IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance: Exam Project

Group C	
Mathias Gleitze Hoffmann	mglh@itu.dk
Martin Lupa Groppelli	mlup@itu.dk
Anne Sofie Rasmussen	soad@itu.dk
Cecilia Maria Fröhlich	cefr@itu.dk
Tomas Federico Maseda	tofm@itu.dk
Eleonora Borovic	eleb@itu.dk

We have received the source code for a miniature version of X (formerly known as Twitter) in Python, for us to rewrite using another programming language. From there we have been asked to containerize our application, create a database, set up infrastructure using code, add a CI/CD chain, add logging and monitoring, and add container orchestration to enable our application to run in a distributed manner.

KSDSESM1KU
Spring, 2024

Contents

1 Systems

1.1 Design of the CSharp-MiniTwit application

We have decided to convert the given source code into an application called 'Csharp-Minitwit' written in C#. We have chosen C# for its well documented ecosystem and robustness, due to it being statically and strongly typed, and we have chosen to build our app using the NET 8.0 runtime, as it is the latest long-term supported release (end of support Nov. 2026)[3].

The application is built using the ASP.NET Core framework[4], with razor pages for rendering HTML pages, and we are following a Model-View-Controller (MVC) pattern and dependency injection, for easier maintainability and testability. The application itself requires a database for storing users, messages, followers and metadata. For this we have chosen to connect to a PostgreSQL using Entity Framework Core (an ORM), making subsequent database alterations and migrations easy.

INSERT PICTURE - NOT WORKING RIGHT NOW

Below we will zoom in on some of the important aspects of the system.

Versioning and CI/CD

The application is versioned, tested, quality assured and deployed using GitHub. Our group has been following the Gitflow workflow to avoid merge conflicts, and ensure tractability. Gitflow also allows us to have a 'develop' and a 'main' branch, enabling us to easily set up a staging server, for integration testing.
Our GitHub repository is also used for project management, creating issues and documentation. Deploying our application to our servers is done using GitHub actions, creating Docker images and pushing them to the relevant environment. GitHub actions also runs unit tests, formats our source code using 'dotnet format' and analyzes it using SonarCloud and Code Climate.

GitLab would have been a good alternative, however the whole group already have GitHub accounts and use it for personal projects — GitLab supports all the same features but is known for having a stronger focus on DevOps and CI/CD[5]. GitLab allows for free self-hosting.

Cloud provider

We have chosen DigitalOcean as our cloud provider, running our application on provisioned virtual machines that we manage our selves. DigitalOcean has been chosen for the sole reason that they offered $200 in credits for students — enough for the entire project.

Since we are not using managed application services, pretty much any cloud provider would suit our needs. Had we gone with managed services, Azure would have been a good alternative due to its integrations for ASP.Net.

Provisioning

For provisioning we use Terraform, which has been chosen specifically for it being declarative. The Terraform scripts are run using the 'bootstrap.sh' script, which takes a single parameter; 'production' or 'staging'. This in turn provisions our entire system (except the database) and pushes the corresponding docker image to the Docker Swarm.

Database

As mentioned, the application uses a PostgreSQL database. We have opted to go with a managed database, for simplicity and to delegate reliability issues to DigitalOcean. A single vertically scaled database will be able to handle many concurrent users. If we needed to scale up substantiably, we would likely have to switch to a specialized distributed database system like Kafka, which the real X uses[6].

Monitoring and logging

We use Prometheus in a pull-based configuration for business monitoring. This was implemented as part of the exercises and has been kept for that reason. We also use DigitalOcean's website for infrastructure monitoring, displaying CPU, memory and disk usage. We have no alarms / emails set up.

For logging we use Loki, which is inspired by Prometheus and chosen for that same reason[2].

Grafana is used to display Prometheus data as a dashboard, and to query the logs from Loki.

Containerization and container orchestration

The application is built as a Docker image through the CI/CD chain. From there it is deployed to an instance of Docker Swarm, using a compose file that connects our application with a Prometheus container. This setup has 2 replicas running on the swarm, more could be appliued, to scale the application if there is an increase in requests.
The Grafana server is setup using standard Grafana, Prometheus and Loki images,

composed together in a `docker-compose.yml`.

The Docker Swarm is setup with 1 leader node, 2 manager nodes and 1 worker node. We only have a single worker node, as we have hit a max number of VMs, and we want to have both a staging and a production environment. To enable the leader node to crash and its role be taken by another node, it is required to have 2 manager nodes.

1.2 Infrastructure

Application Dependencies

Dependencies are handled using dotnet's package manager, NuGet:

- Npgsql.EntityFrameworkCore.PostgreSQL
- OpenTelemetry.Exporter.Prometheus.AspNetCore (exports telemetry data to Prometheus)
- OpenTelemetry.Extensions.Hosting
- Microsoft.EntityFrameworkCore (ORM)
- Serilog (structured logging library)
- Swashbuckle.AspNetCore (auto-generated Swagger documentation)

Technologies used

A quick overview of the technologies used:

- **GitHub**: Code repository, versioning and project management.
- **SonarCloud**: ensuring quality in the code.
- **Digital Ocean**: hosting of virtual servers, databases and volumes.
- **Terraform**: provisioning and management of infrastructure on Digital Ocean.
- **Docker**: containerization of the application for deployment.
- **Prometheus**: monitoring system and time series database that collects metrics.
- **Loki**: log aggregation system integrated with Grafana for viewing logs.
- **Grafana**: visualization of metrics and logs.

put in same order as sub-sub-sections below

1.3 Interactions of subsystems

APIController/HomeController - ORM - Database

Incoming requests to the application are handled by either the `APIController` or the `HomeController`. Both provide similar endpoint logic but are meant to be used by the simulator and regular users respectively. A request's path will match a specific endpoint, prompting the application to perform a corresponding database operation. For instance a POST request to */login* will invoke the `GetByUsername` method from the `userRepository` and request the user information through an SQL SELECT query. The ORM will map the results back to C# objects, which the controller returns in the HTTP response to the user.

Refer to Figure 4 in 5.1 for a sequence diagram showing how a user successfully follow another user and gets a response from the system. See Figure 5 in 5.1 for a sequence diagram illustrating how a request from the simulator is handled in the system.

1.4 Current state of the systems

2 Processes

2.1 Interaction as developers

2.2 Tools and stages included in CI/CD pipelines

Our CI/CD pipelines are implemented using Github Actions. Our workflows are splitted in the following:

quality.yml

Triggered on pull request actions (opened, reopened, edited, synchronize, and ready for review). The main purpose is to run quality checks over the changes existing on the pull request branch.

- .NET SDK (for running dotnet commands)
- Hadolint (for Dockerfile linting)
- SonarQube (for code quality and security analysis)

release.yml

Triggered on pushes to `main` branch. It uses a base `release.config.js` file at the root of the repository and it is meant to automate the generation of releases and its changelogs through the use of conventional commit messages.

- Node.js (for running npm and semantic-release)
- Semantic Release (for versioning and changelog management)

deploy.yml

Triggered on pushes to `develop` and `main`. It internally checks the targeted environment (`staging` or `prod`) retrieving the corresponding secrets,pushing the application image to Dockerhub and deploying to Digital Ocean.

- Docker (login, build, push actions)
- SSH (for deploying to servers)

2.3 Organization of the repository

The repository is organised in two main folders: csharp-minitwit and infrastructure. The first contains all application specific logic, while the second holds all infrastructure related configuration files.

By organising the repository in this manner, there is a clear separation of concerns, making it easier to manage the project. Developers concerning with the development of the application do not need to understand the intricacies of the infrastructure and vice versa.

csharp-minitwit

Main folders and files explained:

- Controllers: APIController and HomeController handle all the logic for each endpoint.
- Databases: contains the `schema.sql` and `minitwit.db` files used to initialise the database in the development environment.
- Metrics: contain all metrics logic and configuration.
- Middlewares: currently only a CatchAllMiddleware required to catch incoming requests and record metrics.
- Models: these represent the way the data is shaped in the application.
- Services: include the business logic and service layer components. ORM logic is reflected under Repositories.
- Views: templates representing the user interface.

infrastructure

Main folders and files explained:

- archive
- grafana
- scripts
- ssh_key
- stack

2.4 Applied branching strategy

We have decided to use Gitflow branching model[1]. In this model the repository holds two main branches:

- `main`: represents the main branch where the source code of HEAD is always production-ready.
- `develop`: it serves as an integration branch where source code of HEAD always reflects a state with the latest delivered development changes for the next release.

Next to these main branches there are some other supporting branches, being `feature` the most widely used. feature branches are meant to be used to develop new features for future releases. They usually branch off and back to develop.

2.5 Applied development process and tools supporting it

To organise the development process and make work visible we decided to capitalise Github's built in Project tab, which provides an adaptable spreadsheet for tracking work, and which also integrates with Issues and Pull Requests. This tool provides a comprehensive visibility to the current state of each ticket and an easy-to-understand interface for developers and stakeholders.

A new Issue is created to document a new requirement and is automatically shown in the Project board as a "Todo" item. When a team member is assigned to such issue, it is moved into the "In Progress" state. The development phase takes part, and when the issue is closed, it is automatically moved to the "Done" status in the board. Closing an issue as "not planned" takes the ticket to the "Aborted" status.

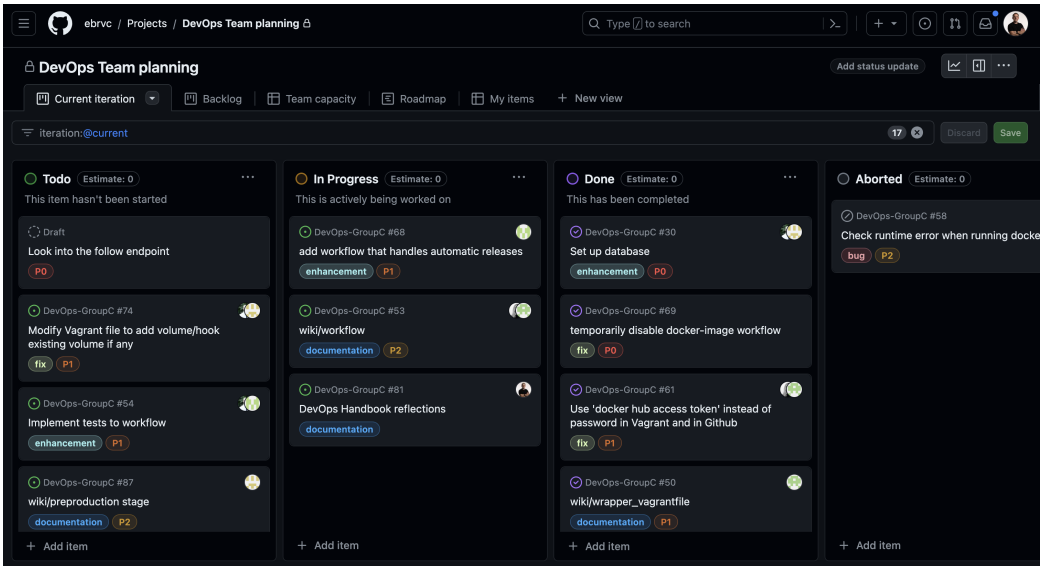

Figure 2: Github's Project tracking tool

2.6 Monitoring and logging

Monitoring is done through a self hosted image of Grafana using a Digital Ocean Droplet that receives the metrics data from Prometheus servers attached to the application droplets. Moreover, logging is configured through a log aggregation

system called Loki, which acts as a Grafana datasource. These logs are pushed directly to Loki by using Serilog, a logging library for .NET applications.

Monitoring the business

These metrics are registered by querying the database on some specific indicators, such as:

- Messages registered (application usage)
- Users registered (conversion)
- Follower registrations (users interaction level)

Application monitoring

- Rate of HTTP requests received per endpoint
- Total number of requests (last 24hs)
- Total count of errors per status code (last 24hs).
- Top 10 unhandled exception endpoints.
- Top 10 Requested endpoints (API).

2.7 Application logging

The reason why we have chosen Loki as a logging mechanism is due to its inherent simpler architecture. It requires less computational power and memory, this aligns perfectly with our MiniTwit application. Below, we present the log events throughout our application:

- Message posting
- User logging / Failure logging / Invalid username or password
- User logout
- User registering / Failure registering
- API requests that are not from the simulator
- API registering / Failure registering
- API message posting from users / Failure or incorrect usage of endpoint
- API retrieval of messages for a specific user / Failure in the retrieval
- API retrieval of followers for a specific user / Failure in the retrieval
- API follow/unfollow requests / Failure in the execution

Infrastructure monitoring

Even though we haven't set this monitoring ourselves, Digital Ocean provides some out of the box monitoring for its droplets, such as CPU usage, memory usage, DISK I/O, Disk Usage, Bandwith, etc.

Describe how we aggregate logs

2.8 Security assessment

Brief results of the security assessment.

2.9 Scaling and upgrades

Because we monitor our system we can see when there would be a need to scale our system up or down. Our strategy for scaling our system is to change the number of nodes in the terraform file minitwit_swarm_cluster.tf and run the bootstrap.sh script. Because the state of our system is kept in the Spaces Object Storage in DigitalOcean it would allow for a quick scaling of the number of nodes.

Describe our strategy for upgrades

3 Security Assessment

We have the following assets:

- Web application
- Database
- CI/CD
- Image repository
- Container orchestration
- Secrets

3.1 Threat sources

Web application: Our current web application runs on HTTP, leaking data (especially when signing up) and making our application vulnerable to replay attacks. All dependencies for the application are Microsoft's own packages. Messages are rendered in HTML, it is possible that this somehow allows for cross-site scripting (XSS).

Database: In the database the username and email is in clear-text, passwords are salted and hashed to defend against rainbow-table attacks. The user ID is an integer, incremented on each sign-up, this is not best practice. To connect to the database, we use the same user and password for both our staging and production environments.

CI/CD: GitHub actions has access to all of our secrets. We use the following NPM packages for versioning our releases:

- "semantic-release-github-actions-tags": "^1.0.3"
- "semantic-release/git": "^8.0.0"

Our images are stored publicly on DockerHub. On our staging server, we don't use tokens for logging into DockerHub, instead the credentials are stored in the environment, in clear text. Our codebase is public on GitHub.

Other: We have a lot of open ports on our servers, possibly also for our logging endpoint.

3.2 Risk scenarios

Scenario	Probability	Impact	Risk	Strategy / Action
An adversary listens to the non-encrypted data sent on our web application. This is possible because we use HTTP and not HTTPS	5	5	25	Get an SSL-certificate and redirect all traffic to HTTPS
An adversary finds an open port on one of our nodes and gains access to our system	3	5	15	Run Nmap, close unused ports
An adversary constructs a message that escapes HTML and runs code when rendered (XSS)	3	5	15	Sanitize user input
An adversary finds our Docker images online	3	1	3	Make image repository private
An adversary hijacks the cookie of a user	1	2	2	Unavoidable

brief description of how we harden the security of our system based on the analysis

4 Lessons learned

4.1 Monitoring - Performance improvements

By analyzing metrics, specifically ”average request duration by endpoint”, and incorporating research conducted by one of the team members, we successfully reduced latency across our endpoints. This improvement is evident in the screenshot below:

Figure 3: Significant reduction of latency metrics measured in Grafana

The primary cause of the high latency was the geographical separation between our Database (hosted in the NYC region) and our application server (in the FRA region).

Issue link:

4.2 Monitoring - Bug fixing

The follow/unfollow endpoint was returning a 200(OK) status code, instead of 204(NO CONTENT) as requested by the simulator API, resulting in cumulative errors. It was possible to detect that given the information provided by the http://206.81.24.116/status.html dashboard.
Issue: Grafana and Loki memory issues In addressing the disk space overloading issues on our Grafana server where we hosted Loki as well, we utilized DigitalOcean's recovery console to identify and resolve the root cause. By executing the command $sudolsof + L1$, we detected files that had been deleted but were still held open by processes, which in turn occupied disk space. We terminated these processes using $sudokill - HUP[PID]$, which released the disk space and restored server functionality, allowing us to regain access to the Grafana platform and establish SSH connecitons to the server. This intervention resolved the memory issue and retored normal operations.

Issue link:

Slow response on the /public endpoint ... ??

Issue link: Look into adding indices to our database #141

5 Appendix

5.1 Systems architecture

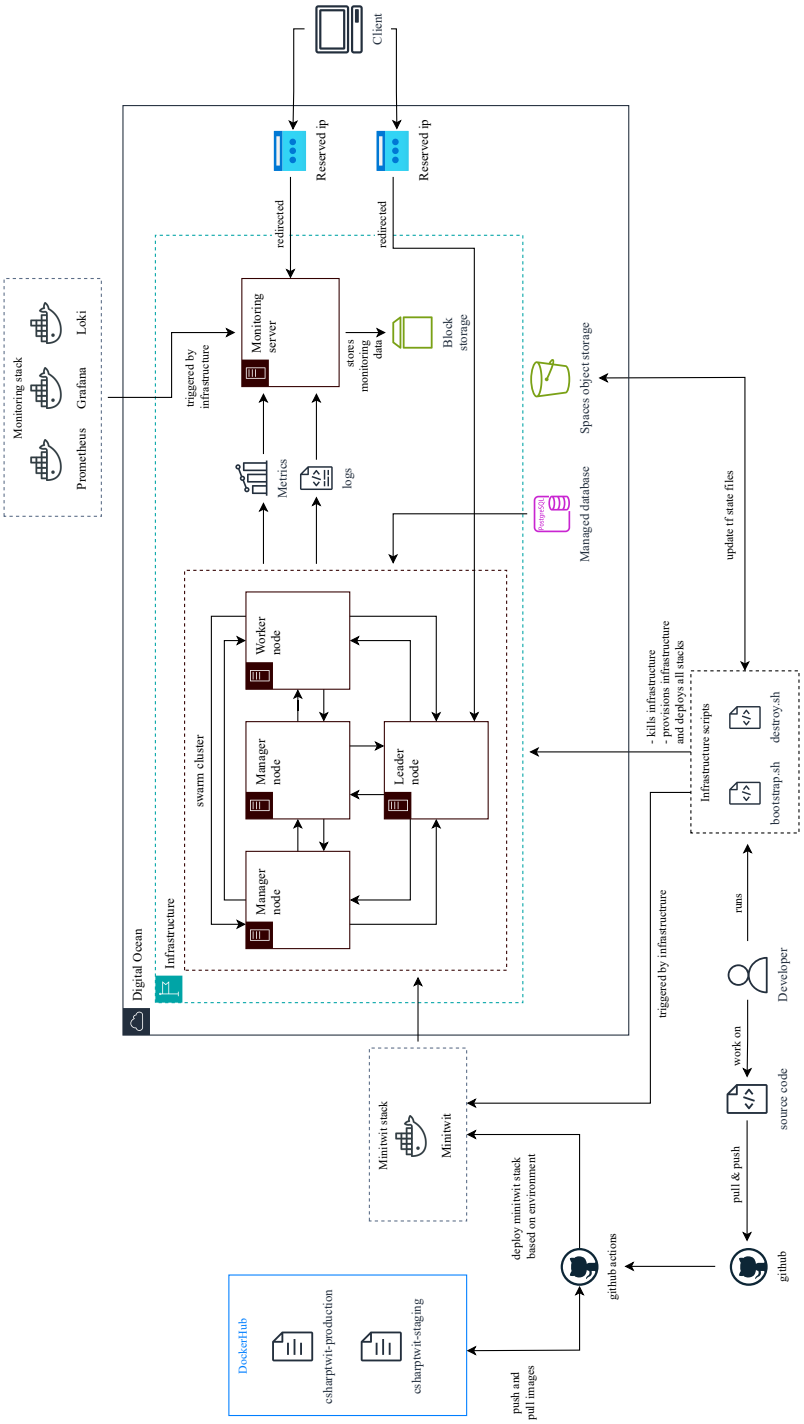

Figure 1: System Architecture Diagram

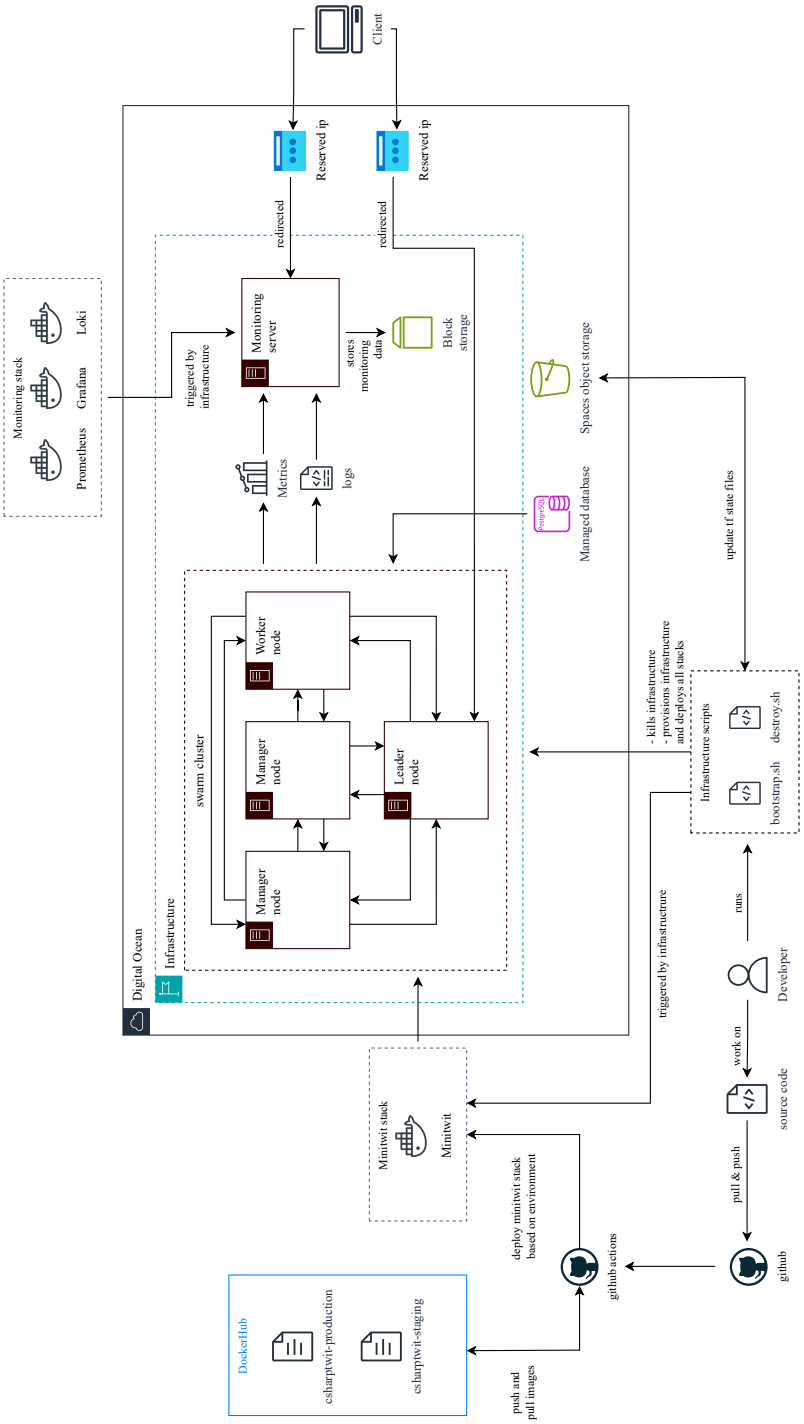

Figure 4: Systems architecture diagram

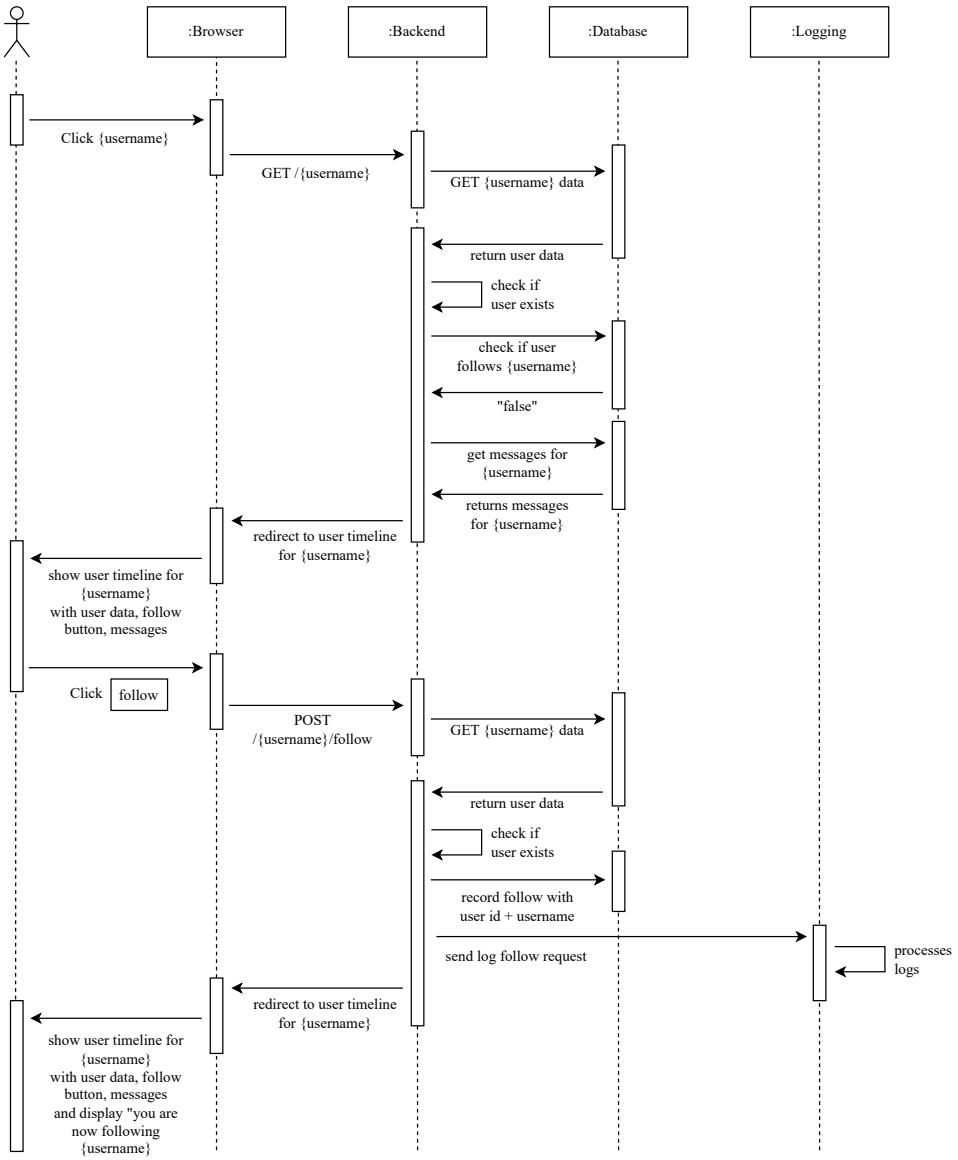

Figure 5: Sequence diagram of a successful follow scenario.