Final Report - DevOps

EvilTwitter

Group E

abea, beba, gujo, luka, sena

IT University of Copenhagen
Denmark
19 - 5 - 2021

Contents

1 System's Perspective

1.1 Architecture of the system

This section will start with a walk through of the overall structure of the project, followed by a further explanation of its sub-parts.

The general structure of the project is a Three Layered Architecture, because it formed a nice separation of concerns in the application. This resulted in three applications namely, EvilClient for the presentation tier, EvilApi for the business/logic tier and the PostgreSQL (PSQL) database cluster for the data tier. For an overview of the architecture, see the model diagram in figure 1.

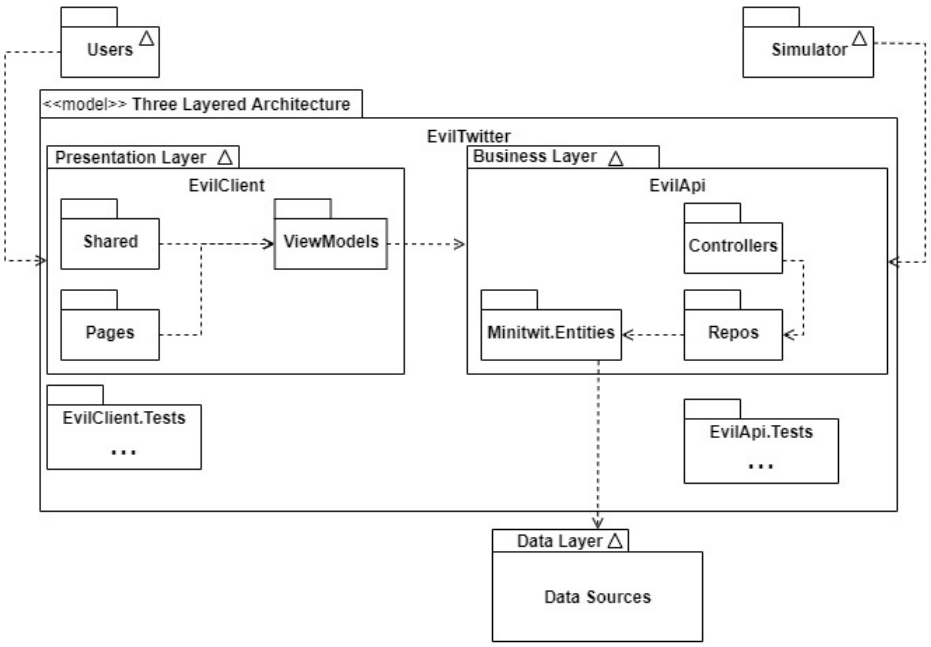

Figure 1: Model diagram of the project, depicting important folders of the source code. Arrows have been added to show the flow of data in the system which also coincides with dependencies between parts of the applications. The flow of data starts at the user or simulator and pointing to the next source 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 path is between EvilClient and EvilApi, the second path is between EvilApi and the database. This flow can be seen in figure 1 and 2.

Communication between EvilClient and EvilApi is done via TCP/IP using the http protocol to deliver optional JSON files in the http request. The EvilApi follows the REST Architecture

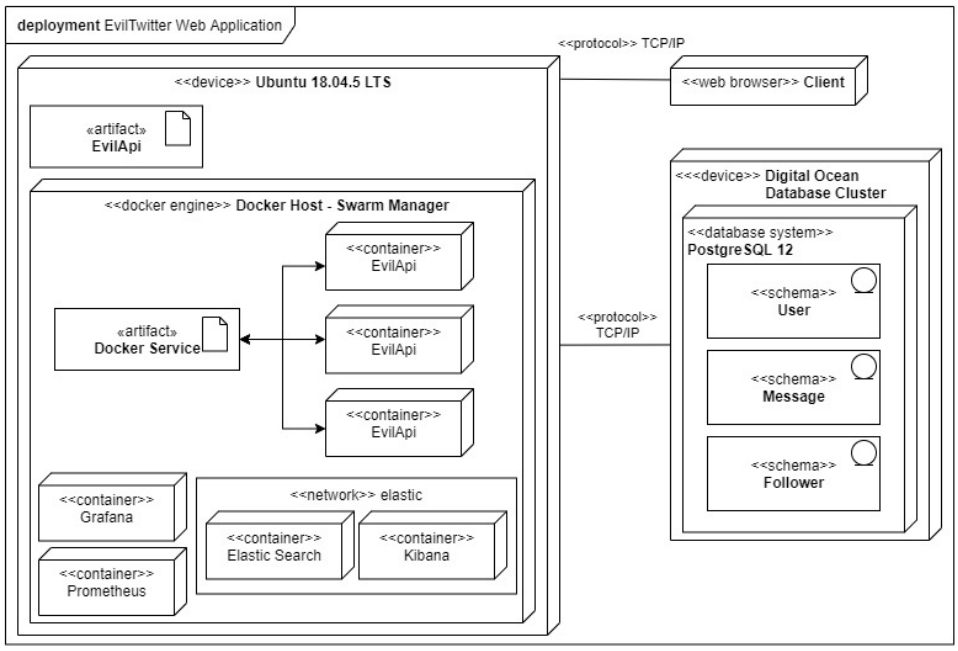

Figure 2: Deployment diagram of the project EvilTwitter showing the current state of the project

(Representational State Transfer), while the EvilApi and the database communicates via an ORM (Object Relational Mapping) specifically the EF Core framework.

Finally, as seen in figure 2 the application attempts to create a microservice by orchestrating some of the docker containers in a swarm mode, to utilise load balancing and making the service more resilient. This had some shortcomings by having only the EvilApi as a microservice and by the EvilClient and EvilApi talking together over the droplet's ports instead of internally i.e. the docker network.

1.2 Design of the system

The architecture outlined in section 1.1 separated the application into 3 distinct entities. Their design will be covered 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 sent by the EvilApi, and handle inputs from 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-View-Model) 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. To display the razor files in a browser the Microsoft Blazor Framework[1] converts 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 versa 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 sent. To handle the conversion from C# data types to PostreSQL data types the application is using an ORM, which in this case is the EF Core framework[2].

1.2.3 Design of the database

As seen on figure 2 the data consisted of three different types and was very structured, hence a relational database seemed like an optimal choice. From the many SQL flavours, we decided to use Postgres, due to its open source nature. The database was arranged in a cluster structure to increase scalability, data redundancy and availability.

1.3 Dependencies

Below is a summarised version of the dependency graph on Github, where all the dependencies of the project is listed under the component depending on it.[3]. Important dependencies will be elaborated on later, in sections where they are deemed relevant.

[1] Blazor framework
[2] EF core framework
[3] Dependency graph

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			

Table 1: All the dependencies EvilClient and EvilApi is using grouped by license

1.4 Interactions of subsystems

Taking the current state of the system as a starting point with only one droplet, helps describe the interactions of the subsystems. Hence, a setup having multiple droplets is possible and each droplet would be identical in regard to interactions of subsystems. The setup is depicted in figure 3, following the Three Layered Architecture presented in section 1.1, in this example the presentation tier and business tier is located on the same droplet, the data tier located on another device. Also, it is observed that the EvilClient communicates directly with the EvilApi[4]. Via sending http requests back and forth, as described in section 1.1. Furthermore, the 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 internally on the droplet i.e. via the docker network

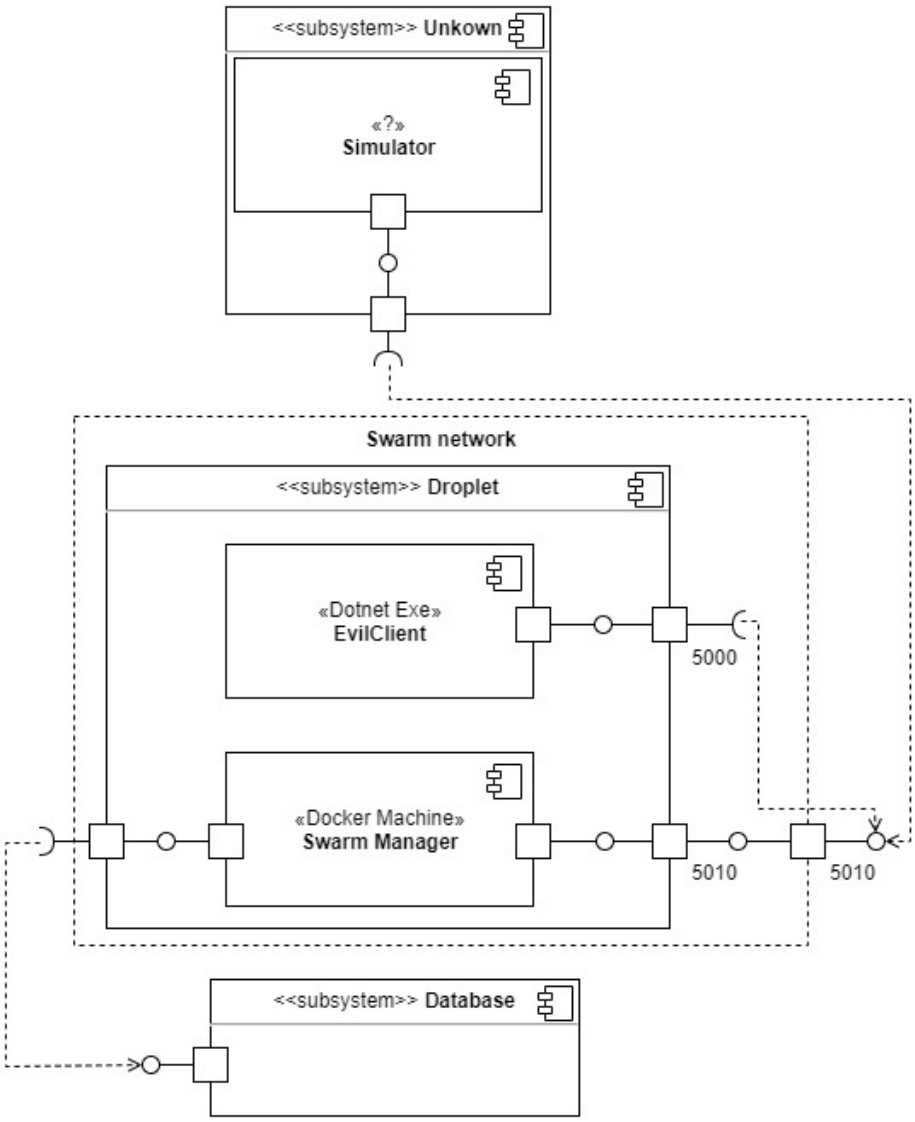

Figure 3: Component diagram depicting the interactions of the subsystems that make up the EvilTwitter project

1.5 Current state of the systems

To assess the state of the system, we'll have a look and discussion of the SonarCloud report produced in the CI pipeline from the main branch.[5]

[5]SonarCloud report

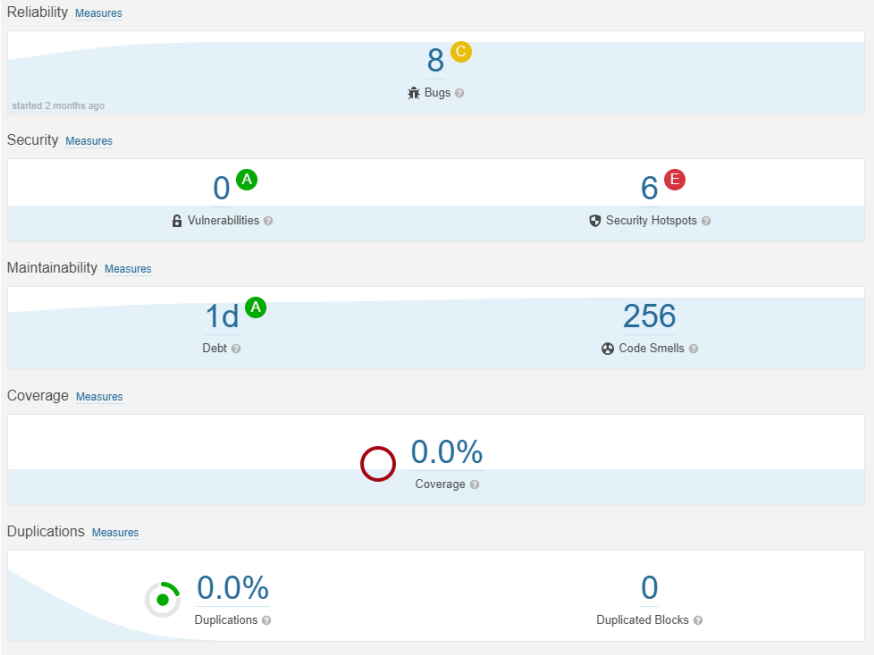

Figure 4: SonarCloud report of the EvilApi. Note that there's a mistake in the latter reports as it couldn't find the tests written for the application

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 publicly available on people's profiles, it is not seen as a risk.

Due to messages received only containing hashed information, logging injection is seen as less of a risk. Furthermore, the application is in a process of moving the logs to a separate database, hence a solution might become invalid and have to be redone shortly after implementation.

It is of course not recommended to make use of the http protocol instead of the https protocol, but we simply didn't prioritise getting a https certificate[6] for the ASP.NET core framework used.

[6]https certificate for ASP.NET core

We have an overall acceptable technical debt of 8 hours[7] with 256 code smells to cover.

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[8], 8 of them under the MIT license[9], while the BSD 3-clause[10] and PostgresQL license[11] cover 1-1 dependency each. Since all 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 third parties don't profit from our work, we released the project under the GPL version 3[12].

[7] SonarCloud documentation
[8] apache
[9] MIT license
[10] opensource.org
[11] psql
[12] gnu

2 Process' Perspective

2.1 Development Team

To adopt a DevOps organisation and development style as specified in Kim, Humble, and Debois 2016 part 1, a mix of various software development strategies and frameworks was chosen, supported by various tools to enhance these strategies. In this section the development strategies will be covered first, followed by the tools used.

2.1.1 Development Strategies

To accommodate the DevOps principles of 'smaller batch sizes' and 'Reduce the number of handoffs' (Kim, Humble, and Debois 2016 p. 9-10), an agile approach was taken to the development by using the agile principle of "Deliver working software frequently ..." (Beck et al. 2001 para. 5). This led to a delivery interval of 1 week with a release on Sunday evening marking the end of an interval.

Team members had other commitments which made it hard to find common working hours. Hence, a self organising approach was chosen (Beck et al. 2001 para. 13) to handle this issue. With this short overlapping working time, it was decided to practice the Scrum daily meetings (Schwaber and Sutherland 2020 p. 9) two times per week, one on Mondays and one on Thursdays. The Monday meetings would be used for prioritising and planning, while Thursday meetings were more of a catch up and experience exchange.

2.2 Development Tools

2.2.1 Communication Tools

Communication and meetings between team members were done using a Teams Group. The Group contained 3 channels a general chat, a chat for arranging meetings/hosting meetings and a chat that contains various useful links.

2.2.2 Planning tools

A Kanban board was created with Github Projects[13] in combination with Github Issues[14] as the sticky notes in it. Using a Kanban board helped to keep an overview of the tasks at hand.

Issues could be added to the project in two ways. Every Monday once the tasks for the following week was known, issues would be created and added to the project. They would

[13] GitHub project board
[14] GitHub issues

then be prioritised with regards to the severity for the application, i.e. security risks would be handled as quickly as possible, while minor UI bugs would be fixed last. If any bugs became apparent at any point, they could be added to the project, where after the developers would assess severity as soon as possible. All of this can be seen in figure 5 together with part of the column setup, which can be seen below.

To Do ⟶ In Progress ⟶ Review ⟶ Done ⟶ Deployed ⟶ To Be Archived

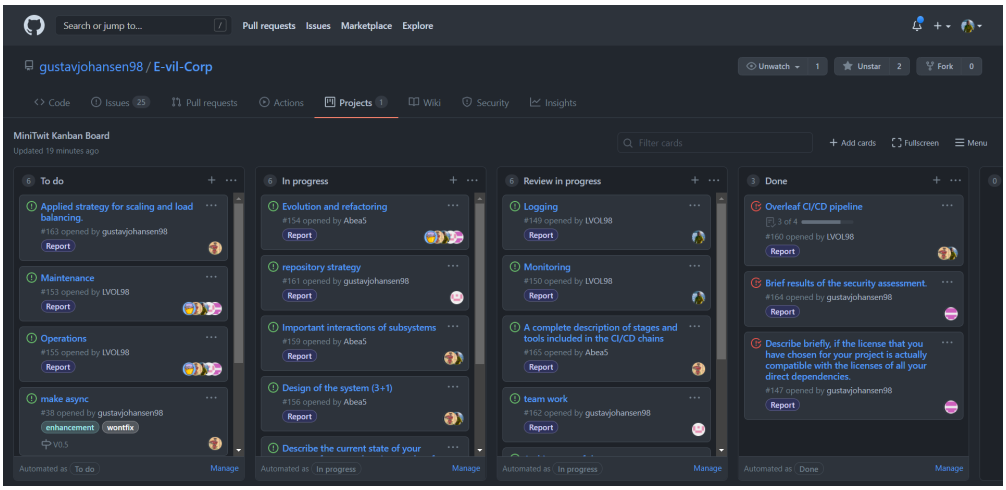

Figure 5: Screenshot of the Kanban board on github projects used by the group throughout the project

2.2.3 Version Control

Git was used as the version control system and Github was used to host the repositories. A mono-repository setup was utilised to host the source code, with a task based branching strategy (Radigan 2018), combined with a development and main branch. Meaning that main contained code that was deployed and develop was a branch for testing and merging the different task branches.

2.3 Monitoring

To retrieve useful information for monitoring from the EvilApi two dotnet package were used. Prometheus-net.SystemMetrics[15] was used to retreive system information from where the application was running, and prometheus-net[16] to gather information from the API controllers. All of this data was propagated to *http://159.89.213.38:3000*, where the monitoring and alerting toolkit Prometheus[17] could gather needed information.

[15] SystemMetrics for Prometheus in dotnet
[16] prometheus-net
[17] Prometheus

Prometheus then made its services available at *http://159.89.213.38:9090* where Grafana[18] 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 set up in two categories: general and controller usage.

Figure 6: The first four graphs from the Grafana dashboard presenting data surrounding the health status of the system and the number of requests in progress

Figure 7: Four graphs from the Grafana dashboard presenting the request duration of the different controllers and the total amount of requests recieved

First, at the top row of figure 6 the general view of the application is given. The top left is a graph over how many request is occurring at a given time, to give an overview of the

[18] Grafana

incoming traffic. To the top right the alert graph can be seen, which is a value that can be either 1 or 0. This translates to what value *latest* returned last. Any latest value greater than 0 would resolve to a value of 1 and anything else would resolve 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 web hook that sends messages to a discord server as seen in the figure 8.

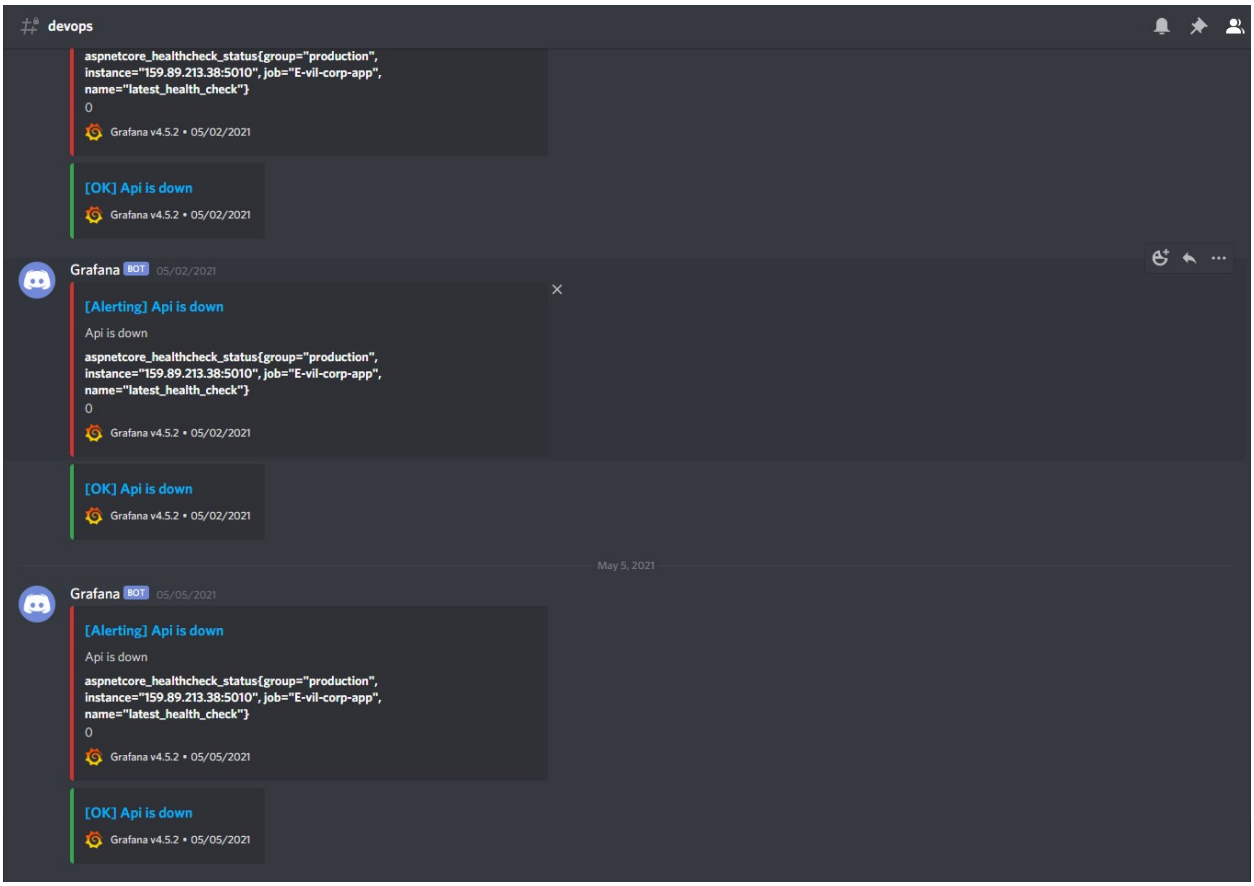

Figure 8: Screenshot of the developer discord containing status messages of the API from Grafana

Second, information from controllers was very detailed[19], 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")

[19]The data posted can be seen at EvilTwitter metrics

Everything else on the Grafana dashboard is the query above executed on all controllers.

2.4 Logging

The application uses Elastic Search[20], Kibana[21] and Serilog[22] to aggregate the logs. This is done by having C# log useful information, that is then propagated to *http://0.0.0.0:9200* by Serilog where Elastic Search monitors and collects log. Elastic Search then exposes itself on port 9200 where Kibana can import the data needed in order to display and query the logs.

Due to several issues regarding the logging implementation, it crashed the service often hence why logging level was increased to error. A solution with having the logs written to a database and having another droplet running the logging stack was being implemented, but sadly it wasn't finished in time. It would be more advisable to increase the logging level to information, such that messages in http requests would be stored. This could be used to look through if the service crashed or malicious activities was tried.

2.5 Security assessment

To assess the security of our system[23], 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 and 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[24]. 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

[20] elasticSearch
[21] kibana
[22] serilog
[23] See appendix for the full report
[24] Let's Encrypt

passwords could be easily looked up from a rainbow table[25]. To provide better guarantees, we could also add salt and pepper[26] 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.

2.6 Description of CI/CD pipeline

Overall, the project consists of three workflows all created 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 checkout[27] action, 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[28] 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 on 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[29]. We then use the push action[30] to push the pdf document back into the correct folder in the repository.

[25] Rainbow table
[26] Salt and Pepper hashing
[27] checkout action
[28] release action
[29] latex compiler action
[30] push action

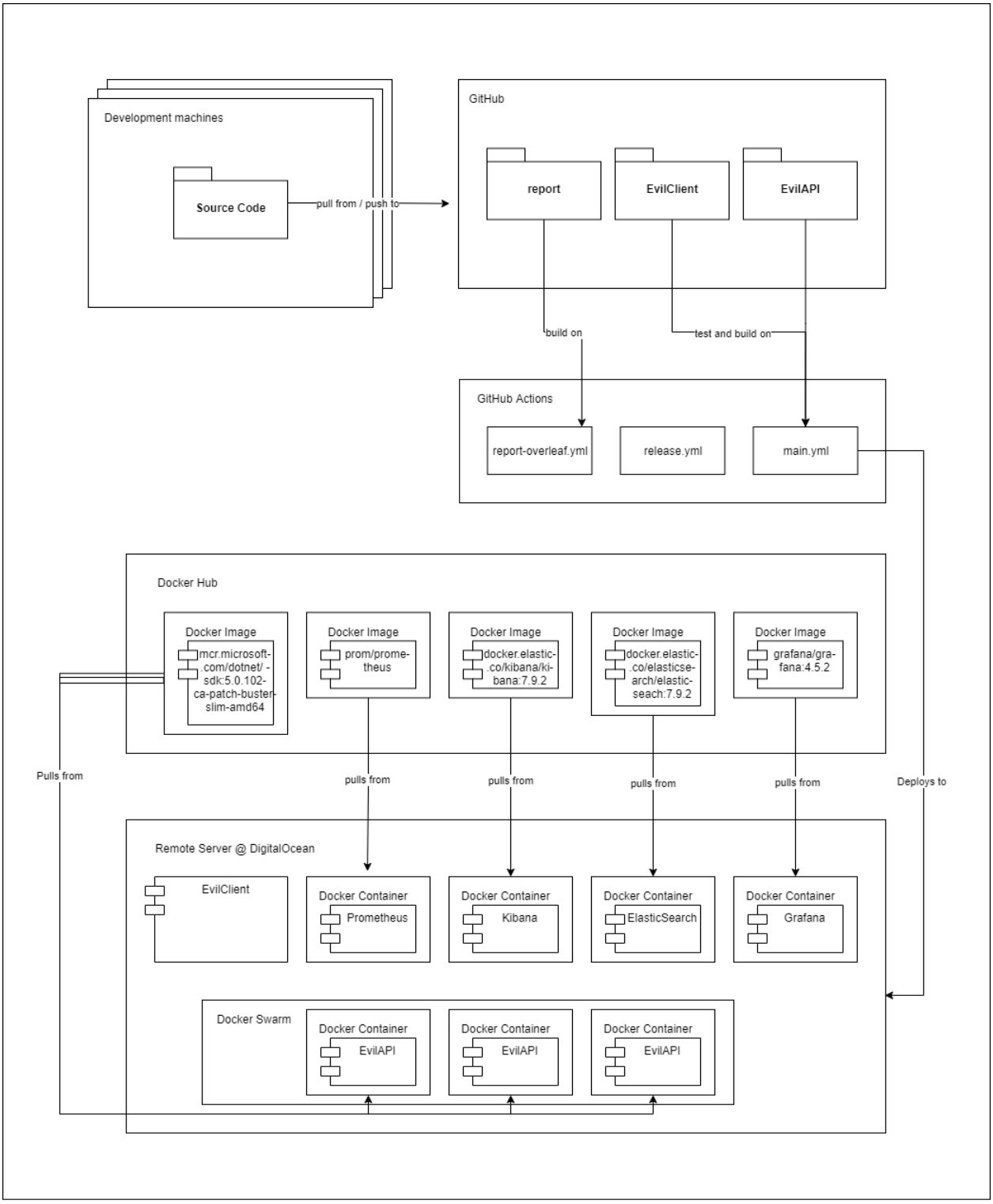

Figure 9: Overview of the overall pipeline and how code is taken from local machines to production

2.6.3 From development to production

Figure 9 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[31] 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[32], that will detect security leaks such as exposed connection strings for instance.
   - **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[33] 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.

[31] CodeCracker
[32] Infersharp action
[33] dotnet action

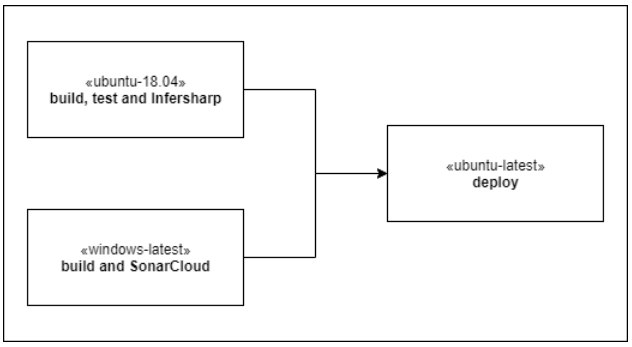

Figure 10: Dependencies between jobs in main.yml

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[34].
- SSH into the droplet via SSH action[35].
- 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.

2.7 Applied strategy for scaling and load balancing

We've utilised vertical scaling once during the project lifetime, when we implemented logging for the system due to required increase in RAM usage on the droplet. This was done manually in the DigitalOcean interface. For load balancing we've managed to use docker swarm for the EvilAPI with only one swarm node. Ideally, we would have utilised multiple droplet nodes as replicas, where each would join the swarm as a worker node, hence achieving horizontal scaling and avoiding single point of failure, which has been an ongoing issue during the project lifespan.

[34] scp action
[35] ssh action

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 that we could pick up a language quickly, we ended up settling for C# after some gruesome weeks of learning go (as we see with the initial go commit[36] being on the 1st of February and the first C# commit[37] being on the 15th of February). This choice created some discrepancies in our abilities to interact with the code base, which was hard to bridge for the entirety of project period. The most important lesson learned in regards to the refactoring, is the importance of researching and having an open discussion about a solution, that is both suited for the functionality of the program as well as individual experience on a team level.

Creating the CI/CD setup, 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. Overall we had a better and more productive experience with GitHub Actions, and after this revelation we made sure to always look into alternative solutions, than just leaning on the course material.

3.2 Operations

Implementing load balancing and zero downtime deployment became a major hurdle towards the end of the project, by dependencies of other implementation like the database migration elaborated on in section 3.3 and how to handle connection strings. After some researching docker secrets was discovered which solved all the issues. Often documentation is a second though with the main knowledge searching happening on Google, but often taking the time and studying the documentation can be more efficient, which is the knowledge gained through this experience.

3.3 Maintenance

Having initially deployed with an SQLite database is sub optimal hence an issue for changing to another provider was created the 25th of February and was finally moved to deployed on the 16th of April[38]. By the issues being unresolved for such a long time, several problems was connected to it with their own solution. One problem does however standout by slowing

[36] Initial go commit
[37] Initial C# commit
[38] See this issue

the development down the most. Storing the connection string in a secure manner was a major hurdle, both before implementing Docker Swarm and after. The solution before was a .txt file located at the droplet which was a sub optimal but temporary solution, that was solved with the implementation of Docker Swarm as the connection string could be stored as a Docker secret.

The activity of planning is often neglected in favour of developing, which time and again shows up with major bugs and issues later on. The neglect not to consider the database situation at deployment time is a prime example of this, and is if not a lesson learned then at least a reminder of the importance of thoughts before actions.

3.4 Final Reflections on DevOps

As everyone had limited experience with deployed software, it is hard to define the difference from previous projects, as everything regarding operations and maintenance was new. That said experience from other school subjects and student jobs does still provide some reflections.

This became clear when two developers from the group tried setting up a CI/CD pipeline in another course - BNDN second year project - such that an mobile app release of the code base would automatically be triggered when merging into the main branch. However, due to the Product Owner not being able to provide Apple developer certificates, this was not successful. The failure of the pipeline resulted in a lot of time used building a release and manually uploading and notify the beta testers, that a new version was available. This really highlights the importance and efficiency of a working CI/CD pipeline when developing in a large group.

4 Links

- GitHub Repository
- GitHub repository
- EvilClient
- EvilAPI
- Prometheus
- Grafana
- Elastic Search
- Kibana

References

[Bec+01] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: https://agilemanifesto.org/principles.html. Accessed: d. 15.05.2021.

[Fie00] Roy Thomas Fielding. *CHAPTER 5 - Representational State Transfer (REST)*. 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Accessed: 11.05.2021.

[KHD16] Gene Kim, Jez Humble, and Patrick Debois. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2016. URL: https://ituniversity-my.sharepoint.com/:b:/g/personal/ropf_itu_dk/Eafg4B4afaxIqYGDYJqOJLQBycrIZ8JwkokFy4j9JuWiuQ?e=OH5SzC. (accessed: 27.03.2021).

[Mic12] Microsoft. *The MVVM Pattern*. 2012. URL: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)?redirectedfrom=MSDN. Accessed: 11.05.2021.

[Rad18] Dan Radigan. *Feature branching your way to greatness*. 2018. URL: https://learnit.itu.dk/pluginfile.php/281083/mod_page/content/5/2020-Scrum-Guide-US.pdf?time=1612081998778. Accessed: d. 15.05.2021.

[SS20] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. 2020. URL: https://learnit.itu.dk/pluginfile.php/281083/mod_page/content/5/2020-Scrum-Guide-US.pdf?time=1612081998778. Accessed: d. 15.05.2021.

5 Appendix

5.1 Risk Identification

5.1.1 Assets identification

The following assets was used in the EvilTwitter project

- **Droplet on DigitalOcean**: Contains both the webserver and webclient. Also holds the connectionstring for the postgres cluster as a docker secret.
- **Public repository hosted on github.com**: Contains the repository, and stores relevant keys for connecting to the droplet in github secrets.
- **Postgres cluster on DigitalOcean**: Postgres server holding all data provided by users.

5.1.2 Risk Sources and Scenarios

The assets identified in section 5.1.1 have the various vulnerabilities, which will be identified in this section followed by an assessment of the severity via a risk matrix in section 5.2.1. Finally solutions to these risks will be proposed in section 5.2.2. The identified risks are:

1. **Github Actions**: The adversary clones the repository and changes the Github Actions Workflow to get information such as the connection string from docker secrets
2. **Github repository owner's profile gets hacked**: The owner of the repository gets hacked and the attached secrets and deploy keys can be exploited. This will give the adversary direct access to the server.
3. **DDoS attack**: The adversary could do a DDoS attack on the MiniTwit service, making it unavailable for users.
4. **The cloud provider getting hacked**: an adversary could gain access to all the infrastructure and destroy the data, leak it, or extort us.
5. **Nuget Package getting hacked**: A Nuget package which the system depends on could be tampered with and get infected with malware. This would be a supply chain attack and could be used to for instance exploit the connection string for the PostgreSQL cluster.
6. **IP Spoofing/Eavesdropping to get password**: The adversary could gain access to a user's account by IP Spoofing/Eavesdropping to get the hashed passwords of a user, and if they know our hashing algorithm, and the user has a simple password, that they can look up from a rainbow table.

5.2 Risk Analysis

5.2.1 Risk Matrix

		Probability		
		High	Medium	Low
Impact	High	1, 3		2, 4
	Medium		6	5
	Low			

Figure 11: Assessment of risks to the system with the probability of the given risk happening on the x-axis, and the impact of said risk on the system on the y-axis. The number references the risks shown on the list in section 5.1.2

5.2.2 Discuss Scenarios

Solutions to the risk scenarios identified in section 5.1.2 will be provided

- **Github Actions** : Limit commit access to only trusted collaborators
- **Github repository owner's profile gets hacked**: Enable two-factor authentication and setting a strong password.
- **DDoS attack**: sign up for some dos protection service such as Cloudfare, limit the number of request each user can send, add CAPTCHA to stop bots from creating accounts.
- **The cloud provider getting hacked**: store backups at different providers as well as offline
- **Nuget Package getting hacked**: peg the version of every nuget package to a specific version and only update that said package once it its integrity can be confirmed
- **IP Spoofing/Eavesdropping to get password**: We can use an https protocol and we can make strict restrictions on the passwords like minimum 8 characters with upper and lower chase characters and at least one numerical digit. Furthermore, we can use salt and pepper in our password hashing algorithm.