DevOps – Project Rapport

Group e - Soufflé

May 23, 2023

When a french person says "Bonjour" and you replie with "Gracias"
Mr. Worldwide

Course: DevOps
Course ID: BSDSESM1KU
Course manager: Helge Pfeiffer (Niceness)

Group members:

Name:	**E-mail:**
Asger Clement Nebelong Lysdahl	asly@itu.dk
Laurits Munk Kure	laku@itu.dk
Patrick Bohn Mathiassen	pmat@itu.dk
Rasmus Overgaard Olesen	raoo@itu.dk

Contents

1 System's Perspective

1.1 MiniTwit Application

We chose to rewrite the MiniTwit application to C#, which has a nice web framework, Blazor WebAssembly, that allows us to write all our code using the same language. Blazor additionally runs as a single-page application which is faster at loading, partly because data can be fetched in the background and because full-page reloads are rare. One downside of WebAssembly is the long initial load of the application, which hopefully can be reduced or removed with future versions of .NET. Our application depends on a few NuGet packages to communicate with the rest of the system.

Identity Server	An easy and secure way of managing users.
Prometheus-net	Sets up the /metrics endpoints for Prometheus.
Serilog.Sinks.ElasticSearch	Gives Serilog a log sink for ElasticSearch, letting it send logs directly to ElasticSearch
EF Core	O/RM for communicating with the database

We aimed for our system to have a simple and maintainable architecture, by utilizing as few different tools as possible and by using software architectural patterns. The patterns we chose to strive for have been Clean Architecture, Repository Pattern, and Client-Server. We use several other patterns too, but we don't consider them something we strive for, as they are simply a byproduct of the tools and framework we use.

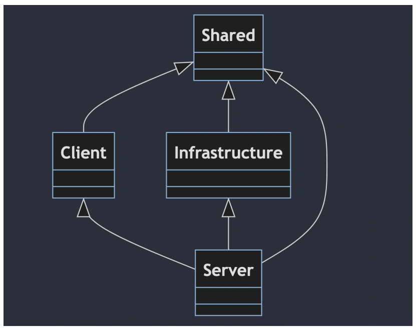

Figure 1: Project Architecture

`Clean architecture` gives us separation of concern and greater modularity. We achieve this by having our code split into 4 C# projects. One project, `Shared`, is for our data transfer objects (`DTO's`) and repository interfaces, so we will be able to mock our repositories without a dependency on them. A second project, `Infrastructure`, is for the repositories, database context, and models for the database schema, where the repositories depend on the `Shared` project. The third project, `Client`, contains the `Blazor` files, responsible for the presentation layer and user interface. Finally, the fourth project, `Server`, is the actual server that contains the API controllers, serves the frontend, and the application configuration. The controllers have a dependency on `Infrastructure` and `Shared`, as the controllers need the repositories and the DTO's. For the `Server` to serve the frontend it has a dependency on the `Client`. Thus none of our projects are mutually dependent and the core is the `Shared` project, from where it aggravates outward to a layer containing the `Infrastructure` and `Client`. Lastly a layer with the `Server` as the part that other systems can interact with.

We use a `Repository pattern` to add abstraction around our communication with the database. Even though we use EF core which some would say is enough abstraction, then repositories allow us to make a switch over to use something like `Dapper` instead of `EF Core`.

1.1.1 API request route

The route of data in our application starts with the user. When a user sends a request to the server, then it needs to pass through a few places before coming back to the user. Below is a sequence diagram illustrating the journey of an arbitrary request along its designated 'route'.

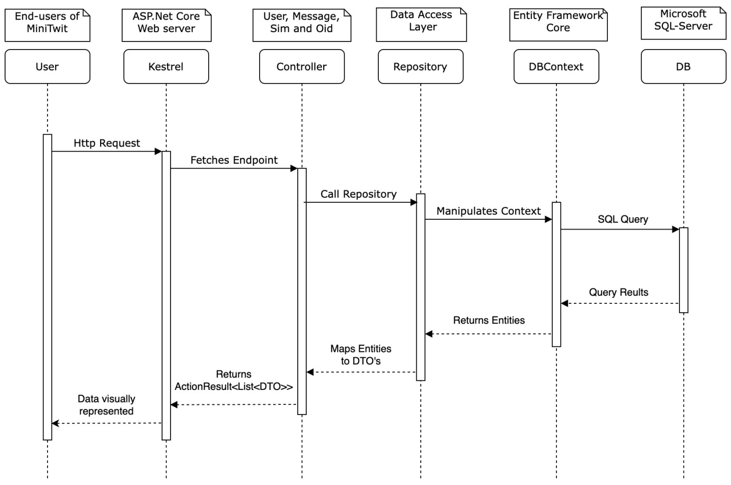

Figure 2: Data journey

The route starts with the user sending a request to the server. The server, Kestrel, receives the request and sends it to the controller method that matches the endpoint the request targeted. The controller then uses one of the repositories, that asks EF Core to format a query and send it to the database. When the database returns data, it aggregates up the ladder and then back again to the user.

1.1.2 Database

For the server to communicate with the database we chose to use the popular open-source O/RM Entity Framework Core, also known as EF Core. EF Core makes it possible to define the database schemas and maintain the database state. Additionally, it removes the type barriers between code and database while making it possible to query the database without writing SQL, which blocks SQL injection.

1.1.3 Dependencies

At a low level of abstraction, our application has specific dependencies such as libraries and packages. These dependencies are internally within our program and are imported and installed using dotnet as NuGet package-manager. A visual representation of package dependencies for our application can be seen in figure 3.

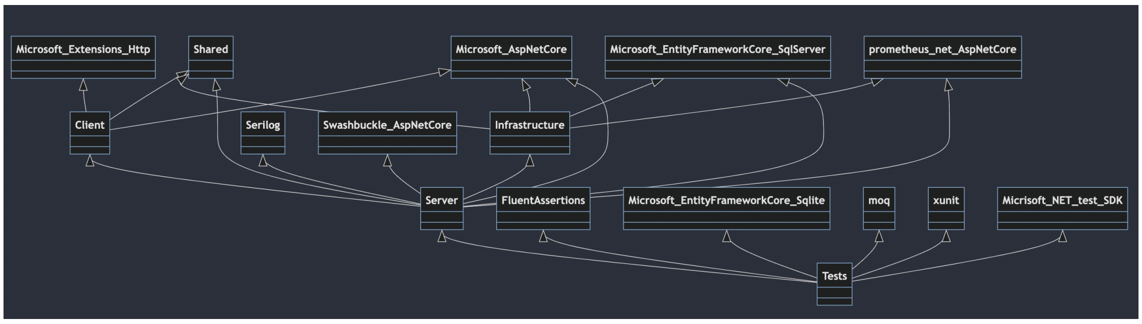

Figure 3: Package dependency graph for C# application

1.2 Tools & Expanded System Architecture

This section briefly describes the different tools the system utilizes, and for what purpose. Most are talked about in further detail in the Proces' Perspective section. Figure 4 shows an overview of the infrastructure and dependencies of the system and its tools.

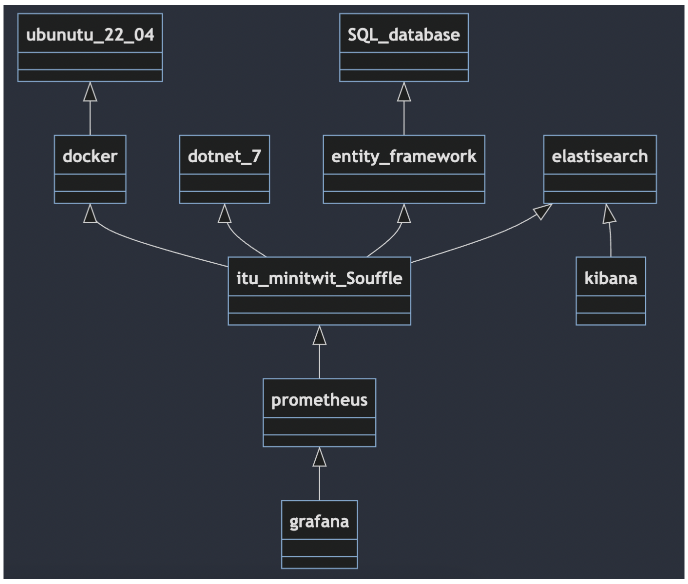

Figure 4: Infrastructure dependency graph

1.2.1 .NET 7

This is the platform our project is built on. We chose this runtime since it is the one we are most familiar with and it contains a lot of libraries that suits our needs.

1.2.2 Docker & Ubuntu 22.04

Docker is responsible for building, shipping, and running our application through containers. It makes it possible to have complete control of the environment in which the programs must run (we use Ubuntu 22.04). This ensures compatibility and portability, all it needs to run is docker engine. Docker Swarm also enables scaling of the system within virtual machines.

1.2.3 Digital Ocean

As recommended, Digital Ocean is our chosen provider of cloud infrastructure. It is user-friendly and makes controlling our domain name simple by allowing it to be routed through Digital Oceans NS records. However, as providers go, it is expensive. We had a limited amount of free credits, but we were restricted in the number of virtual machines we could afford. Our system has 2 droplets; one for our server, and one for all our tools. We would have preferred to separate our tools on individual droplets, potentially managed as a swarm for easy balancing, but it was not possible on our budget.

1.2.4 GitHub Actions

GitHub Actions allows us to integrate our workflows, and apply our tools to our branches before merging them together, ensuring a certain level of quality control. E.g. by integrating testing as a prerequisite before merging code changes.

1.2.5 Prometheus & Grafana

Prometheus is responsible for monitoring our system by periodically querying its specified metric endpoints for data. This data is then accessed by Grafana, which is the platform responsible for visualizing it on its dashboards. See 2.4.1 for more detailed information.

1.2.6 Superlinter & Snyk

Lint Code Base (SuperLinter) is a static analysis tool we use to identify and reduce possible code smells, such as duplicated code segments and long methods since these smells may affect the maintainability and readability of the code. The SuperLinter was chosen for its versatility, as it includes pre-configured linters for multiple languages and allows easy customization to align with our project's specific requirements.

Snyk is another static analysis tool we use due to its ability to identify and address security vulnerabilities. It scans open-source libraries and dependencies, ensuring our code is as robust and secure as possible.

1.2.7 ElasticSearch & Kibana

ElasticSearch is a search and analytics engine that enables accurate, and near real-time searching of data, allowing rapid data retrieval and analysis, which makes it perfect for reading aggregated system logs. Though it is scalable and distributable, we only run one instance of it, even though it means we lose the redundancy. Kibana on the other hand is a data visualization and exploration tool that allows for the creation of dashboards to analyze and present data.

Together these tools store, search, analyze, and visualize log data in our MiniTwit system. See 2.4.2 for more details.

1.3 Reflection of System State

DevOps is a crammed course. It has not been possible to implement everything. These are the tools the current system does not have, either because it was skipped, down-prioritized, or seemingly incompatible with our setup.

1.3.1 Terraform

This was never set up. We had a hard time understanding how the system should be set up and what parts of our infrastructure it should manage. Our rough understanding is that it could have made it easy to switch cloud providers and scale the number of machines the system should be distributed on.

1.3.2 NGINX in the Swarm

We have made a script that allows us to pass in a few addresses of VM's, and then have our Server deployed with an NGINX load balancer in front. We have had some issues with loading times when going through the load balancer to the servers though.

1.3.3 Sonarqube & Code Climate

We never set up SonarQube and CodeClimate in our project due to time limitations. Theoretically, these static analysis tools would detect bugs, vulnerabilities, and code smells in order to improve code quality. Since we already implemented the Superlinter and Snyk we decided to down-prioritize these tools.

1.4 License

We chose the GNU General Public License v.3.0 (GPL-3.0). As it is more restrictive in future third-party usage than the MIT license, while remaining compatible with our tools, and promoting collaboration with the development community.

- Docker, Snyk, Grafana and Prometheus is distributed under the Apache License 2.0, which is compatible with GPL-3.0.
- Kibana and ElasticSearch are licensed under the Elastic License, which has some limitations that we are in accordance with.
- .NET and Superlinter is under the MIT License, which is compatible with GPL-3.0.
- NGINX is available under the NGINX Open Source License, which is compatible with GPL-3.0.
- Ubuntu is itself released under GPL.
- Identity Server has an open and free license while we don't sell our software or our services.
- Terraform is under the MPL 2.0 license, which is compatible as long as we do not distribute their source code.

2 Process' Perspective

2.1 Collaborative Development & Team Organization

When we initially talked about teaming up we made sure to align our expectations. We agreed that each member expected to spend 8-12 hours a week on the course. Thus our way of working became something like this:

- Tuesday, 10-14: Project Work
- Tuesday, 14-16: Lecture
- Tuesday, 16-18: Exercises/Project Work
- Friday, 10-15: Project Work (When possible, not each Friday)

The above points were all physically present sessions for all members. Each member had the freedom to do additional work at other times during the week. All communication was through a Discord channel.

When working, we would at times do versions of pair programming. Teamwork was however difficult, since all concepts were new to everybody, and a lot of the understanding and progress came from trial and error. All sessions were an open forum where questions were welcome, and help was always possible to get. Sadly, we were not always good at remembering co-authorship on commits which might make the repository contributions look rather skewed.

We intended to utilize GitHub Issues to share the tasks between us, making sure not to be wasting time on duplicate work.

2.2 CI/CD Chain

In this project, we have implemented a CI/CD chain by defining a handful of `.yml` files that utilize GitHub Actions.

Throughout the development stage of our chain, every time we pushed to a feature branch (more on branch strategy in section 2.3), the Super-linter is run except for pushes to main. This isbecause we primarily interact with main through pull-requests. During the development and refactoring of workflows, however, we did permit direct pushes to the main branch. The reason behind this was twofold: firstly, setting up secrets and forking the repository proved to be excessively time-consuming, and secondly, a failed workflow would not disrupt the functionality of our main branch. We also had a daily workflow using Dependabot, that scanned for outdated dependencies and informed us of updates through pull requests. Before any pull requests were merged into main, another developer had to perform a code review as part of our workflow.

The first thing that happens when a pull request is created is that the application is built. Only building when a pull request is made to the main branch ensures multiple things:

- Changes are isolated
- Since code is reviewed before a pull request is merged into main, code that is not reviewed is never deployed
- Overhead is reduced, by not running unnecessary build cycles on each push to a feature branch

As part of our build stage, we incorporate an additional formatter/linter that operates on the build. This step is implemented to guarantee a greater level of code consistency. Our build workflow proceeds by checking out the branch, restoring the dependencies, building the application and executing the unit tests. This makes the test-stage dependant on the build-stage; if the build fails, the tests are never run. Depending on the test-suite, stability is ensured by making sure changes are tested, and old functionalities do not break (regression testing).

As part of our testing we also implemented a Snyks workflow. Snyks is a security platform designed for finding vulnerabilities in open-source dependencies and doing security scans on the repository.

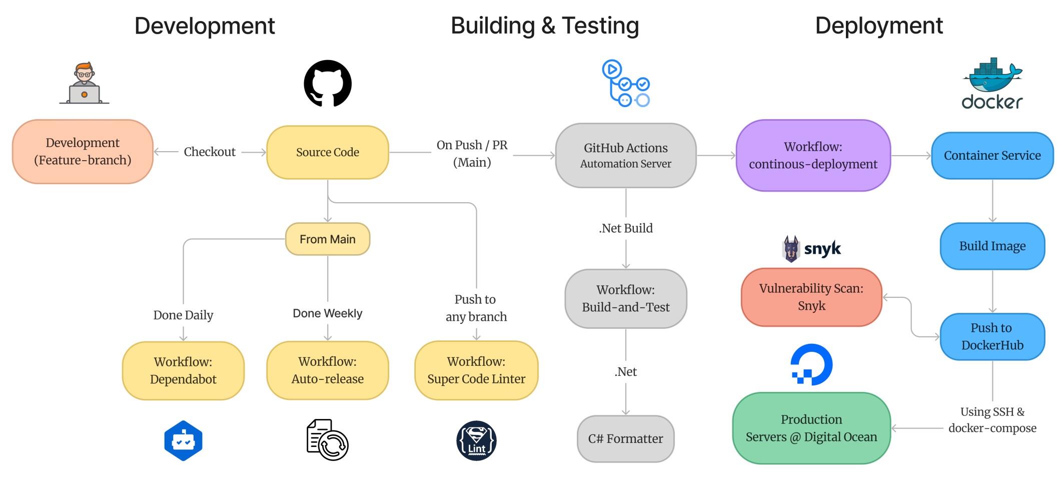

Figure 5: CI/CD tech-stack

Snyks is a great tool for notifying developers of any found critical vulnerabilities, that would pose a high threat to our application. Additionally, we used a Snyk feature on our code base, that scans the code base for sensitive data like connection strings, secret tokens, API keys, etc. This measure was taken in an effort to mitigate the risk of inadvertently exposing secrets.

After the build-and-test stage of the chain is run successfully, we move to the deployment stage. If either the building or testing fails, the pull request will be rejected.

We made a workflow, responsible for the continuous deployment aspect of the pipeline. Through this workflow, we have linked the building, testing, and deployment stages. Combined with the use of GitHub secrets, these workflows provide a powerful solution to automated deployment. Furthermore, it's a convenient way to manage our application's deployment process while maintaining confidentiality and flexibility. However, it does come with a drawback: Scalability is somewhat limited. One example of this is because the workflow uses repository secrets to reference the IP-addresses of our servers. If we had been a 'real' company that wanted to change our Server-provider to something other than Digital Ocean, or perhaps just looking to add more servers, we would have to add a separate paragraph to the workflow and update the repository secrets *manually*.

As an additional last step, we implemented an auto-release function to further streamline the deployment process. This step ensured that the latest changes were automatically released as requested by TA's and teachers. This was a scheduled event, done each Sunday at 20:00 CEST. Another workflow that has to do with automating release, is the Latex workflow. This pulls the current version of the report, each time a push to main is made.

2.3 Repository Organization & Development Strategy

2.3.1 Repository Organization

We structured our code in a mono-repository for two main reasons:

1. It provides improved collaboration, as all developers are working on the same codebase.

2. It simplifies version control as we only have a single history for all code.

Another consideration was, that we didn't feel the need to divide it up, as the application is still relatively small. It would also add what we deemed to be unnecessary complexity by requiring different repositories to be cloud-hosted for cross-access since the necessary references in our C# code would no longer reside in the same .NET solution.

2.3.2 Development Strategy

We utilized feature branching as our strategy as it has multiple advantages that work very well with the DevOps way of working. An example of this is continuous integration. The GitHub Actions make it simple to automate a require-to-pass test suite before any feature branch is merged into the main one. Other advantages include:

- It isolates changes to specific features, minimizing conflicts with separate work.
- It facilitates the code review process that in turn ensures quality assurance.
- It provides a clear separation of work, which makes collaboration easier.
- It makes parallel development possible.

Our tasks were organized in GitHub Issues. This made it possible for each member to always be able to see which tasks were up for grabs, and who were working on what. It also acted as a task backlog giving an overview of the work that was yet to be done.

As the project progressed, however, the tasks began including the whole group, and the Issues board became slightly irrelevant.

2.4 Monitoring & Logging

2.4.1 Monitoring

We monitor our system by having Prometheus gather metrics from our metric endpoints regularly. This is made possible by the NuGet package Prometheus.net, which is responsible for exposing a metrics endpoint in our API, which Prometheus can then send requests to. This data is then queried again by Grafana, which is responsible for visualizing it. We can access Grafana's dashboard through the user we set up. Our current metrics are:

- Http-requests (Duration)
- Total Users
- Total Messages
- Average request duration (Last 2 minutes)

Together, these metrics provide information about how much stress we can expect our system to experience, and what kind of request intensity we can expect. It also provides an overview of user growth, giving us a sense of the direction our service is headed.

Additionally, we are also making use of Digital Ocean to monitor our hardware such as the CPU and memory usage of our droplets.

2.4.2 Logging

To implement logging in our project, we used Serilog to do the actual logging, elastic search as the database to hold the logs and Kabana which serves as a UI to view them.

Serilog is a popular nuget package for structuring a logging system. In order to log we are making use of the Serilog.Sinks.Elasticsearch NuGet Package. In our application, we have specified some metrics that ElasticSearch

Kibana then retrieves the logging data from this sink and displays it. We are logging all exceptions, requests to the controllers, and different functionalities happening in the middleware.

skiftet default logger ud med serilog - andet format alle logs vi får er logs fra middleware default configuration logs -

Log messages are generated to capture important events, errors, or relevant information during runtime. These logs are collected and forwarded to ElasticSearch, that then indexes and stores the data. Kibana then connects to ElasticSearch we use - sink direkte fra .NET direkte ind til (serilog - logging framework). en sink, der hvor alle logs glider hen. Den sink er sat til elastik search hvor kibana kan vise os logs på en pæn måde. sender det videre til elastisearch direkte det er kun logs fra .Net så vi kan formattere det her og fortælle hvad vi gerne vil logge og hvordan det skal se ud direkte i serilog. der er formartering i serverens programfil, men bliver måske ikke brugt ved deployment. triggers i middleware. når vi modtager request bliver der lavet en log, når den er færdig med at processe - værktøjer der laver logs for os.

alle exceptions request til controllers masse andre ting der er I middleware -

2.5 Security Assessment

Our security assessment consisted of a penetration test - Zed Attack Proxy, or ZAP, that showed some flaws in our program. Most seemed to be relatively simple to fix, and none seemed to be too severe in nature:

- No Anti-CSRF tokens were found in a HTML submission form
- Passive (90022 - Application Error Disclosure)
- Content Security Policy (CSP) Header Not Set
- Missing Anti-clickjacking Header
- X-Content-Type-Options Header Missing
- Hidden File Found
- Vulnerable JS Library
- XSLT Injection might be possible.
- Cloud Metadata Potentially Exposed

These items resulted in GitHub Issues; some of which were dealt with while others were down-prioritized.

2.6 Scaling & Load Balancing

Our applied strategy for scaling and load balancing involved utilizing Docker Swarm for fault tolerance and easy container management, together with Nginx working primarily as a load balancer and reverse proxy. Docker Swarm uses a cluster of nodes (workers and managers) and allows us to automatically adjust the number of running Docker containers based on the current workload. Another important feature of the swarm is that it automatically evenly distributes incoming requests across the worker containers running the same service (internal load balancing with an ingress mesh).

The swarm has a maximum capacity limited to the combined resources (CPU, memory, disk space) of the server nodes within the swarm clusters. We used two servers for our application swarm and one server for the load balancing swarm.

The first swarm consists of a server with manager nodes, and a server with worker nodes running containers with our MiniTwit application. The second swarm includes a single manager node responsible for load balancing and reverse-proxying with Nginx. We designed it this way to separate concerns and allow the two swarms to scale independently. Combining Docker Swarm's automatic and dynamic scaling with Nginx's reverse-proxy capabilities resulted in an improvement in the system's availability and reliability.

By having Nginx handle and redirect request to our service, we could eleminate the 'single point of failure', and decrease the impact a server-crash has on our system. In case the primary replica-node running our swarm was flooded or broke down, we could redirect the trafic from that server-node to another upstream server, while docker swarms dynamical scaling would ensure that the system would be able to keep up with accelerating request and users.

However, in our setup, there were a couple of things that could be improved. Initially, we only tested our system using a single instance of Nginx, which essentially only shifted the singel point of failure from the application service to the load-balancer in the Swarm. This decision involved weighing the trade-offs between reliability, complexity and also the financial costs of creating more droplets. Also, we encountered some non-negligible internal network delays that affected the system's performance. When Nginx was responsible for the load balanceing, request took a long time to complete or re-direct. Due to the way Nginx was implemented it didn't function as originally intended.

2.7 AI Assistance

As developers, we have embraced the tool that is AI. We have used ChatGPT as a sparring partner whenever we got a task we did not know how to solve, or when we got stuck. It has been mostly great at providing ideas or starting points, but it has probably been at its best when troubleshooting or debugging. If nothing else it has provided the services of a rubber duck.

We have also used it in the formulation of documents like the SLA agreement, as it is mostly boilerplate text anyways. We provided it with some information about our system and what guarantees we could give, and it returned a rough draft we could finalize.

Finally, one team member has been using GitHub Copilot as an advanced intellisense tool, but not for making several lines of code. Meaning that it was used to finish individual lines of code, but not for making entire methods or blocks of code.

3 Lessons Learned Perspective:

3.1 Evolution & Refactoring

3.2 Operation

3.3 Maintenance

4 Appendix

4.1 Diagrams Baby

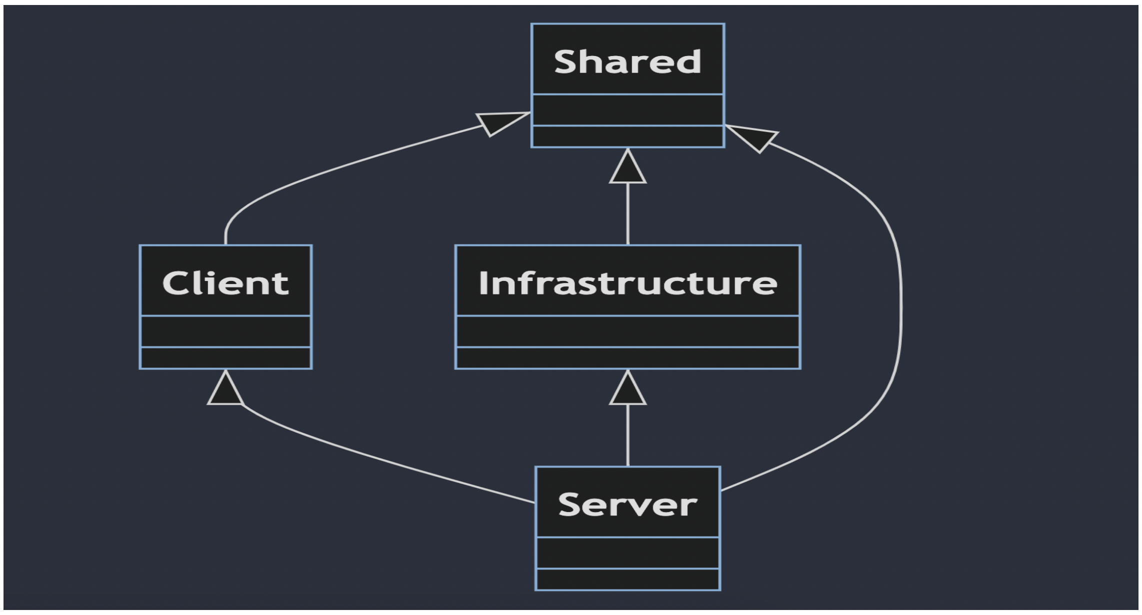

Figure 6: Larger version of the project dependency graph

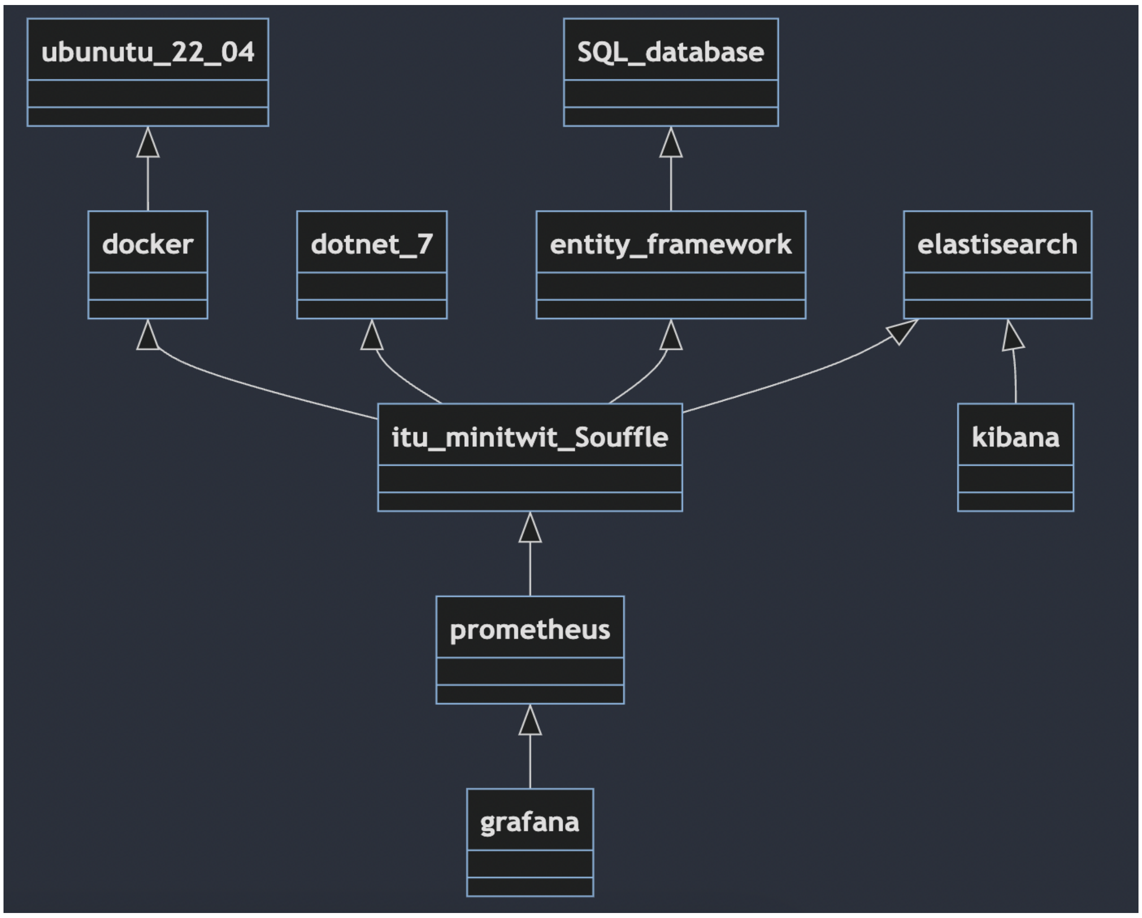

Figure 7: Larger version of the application dependency graph

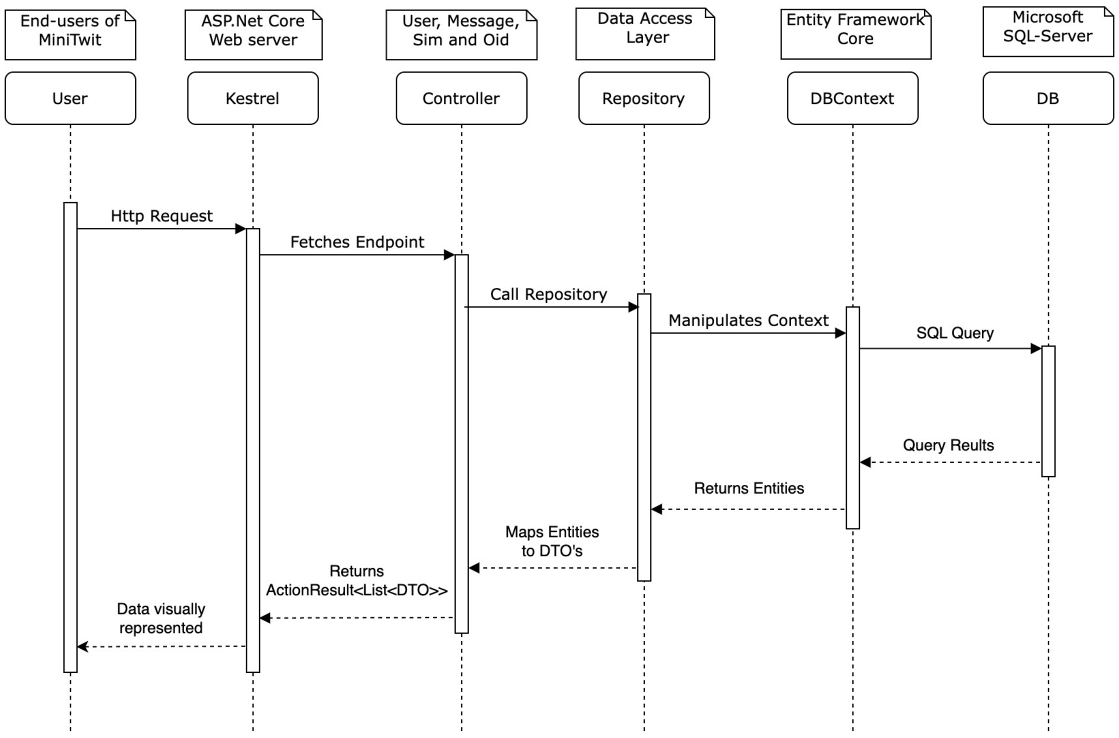

Figure 8: Larger version of API sequence diagram

4.2 memes