensuring that all library licenses comply with your project's licensing.
- 3. System Observability and Monitoring: Logs Analysis: Insights into errors and unusual system behavior captured in logs. Metrics Dashboard: A real-time dashboard showing key performance metrics. Alerts History: Review of past alerts to understand recurring issues or spikes in resource usage.
- 4. Development Practices: Code Review Practices: Regular code reviews help maintain code quality and mentor junior developers. Build and Deployment Frequency: Frequency of deployments can indicate the agility and health of the development process. Feature Development Speed: How quickly new features move from conception to production

2 Process' perspective

2.1 CI/CD Chain

We implemented a GitHub Actions workflow to automate the process of testing, building and deploying the most recent version of Minitwit. It is set to execute on each push to the 'Main' branch of our GitHub repository. The workflow is separated into three main jobs: 'BuildAndTest', 'Deploy', and 'Release' which are executed sequentially, each dependent on the last executing successfully.

2.1.1 Triggers

The workflow is triggered on two specific GitHub events: manual triggers via workflow_dispatch and on pull requests to the main branch.

2.1.2 Jobs

BuildAndTest

In this step all relevant images are built and pushed to the Docker hub and then run and tested to ensure that the system will work as expected. The key steps are as follows:

1. **Checkout:**

- Uses `actions/checkout@v2` to fetch the codebase from the repository.

2. **Environment Setup:**

   - Dynamically creates a `.env` file with database configurations sourced from GitHub secrets.

3. **Docker Operations:**

   - Logs into Docker Hub using credentials from GitHub secrets to push built images.
   - Builds and pushes multiple Docker images, including:
     - The application image
     - API image
     - A test database image

4. **Python Setup and Dependency Installation:**

   - Configures the Python environment.
   - Installs necessary dependencies to ensure consistent testing conditions.

5. **Testing:**

   - Executes integration tests by setting up the application and its dependencies in Docker containers.
   - Conducts API tests and application-specific tests to ensure functionality and reliability.

Deploy

The **Deploy** job is activated on successfully completing the **BuildAndTest** job. This job manages the deployment of the application to the remote server where the application is hosted. The steps are:

1. **Checkout:**

   - Retrieves the latest codebase from the repository, uses `actions/checkout@v2` similar to the first job.

2. **SSH Configuration:**

- Prepares SSH keys to establish a secure connection to the deployment server.

3. **Deployment Execution:**

   - Updates environmental configurations and transfers necessary files to the server using SSH.
   - Employs Docker Compose on the server to pull the latest Docker images.
   - Deploys the images using Docker Stack, which effectively updates the running application on the server.

Release

After the previous jobs have been successfully completed, this job manages software versioning and public release.

1. **Version Calculation:**

   - Executes a script to determine the new version number for automated version tracking.

2. **GitHub Release Creation:**

   - Uses `actions/create-release@v1` to create a formal release on GitHub.
   - Tags the release with the new version number and provides release notes outlining the changes in this version.

2.2 Monitoring

For monitoring we use Prometheus to collect metrics, and integrate with Grafana for live visualization of key metrics. We have two dashboards, one monitoring the API and the other the database.

2.2.1 API dashboard

1. **Response time:** This chart displays the response times of requests over the past six hours, with separate lines for the slowest 95th and 99th percentile requests.

2. **Rate of HTTP requests:** Shows the number of HTTP requests per second, categorized by endpoints.

3. **HTTP Requests Total:** Shows the total number of HTTP requests received, categorized by endpoints.

4. **Total HTTP request errors:** Shows a chart of total HTTP request errors by endpoint.

5. **Bad HTTP request total:** Shows the total number of bad requests made, by endpoint.

2.2.2 Database dashboard

1. **Amount of users:** Gauge charting user count.

2. **Amount of messages:** Gauge charting message count.

3. **Amount of follows:** Gauge charting count of follows between users.

4. **Distribution of users by message count:** Bar chart showing count of users by number of messages.

5. **Distribution of users by following count:** Bar chart showing count of users by how many they follow.

2.3 Logging

In the figure below, you can see how our ELKB stack is structured to ship logs to Elasticsearch and further analyze them.

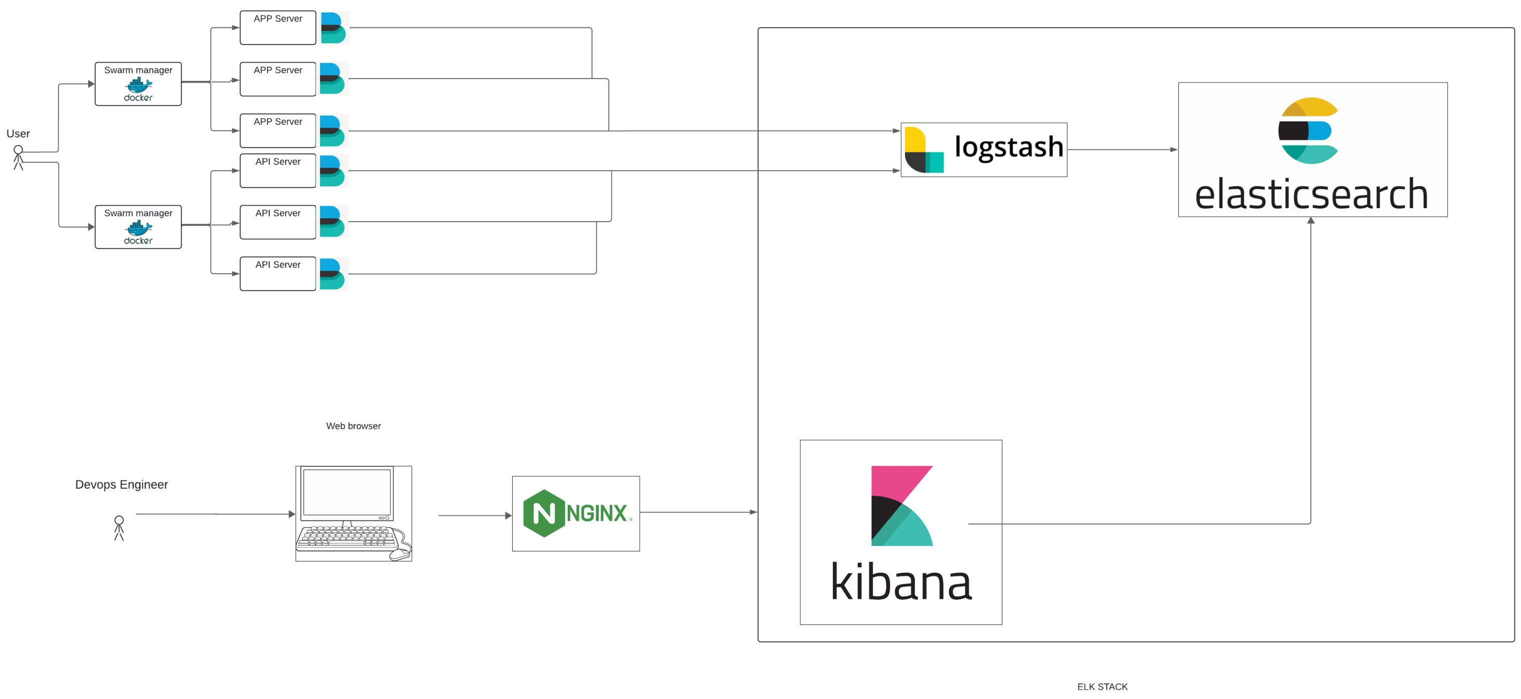
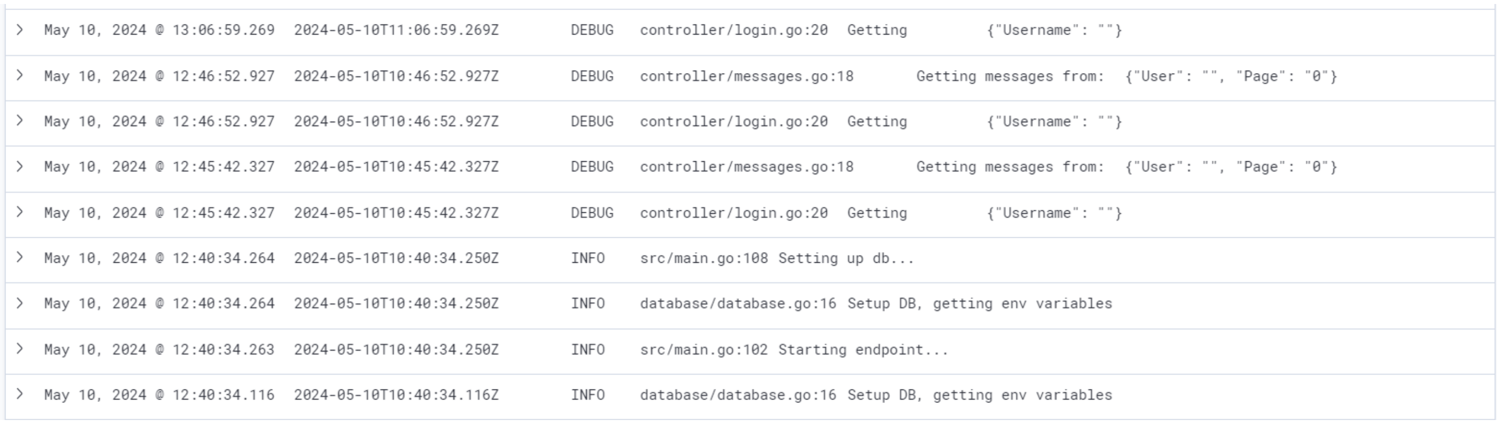


Figure 2.1: ELK stack

What enables the Filebeat to ship logs is a label tag added to our "app" and "api" services which enables Filebeat to autodiscover logs from those containers which has that option enabled. Filebeat will collect logs only containing a timestamp, which are those generated by the logger of our code, and ship them to Logstash where they will be further processed to be outputted to Elasticsearch. Finally, and Devops engineer can analyze and visualize this logs through Kibana through their browser, where the stack will be reversed proxied by Nginx for authentication reasons.

What we log in our application is the main events that occurs, we mainly use 4 levels for this purpose: info, debug, error and fatal. We used "info" level to communicate the events in a general level of how things are happening; "debug" for debugging purposes, how parameters are being changed and passed; "error" when the program encounters some non-fatal errors for example in our Go app when the "err" parameter is not nil we notify it with the error level; and finally "fatal" is used for errors that halts or crashes the application.

All logs are collected in a single index pattern, from all sources from the "app" and "application. We can distinguish the sources in Kibana because you can see from which folder and which file is the log generated from. Please, see image below:



```
> May 10, 2024 @ 13:06:59.269  2024-05-10T11:06:59.269Z  DEBUG  controller/login.go:20  Getting  {"Username": ""}
> May 10, 2024 @ 12:46:52.927  2024-05-10T10:46:52.927Z  DEBUG  controller/messages.go:18  Getting messages from:  {"User": "", "Page": "0"}
> May 10, 2024 @ 12:46:52.927  2024-05-10T10:46:52.927Z  DEBUG  controller/login.go:20  Getting  {"Username": ""}
> May 10, 2024 @ 12:45:42.327  2024-05-10T10:45:42.327Z  DEBUG  controller/messages.go:18  Getting messages from:  {"User": "", "Page": "0"}
> May 10, 2024 @ 12:45:42.327  2024-05-10T10:45:42.327Z  DEBUG  controller/login.go:20  Getting  {"Username": ""}
> May 10, 2024 @ 12:40:34.264  2024-05-10T10:40:34.250Z  INFO   src/main.go:108 Setting up db...
> May 10, 2024 @ 12:40:34.264  2024-05-10T10:40:34.250Z  INFO   database/database.go:16 Setup DB, getting env variables
> May 10, 2024 @ 12:40:34.263  2024-05-10T10:40:34.250Z  INFO   src/main.go:102 Starting endpoint...
> May 10, 2024 @ 12:40:34.116  2024-05-10T10:40:34.116Z  INFO   database/database.go:16 Setup DB, getting env variables
```

Figure 2.2: Logs from Kibana

2.4 Security assessment

2.5 Scaling strategy

3 Lessons Learned Perspective

3.1 Evolution and refactoring

3.2 Operation

3.3 Maintenance

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec venenatis enim a nulla molestie, ac feugiat justo egestas. Nullam interdum lorem et neque ullamcorper volutpat sed sit amet tortor. Praesent sit amet aliquet risus, et accumsan mi. Sed facilisis condimentum varius. Praesent et nunc cursus, laoreet nisi a, venenatis nisl. Integer diam magna, iaculis at dapibus et, rutrum in leo. Pellentesque feugiat diam felis, quis ornare eros dignissim et. Maecenas sed laoreet nunc. Morbi porttitor massa id dui aliquam, non ultrices dolor malesuada.

Cras et porta ex. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam erat volutpat. Phasellus eget ipsum sit amet nulla porttitor rhoncus at eget elit. Aenean vulputate, urna sed lacinia luctus, sem nisi iaculis nunc, vitae mattis neque sem in felis. Sed tempor tincidunt dapibus. Morbi porta ex erat, sed ornare mi gravida nec. Phasellus ut nunc venenatis, mollis est vestibulum, laoreet nibh. Mauris in vulputate diam. Nunc ullamcorper vestibulum velit, eget volutpat leo vulputate at. Nam in tortor id dolor elementum lobortis at ut elit. Nulla in interdum mi. Sed aliquam ullamcorper blandit.

4 Appendix