DAT515 Project

Static Website with Contact Form

Marodeen Shamonei, Maciej Ruchala
University of Stavanger, Norway
m.shamonei@stud.uis.no,m.ruchala@stud.uis.no

KEYWORDS

AWS, Flask, Python, SQLite3, Docker

1 INTRODUCTION

Throughout the DAT515 course, we learned to implement different technologies such as Docker on a cloud based virtual machine (VM). It was the goal of this project to utilize all of these technologies on one project in order to demonstrate our knowledge.

The selected project was a simple static website with a contact form. Given that web design was not a focus in the course, the final web page design is rudimentary and our main focus was on the back-end. In the course, we were given access to an OpenStack VM, but given the popularity of cloud computing platforms like Amazon Web Services (AWS), we opted instead to put our knowledge from Lab 3 into practice and use AWS for compute with an EC2 instance. On the EC2 instance, we set up a web server using Nginx, which hosted the HTML and CSS of our web page. Again, because web design was not a primary focus, the Bootstrap front-end framework was used to design the web page. The web page's back-end functionality, which processes the form and stores it in a database, was written using Flask. Docker was leveraged to implement containerization of the web server and Flask app.

2 DESIGN

The project was designed to be a web application that could operate within the constraints of the AWS free tier. By selecting a t3.micro EC2 instance, we aimed to achieve a balance between resource availability and the low-cost requirements of the project. Though this instance provided only 1 GB of memory, which was insufficient for running Kubernetes, it was capable of hosting our core application. To ensure consistency in access, an Elastic IP was assigned to the instance, allowing a persistent public IP address for the server, even if the instance was stopped and restarted.

The front-end of the application was designed with simplicity and responsiveness in mind. Using the Bootstrap framework, we developed a clean, user-friendly interface that would display consistently across devices. The layout included a navigation bar and a form where users could submit basic information, such as their name, email, and feedback. This form was structured to capture essential data without overwhelming users, while maintaining an organized, professional appearance.

For the web server, Nginx was chosen due to its high performance and compatibility with our static HTML setup. Nginx was configured to serve the static front-end content while also acting as a reverse proxy to the back-end Flask application. This setup allowed Nginx to efficiently manage incoming requests, while directing application-specific requests to Flask, making the architecture both modular and manageable.

The back-end was built with Flask, chosen for its simplicity and effectiveness in handling lightweight applications. Flask was responsible for processing form submissions, connecting to a SQLite3 database to store user inputs, and sending email confirmations upon successful submissions. The database was kept minimal with SQLite3, a file-based database, which suited the limited scope of data and fit well with the small-scale requirements of the project.

To streamline deployment and improve maintainability, we opted to containerize the application using Docker Compose. By separating Nginx and Flask into individual containers, we simplified the setup, allowing for independent configuration and easy replication of the environment. Though we could have containerized the SQLite3 database, we decided against it given its simplicity and limited data requirements.

Security was a key consideration in the design. We integrated HTTPS for secure data transmission, using a free SSL certificate from Let's Encrypt. This required a custom domain, so we purchased marodeen.com through Cloudflare. Cloudflare's DNS services allowed us to point this domain to our Elastic IP, ensuring users could access the site securely through a dedicated URL.

3 IMPLEMENTATION

3.1 Compute Setup on AWS

To meet project budget and requirements, we opted for the AWS free tier, deploying a t3.micro EC2 instance with 1 GB of memory. This limited our ability to implement Kubernetes due to memory requirements but provided adequate resources for other project functions.

We assigned an Elastic IP to the EC2 instance, ensuring the public IP address would remain consistent across reboots, which helped in maintaining persistent access and was particularly useful for domain and SSL configuration. SSH access was configured to connect to the instance securely. Key-based SSH access was used, ensuring security while avoiding password-based login risks.

3.2 Front-end Development

The front-end interface was built using Bootstrap for responsive design and ease of use. A simple HTML file contained a navigation bar and a form, which asked users for basic information such as email, name, and feedback.

The HTML file was moved to /var/www/html on the EC2 instance to make it available through the Nginx server. This allowed static serving of the front-end content.

Supervised by Hein Meling.

Project in Computer Science (DAT515),
2024.

3.3 Web Server Setup with Nginx

We installed Nginx on the EC2 instance as our web server. Nginx's configuration was tailored to point to the /var/www/html directory, where the front-end HTML file was stored.

Nginx served the static HTML content of the page, while also acting as a reverse proxy for the Flask app.

3.4 Back-end Implementation with Flask

A Flask app was developed to handle server-side functionality. The app contained logic to connect to an SQLite3 database for storing form submissions. This database was selected for simplicity and ease of setup.

Flask's built-in support for SQLite3 was used, and the database file was stored locally on the instance.

To confirm form submission, the Flask Mail library was used to send an email to the user upon successful form receipt. We configured Flask Mail with SMTP settings appropriate for sending these confirmation emails.

We encountered some challenges regarding the deployment port for the Flask app and configuring it to handle submissions from the webpage. Since ports 80 and 443 were already allocated to the server for hosting the main webpage, we decided to run the Flask app on port 5001. To connect these components, we configured Nginx to use a reverse proxy. While the main website traffic is directed to port 443, any requests to /submit are forwarded to port 5001, where the Flask app can handle them. The Flask app then listens specifically for POST requests to /submit, processes the form data, stores it, and sends a confirmation email to the user.

3.5 Containerization with Docker Compose

To streamline deployment, Docker Compose was used to define two main containers: one for Nginx and another for the Flask app. Docker Compose simplified the orchestration of these containers and allowed quick setup and teardown of the environment.

Nginx and Flask were set up as separate services, with Nginx handling front-end requests and proxying back-end requests to Flask. The SQLite database, being file-based and small in size, was not containerized.

One challenge we encountered was ensuring the Flask app could run automatically without manually starting it from the terminal each time. To address this, we utilized the Gunicorn tool, which is designed to serve Python web applications in a production environment. We modified the Dockerfile to include a command that runs the Flask app on port 5001 using Gunicorn. This setup allowed us to skip the manual startup process, as each time we deploy the containers with docker compose up, the Gunicorn command automatically starts the Flask app for us.

3.6 HTTPS and SSL Configuration

To ensure secure data transmission, we configured the web page to use HTTPS. We obtained an SSL certificate from Let's Encrypt, which provides free SSL/TLS certificates. This certificate was installed and configured within Nginx.

As Let's Encrypt requires a domain, we purchased the domain marodeen.com from Cloudflare. Cloudflare was also configured to handle DNS settings, pointing the domain to the Elastic IP of our EC2 instance.

4 DEPLOYMENT

The deployment process for this application was designed to be as efficient and straightforward as possible, leveraging Docker Compose to simplify the setup of our containerized environment. With both Nginx and the Flask app containerized, deploying the application became as simple as running the docker compose up command. This single command initiates both the Nginx server and the Flask app on their designated ports, bringing up the entire application stack in one step.

One of the primary goals in the deployment phase was to ensure that no manual intervention was required to start or manage individual services. By configuring the Dockerfile for the Flask app to automatically start with Gunicorn, we eliminated the need to manually execute the Flask script each time the application was deployed. This approach allowed the Flask app to reliably start on port 5001 whenever docker compose up is executed, saving time and ensuring consistency across deployments.

One issue that arose during deployment was that the server within the Docker container couldn't access the SSL certificates, which were stored directly on the EC2 instance. This happened because Docker containers are isolated and don't automatically have access to files on the host machine. After some troubleshooting, we realized that to resolve this, we needed to modify the docker-compose.yml file. Under the volumes section for the Nginx container, we specified the directory containing the SSL certificates on the EC2 instance, effectively mounting it within the container. This adjustment allowed Nginx to locate and use the certificates for HTTPS, ensuring secure connections without further issues.

Overall, Docker Compose proved to be an effective solution, enabling a smooth and automated deployment process. This setup not only minimized the risk of misconfiguration but also ensured that each deployment was quick, reproducible, and aligned with the intended design of the application.

5 CONCLUSION

In conclusion, this project successfully demonstrated the integration of various technologies to create a functional web application hosted on AWS. We effectively utilized the free tier of AWS and implemented a containerized architecture with Docker Compose, allowing us to run both the Nginx server and Flask app seamlessly. While we achieved our immediate goals, there are several areas for improvement that could enhance the application's performance and scalability in the future.

One significant improvement would be to upgrade to an instance with more memory, which would allow us to implement Kubernetes for container orchestration. By leveraging Kubernetes, we could gain better control over scaling, load balancing, and managing our containerized services. This would not only improve the reliability of our application under varying loads but also streamline deployment processes and enhance overall resource utilization.

Additionally, migrating to a more sophisticated database solution, such as PostgreSQL or MongoDB, could provide greater capabilities for data management and scalability. Containerizing this database

would further ensure that our development and production environments remain consistent and easily manageable.

Furthermore, we could enhance the application's security by implementing additional measures, such as automated security updates for the containers and using tools like Let's Encrypt for SSL certificate renewal. Implementing a continuous integration and deployment (CI/CD) pipeline would also streamline our development workflow, allowing for more efficient testing and deployment of new features.

Overall, the foundation laid in this project presents a strong starting point, and with these enhancements, we can significantly elevate the application's performance, scalability, and security in future iterations.