Final Report - DevOps

EvilTwitter

Group E

abea, beba, gujo, luka, sena

IT University of Copenhagen
Denmark

26 - 4 - 2021

Contents

1 System's Perspective

1.1 Architecture of the system

This section will start with an overall walk through of the overall structure, followed by a further explanation of the various parts.

For the overall structure of the application a Three Layered Architecture was chosen, As this resulted in a nice separation of concern in the application. This resulted in three distinct applications namely, EvilClient for the presentation tier, EvilApi for the business tier logic tier and the PostgreSQL (PSQL) database cluster for the data tier.

To give an overview of this, see figure 1 that contains a model diagram depicting said flow via its dependencies.

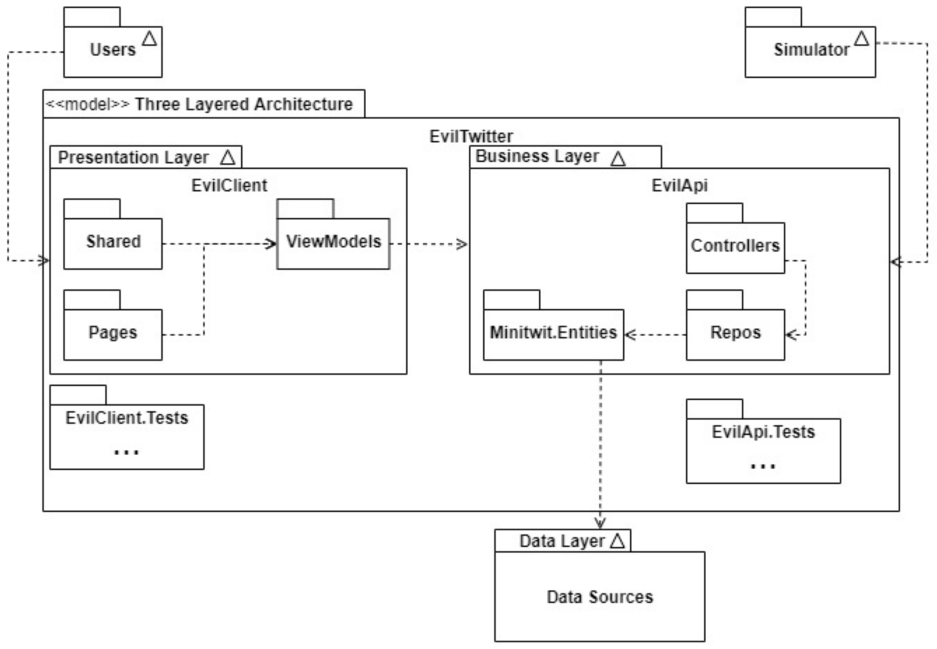


Figure 1: Model diagram of the EvilTwitter project, depicting important folders of the source code. Arrows have been added to show flow of data in the system, starting at the user or simulator and pointing to the next source file that would further the request along, resulting in all arrows leading to the database at the bottom

By following the Three Layered Architecture only two paths of communication exists, where both use a different architecture. First is between EvilClient and EcilApi, the second is between EvilApi and the database. This flow can be seen in figure 1 which is also depicted in figure 2.

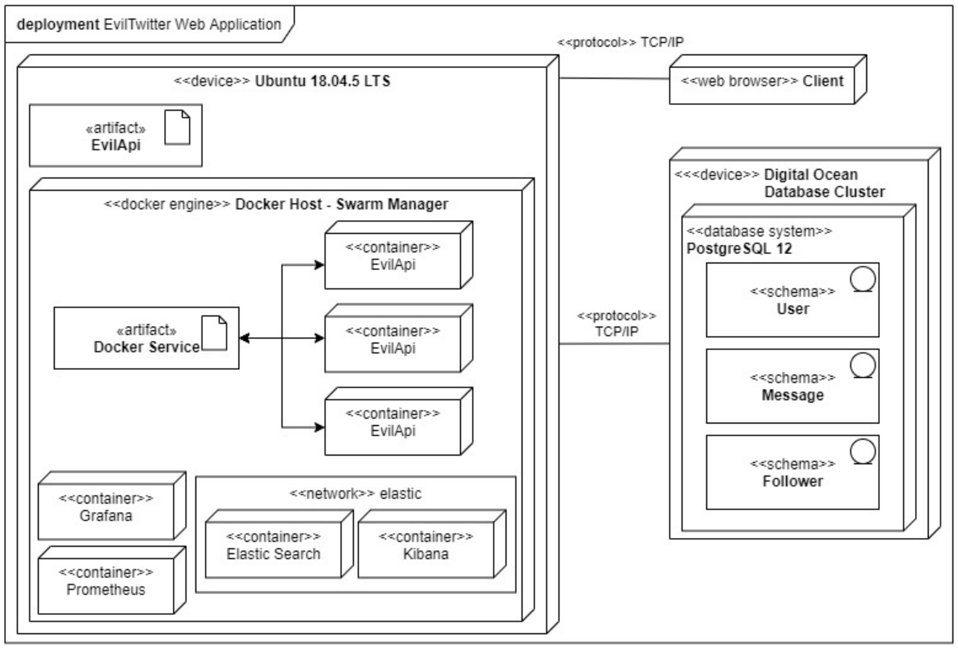


Figure 2: Deployment diagrom of the project EvilTwitter showing the current state of the project

Communication between EvilClient and EvilApi is done via TCP/IP using the http protocol to deliver deliver json files. The EvilApi follows the REST Architecture (Representational State Transfer). The EvilApi and the database communicates via an ORM, specifically the EF Core framework.

TODO: communication between EF Core and PSQL

Finally as seen in figure 2 the application attempts to create a microservice by orchestrating some of the docker containers in swarm mode. In order to handle increase in traffic, thereby handling scaling in a horizontal way while also making the service more resilient. This had some shortcomings that will be discussed in section 1.5.

1.2 Design of the system

The overall architecture outlined in section 1.1 separated the application into 3 distinct entities, namely EvilClient, EvilTwitter and the PSQL Database. Their design will be covered in separately in the following sections

1.2.1 Design of EvilClient

The main responsibility of the EvilClient is to display data from the database to the user send by the EvilApi, and handle inputs to the user that could manipulate data in the database. Hence the responsibility of the EvilClient is data conversion between displaying data to the user, and converting user input to usable data in the database thereby, the MVVM (Model-View-ViewModel) pattern was chosen (Microsoft 2012).

Of notice should be the Pages, Shared, ViewModels folders of the EvilClient folder. Here Pages and Shared contains the View part of MVVM which is handle by the Microsoft Blazor Framework[1] to convert the code into a application that is executable in a web browser. The ViewModels folder holds the code that converts data from the EvilApi into usuable information to the user, and vice verse converts input from the client into data that is usable for the EvilApi.

1.2.2 Design of EvilApi

EvilApi is a REST Fielding 2000 Api that updates a database according to the requests send. To handle the conversion from C# to PostreSQL an ORM is used, that in this case is EF Core[2].

1.2.3 Design of the database

TODO

1.3 Dependencies

A dependency graph for all used dependencies can be found in our GitHub repository[3].

1.4 Interactions of subsystems

Taking the current state of the system as a starting point helps describe the interactions of the subsystems, hence a setup having multiple droplets is possible and each droplet would be identical in regards to interactions of subsystems. The setup is depicted in figure 3, following the Three Layered Architecture presented in section 1.1, but in this example the presentation tier and and business tier is located on the same droplet with the data tier located on another

[1]https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor
[2]https://docs.microsoft.com/en-us/ef/core/
[3]https://github.com/gustavjohansen98/E-vil-Corp/network/dependencies

Table 1: Dependencies grouped by license

apache 2	MIT	BSD 3clause	PostgresQL license
AspNetCore.Diagnostics.HealthChecks	Newtonsoft.Json	moq4	npgsql/efcore.pg
code-cracker	ProfanityDetector		
dotnet/efcore	prometheus-net		
aspnet/Diagnostics	prometheus-net.SystemMetrics		
serilog-aspnetcore	RehanSaeed/Serilog.Exceptions		
serilog-enrichers-environment	Swashbuckle.AspNetCore		
serilog-sinks-debug	coverlet		
serilog-sinks-elasticsearch	vstest		
Roslynator			
xunit			
visualstudio.xunit			

device. Also it is observed that the EvilClient communicate directly with the EvilApi.[4], by sending http requests back and forth that depending on the request might contain a data in a json format, as described in section 1.1. Further, a simulator talks directly to the EvilApi in the same manner as the EvilClient, which leaves only the EvilApi communicating directly with the database.

[4]Note should be taken that the EvilClient actually sends out an http request that leaves the droplet only to return shortly after. This is not optimal and the services should talk internaly on the droplet i.e. via the docker network

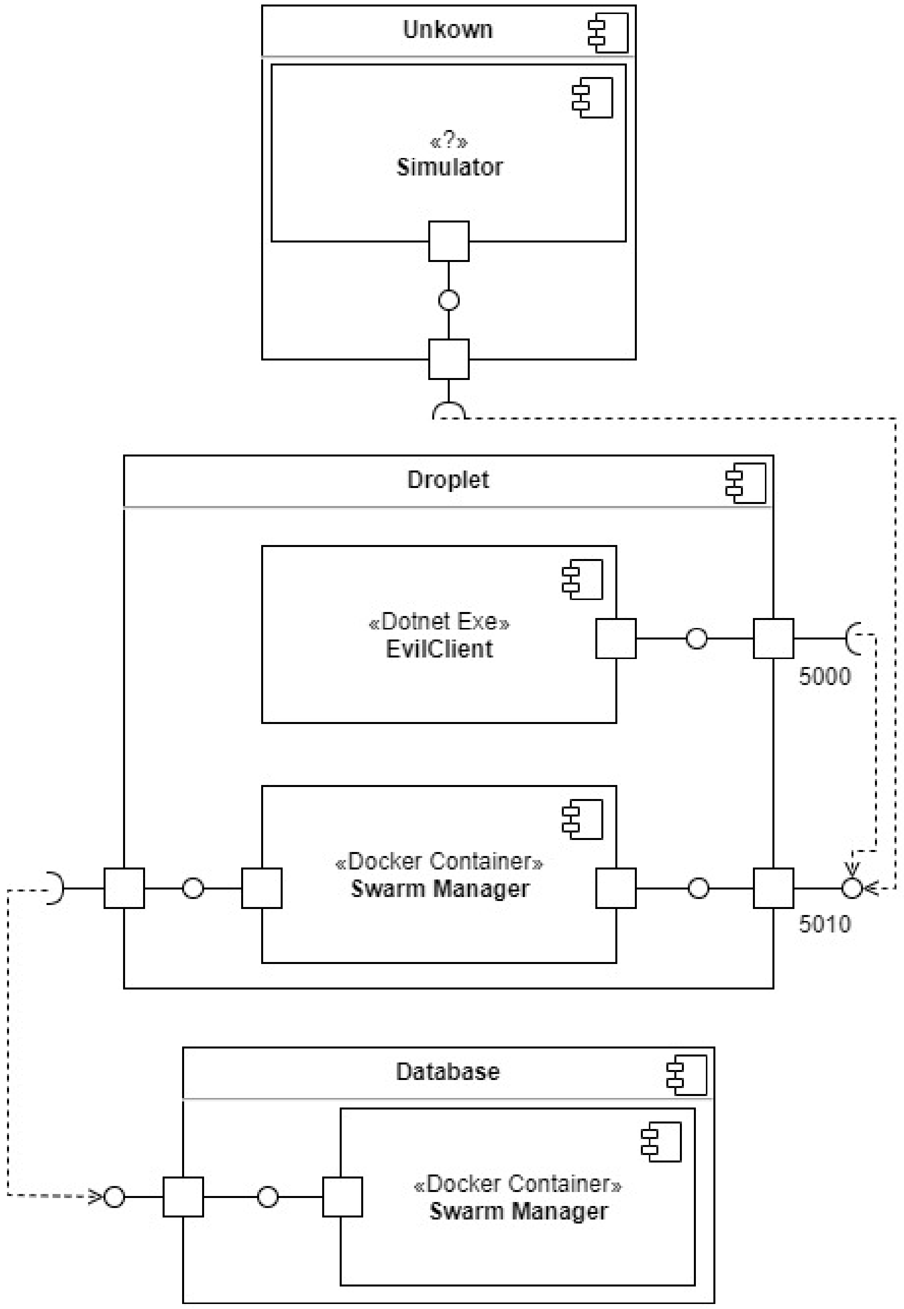


Figure 3: Component diagram depicting the interactions of the subsystems that make up the EvilTwitter project

As seen

1.5 Current state of the systems

To assess the state of the system, first a look and discussion of the SonarCloud report will be done.[5]

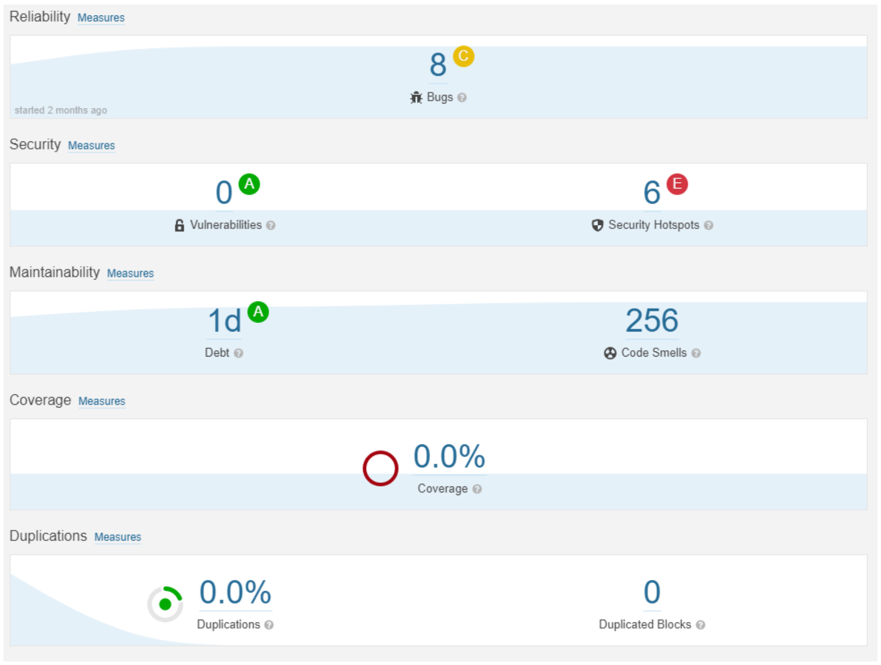


Figure 4: TODO

By looking at the security hotspots 3 clear risks can be seen:

- Use of MD5 hasher
- Logging Injection
- Use of the http protocol

From this the MD5 hash was kept due to the lack of a password reset function, as the application was first deployed with a MD5 hash, hence users made initially would not be able to log in. Though the MD5 hash is used in the creation of the user avatar this only hashes the email, and since this is public available on people profile it is not seen as a risk.

Due to two factors logging injection is seen as less of a risk. Firstly messages received containing the hashed password isn't logged or un-hashed, and secondly the application is in a transition of moving the logs to a PostgreSQL database.

TODO: use of http protocol

[5]https://sonarcloud.io/dashboard?id=gustavjohansen98_E-vil-Corp

1.6 Software license agreement

As listed in section 1.3, we have 21 direct dependencies. 11 of them are licensed under Apache version 2[6], 8 of them under the MIT license[7], while the BSD 3-clause[8] and PostgresQL license[9] cover 1-1 dependency each. Since all of these licences are permissive, we had a lot of freedom to choose how to license our software. While we must preserve the original license notices in the files which use code covered by the aforementioned licences, we are permitted to license the project *as a whole* as we see fit. Therefore, to make sure evil capitalists don't profit from our work, we released the project under the GPL version 3[10].

2 Process' Perspective

2.1 Team Work

2.1.1 Organisation

The teams aimed at doing agile development via including practices like self organising teams and iterative delivery. We worked in weekly iterations. Every week we had two meetings one on Mondays and one on every Thursday. Each meeting started with stand up meeting where everyone summarised what have been achieved since the session and what we expect to be done until next week. On Mondays we defined the task that we plan to do on the given with priorities assigned to each of them. Also, we picked the individual or group tasks for the week through out the week the individuals and groups work on the tasks in a self-organising manner. The Thursday session had the main focus demonstrating the progress ,helping each other out if one of the group or group member is stuck with a task and deciding on the which task to pick if we were done with the one picked on Monday.

2.1.2 Communication Tools

We used Teams Groups for the meetings and communication , which contained a general chat, a chat for arranging meetings and a chat that contains various useful links. This enabled us ask questions and provide feed back out side of the meetings, thus we could work on the project more continuously. We organised the tasks in Kanban, this way we could easily track the progression of them and see which ones need to be worked on urgently.

[6]https://www.apache.org/licenses/LICENSE-2.0
[7]https://opensource.org/licenses/MIT
[8]https://opensource.org/licenses/BSD-3-Clause
[9]https://www.postgresql.org/about/licence/
[10]https://www.gnu.org/licenses/gpl-3.0.en.html

2.2 Development Strategy

We used repository. We developed on our local machines and pushed to the development branch first. The have been deployed when it was pushed to the main branch. Before merging the development branch with the main, at least two group members reviewed the code and tested the new functionality.

2.2.1 Tools used

To host the code a Github Repository was used, which was owned by one group member

Github Actions

Github Projects

Digital Ocean Droplet

Digital Ocean Database Cluster

2.3 Monitoring

The monitoring of the EvilTwitter application was done using the monitoring and alerting toolkit Prometheus[11] to gather information from the application. Two dotnet package was used to retrieve data from the application. Prometheus-net.SystemMetrics[12] was used to retreive system information from where the application was running, and prometheus-net[13] to gather information from the Controllers. The application posted all the collected data at *http://159.89.213.38:5010/metrics* where Prometheus could then gather the necessary data.

Prometheus then made its services available at *http://159.89.213.38:9090* where Grafana[14] could connect and retrieve the necessary data. Via grafana this data was changed to a more readable format collected in a dashboard that was made available at *http://159.89.213.38:3000*.

The information gathered in Grafana was setup in two categories: General and controller usage

[11] https://prometheus.io/
[12] https://github.com/Daniel15/prometheus-net.SystemMetrics
[13] https://github.com/prometheus-net/prometheus-net
[14] https://grafana.com/

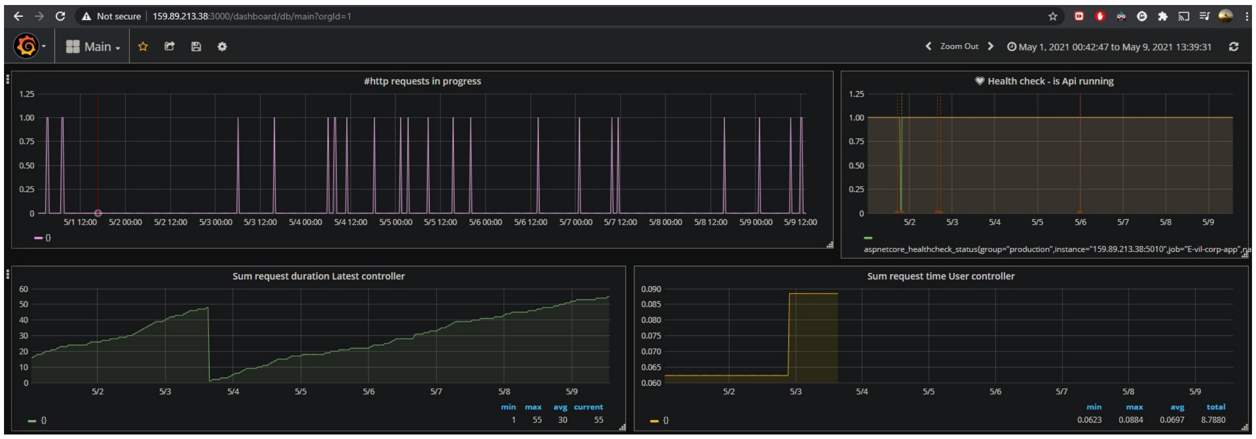

#http requests in progress
Health check - is Api running
Sum request duration Latest controller
Sum request time User controller
May 1, 2021 00:42:47 to May 9, 2021 13:39:31

Figure 5: TODO

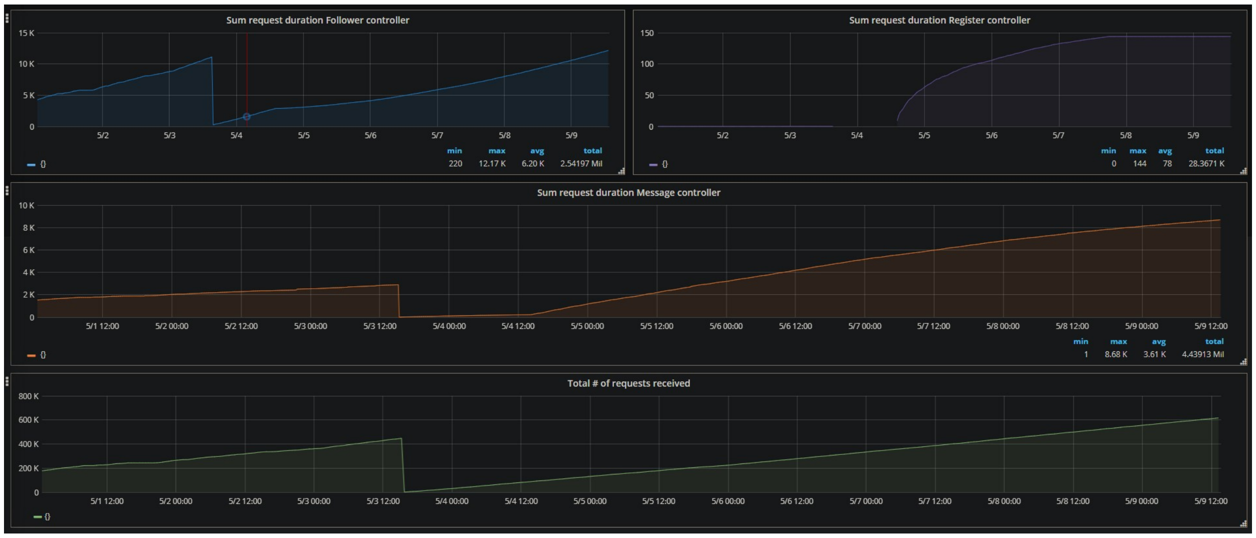
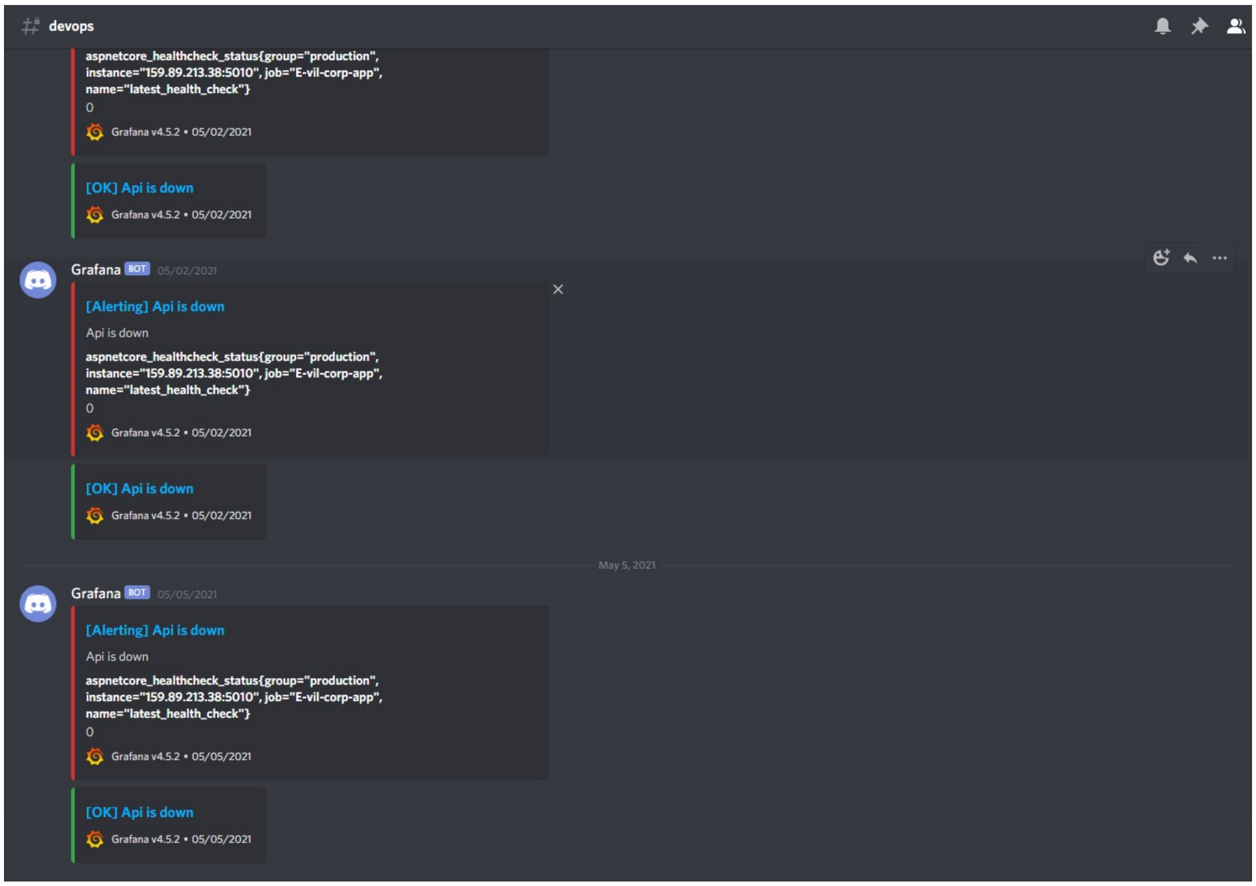

Sum request duration Follower controller
Sum request duration Register controller
Sum request duration Message controller
Total # of requests received

Figure 6: TODO

First at the top row of figure 5 the genral view of the application is given. This includes a graph over how many request is occurring at a given time (the graph to the left), to give an overview of the incoming traffic. Further, the alert graph can be seen at the top right, which is a value that can be either 1 or 0. This translates to what value latest returned last, with any latest value greater than 0 would resolved to a value of 1 and anything else would resolved to a value of 0. Hence if the Api is down or returns odd values this would be registered by Grafana. This tracker could be used to notify the developers if the Api behaved oddly, which was utilised by having a webhook that sends messages to a discord server as seen in the figure 7.


```
devops
aspnetcore_healthcheck_status{group="production",
instance="159.89.213.38:5010", job="E-vil-corp-app",
name="latest_health_check"}
0
Grafana v4.5.2 • 05/02/2021

[OK] Api is down
Grafana v4.5.2 • 05/02/2021

Grafana BOT 05/02/2021
[Alerting] Api is down
Api is down
aspnetcore_healthcheck_status{group="production",
instance="159.89.213.38:5010", job="E-vil-corp-app",
name="latest_health_check"}
0
Grafana v4.5.2 • 05/02/2021

[OK] Api is down
Grafana v4.5.2 • 05/02/2021

May 5, 2021

Grafana BOT 05/05/2021
[Alerting] Api is down
Api is down
aspnetcore_healthcheck_status{group="production",
instance="159.89.213.38:5010", job="E-vil-corp-app",
name="latest_health_check"}
0
Grafana v4.5.2 • 05/05/2021

[OK] Api is down
Grafana v4.5.2 • 05/05/2021
```

Figure 7: TODO

Second information from controllers was very detailed[15], and could be filtered by message type (POST, GET etc), response (204, 404 etc.). This information is valuable in solving performance issues, but creating a graph for every single message type and response would clutter the dashboard and make it less readable. Hence a decision was made that such queries should be done on a case by case basis, and a more general overview was created to monitor each controller as a whole. The query can be seen below.

sum(http_request_duration_seconds_sumcontroller="Follower")

Everything else on the Grafana dashboard is the query above executed on all controllers. An example to the use fullness of this approach can be seen in the following example. At one point the message controller was more time as the other controllers, where by looking on the summarised time per message type revealed that it was the GET call that used a lot of time.

[15]The data posted can be seen at http://159.89.213.38:5010/metrics

2.4 Logging

The application uses Elastic Search[16], Kibana[17] and Serilog[18] to aggregate the logs. This is done by having C# logs useful information that is then propagated to *http://localhost:9200* where Elastic Search monitors and collects log. Finally Kibana imports this data in order to display and query the logs.

The package Serilog came with some off the shelf functionality in case of logging, where at the informational level the following functionality was utilised initially. Logging of database queries and requests and response send to and from the controllers. After evaluation this was deemed unnecessary, as the information given in the logs could be retrieved wither from the monitoring setup or from database queries themselves, hence logging was kept at an error level as this information cannot be retrieved otherwise.

2.5 Security assessment

To assess the security of our system, we considered vulnerabilities of our cloud infrastructure, vulnerabilities of the code we produced, and security of the user data. When assessing the safety of our infrastructure, we found it high impact and high probability that an adversary gains access to the account of the repo owner, gaining access to our secrets. To decrease this risk, the repo owner enabled 2-factor authentication. We also found it high probability high impact to fall victim to a denial-of-service attack, since DoS attacks are cheap to execute, and we have no protection set up against it. The system could also be overwhelmed by automated sign-ups to the platform, so it would be advantageous to set up CAPTCHA against it.

Considering we use the insecure version of http, our users are at risk of an adversary eavesdropping or spoofing our server's IP. This in turn would lead to disclosure of the user's credentials. Could by remedied by self-signing a certificate with Let's Encrypt. We deemed this issue medium impact and medium probability.

One low probability risk we identified was the cloud provider (or our account at the provider) getting hacked. This would be a severe problem, since we would lose access to our infrastructure, with the adversary gaining complete control over the live server and our database. User credentials would be somewhat safe, we only store hashes of passwords, however common passwords could be easily looked up from a rainbow table. To provide better guarantees, we could also add salt and pepper to the passwords. The other low probability risk we identified was a supply chain attack on any of our dependencies. Since it is unfeasible to constantly audit every new version the supplier publishes, the best line of defence is auditing once, and freezing version numbers after.

[16]TODO
[17]TODO
[18]TODO

2.6 Description of CI/CD pipeline

Overall, the project consists of three workflows all utilised with GitHub Actions to make use of the open source workflow actions found in the GitHub Marketplace:

- **release.yml** to automatically make a release every Sunday at 9 pm.
- **report-overleaf.yml** to automatically compile the Latex source code into a pdf.
- **main.yml** to deploy local changes from development to production.

All the workflow files are found in the *.github/workflows* folder of the repository, and each uses the action checkout[19] to checkout the repository to a virtual machine hosted by GitHub to perform operations on.

2.6.1 Automatic release

This workflow uses the create-release[20] and CRON formatting to automatically trigger a release of the main branch every sunday at 9 pm.

2.6.2 Latex report build

We write the report in Overleaf and from their platform, we push the changes in the Latex documents directly to main, which compiles them to a pdf via the latex-action[21]. We then use the push action[22] to push the pdf document back into the correct folder in the repository.

[19] https://github.com/actions/checkout
[20] https://github.com/actions/create-release
[21] https://github.com/xu-cheng/latex-action
[22] https://github.com/ad-m/github-push-action

2.6.3 From development to production

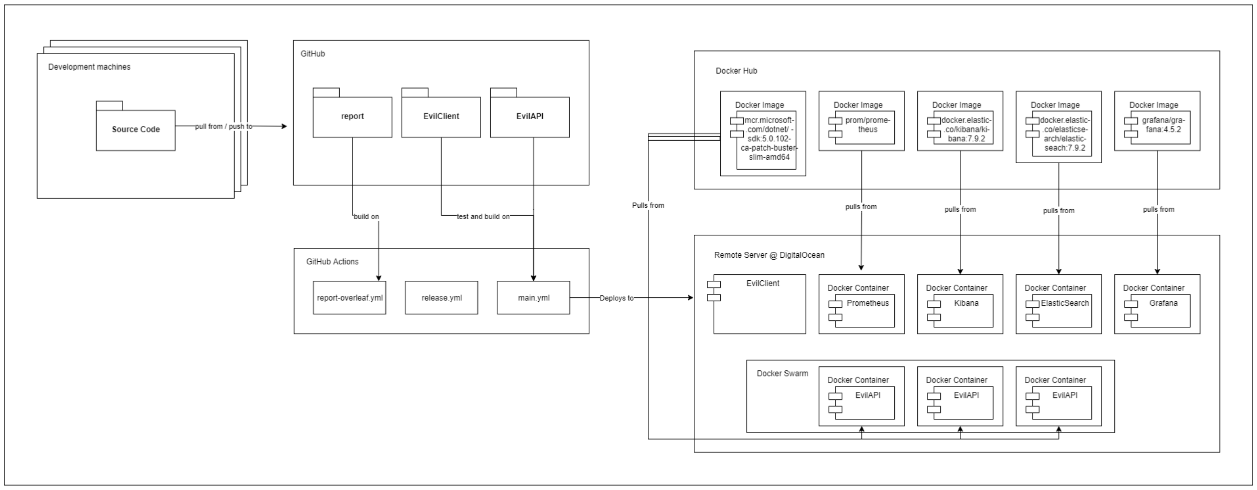


Figure 8: Overall workflow diagram

Figure 4 displays the different stages on how implementations are taken from development into production. We can make use of a tier structure to explain the details:

1. **Development tier**
   Changes to the source code are made in this tier on individual developer workstations. To ensure code quality before pushing to version control, the project uses the dotnet CodeCracker[23] package to provide the developer with a static code analysis when building locally.

2. **Integration tier**
   Upon merging into the main branch, the *main.yml* workflow will be triggered. This workflow handles testing, quality control and staging and is separated into three jobs, that will run on GitHub Actions hosted containers:

   - **Build, test and Infersharp analysis**
     This job runs in an Ubuntu 18.04 environment to resemble the actual target production environment hence staging. Here, dependencies are restored, the source code will be built and all the unit tests will be performed. Furthermore, we make use of the Infersharp action[24], that will detect security leaks such as exposed connectionstrings for instance.

[23] https://github.com/code-cracker/code-cracker
[24] https://github.com/microsoft/infersharpaction

- **Build and SonarCloud analysis**
  The second job runs in a Windows latest version environment, since the SonarCloud static code analysis depends on this environment. When built and completed, a comprehensive code analysis is available on our SonarCloud dashboard.

Both of these jobs run concurrently and utilise the dotnet action[25] with dotnet version 5.0.102 to mirror the dotnet environment in production.

3. **Deployment tier**
   The third job of the *main.yml* workflow handles continuous deployment and is dependent on the aforementioned jobs, as shown in figure 5.

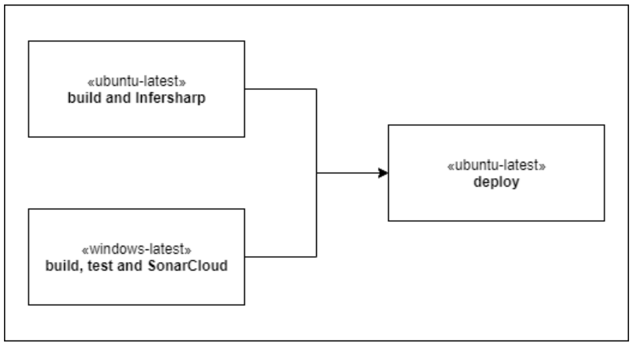


Figure 9: Main Workflow diagram

That is, this final job will be triggered if and only if the other jobs succeed, and upon success the job will run the deployment script:

- Checkout the repository from main.
- Copy repository to DigitalOcean droplet via SCP action[26].
- SSH into the droplet via SSH action[27].
- Run the swarm deploy script, such that the api docker service updates its running containers with the latest build of the EvilAPI subsystem.
- Run an instance of the EvilClient subsystem.
- Run the Grafana, Kibana, ElasticSearch and Prometheus containers if they are not already running via docker-compose.

If the rolling update in the api docker service fails, an automatic rollback to the previous image will be performed. Unfortunately, the EvilClient instance does not have any rollback strategy upon failures as of now.

[25]https://github.com/actions/setup-dotnet
[26]https://github.com/appleboy/scp-action
[27]https://github.com/appleboy/ssh-action

2.7 Applied strategy for scaling and load balancing

3 Lessons Learned Perspective

3.1 Evolution and refactoring

Since we came from different backgrounds - data science and software development-, our first issue in refactoring was settling on a language. As we didn't have a shared programming language all of us knew, we tried learning a common language none of us knew before. While we thought we can pick up a language quickly, since we knew the basic building blocks and had experience learning from documentation, we ended up settling for C# after some gruesome weeks of learning go. This choice created some discrepancies in our abilities to interact with the codebase, which was hard to bridge for the entirety of project period.

When setting up virtualization, we started by deploying local copies to Digital Ocean with the help of Vagrant. Later, we switched from local copies to cloning from github, since it was easier to just specify the production branch in Vagrant than manually switching branches for spinning up the machines. An issue we were still facing was the database. Since we were working with an sqlite database, we had to manually copy the database from our VM before every destruction. At first, we bridged the issue by setting up a vagrant trigger-on-destroy to copy the database file. Later, we switched to a postgres cluster, also deployed on Digital Ocean, so that the database was more independent from the application. importance of database setup initially

Creating the CI/CD setup, we've been through hell. We had a lot of issues with setting up the keys the right way for everything, which is why we ended up not using Travis, but github actions, since it was more tightly integrated with the repository.

Monitoring

Logging

Load balancing

3.2 Operations

3.3 Maintenance

Even more text :O