DevOps 2021

Malou :)

3 May 2021

Contents

System Design

Minitwit is a social media application that provides basic Twitter-like services. It consists of a web app and API services that are publicly available on the internet. Both services allow the user to register a profile, log in, create messages (tweets), follow and unfollow users. Basic authentication is required when creating messages, following, or unfollowing. Most of the application is written in Python, since our web app and API are using the Django framework.

One of the first things that our application needed was a database to store user info and posts. We chose to use a containerized PostGreSQL server.

Minitwit also consists of monitoring tools, which the web app and API communicate with when certain metrics are updated. These tools include Prometheus and Grafana, which allow for collection and displaying of metrics respectively. The monitoring tools are very useful for us developers, since they help us to maintain the system properly.

For logging features, we have implemented an EFK stack that includes Filebeat, Elastic Search, and Kibana. Filebeat is responsible for harvesting the data that we want to log, while Elastic Search is used to store that data in a database. With the logging features implemented, it is much easier for developers to diagnose and debug problems with the system. It also helps us in maintaining the system.

Architecture

Minitwit is hosted on multiple Digital Ocean droplets which form a Docker Swarm. Our logging and production database are each containerized on their own separate Droplets. This allows us to easily horizontally scale everything besides our persistent data, which should not be horizontally scaled anyways.

With a project that requires a web app, database, and API, it is normal to have the web app communicate with the database through the API. However, Django is designed such that direct communication with a database backend is much simpler to implement than communication with a custom backend server. For this reason, our web app and API don't communicate with one another, and therefore don't form a frontend/backend structure. Instead, our database is our backend and our web app/API servers are our frontends.

****We chose to use a PostGreSQL client that initially was hosted on the same, but we changed it to blah bla h ablsdlsad**** django with postgres - lecture 3???

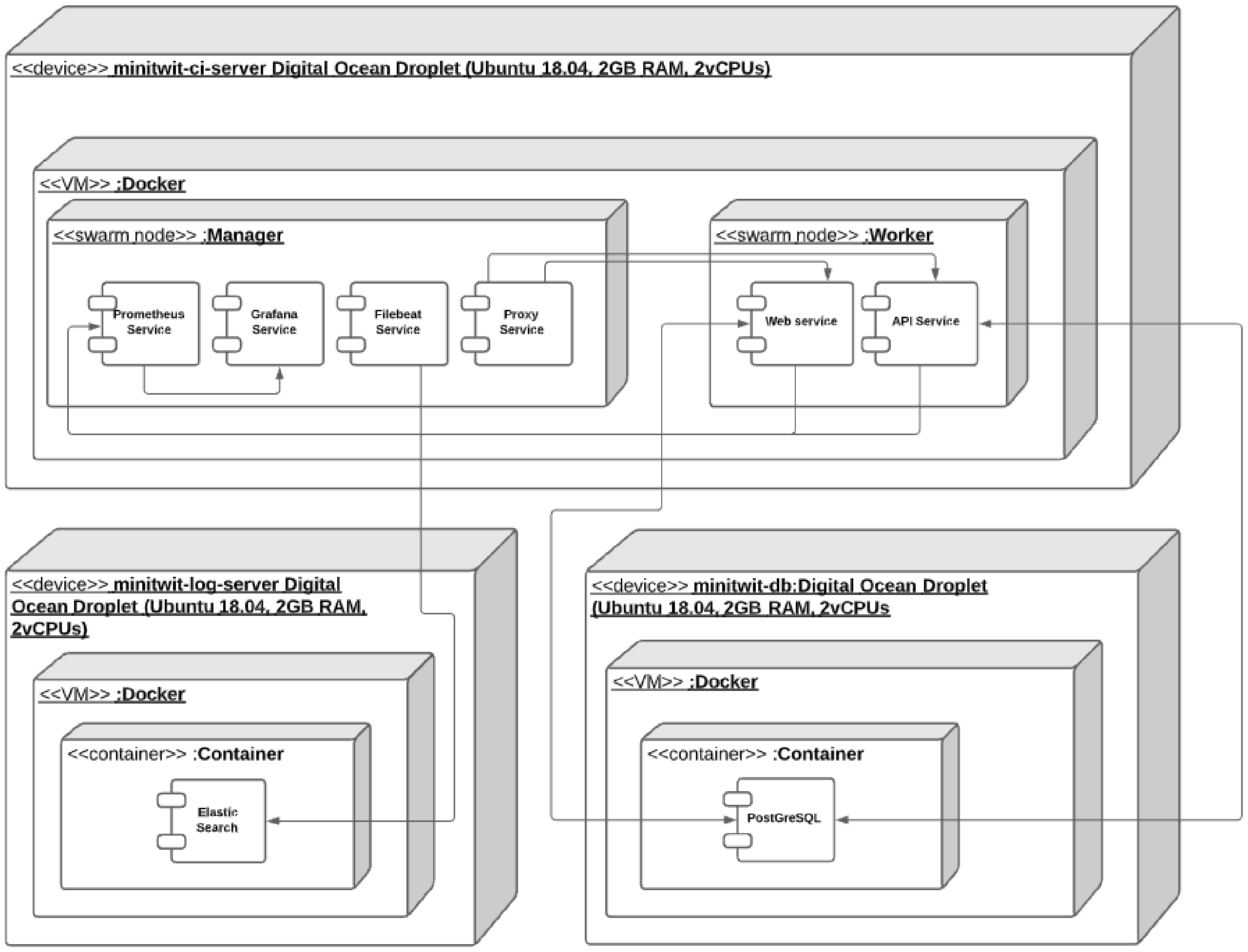

Figure 1: Deployment Diagram

We have a proxy service that uses nginx to route traffic from minitwititu.xyz to our web app server's IP address, and from api.minitwititu.xyz to our API server's address. It only exposes those two IPs, so all of the logging and monitoring related IPs are not exposed.

Our logging is accomplished by our Filebeat service, along with a logging database that hosts an Elastic Search instance. Filebeat scrapes the swarm manager's output, including all standard output for all services in the stack, and then logs relevant data in the logging database (Elastic Search). This logging database is used by Kibana to display our log information in a neat and readable website.

Our monitoring is accomplished by Prometheus, which exposes our metrics on minitwitwitu.xyz/metrics. Our web app and API both make calls to update certain Prometheus metrics, and Prometheus gathers other performance-related metrics from both of them. The /metrics route is also checked by our Grafana service, which hosts a webpage in which metrics can be monitored through customizeable dashboards.

Dependencies

Our dependencies are split into direct dependencies and tools

Tools

- Docker | Cloud computing services
- Digital Ocean | Cloud infrastructure provider
- Travis | Hosted continuous integration service
- ElasticSearch | Distributed RESTful search and analytics engine
- Kibana | Data visualization dashboard software for ElasticSearch
- Filebeat | File harvester
- PostGreSQL | Database Manangement System
- NGINX | (Web Server used for reverse proxy)
- Flake8 | Python style consistency
- Black | Python code formatter
- SonarQube | Code quality inspector, bugs, vulnerabilities
- Code Climate | Test coverage
- Better Code Hub | Quality improvements

why nginx, so we dont have ssl cetificate https?? lying around.. så de ikke centraliced managemant of ssl certicifacte.

Application Dependencies

Web App dependencies are as follows:

- asgiref 3.3.1 - Includes pytest a framework that makes it easy to write small tests
- django 3.1.8 - Python Web Framework
- django-prometheus 2.1.0 - Export django monitoring metrics for Prometheus
- django-rest-framework 3.12.2 - Web APIs for Django
- prometheus 0.9.0 - Prometheus instrumentation library
- psutil 5.8.0 - Python system monitoring
- psycopg2 2.8.6 - PostgreSQL database adapter for Python
- pycodestyle 2.7.0 - Python style checker
- pytz 2021.1 - Cross platform timezone calculations
- requests 2.25.1 - HTTP library
- sqlparse 0.4.1 - SQL query parser / transformer
- uWSGI 2.0.18 - 2.1 - Web service gateway

API dependecies are as follows:

- asgiref 3.3.1 - Includes pytest a framework that makes it easy to write small tests
- django 3.1.8 | Python Web Framework
- django-prometheus 2.1.0 - Export django monitoring metrics for Prometheus
- markupsafe 1.1.1 - Safely add untrusted strings to HTML/XML markup
- psutil 5.8.0 - Python system monitoring
- psycopg2 2.8.6 - PostgreSQL database adapter for Python
- pytz 2021.1 - Cross platform timezone calculations
- requests 2.25.1 - HTTP library
- sqlparse 0.4.1 - SQL query parser / transformer
- toml 0.10.2 - Python library for TOML
- uWSGI 2.0.18 - 2.1 - Web service gateway
- wrapt 1.12.1 - A Python module for decorators, wrappers and monkey patching.

Current state

Get the fucking grafana to tell us something xD

License

We collected all the license for every dependency we have to form the license our product. Here we met the GNU GPL v2 for psycopg2, The GPL series are all copyleft licenses, which means that any derivative work must be distributed under the same or equivalent license terms. To cover the product we therefore chose to go with the GNU General Public License v3.0; can be found in the licence document. In this process we also collected all copyright noticies for the dependencies, these are all placed in the Notice document.

The Team

The team is organized via Discord. We build a server to support all communication through that. Here we meet up, discuss meetings, solutions, problems and more. We also created a Github webhook, so that everyone gets a notification on the Discord server, whenever there are made changes to the project.

We use discord servers to meet up every Monday during and after our lecture, we usually work on the given task most of that day. The work that we did not finish are usually completed in the weekend, because of very different time schedules during the week.

CI/CD pipeline

Our CI/CD chain is run everytime we commit to any branch.

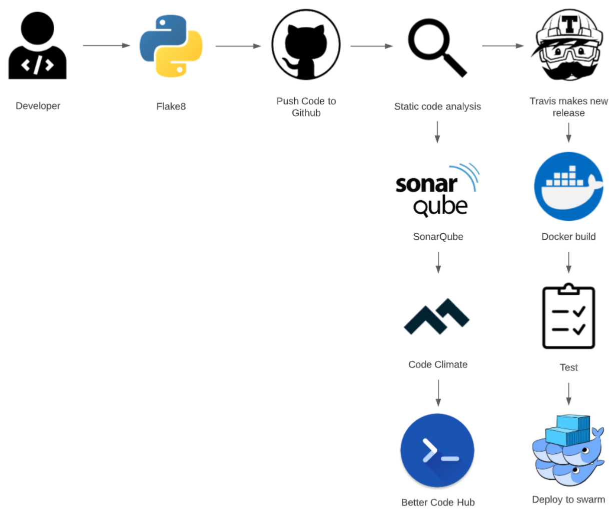

Figure 2: CI/CD pipeline

Git hook

Pre commit

Before commiting, we have a git hook that runs **Flake8** to enforce style consistency across our Python project.

Github

The developer code is then pushed to Github…

Static code analysis

We use 3 static analysis tools as software quality gates into your CI/CD pipeline.

We use **SonarQube** for continuous inspection of code quality. It performs automatic reviews with static analysis of the code to detect bugs, code smells, and security vulnerabilities in our project.

We use **Code Climate** for test coverage?

We use **Better Code Hub** for quality improvements?

Travis

Our Travis setup consist of 3 jobs; build, test, deploy.

Docker build

The first job in our Travis setup is build. But, before we build we start out with getting acess to Travis by using our ssh keys. Afterwards, this job has 3 stages; login to docker, build the 3 docker images web, api and proxy and lastly push the 3 images.

Test

The second job is testing. Here we start out by migrating. Migrations are Django's way of propagating changes we make to our models (adding a field, deleting a model, etc.) into our database schema.

We only test frontend…

Deploy

The third job is deploy and we only deploy when we commit to the main branch, which is how we make a new release. Before deploying we set up the git user and tag the commit. The final step is pulling the images and deplying to the swarm.

- deploy token?

Repository

what, where, why We have organized the project in one repository … We split logging into a separate repository because …

Branching strategy

Our branching strategy utilizes the Gitflow Workflow. Basically we will have 2 main branches and a number of feature branches:

- main branch - This is the main branch and contains the production code.
- developer branch - This is the development branch containing the development code. This code is merged and pushed into main at the end of each completed weekly assignment.
- feature branches - Used to develop specific features relating to a specific assignment and is merged and pushed into the development branch when it is completed.

Branches should always be branched from develop

Always pull the newest development branch before creating a feature branch

Experimenting is preferably done in branches. In some cases when working on a specific task it might make sense to further branch out from the feature branch eg.

Development process

For this project we worked with an agile development process. Agile is all about moving fast, releasing often, and responding to the needs of your users, even if it goes against what's in your initial plan. Every week we would plan a new implementation, work on the implementation in smaller bids, test it and release it for feedback session the next lecture. If we had any bugs or backlog from the last week we would split up and work on tasks in smaller groups.

We used Github Projects to organize tasks in Kanban boards. We created 2 boards to separate tasks on the frontend and backend at the beginning of the project.

We later used Github issues to maintain a backlog over the tasks for the group to take on.

Monitoring

We use the monitoring service Prometheus, to monitor our application… To display the data that we get from Prometheus, we use the web-based multi-source graph interface, Grafana.

We split our monitoring into 2 Grafana dashboards; Business Monitoring (e.g. images/Business Monitoring), which displays our PostgreSQL Query data, and Infrastructure Monitoring (e.g. images/Infrastructure Monitoring), which displays our Prometheus metrics.

The Business Monitoring dashboard contains amount of: Users, Messages and Followers. We used these data to monitor the correctness of successful requests.

The Infrastructure Monitoring dashboard contains CPU Load percent, used for up-time calculation as well as strain on the system and HTTP Responses (Frontend / Backend), used to monitor system failure as well as performance in regards to correct response.

Logging

…

Security

…

Scaling and load balancing

… # Leasons learned