MODULE 169: FULL GUIDE FOR DOCKER CONTAINER

Microservice Application written in python. Application makes simple requests to the database and displays the values on a frontend.

M169: Deploy services with containers

Table of Contents

Introduction

This document serves as a comprehensive educational guide, providing essential information for replicating this project. Some steps in this guide are written in a general context and may not be explicitly tailored to the specifics of my project. For a more concise version that closely follows the guidelines, please consult my GitHub repository.

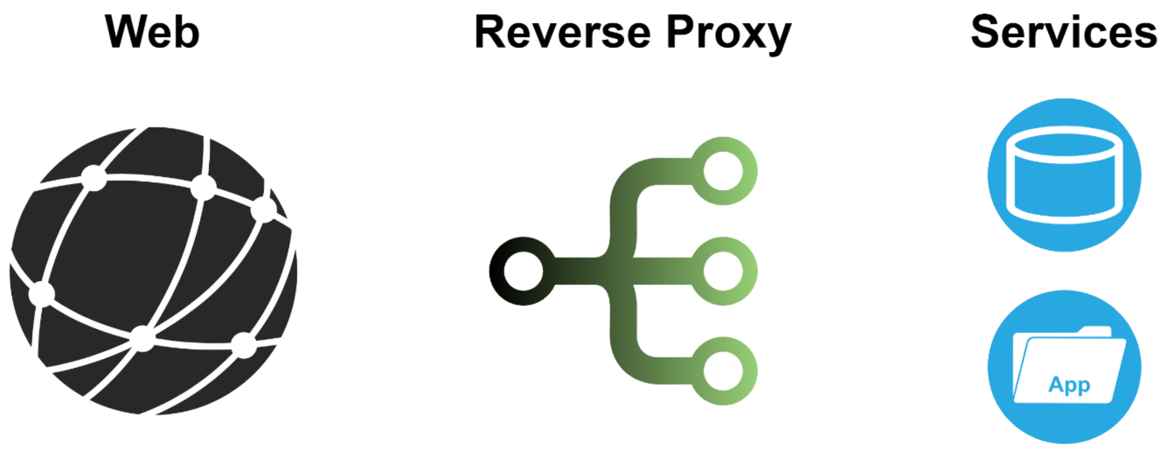
Introduction: Scope of Project

My goal is to develop a functional web application capable of interacting with a database. The application will be programmed using Python, specifically utilizing the Flask framework. While Flask's built-in server is an option, it's not suitable for production environments. Therefore, I've opted to use Gunicorn (Green Unicorn), a WSGI (Web Server Gateway Interface) server designed for such purposes. WSGI servers facilitate communication between Python web applications and web servers.

To containerize the application, I'll create a customized Docker image and deploy it within a Docker network alongside the database server. For database management, I've selected MariaDB along with PhpMyAdmin to simplify user creation and credential management. This choice also improves readability within the codebase, as the mysql-connector-python library utilizes a more intuitive syntax.

While enhancing security, one consideration is implementing a credential or configuration file to store database credentials in hashed form. This approach aligns with standard security practices. However, in the interest of readability, I've chosen not to abstract the code further. It's important for beginners and students with varying levels of expertise to be able to comprehend the code easily.

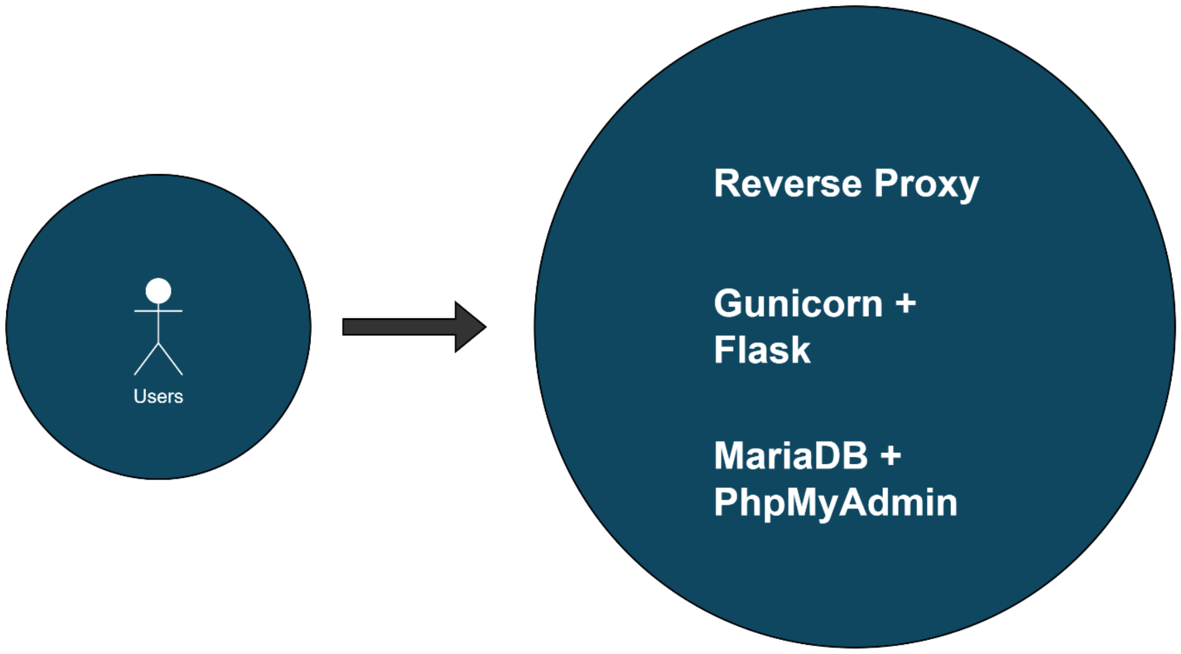
Furthermore, I'll deploy a reverse proxy to enhance security by preventing direct access to the database server from the public network. A reverse proxy acts as an intermediary, directing external requests to the appropriate service behind it, thus bolstering the security of the backend infrastructure. However, it's important to acknowledge that while these measures enhance security, they aren't without limitations.



Introduction: Logical docker network

I mentioned in the previous step how I'll make the docker application more secure. This can be done as previously mentioned with a reverse proxy (nginx, Cloudflare, Apache) and following best practices for the coding of the main application like utilizing .conf files for credentials.

I began to make a rough draft of the logical network; this is my first draft:



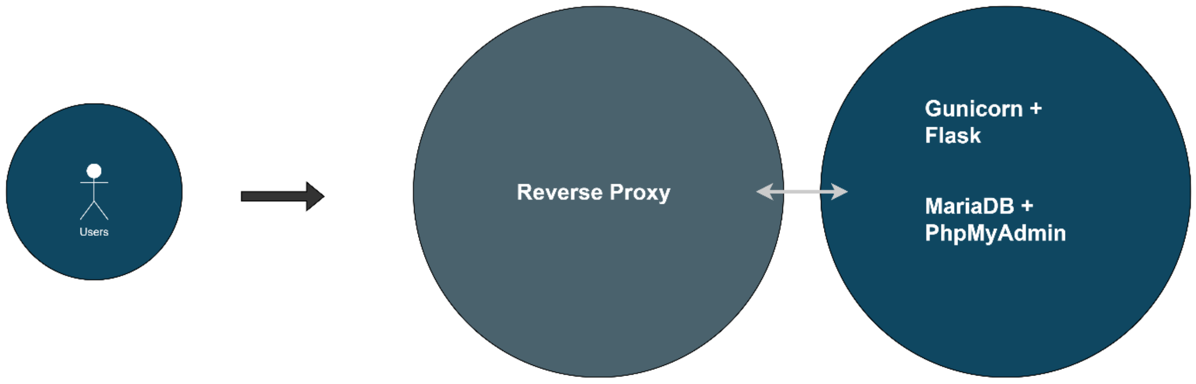
I then analysed my first draft thoroughly and there were some points that stood out to me:

1. **Insecure communication:** If not correctly configured the communication between the reverse proxy and the backend services could be vulnerable to attackers that eavesdrop on sensitive data.
2. **SSRF (Server-Side Request Forgery):** A poorly configured reverse proxy can be vulnerable to SSRF attacks. An attacker might be able to trick the reverse proxy into making unintended requests to internal systems.
3. **Lack of segmentation:** Flat networks where all services have unrestricted access to each other increase the surface for an attacker to attack on. Segmenting the network into two separate or using VLANs would be a better solution to this.

It also sports some points that would speak for it:

1. **Simplified Network Architecture:** Having the reverse proxy in the same network as other services simplifies network configuration.
2. **Performance:** With the reverse proxy in the same network there is lower latency and faster response times since requests don't need to traverse between networks and overstep their origin networks boundaries.
3. **Ease of configurations:** Placing the reverse proxy in the same network as other services help simplify the configuration and maintenance of said reverse proxy and network.

I wasn't satisfied with the outcome of my initial draft, so I decided to make another draft, this time taking the flaws into consideration and doing my best to correct them:



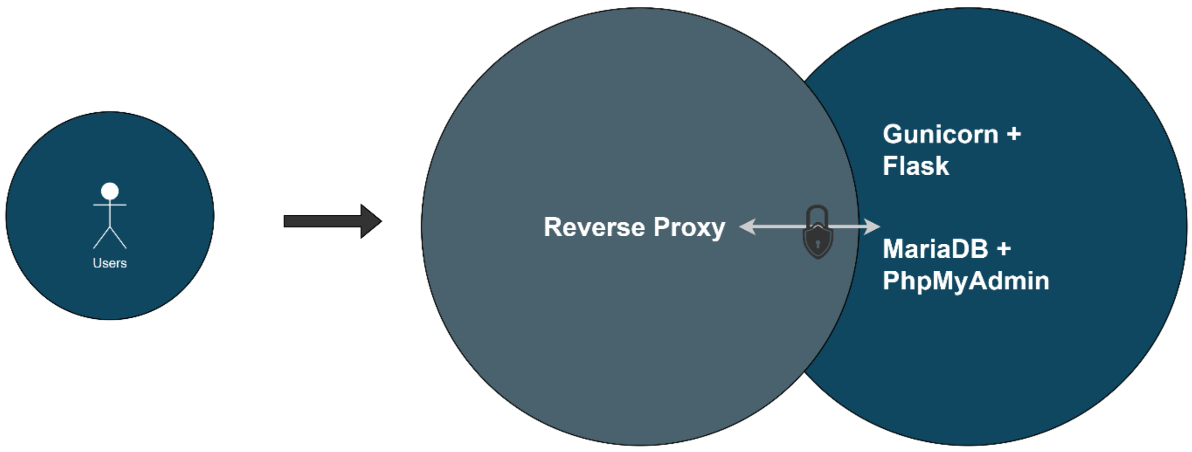
After sleeping a day and airing out my head at the gym, I've noticed some few details that might've gone by if not looked at with a fresh mind, here are some of them:

1. **Single Point of Failure:** The reverse proxy is a single point of failure in this configuration. Meaning if it goes down or gets compromised users won't be able to access the application. However, this is unavoidable since it's the safest option.
2. **Lack of encryption:** In this diagram I completely forgot to encrypt the connection between the reverse proxy and the backend services. Thus, attackers could eaves drop on the conversation if they'd position them between the reverse proxy and the backend services.
3. **Complexity in configuration:** In this diagram the reverse proxy is in another network with a different subnet. This means that the two networks natively can't communicate with each other. Making facilitating a stable communication between the two networks that more complicated.

There are some very good points as well:

1. **Security Isolation:** Placing a reverse proxy in an outwards facing network (also called DMZ) provides and additional layer of protection for the internal network. It acts as a buffer between the public and internal network.
2. **SSL Termination:** The outward-facing reverse proxy can handle SSL termination, decrypting incoming HTTPS traffic and then forwarding it to internal services over plain HTTP. This in turn offloads the SSL/TLS decryption workload from internal servers, improving performance.
3. **Web Application Firewall (WAF)**: Placing a reverse proxy in the outward-facing network allows to implement a WAF to inspect incoming traffic, filtering out malicious requests and thus protecting the applications from common web-based attacks such as SQL-injection and cross-site scripting (XSS).

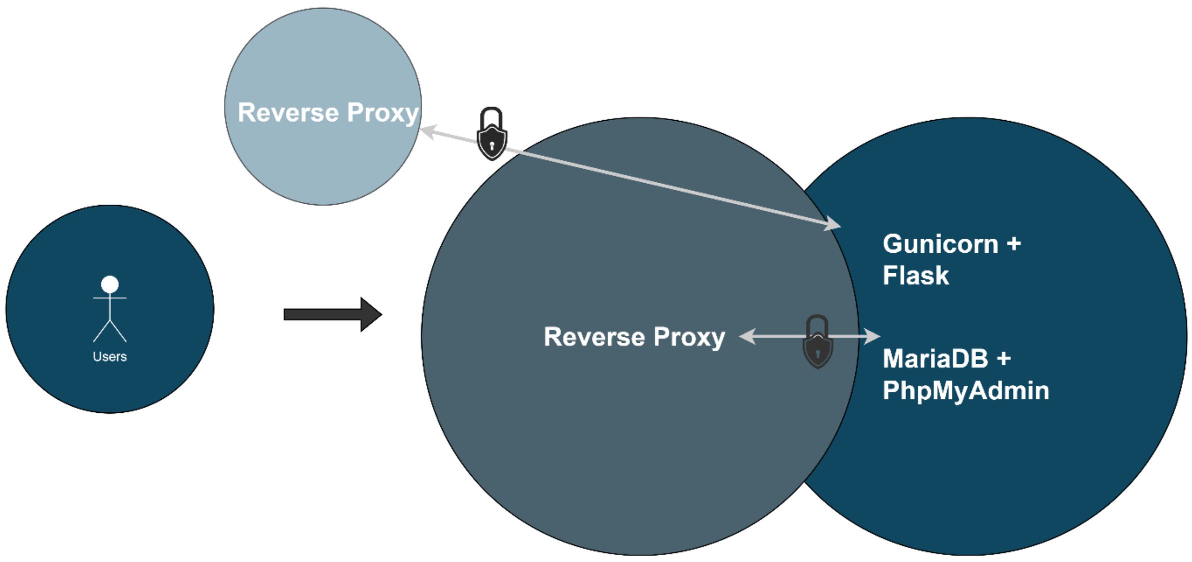
With all the consideration in mind, I aimed to create one final draft that would be the final logical network plan / docker network plan which I would use in my project:



Looking over this logical network architecture it combines the best of both drafts:

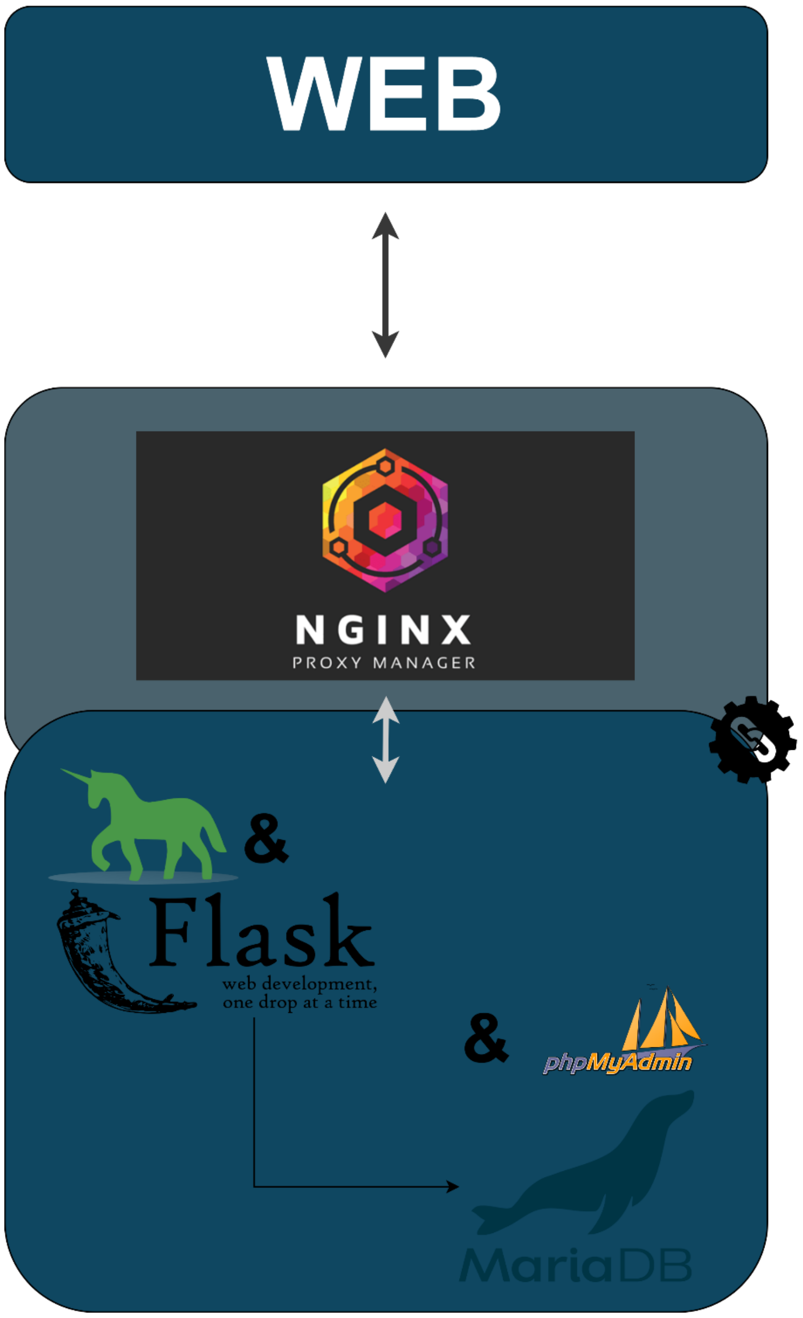
1. Ease of configuration: Making the reverse proxy be in two networks Internal and external, makes configuring the communication between the two networks that easier.
2. Encryption: In this draft I've made sure that the communication/connection between the two networks is secured via SSL/TLS. This can be done inside the reverse proxy by saying that it uses a certificate by Let's Encrypt.
3. DMZ Segmentation: Placing the reverse proxy in a DMZ, which is a network segment that exposes services to the internet while maintaining separation from the internal network, allows for controlled access to internal resources.
4. Granular Logging and Monitoring: Deploying a reverse proxy in both network zones allows for comprehensive logging and monitoring of inbound and outbound traffic. Allowing to track patterns, detect potential security threats and troubleshoot issues more effectively.
5. Security Isolation: Placing the reverse proxy in two different network zones enhances security by isolating external-facing and internal-facing traffic. It creates a clear boundary between the external and internal network.

These are only some of the many benefits of the final draft however there is still the problem of the single point of failure, to mitigate this you can create two reverse proxies that do the same:



Also, in this kind of configuration you could enable load balancing so that traffic would be distributed evenly between the two reverse proxies. However, this would be very taxing for my computer’s performance, thus higher latency would be a problem, This wouldn’t be a problem on a more performant computer.

Introduction: Docker container architecture



Proxy-Network

Where Nginx reverse proxy manager will be.

Linked via R-Proxy

The two networks are "linked" in the sense that you can access the services in the backend over the Reverse Proxy, because the reverse proxy is in both networks at the same time.

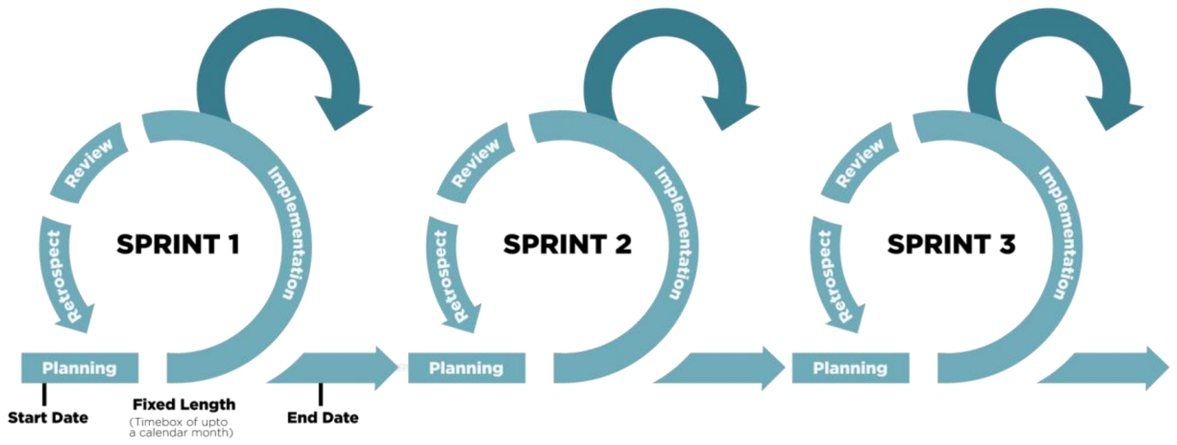
Backend-Network

Where all the backend services will be like MariaDB, phpMyAdmin and the WSGI server

Which project planning method

For the project planning method, I will use the Agile Sprint methodology, a staple of modern software development. Agile Sprint is a dynamic framework known for its iterative and incremental approach, designed to foster flexibility and responsiveness.

Within the Agile Sprint framework, development cycles, known as sprints, are structured into short, time-boxed iterations. These iterations typically last from one to four weeks, during which cross-functional teams collaborate closely to deliver a tangible, potentially shippable product increment.



One of the core principles of Agile Sprint is its emphasis on continuous improvement. At the beginning of each sprint, the team collectively defines a set of prioritized user stories or tasks from the product backlog. Throughout the sprint, the team works diligently to implement these requirements, holding daily stand-up meetings to discuss progress, identify potential obstacles, and adjust the plan accordingly. This iterative approach allows for frequent feedback from stakeholders and end-users, enabling the team to make timely adjustments and deliver value incrementally.

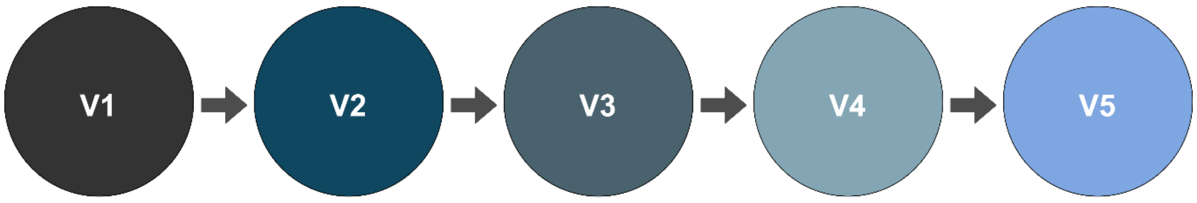
By adopting the Agile Sprint methodology, we aim to achieve several key benefits. Firstly, it promotes transparency and visibility, as progress is regularly tracked and communicated to all stakeholders. Secondly, it fosters collaboration and accountability among team members, encouraging a shared sense of ownership over project outcomes. Thirdly, it enables us to deliver tangible results quickly and adapt to changing priorities or market conditions with ease.

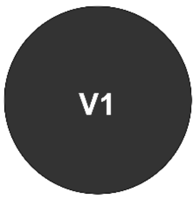
In summary, the Agile Sprint methodology offers a structured yet flexible approach to project planning and execution, aligning closely with our goal of delivering high-quality software solutions efficiently and effectively.

Major milestones

As I said before I will use the project planning method, agile sprint. The agile sprint method is a framework used in agile software development for iterative and incremental development. It does so by breaking down a project into smaller, manageable chunks.

For this I want to break my project down into five smaller chunks note this only concerns the application:





Version 1: Creating the Framework

The goal of the first version is to create a framework or a base on which I can slowly but surely add more and more to it.

Meaning that this version is basically only a flask application that serves a basic index.html which I then can configure to my liking.

Version 2: Adding Character

The goal of the second version is to add more character to the plain index.html for that I used a CSS and JS Framework called UIKIT.

I like UIKIT since it has a minimalist look to it and is easy to implement and does not require any further dependencies like Node.js or package loaders.

In the initial testing phase of the second version the CSS and JS framework wouldn't want to work, so I researched a little bit and I found out that you need the following lines of code to make it work:

For CSS linking:

```
<link rel="stylesheet" href="{{ url_for('static', filename='css/uikit.min.css') }}">
```

For JS linking:

```
<script src="{{ url_for('static', filename='js/uikit.min.js') }}"></script>
<script src="{{ url_for('static', filename='js/uikit-icons.min.js') }}"></script>
```

V3

Version 3: Adding functionality

The goal of the third version is to add a functionality to the application meaning that the main file `app.py` could use blueprints.

In other words, blueprints can be used like additional modules you would install into your car, meaning that they add more functionality to your project.

Each blueprint has its own file for example you want to make the main file `app.py` use an API that constantly requests performance data from a device and display it onto to the frontend then you might use a blueprint to do so.

To make the main file `app.py` use the function from the perfmon functions you can do so the following:

Important the file/module from where it's located in the code repository:

```
From pymodules.perfmon import perfmon_bp
```

This way you can use the methods/function from the blueprint `perfmon` to link the blueprint to the main file, do the following now:

```
app.register_blueprint(perfmon_bp)
```

Of course, this is just an example in my code I actually used the following lines:

```
From pymodules.inserter import inserter_bp
From pymodules.fetcher import fetcher_bp
```

And

```
app.register_blueprint(inserter_bp)
app.register_blueprint(fetcher_bp)
```

V4

Version 4: Adding a layer of security

In this iteration of the application, the primary focus is on bolstering security measures.

In previous versions, I relied on the root user for database access, which isn't considered a best practice. To mitigate potential risks, I've now established a dedicated user within MariaDB via phpMyAdmin.

This new user is specifically configured with limited privileges, allowing only select statements and table creation. This approach significantly enhances security compared to using the root user, which inherently possesses the ability to execute potentially destructive commands.

V5

Version 5: Final version | containerizing the application

In the latest iteration, version five, I aimed to containerize the application from the previous iteration, bringing it into a Docker environment.

However, this transition required some adjustments to the credentials. The primary change involved specifying a different hostname. Why the change? Well, it's because the container operates within the Docker network.

By utilizing the hostname 'host.docker.internal', the application can seamlessly access the database within this environment. This tweak ensures that the application functions smoothly within its Dockerized setup.

But how to containerize a application, its actually very easy. Create a Dockerfile and input the following:

```
FROM python:3.10
MAINTAINER Rayan Lee bopp <rayanleeboppcastoulooraya@gmail.com>
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir --upgrade -r requirements.txt
EXPOSE 5000
EXPOSE 3306
COPY . .
CMD ["gunicorn", "--bind", "0.0.0.0:80", "app:create_app()"]
```

Running this with the command: `docker build -t rayanleebopp/m169-flask-application:v1` will create a image called rayanleebopp/m169-flask-application with the tag v1.

Then after successfully creating the image you can run your own docker container with the following command:

```
docker run --network m169-project -dp 5000:5000 -w /app -v "$(pwd):/app"
rayanbopp/test-flask:v1 flask run --host 0.0.0.0
```

Congratulations the application is now done.

EXTRA

Version 6 (Extra): reverse proxy | docker compose

As previously discussed, ensuring a secure environment is a priority for me. As part of this effort, I've incorporated an Nginx reverse proxy manager into the setup.

To streamline the process further, I created a custom Nginx reverse proxy manager image and developed a Docker Compose file. This approach significantly simplifies the setup of the entire Docker environment.

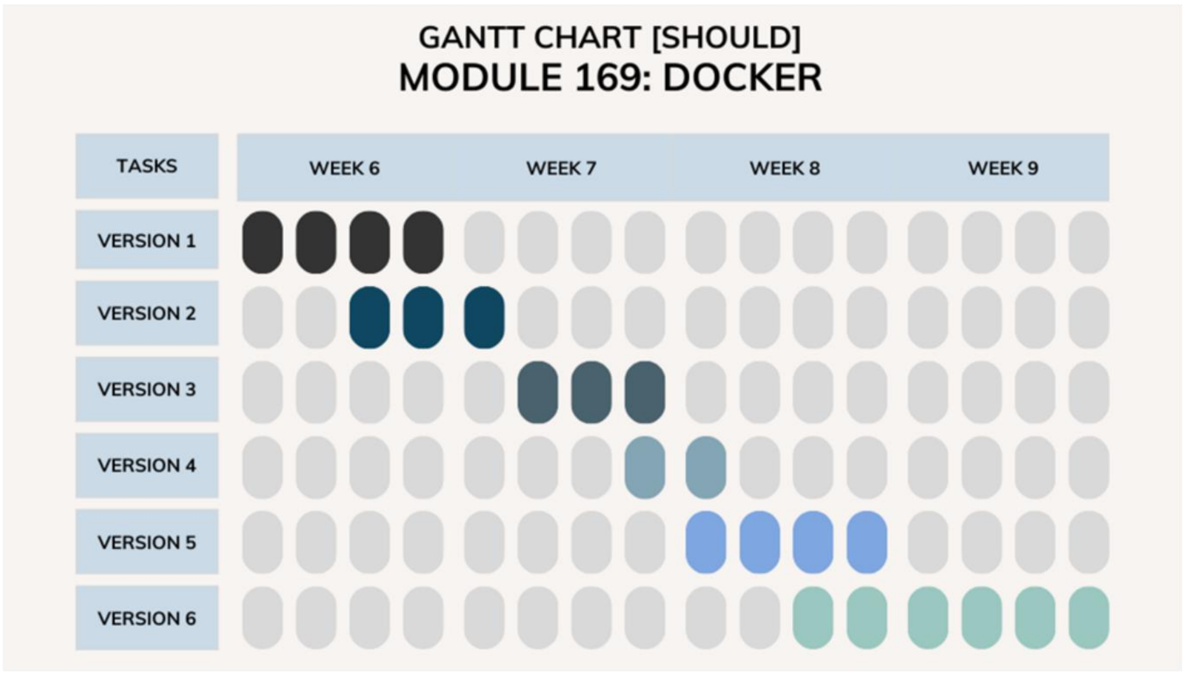
Project Management

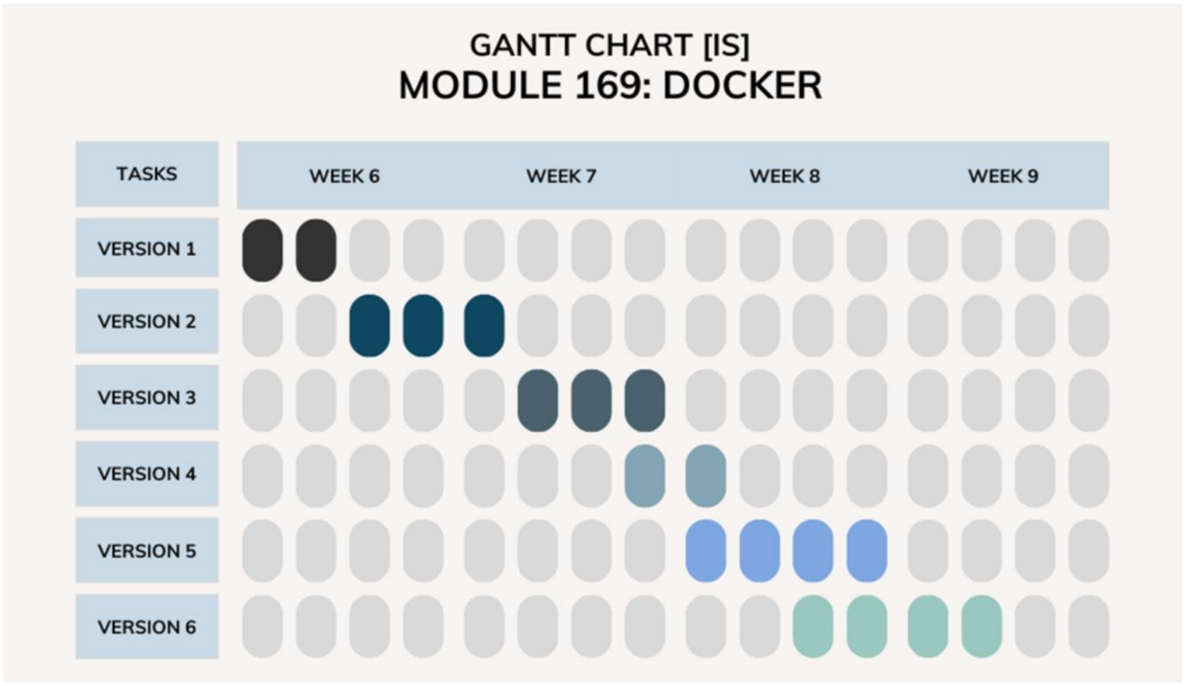
Timeframe for each Task

Version	Description	Timeframe (hours)
V1	The primary objective of the initial version is to establish a foundational framework that can be expanded upon gradually. This version consists primarily of a Flask application serving a basic index.html file. Its purpose is to provide a flexible starting point that can be easily customized and extended according to specific project requirements.	
V2	The goal of the second version is to enhance the basic index.html by integrating the UIKIT CSS and JS framework. Despite encountering initial issues during testing, research led to the inclusion of specific lines of code to ensure the framework's functionality.	
V3	The objective of the third version is to introduce functionality to the application by incorporating blueprints into the main file, app.py. Blueprints act as modular extensions, enhancing project capabilities akin to installing additional features in a car. Each blueprint corresponds to a specific functionality, such as integrating an API to retrieve and display performance data from a device onto the frontend.	
V4	In this iteration, the focus is on improving security measures. I've transitioned from using the root user to a dedicated user with restricted privileges in MariaDB via phpMyAdmin. This enhances security by minimizing potential risks associated with database access.	

V5	In version five, the focus was on containerizing the application from the previous iteration into a Docker environment. This transition necessitated adjustments to the credentials, primarily changing the hostname. As the container operates within the Docker network, using the hostname 'host.docker.internal' enables seamless access to the database within this environment. This adjustment ensures smooth functionality within the Dockerized setup."	
EXTRA	Prioritizing a secure environment, I've integrated a Nginx reverse proxy manager into the setup, as discussed earlier. To enhance efficiency, I've crafted a custom Nginx reverse proxy manager image and crafted a Docker Compose file. This streamlined approach greatly simplifies the setup of the Docker environment.	

Gantt chart

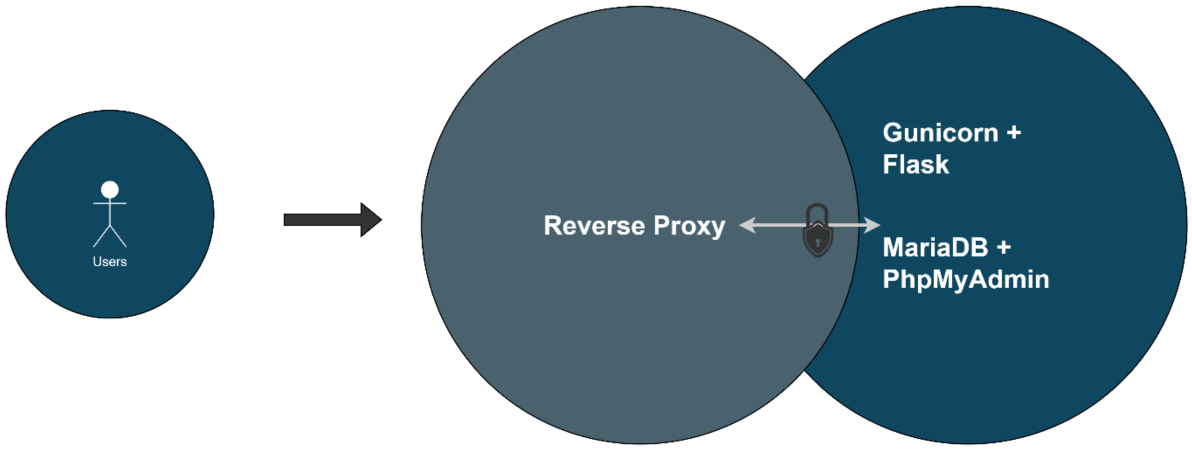

GANTT CHART [SHOULD]
MODULE 169: DOCKER
TASKS
WEEK 6
WEEK 7
WEEK 8
WEEK 9
VERSION 1
VERSION 2
VERSION 3
VERSION 4
VERSION 5
VERSION 6


GANTT CHART [IS]
MODULE 169: DOCKER
TASKS
WEEK 6
WEEK 7
WEEK 8
WEEK 9
VERSION 1
VERSION 2
VERSION 3
VERSION 4
VERSION 5
VERSION 6

Functionality

Functionality: project

The functionality of the project is actually pretty simple, lets take the high level overview graphic from the beginning in this document:



This graphic as mentioned before provides a high-level overview, although it isn't a docker container structure it nevertheless provides us with a good level of understanding of how the docker application operates.

The users which come over the internet will always interact with the reverse proxy as its the only service, which is pointing outwards, meaning its the only point of entry.

The reverse proxy intercepts incoming user requests and redirects them to the backend service. It sits between the clients (users) and the servers (backend), as depicted in the graphic above. When a client sends a request to access a resource, such as a web page or an API endpoint, the reverse proxy receives the request on behalf of the server.

It knows where to redirect by evaluating the requests and determining which server it should handle by analysing various factors. These factors can be load balancing algorithms, server health checks, URL patterns or header information.

After redirecting the request of the user, the backend server processes this request and sends it back to the reverse proxy, the reverse proxy then sends this back to the user. The user then receives his processed request directly from the reverse proxy unaware of the fact that it wasn't the reverse proxy processing the request.

Now you have a good overview on how the user interacts with the application.

Functionality: application

Here I will focus more on the functionality of the application.

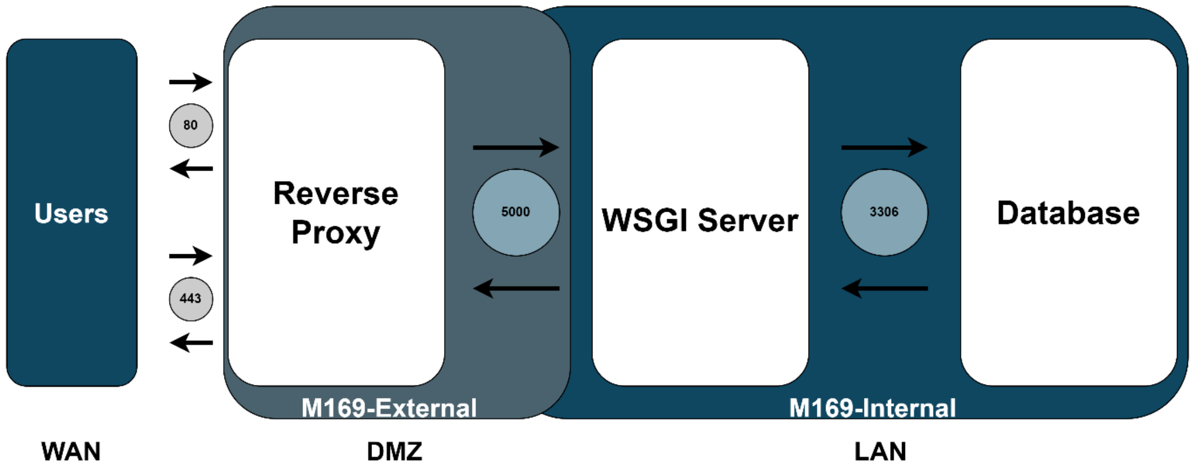
As Explained before the Users access the backend services over the reverse proxy, they communicate with the reverse proxy over port 80 (HTTP) and port 443 (HTTPS). The reverse proxy will then process these requests based on information and will redirect these requests to the appropriate service and port.

In our case the reverse proxy communicates over 5000 a nonspecific port, with the WSGI server on which the application runs.

The application, or rather the flask script that runs on the WSGI Server under port 5000 then establishes a communication with the database over port 3306. This can happen because the script knows which credentials to enter due to them being defined within the script. It also can select statements because it uses an object called `.cursor`.

The WSGI Server that runs the flask script runs a synchronous service, since it only performs one action at one given time, meaning that when the user requests info from the database it only processes this one request from the user and not anything further. That means a user can get info from the database to be displayed on the frontend and request another set of data, with the previous data displayed being overridden on the frontend.

- **Data Retrieval**: Initially, the data was fetched from the database, which may have involved either synchronous or asynchronous interactions with the backend, depending on the implementation.
- **Data Display**: Once the data is fetched and available to the frontend, displaying it on the frontend is a synchronous process. The frontend simply renders the pre-fetched data without needing to wait for any additional requests or interactions with the backend.



After sending back the data requested data the reverse proxy sends the processed user request back to his browser and it seems as if the reverse proxy had processed his request.

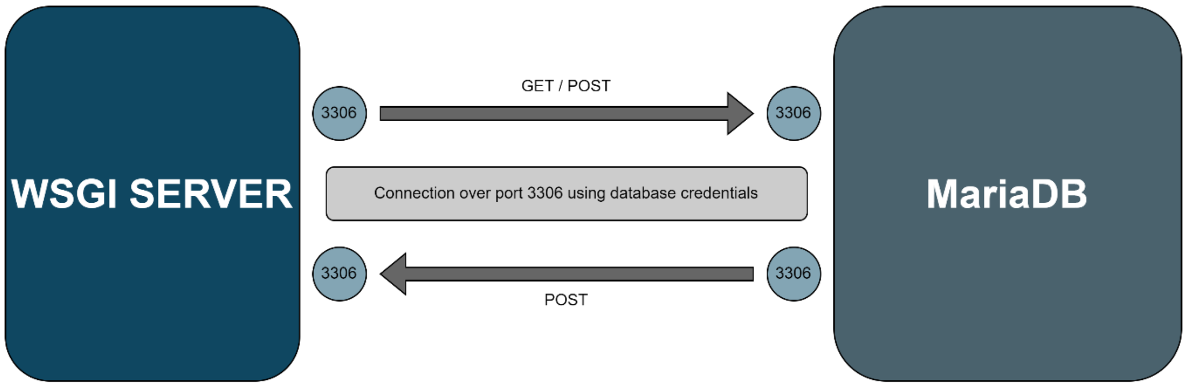
Functionality: docker container communication

Docker containers can communicate in different ways with each other. Take for example MariaDB and phpMyAdmin. When I initially setup the two containers to test over localhost I didn't add them to a docker network I used the link function to link them together.

The `--link` function under docker creates a network link between the two containers in question. However, this is only one of the many ways to make them communicate with each other. Let me tell you three more:

1. **Shared Volume:** Docker containers can share data between them by using shared volumes. This is done by mounting a volume from the host machine or other container to another one. This allows for data sharing and synchronization. It also allows, as mentioned in my GitHub wiki, horizontal scaling. Horizontal scaling adds nodes and using shared volumes for them allow for seamless horizontal scaling since they run on the same data.
2. **Inter-Container Communication:** Docker containers can communicate with each other directly using inter-container communication by making use of UNIX-sockets or named pipes. Making use of such methods allows containers to interact with each other's filesystems, execute commands and/or exchange data.
3. **Orchestration Tools:** Orchestration Tools like Docker Swarm or Kubernetes (K8s) enable containerized applications to be deployed and managed across multiple hosts or clusters. They utilize `.yaml` files often named `route` or something like that to establish connection between the containerized applications.

In my case my containerized application only communicates via network, meaning over TCP/IP connections since I use the database credentials which include port and hostname. As seen in this graphic:



Synchronous and Asynchronous

Sync and Async: Definition

1. **Synchronous:**
    - In synchronous operation, tasks are executed one after the other in a sequential manner.
    - Each task waits for the previous one to complete before starting.
    - Synchronous operations are blocking, meaning that the execution of code halts until the current task finishes.
    - Examples of synchronous operations include traditional function calls, where the program waits for the function to return a result before proceeding.
2. **Asynchronous:**
    - In asynchronous operation, tasks are executed concurrently or independently of each other.
    - Tasks can start and run in parallel without waiting for each other to finish.
    - Asynchronous operations are non-blocking, meaning that the program can continue to execute other tasks while waiting for asynchronous tasks to complete.
    - Examples of asynchronous operations include callback functions, promises, and async/await constructs commonly used in event-driven programming or when dealing with I/O operations.

Aspect	**Synchronous (Sync)**	**Asynchronous (Async)**
Execution Model	Sequential execution; tasks run one after the other	Concurrent execution; tasks run independently
Blocking	Blocks further execution until task completes	Does not block further execution; allows parallelism
Wait	Waits for each task to finish before moving on	Does not wait for tasks to finish; allows non-blocking behavior
Performance	May lead to slower performance if tasks are lengthy	Often leads to better performance due to parallelism
Example	Traditional function calls	Callbacks, Promises, async/await constructs

Sync and Async: Example

Synchronous (Sync) File Download Service:

In a synchronous file download service, when a client requests to download a file, the client sends a request to the server and waits until the file is fully downloaded before proceeding with any other tasks. Here's a basic pseudo-code example:

```python
# Synchronous File Download Service

def download_file_sync(file_url):
    # Send request to server
    response = send_request(file_url)

    if response.status_code == 200:
        # Save the file locally
        save_file_locally(response.content)
        print("File downloaded successfully")
    else:
        print("Error downloading file")

# Example usage
download_file_sync("http://example.com/file.txt")
```

In this synchronous example, the client initiates the download and waits until the entire file is downloaded before moving on to any other tasks. If the file is large or the server response is slow, the client will be idle during this time.

Asynchronous (Async) File Download Service:

In an asynchronous file download service, the client doesn't wait for the file to be fully downloaded. Instead, it continues with other tasks and is notified once the download is complete. This can improve efficiency, especially when dealing with large files or multiple downloads simultaneously. Here's a pseudo-code example using Python's asyncio library:

```python
import asyncio

# Asynchronous File Download Service

async def download_file_async(file_url):
    # Send request to server asynchronously
    response = await send_request_async(file_url)

    if response.status_code == 200:
        # Save the file locally
        save_file_locally(response.content)
        print("File downloaded successfully")
    else:
        print("Error downloading file")

async def main():
    # List of file URLs to download asynchronously
    file_urls = [
        "http://example.com/file1.txt",
        "http://example.com/file2.txt",
        "http://example.com/file3.txt"
    ]

    # Start asynchronous downloads
    await asyncio.gather(*(download_file_async(url) for url in file_urls))
# Example Usage
asyncio.run(main())
```

In this asynchronous example, multiple files can be downloaded simultaneously without waiting for each download to complete. The asyncio.gather() function is used to execute multiple coroutines concurrently. Once all downloads are complete, the program exits.

Sync and Async: My Project – MariaDB

The service provided by MariaDB can be considered synchronous in most cases, especially when handling traditional SQL queries.

Synchronous Nature of MariaDB:

- **SQL Queries:** MariaDB primarily processes SQL queries in a synchronous manner. When a client sends a SQL query to MariaDB, the database engine executes the query sequentially and returns the result to the client. During this process, the client typically waits for the database to finish executing the query before proceeding.
- **Blocking Operations:** SQL queries in MariaDB are blocking operations, meaning that the database connection is typically occupied while the query is being processed. Other operations or queries may need to wait until the current query completes before they can be executed.

Giveaways that the Service is Synchronous:

- **Blocking Behavior**: MariaDB's synchronous nature is evident from its blocking behavior. When a client sends a query to MariaDB, it typically waits for the query to complete before proceeding with further operations.
- **Sequential Execution**: MariaDB processes SQL queries sequentially, one after the other. This sequential execution suggests a synchronous workflow, where each query must finish before the next one can be executed.

Communication Process:

The communication process between a client and MariaDB typically follows these steps:

1. **Client Sends Query**: The client sends a SQL query to the MariaDB server over a network connection or a local socket.
2. **MariaDB Receives Query**: The MariaDB server receives the query and processes it sequentially. The server may perform various operations, such as parsing, optimizing, and executing the query.
3. **Query Execution**: During query execution, the database server accesses data stored in tables, performs calculations, and applies any necessary locks to ensure data consistency.
4. **Result Retrieval**: Once the query execution is complete, MariaDB returns the result set to the client over the network connection or socket.
5. **Client Receives Result**: The client receives the result set and can process or display the data as needed.

Conclusion:

In summary, the service provided by MariaDB is predominantly synchronous, especially when processing SQL queries. The blocking behavior and sequential execution of queries are key indicators of its synchronous nature. The communication between clients and MariaDB involves sending queries and receiving results over network connections or local sockets, following a request-response pattern.

Sync and Async: My Project – phpMyAdmin

Determining whether phpMyAdmin is a asynchronous or synchronous service depends which distribution/configuration you use. Below is the explanation for both asynchronous and synchronous.

Synchronous nature of phpMyAdmin

phpMyAdmin, being a web-based application, often operates **synchronously**. When a user interacts with **phpMyAdmin**, such as querying a database or navigating through tables, the application typically processes each request sequentially.

Upon receiving a request, **phpMyAdmin** performs the necessary operations, such as executing SQL queries or fetching data from the database, before generating a response and sending it back to the user.

Each user action typically blocks subsequent actions until the current one is completed, which is characteristic of synchronous behaviour.

Giveaway that the service is synchronous:

- **Blocking Behavior**: Users may experience delays or unresponsiveness if phpMyAdmin is busy processing a long-running task, indicating synchronous behavior where one operation must complete before the next begins.
- **Sequential Execution**: Operations performed in phpMyAdmin, such as executing SQL queries or exporting data, typically occur in a step-by-step manner, reflecting synchronous behavior.

Communication Process:

1. **Client Request**: The user opens a web browser and navigates to the URL where PHPMyAdmin is hosted, typically something like http://example.com/phpmyadmin.
2. **HTTP Request**: The browser sends an HTTP request to the web server hosting PHPMyAdmin. This request includes information like the URL, HTTP method (usually GET or POST), and any additional headers.
3. **Web Server Processing**: The web server (such as Apache or Nginx) receives the request and forwards it to the PHP interpreter if the requested resource is a PHP script.
4. **PHP Script Execution**: PHP interprets the PHPMyAdmin script. PHPMyAdmin consists of multiple PHP scripts that generate HTML content dynamically based on user interactions.
5. **Database Connection**: PHPMyAdmin establishes a connection to the MySQL or MariaDB server specified in its configuration. This connection allows PHPMyAdmin to send queries to the database and retrieve results.
6. **Authentication**: If authentication is enabled, PHPMyAdmin prompts the user to enter their username and password. These credentials are typically compared against a

MySQL/MariaDB user account to ensure the user has permission to access the databases.

7. **User Interface Generation**: Once authenticated, PHPMyAdmin generates the user interface dynamically based on the user's privileges and the structure of the databases and tables available on the MySQL/MariaDB server.
8. **User Interaction**: The user interacts with the PHPMyAdmin interface by clicking links, submitting forms, and entering SQL queries. These interactions trigger further HTTP requests to PHPMyAdmin.
9. **Query Processing**: When the user submits a query or performs an action (such as creating a table or importing data), PHPMyAdmin processes the request. It may sanitize and validate user input to prevent SQL injection attacks.
10. **Database Operations**: PHPMyAdmin translates the user's actions into SQL queries and sends them to the MySQL/MariaDB server via the established database connection. The server executes these queries and returns the results (if any) to PHPMyAdmin.
11. **Response Generation**: PHPMyAdmin processes the results received from the database server and generates HTML content to display to the user. This content might include tables of data, forms for further interaction, or error messages.
12. **HTTP Response**: PHPMyAdmin sends the HTML content back to the user's web browser as an HTTP response. The browser renders this content, allowing the user to view and interact with the database management interface.
13. **Repeat**: The user can continue to interact with PHPMyAdmin, initiating further requests and receiving responses as needed until they decide to log out or close the browser.

Conclusion:

phpMyAdmin's synchronous behavio aligns with its traditional web application architecture, where user interactions are processed sequentially, one at a time.

While this approach ensures simplicity and ease of understanding for users, it may lead to potential scalability challenges under heavy load, as each request must be processed before the next one can be handled.

Asynchronous nature of phpMyAdmin

As said before determining whether phpMyAdmin is synchronous or asynchronous is based on the configuration the user/admin has made.

I will now explain when a phpMyAdmin service can be considered and asynchronous service and what the giveaways are:

In some configurations or deployments, phpMyAdmin can be made to operate asynchronously by leveraging technologies such as AJAX (Asynchronous JavaScript and XML) or background processing.

Asynchronous behavior can be introduced to handle long-running tasks, such as executing complex SQL queries or performing data imports, without blocking the user interface.

By offloading intensive tasks to background processes or worker threads, phpMyAdmin can continue to respond to user interactions while simultaneously processing tasks in the background.

Knowing all that, we can now determine what my deployment of phpMyAdmin is. In this case my deployment of phpMyAdmin is a synchronous service.

Giveaways that the service is asynchronous:

- **Non-Blocking Operations:** Tasks that would typically block the user interface, such as importing large datasets or executing time-consuming queries, may complete in the background while the user continues to interact with phpMyAdmin.
- **User Feedback:** Asynchronous operations may provide feedback to the user on task progress or completion through notifications or status indicators, allowing them to continue working without waiting for tasks to finish.

Communication Process:

Similar to the synchronous version of phpMyAdmin the asynchronous version of phpMyAdmin communicates with the user's web browser via HTTP(S) requests and responses, However the asynchronous model involves additional communication between the client and server to provide real-time updates or monitor task progress.

1. **User Interaction**: The user interacts with the phpMyAdmin interface by performing actions such as executing SQL queries, browsing databases, modifying table structures, etc.
2. **Client-Side JavaScript**: When the user initiates an action, such as submitting a form or clicking a button, JavaScript code embedded in the phpMyAdmin interface intercepts this action.
3. **AJAX Request Initiation**: The JavaScript code constructs an AJAX request using the XMLHttpRequest object or fetch API. This request typically includes information such as the action to be performed (e.g., executing a SQL query), any parameters or data required for the action, and the target URL (PHP script) on the server.
4. **Sending the Request**: The constructed AJAX request is sent asynchronously to the server. Asynchronous means that the request is sent without blocking the user interface, allowing the user to continue interacting with the page while the request is being processed.
5. **Server-Side Processing**: On the server side, PHP scripts receive the AJAX request. These scripts are responsible for handling the request, executing the necessary actions (e.g., querying the database), and generating a response.
6. **Database Interaction**: If the AJAX request involves database operations (which is often the case in phpMyAdmin), the PHP script interacts with the MySQL or MariaDB database server. This interaction may involve executing SQL queries, retrieving data, updating records, etc.
7. **Response Generation**: After processing the request and performing any necessary database operations, the PHP script generates a response. This response typically includes data to be sent back to the client (e.g., query results, status messages) and is often in formats such as JSON or XML.
8. **Sending the Response**: The generated response is sent back to the client asynchronously.
9. **Client-Side Response Handling**: Once the response is received by the client-side JavaScript code, it processes the response data and updates the user interface accordingly. For example, if the user executed a SELECT query, the JavaScript code may render the query results in a table within the phpMyAdmin interface. If there are any

errors or status messages returned by the server, they are displayed to the user as appropriate.

10. **Continued Interaction**: The user can continue interacting with the phpMyAdmin interface, initiating further actions that trigger additional AJAX requests and responses, thus maintaining an asynchronous flow of communication between the client and server.

Conclusion:

Asynchronous phpMyAdmin deployments offer improved responsiveness and scalability by allowing long-running tasks to execute in the background while users continue to interact with the application.

Leveraging asynchronous techniques can enhance the user experience by reducing wait times and providing real-time feedback on task progress.

In summary, phpMyAdmin can operate synchronously or asynchronously depending on its configuration and deployment environment. While synchronous operation is common for traditional web applications.

Asynchronous techniques can be employed to improve responsiveness and scalability in certain scenarios.

Sync and Async: My Project – flask application

Given that I've already explain how my flask application function I don't want to waste much time here explaining it over again. Normally a WSGI server that runs a flask application can either be synchronous or asynchronous depending on what modules/libraries the developer used and what the intended use of the flask application is.

The intended use of my application is trivial in nature. Making simple database requests over the frontend and then waiting for the response from database to the script to then paste the given values onto the frontend.

Synchronous nature of flask application

- **Simple Database Requests**: Since the service only makes simple database requests, it's likely that these requests are straightforward and do not involve long running or blocking operations. Synchronous execution is sufficient for handling such requests efficiently.
- **Displaying Data on the Frontend**: Similarly, displaying data on the frontend typically does not require asynchronous processing. The WSGI server can handle requests synchronously and serve the frontend content as needed.

Giveaways that the service is synchronous:

- **No Complex Processing:** The absence of complex or long-running operations suggests that synchronous processing is suitable.
- **No Concurrent Operations**: As there's no need for concurrent or parallel processing, synchronous execution is sufficient.

Communication Process:

The WSGI server (e.g., Gunicorn, uWSGI) communicates synchronously with the Flask application, handling each request sequentially.

Flask, in turn, communicates synchronously with the database server, making simple database requests and awaiting their responses before proceeding.

Conclusion:

Given the simplicity of the service, synchronous execution is suitable and efficient. It effectively handles simple database requests and serves frontend content without the need for asynchronous processing.

Sync and Async: My Project – nginx reverse proxy manager

As with phpMyAdmin a reverse proxy can be either synchronous or asynchronous based on its configuration. You can easily determine whether a reverse proxy is synchronous or asynchronous by looking at how the traffic is being handled but also what additional features it uses. (**Port forwarding, load balancing, SSL termination, web acceleration**)

Now let me explain what traffic processing type is synchronous and what type is is asynchronous.

Synchronous nature of nginx reverse proxy manager

In a synchronous reverse proxy manager, each incoming request is handled one at a time, with the server waiting for each request to be fully processed before moving on to the next. This approach may be suitable for environments with low to moderate traffic where simplicity and predictability are prioritized.

Giveaways that the service is synchronous:

1. Requests are processed in the order they are received, with each request blocking subsequent requests until completed.
2. Resource usage (CPU, memory) may be higher during periods of high traffic or long-running requests.

Communication:

1. **User Interaction**: The process begins with a user interacting with the Sync NGINX Reverse Proxy Manager, typically through a web interface or a command-line interface.
2. **Configuration Input**: The user provides input to the Sync NGINX Reverse Proxy Manager, such as specifying the backend servers, the domains or paths to be proxied, SSL certificates, caching settings, and other configuration parameters.
3. **Validation and Parsing**: The Sync NGINX Reverse Proxy Manager validates and parses the user's input to ensure it conforms to the NGINX configuration syntax and security best practices. This step helps prevent errors and ensures the resulting configuration is correct and secure.
4. **Generation of NGINX Configuration**: Based on the validated input, the Sync NGINX Reverse Proxy Manager generates the NGINX configuration files required to set up the reverse proxy. These configuration files typically include directives such as `server`, `location`, `proxy_pass`, `ssl_certificate`, `ssl_certificate_key`, and others, depending on the user's input.
5. **Communication with NGINX**: Once the NGINX configuration files are generated, the Sync NGINX Reverse Proxy Manager communicates with the NGINX server to apply the new configuration. This communication can occur through various methods, such as:
   - **File System**: The Sync NGINX Reverse Proxy Manager may write the generated configuration files directly to the NGINX configuration directory on the file system, triggering NGINX to reload its configuration.
   - **API**: Some implementations of the Sync NGINX Reverse Proxy Manager may communicate with NGINX via its API, sending commands to reload the configuration or dynamically update specific parts of it without requiring a full reload.

6. **NGINX Configuration Reload:** NGINX reloads its configuration to apply the changes made by the Sync NGINX Reverse Proxy Manager. During this process, NGINX checks the syntax of the new configuration files and, if they are valid, applies the changes without interrupting ongoing connections.
7. **Proxying Requests:** With the new configuration in place, NGINX now proxies incoming requests to the specified backend servers based on the rules and settings defined by the user through the Sync NGINX Reverse Proxy Manager.
8. **Response Handling:** NGINX receives responses from the backend servers and forwards them back to the client making the request. It may also apply additional processing, such as caching or modifying headers, based on the configuration provided by the user.
9. **Feedback to User:** Depending on the implementation, the Sync NGINX Reverse Proxy Manager may provide feedback to the user about the status of the configuration update, such as whether it was successful or if any errors occurred during the process. This feedback helps the user troubleshoot issues and ensures the proper functioning of the reverse proxy setup.

Conclusion:

A synchronous reverse proxy manager can be beneficial for scenarios where simplicity and ease of understanding are critical, but it may not be as efficient in high-traffic or latency-sensitive environments.

Asynchronous nature of nginx reverse proxy manager

In an asynchronous reverse proxy manager, incoming requests are handled concurrently, allowing the server to process multiple requests simultaneously. This approach can improve performance and scalability, especially in high-traffic environments.

Giveaways that the service is asynchronous:

Requests are processed concurrently, allowing the server to handle multiple requests simultaneously without blocking.

Resource usage may be more efficient, as the server can optimize resource allocation and prioritize tasks based on demand.

Communication Process:

1. **User Initiates a Request**: The sequence starts when a user initiates a request by accessing a web application or service through a web browser or any other client.
2. **DNS Resolution**: If the user is accessing a domain name, the client performs a DNS resolution to find the IP address of the server hosting the application.
3. **Connection Establishment**: The client then establishes a TCP connection with the server. If SSL/TLS is enabled, there might be an additional TLS handshake to establish a secure connection.
4. **Request Transmission**: After the connection is established, the client sends an HTTP request to the server. This request typically includes details such as the HTTP method (GET, POST, etc.), the requested URI, headers, and sometimes a payload for methods like POST.
5. **Reverse Proxy Receives the Request**: The request reaches the asynchronous Nginx reverse proxy manager. The reverse proxy listens for incoming requests on a specified port and IP address.

6. **Load Balancing (Optional)**: If the reverse proxy is configured for load balancing, it might distribute incoming requests across multiple backend servers based on predefined algorithms like round-robin, least connections, or IP hashing.
7. **Handling Asynchronous Requests**: Here's where the asynchronous nature of Nginx comes into play. Instead of blocking a worker process for each incoming request, Nginx employs an event-driven, asynchronous architecture. When an incoming request arrives, Nginx doesn't dedicate a worker process exclusively to handle it. Instead, it adds the request to an event loop and continues to process other requests. This allows Nginx to handle a large number of concurrent connections efficiently.
8. **Proxying the Request to the Backend Server**: The reverse proxy manager forwards the HTTP request to the appropriate backend server based on the configured rules. It modifies the request headers and possibly the URI if needed.
9. **Backend Processing**: The backend server receives the request, processes it, and generates an appropriate response. This could involve executing server-side code, querying databases, or accessing other resources.
10. **Response Transmission**: Once the backend server generates the response, it sends it back to the reverse proxy manager.
11. **Reverse Proxy Sends Response to the User**: The reverse proxy manager receives the response from the backend server and forwards it back to the user who initiated the request. It might also perform additional processing such as caching, compression, or modifying response headers before sending it to the client.
12. **Connection Closure**: After the response is sent back to the user, the connection between the client and the reverse proxy (and optionally between the reverse proxy and the backend server) is closed, freeing up resources.

Conclusion:

An asynchronous reverse proxy manager offers improved performance and scalability, making it suitable for high-traffic environments where responsiveness and resource efficiency are crucial.

In my project, however the reverse proxy manager is configured as a synchronous.

Preparation

To streamline the process, I've taken the initiative to proactively prepare. I've crafted a thorough plan that entails identifying key subject, documenting them, and executing them. This proactive approach ensures not only comprehension but also hands-on experience. This hands-on experience would prove to essential in streamlining the process.

Key subjects

I had the following key subjects:

1. Microservices: What are they? What are the advantages? What are the drawbacks?
2. Monolithic service: What are they? What are the advantages? What are the drawbacks?
3. Packaging services: How can I make a container myself?
4. Automation: Everything regarding automation
5. Security: Everything regarding security
6. Sync and Async: What is the definition? What is the difference?
7. Dockerfile: What is a Dockerfile? How to use a Dockerfile? Basic setup Dockerfile
8. Docker Networks: What is a docker network? What is the use case of a docker network? What are the different types of docker network? Basic setup docker network
9. Docker volumes: What are docker volumes? What is the use case for them? Basic setup docker volume
10. Docker compose: What is a docker compose? What is the use case of docker compose? Basic setup docker compose.

Let me explain all these key subjects very shortly:

1. **Microservices:** Small, independent software components. Advantages: Scalability, flexibility, technology diversity. Drawbacks: Complexity, increased management overhead.
2. **Monolithic service:** Single, unified software application. Advantages: Simplicity, easier deployment. Drawbacks: Lack of scalability, technology lock-in.
3. **Packaging services:** Create a container using Docker. Install Docker, write a Dockerfile, build the container, and run it.
4. **Automation:** Streamlining repetitive tasks using scripts or tools. Includes deployment automation, configuration management, and continuous integration/continuous deployment (CI/CD).
5. **Security:** Protecting systems, data, and users from threats. Includes network security, access control, encryption, and security testing.
6. **Sync and Async:** Sync refers to synchronous communication, where each task waits for the previous one to complete. Async refers to asynchronous communication, where tasks can continue independently.
7. **Dockerfile:** Text file defining the steps to create a Docker image. Write a Dockerfile, specify base image, add dependencies, and build the image.
8. **Docker Networks:** Networking feature in Docker. Enables communication between containers and other Docker entities. Types include bridge, overlay, and host networks. Basic setup involves creating a network and connecting containers to it.
9. **Docker volumes:** Persistent data storage in Docker. Used to share data between containers and host system. Basic setup involves creating a volume and attaching it to containers.

10. **Docker Compose:** Tool for defining and running multi-container Docker applications. Write a Compose file (YAML), specify services, networks, and volumes, and run the application using docker-compose up.

For further and a more in-depth guide to these key subjects please look read my GitHub Wiki. There you will find much more and the basic setup guide as well.

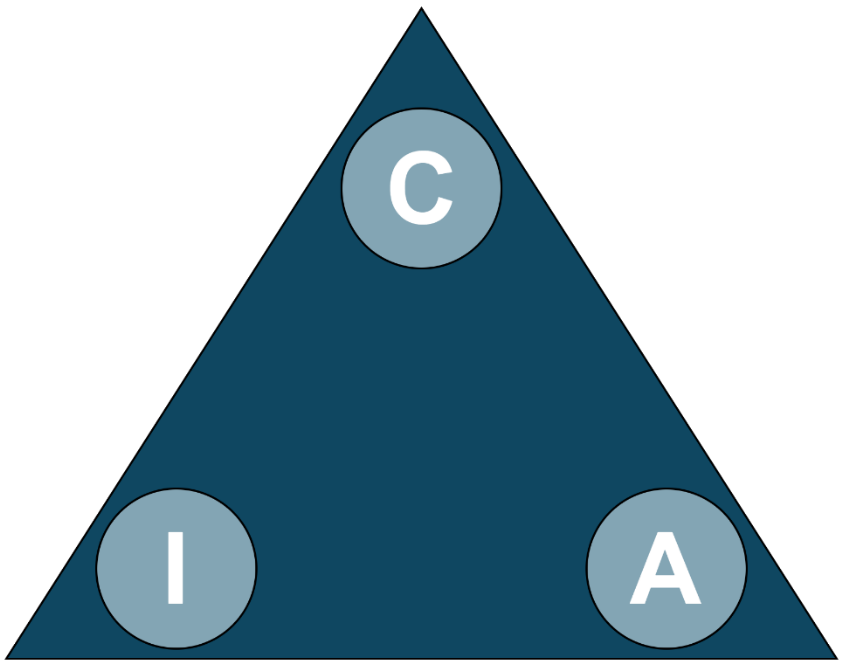
GitHub Wiki: Campus Castolo | m169

GitHub – Camus-Castolo/m169: Docker Services
Services mit Containern bereitstellen. Contribute to campus-Castolo/m169: Services mit Containern bereistellen by creating a pull request

Github.com

Security

When people think of security, they typically imagine protecting their digital devices and online activities from hackers, malware, viruses and other cyber threats, by using the means of VPNs and Software firewalls. While this is not completely wrong cybersecurity far extends beyond just simple VPNs and firewalls.



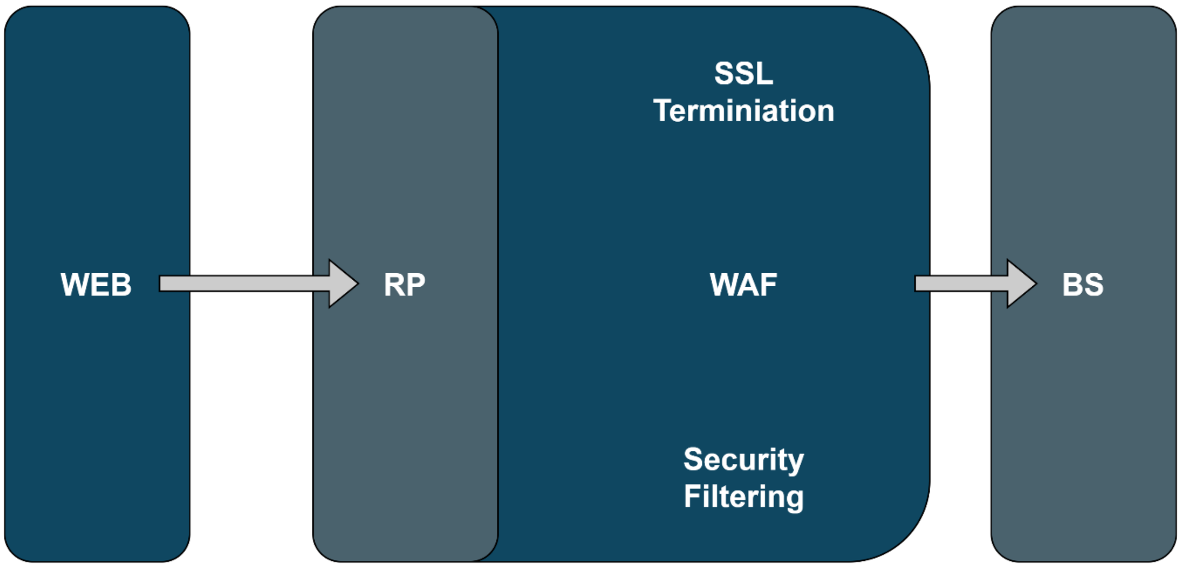
The core principles of cybersecurity, also often referred to as the CIA triad, which stand for:

1. **Confidentiality:** This principle ensures that information is only accessible to those who are intended to use/view it. It involves protecting sensitive data from unauthorized access, disclosure, or alteration.
2. **Integrity:** Integrity focuses on maintaining the accuracy and reliability of data and systems. It ensures that information is not tampered with or altered by unauthorized individuals or malicious software.
3. **Availability:** Availability ensures that information and resources are accessible and usable when needed. It involves implementing measures to prevent disruption to services, systems or data ensuring that they remain available to authorized users.

These three principles form the foundation of cybersecurity strategies and guide the implementation of security measures to protect against a wide range of threats and vulnerabilities.

Security – Network

To fortify the security of my network, I employ a reverse proxy. This strategic measure not only shields the network but also ensures uninterrupted service provision. By positioning the reverse proxy as a buffer between external users accessing the service via the internet and our internal resources, by filtering and inspecting incoming traffic before it reaches the internal network.



WEB: Web users who access the BS

RP: RP stands for reverse proxy.

BS: BS stands for backend services.

This not the only thing however, there are three other things you can do as well:

1. **Use Network Segmentation:** In the above shown graphic I've partially done this I've segmented the network into two distinct networks where the **RP** serves as a bridge between them.
2. **Implement Firewall rules:** Docker has a native firewall. You can configure this native firewall to restrict incoming and outgoing traffic between containers or networks. For example, you could execute the following commands:

```
# Allow traffic from container A to container B on port 8080
docker network create my-network
docker run --name containerA --network my-network -p 8080:8080 -d imageA
docker run --name containerB --network my-network -d imageB
docker network connect --alias containerB my-network containerA
```

This allows containerA and containerB to communicate over port 8080, additionally you can also configure this even if the two containers aren't in the same network. I recommend using a third-party tool for this like Calico, Cilium or K8s Network Policies (K8s Network Policies only if using docker with K8s)

3. **Enabling TLS:** Enabling TLS (Transport Layer Security) allows the network to encrypt traffic, granting additional security between endpoint to endpoint. Do this by utilizing Certificates and enabling TLS within a service if the service has the capability to do so.

Security – Code

As mentioned earlier in this document, my initial intention was to enhance security within the codebase by implementing a configuration file. However, I ultimately decided against this approach due to concerns about further abstracting the code. Nevertheless, it's crucial to acknowledge that security, particularly confidentiality, is paramount. Allow me to demonstrate the alternative method:

For this you want to use the `os` library this library allows you to find files on your operating system by using the file name and `os.walk` and `os.getcwd` and assign it the value of `starting directory`.

After that you also have to assign a value to the variable `filename` this way you can use the value of the variable in a for loop this will look something like this

```python
import os

# Specify the filename you're looking for
filename = "config.ini"

# Get the current working directory
starting_directory = os.getcwd()

# Iterate over all files and directories in the directory tree
for dirpath, dirnames, filenames in os.walk(starting_directory):
    # Check if the filename exists in the list of filenames
    if filename in filenames:
        # If found, print the full path of the file and stop searching
        print(f"The file '{filename}' was found at: {os.path.join(dirpath, filename)}")
        break
else:
    # If the file is not found, print a message saying so
    print(f"The file '{filename}' was not found.")

```

This code snippet only searches for the specified file name. Now what exactly is in this file. These are the contents of the file:

```
[database]
host = host.docker.internal
port = 3306
username = m169databaseadmin
password = 61626364313273387 26b6473
```

And with the following script you can decrypt the base16 encrypted password, parse it form the config.ini file and execute actions within the database via the script:

```python
import os
import configparser
import mysql.connector
import base64

# Function to find the configuration file
def find_config_file(starting_dir, filename):
    for dirpath, _, filenames in os.walk(starting_dir):
        if filename in filenames:
            return os.path.join(dirpath, filename)
    return None

# Find the config file
config_file_name = "config.ini"
config_file_path = find_config_file(os.getcwd(), config_file_name)

# Check if config file is found
if config_file_path:
    # Parse the config file
    config = configparser.ConfigParser()
    config.read(config_file_path)

    # Retrieve database credentials
    db_credentials = {
        "host": config.get("database", "host"),
        "port": config.getint("database", "port"),
        "username": config.get("database", "username"),
        "password_encrypted": config.get("database", "password")
    }

    # Decrypt the password
    decrypted_password = base64.b16decode(db_credentials["password_encrypted"]).decode()

    try:
        # Connect to the database
        conn = mysql.connector.connect(
            host=db_credentials["host"],
            port=db_credentials["port"],
            user=db_credentials["username"],
            password=decrypted_password
        )

        # Connection successful
        print("Connected to the database!")

        # Close the connection
        conn.close()
    except mysql.connector.Error as e:
        # Connection failed
        print("Error connecting to the database:", e)
else:
    print(f"Config file '{config_file_name}' not found.")

```

However, like I said in the beginning I didn’t implement this because I want other people to know how the script establishes a connection and I didn’t want to abstract it to much for newer users.

Security – Database

Database security encompasses the implementation of measures aimed at safeguarding databases from unauthorized access, misuse, or data breaches. Within the CIA triad, these measures primarily focus on ensuring confidentiality and integrity.

There are several effective safeguards for databases, despite their limited number. Here, I'll outline two that I've implemented and one additional:

1. **User Management:** Effective user management, such as role-based access control (RBAC), helps ensure that only authorized individuals can access the database. This minimizes the risk of unauthorized access and potential misuse of sensitive data. Also make sure to use the least-privilege principle this only give the user the privileges (authority) they really need. For example, only giving **`SELECT, CREATE`** privileges to the user used within the script.
2. **Encryption:** Encrypt sensitive data stored in the database to protect it from unauthorized access, for example hashing passwords and other sensitive information. This also involves encrypting data and rest (stored form) and data in transit (data being transmitted between systems) for example using TLS Certificates for the transit and hashing for passwords.
3. **Regular Auditing and Monitoring (additional):** Set up robust auditing and monitor systems to track database activity and detect any suspicious behaviour or unauthorized access attempts. Regularly review audit logs and monitor database activity in real-time.

The following video shows the least privilege principle quite well:

IaC

IaC – Introduction

Deploying each container individually is quite a time-intensive task. This process entails consistently referring to the documentation of the Docker image you wish to pull from the container registry. Sometimes, these images lack documentation altogether. To save time and avoid headaches, one can utilize docker-compose.yaml files. These files contain sets of instructions that the Docker engine can use to deploy a fully-fledged Docker environment.

These docker-compose.yaml files can be found in abundance if one searches docker-compose [service-name] under GitHub. Meaning that the docker-compose file does not need to be written from the ground up, but only modified to fit into the docker environment one wants to deploy.

IaC – Intermediary – Example

```yaml
version: '3.8'

services:
  web:
    image: python:3.9-slim
    container_name: flask_web
    command: python app.py
    volumes:
      - ./app:/app
    ports:
      - "5000:5000"
    depends_on:
      - db
    networks:
      - app_net

  db:
    image: mysql:5.7
    container_name: mysql_db
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: example_password
      MYSQL_DATABASE: example_db
      MYSQL_USER: example_user
      MYSQL_PASSWORD: example_password
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql
    networks:
      - app_net

networks:
  app_net:

volumes:
  mysql_data:

```

In conclusion, this Docker compose file will create an environment consisting of two interconnected services:

1. **Flask Web Application (web service):**
   - It runs on a Python 3.9 environment (python:3.9-slim image).
   - The Flask application is expected to be defined in a file named app.py within a directory named app.
   - Port 5000 of the host is mapped to port 5000 of the container, allowing access to the Flask application.
   - It depends on the db service, meaning it will wait for the database service to be ready before starting.
   - The container is connected to the app_net network.
2. **MySQL Database (db service):**
   - It runs on a MySQL 5.7 environment (mysql:5.7 image).
   - Port 3306 of the host is mapped to port 3306 of the container, allowing external tools to connect to the MySQL database.
   - It sets up the MySQL root password, database name, user, and password via environment variables.
   - A volume is mounted to persistently store MySQL data.
   - The container is connected to the app_net network.

These services are additionally connected to a custom docker network called **app_net** allowing them to communicate with each other via service name, as well as the volumes that give the MySQL database persistent storage.

IaC – Own docker compose

Version control

Version control: Definition

Version control is a system that records changes to files over time, allowing users to track modifications, revert to previous versions, and collaborate with others on the same files. It enables developers to manage code efficiently, maintain a history of changes, and coordinate teamwork seamlessly by providing a structured way to manage and merge code changes. Popular version control systems include Git, Subversion (SVN), and Mercurial, each offering features tailored to different needs and workflows.

Version control: How to setup

To setup version control is relatively simple, for this you'll need to do the following steps:

Step	Description
1	**Download and Install GitHub Desktop** • Go to the GitHub Desktop website and download the installer for your operating system. • Once the download is complete, run the installer and follow the on-screen instructions
2	**Sign in to Your GitHub Account:** • After installation, launch GitHub Desktop. • If you already have a GitHub Account, use the sign in option. • If you don't have a GitHub Account, sign up for free.
3	**Configure Git** • GitHub Desktop comes bundled with Git • Authenticate GitHub Desktop with your GitHub account and repositories, by making a Key • If you have big files use LFS enable this over the terminal
4	**Clone a repository** • To work on a project, you'll need to clone a repository from GitHub to your local machine. • In GitHub Desktop, click on the "Current Repository" dropdown menu and select "Clone a Repository..." • Choose the repository you want to clone from the list or paste its URL. • Select the local path where you want to clone the repository. Click on the "Clone" button to download the repository to your computer.
5	**Start Collaborating** • Once you've cloned a repository, you can start making changes to the files locally. • GitHub Desktop provides an easy-to-use interface for viewing changes, committing them, and pushing them to the remote repository on GitHub. • Make your changes, stage them, write a commit message, and then click "Commit" to save your changes locally. • To push your changes to the remote repository on GitHub, click on the "Push origin" button.

Version control: Use case

I diligently employed version control systems to manage both my codebase and Dockerfiles. This strategic approach not only ensures systematic tracking of changes to my files but also facilitates seamless collaboration within the development team. By leveraging version control, I maintain an organized and transparent workflow, enabling swift identification and resolution of issues. How ever I must say that I’ve only made a version of my code after every bigger iteration of my application.

The rollback function of VCS (Version Control Systems) has saved me multiple times from having to rewrite and debug my code for hours upon hours on end. However I not only used VSC (Version Control System) for my code but also mainly my documentation of theory and knowledge inside my GitHub Wiki.

All my code has been stored in the GitHub repository:

GitHub Wiki: Campus Castolo | m169

GitHub – Camus-Castolo/m169: Docker Services
Services mit Containern bereitstellen. Contribute to campus-Castolo/m169: Services mit Containern bereistellen by creating a pull request

Github.com

Interconnectedness

Interconnectedness: necessities

Ensuring the persistence and security of data connections between Docker containers is critical for maintaining a robust and secure system architecture. Here are some additional considerations to enhance these aspects:

1. **Continuous Monitoring:** Implementing continuous monitoring mechanisms can help detect and respond to any potential security threats or anomalies in real-time. By employing tools like intrusion detection systems (IDS) or security information and event management (SIEM) solutions, you can proactively identify and mitigate security risks within your containerized environment.
2. **Access Control Policies:** Implementing granular access control policies within Docker containers can help restrict unauthorized access to sensitive data and resources. By defining and enforcing access controls based on the principle of least privilege, you can minimize the attack surface and prevent malicious actors from compromising your containerized applications.
3. **Regular Audits and Compliance Checks:** Conducting regular security audits and compliance checks ensures that your Docker containers adhere to industry best practices and regulatory requirements. By performing vulnerability assessments and penetration testing, you can identify and remediate any security weaknesses or non-compliance issues before they are exploited by attackers.
4. **Container Image Security:** Paying attention to the security of container images is crucial for maintaining the integrity of your Docker environment. Implementing secure image build practices, such as using official base images from trusted sources and regularly updating dependencies, helps mitigate the risk of deploying vulnerable containers into production.
5. **Network Segmentation:** Segmenting your Docker network into distinct zones based on trust levels can help isolate sensitive workloads and prevent lateral movement in the event of a security breach. By implementing network segmentation controls, such as firewalls and virtual private networks (VPNs), you can limit the impact of potential security incidents and protect critical assets within your containerized infrastructure.

By incorporating these additional measures into your Docker container environment, you can further strengthen the persistence and security of data connections, reducing the risk of unauthorized access, data breaches, and other security incidents.

Interconnectedness: Persistency

In the realm of Docker, interconnectedness between containers and the aspect of persistency are pivotal for maintaining the integrity and functionality of services, particularly when it comes to databases. Persistency in Docker refers to the ability to retain data across container instances, ensuring that data remains intact even when containers are stopped, restarted, or scaled.

To achieve persistency in Docker containers, one common approach involves leveraging Docker volumes. Docker volumes are specially designated directories within the Docker environment that exist outside of individual containers but can be mounted into them, allowing data to persist beyond the lifespan of any single container.

Firstly, we need to create a Docker volume to store the database data persistently. We can accomplish this using the following Docker command:

```
docker volume create my_pgdata_volume
```

This command creates a volume named `my_pgdata_volume,` which will serve as the persistent storage for our PostgreSQL data.

Next, when we run our PostgreSQL container, we need to mount this volume into the container at the appropriate location where the database stores its data. We can do this by specifying the -v or --volume flag along with the volume name and the container directory where the data should be mounted. Here's an example:

```
docker run -d --name my_postgres_container -v my_pgdata_volume:/var/lib/postgresql/data
postgres:latest
```

In this command:

`-d` flag detaches the container and runs it in the background.

`--name my_postgres_container` assigns a custom name to the container for easy reference.

`-v my_pgdata_volume:/var/lib/postgresql/data` mounts the `my_pgdata_volume` volume into the `/var/lib/postgresql/data` directory within the container.

`postgres:latest` specifies the PostgreSQL image and its version.

With this configuration, the PostgreSQL database within the container will utilize the `my_pgdata_volume` volume for persistently storing its data. Even if the container is stopped, removed, or replaced, the data will remain intact within the volume, ensuring persistency across container instances.

Interconnectedness: Persistency – MariaDB

To make a MariaDB container persistent using Docker volumes, you'll need to mount two volumes: one for the database files and another for configuration files. Here's how you can achieve that:

1. **Database Data Volume:** This volume will store the database files to ensure data persistence even if the container is stopped or removed.

```
docker volume create mariadb_data_volume
```

2. **Configuration Volume:** This volume will hold the MariaDB configuration files to maintain custom configurations across container restarts.

```
docker volume create mariadb_data_config
```

When running the MariaDB container, you'll need to specify these volumes to mount them appropriately. Here's an example Docker command:

```
docker run -d \
  --name m169.project.mariadb \
  -v mariadb_data_volume:/var/lib/mysql \
  -v mariadb_config_volume:/etc/mysql \
  -e MYSQL_ROOT_PASSWORD=my-secret-pw \
  -e MYSQL_DATABASE=m169-project-inventory \
  -n m169-project-internal \
  -p 3306:3306 \
  mariadb:latest
```

In this command:

- `-v mariadb_data_volume:/var/lib/mysql` mounts the `mariadb_data_volume` to the MariaDB data directory, ensuring persistent storage of database files.
- `-v mariadb_data_config:/etc/mysql` mounts the `mariadb_data_config` to the MariaDB configuration directory, allowing customization of configurations.
- `-e MYSQL_ROOT_PASSWORD=my-secret-pw` sets the root password for MariaDB. Replace `my-secret-pw` with your desired password.
- -e MYSQL_DATABASE=m169-project-inventory creates a database named m169-project-inventory.
- -n m169-project-internal makes the docker container join the network m169-project-internal
- `-p 3306:3306` exposes the MariaDB port.

With these volumes mounted, your MariaDB data and configurations will persist across container restarts and even if the container is removed.

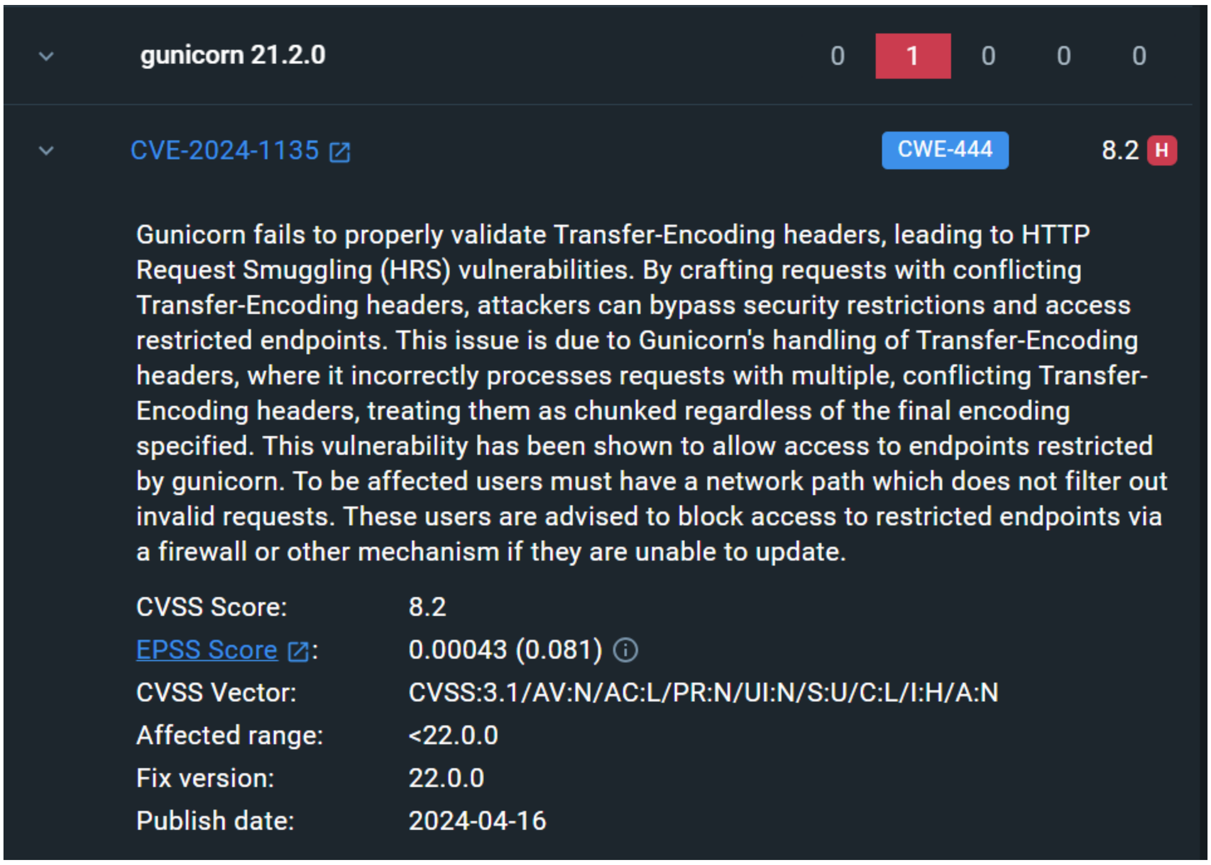
Interconnectedness: Security of container images

Here's more you can add about Docker Scout within Docker Desktop:

Unveiling Security Vulnerabilities:

1. Docker Scout acts as a security watchdog for your container images. It meticulously scans layers within the image, uncovering hidden vulnerabilities in software packages. These vulnerabilities can be entry points for malicious actors, so early detection is crucial.

For example, here is a screenshot of a vulnerability with Gunicorn within my self made docker image:



gunicorn 21.2.0 0 1 0 0 0

CVE-2024-1135 CWE-444 8.2 H

Gunicorn fails to properly validate Transfer-Encoding headers, leading to HTTP Request Smuggling (HRS) vulnerabilities. By crafting requests with conflicting Transfer-Encoding headers, attackers can bypass security restrictions and access restricted endpoints. This issue is due to Gunicorn's handling of Transfer-Encoding headers, where it incorrectly processes requests with multiple, conflicting Transfer-Encoding headers, treating them as chunked regardless of the final encoding specified. This vulnerability has been shown to allow access to endpoints restricted by gunicorn. To be affected users must have a network path which does not filter out invalid requests. These users are advised to block access to restricted endpoints via a firewall or other mechanism if they are unable to update.

CVSS Score:	8.2
EPSS Score:	0.00043 (0.081)
CVSS Vector:	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:H/A:N
Affected range:	<22.0.0
Fix version:	22.0.0
Publish date:	2024-04-16

Streamlined Workflows:

2. By integrating with your existing Docker workflow, Scout saves you valuable time. It seamlessly analyzes images during the build process, providing instant feedback on potential security issues. This allows you to fix problems early on, preventing delays and ensuring your containers are built on a secure foundation.

Actionable Recommendations:

3. Scout doesn't just identify vulnerabilities; it empowers you to address them. It provides clear recommendations for remediation, suggesting specific patches or updates to mitigate the risks. This guidance helps developers prioritize security fixes and make informed decisions.

Beyond the Basics:

4. Docker Scout's capabilities extend beyond vulnerability scanning. It offers valuable insights into the overall structure of your container images, helping you identify inefficiencies and optimize image size. This can lead to faster deployments and improved container performance.

Together, Docker Desktop and Docker Scout form a powerful combination, enabling developers to build secure and efficient containerized applications with greater confidence.

Image management

How to publish your own Image to a container repository

To publish your own Docker images to a container registry such as Docker Hub, the first step is to create an account on the registry. Here's how to do it:

1. **Create an Account:** Navigate to the website of the container registry you want to use, such as Docker Hub. Look for the option to sign up or create an account. Provide the required information, such as your email address, username, and password. Follow the prompts to complete the registration process.

navigate to the directory containing it in your terminal or command prompt and use the docker build command to build your image. For example:

```
docker build -t yourusername/imagename:tag .
```

Replace yourusername with your Docker Hub username, imagename with the name you want to give to your image, and tag with a version or tag for your image.

2. **Login to Docker Hub**: Use the docker login command to log in to Docker Hub from your terminal or command prompt. You will be prompted to enter your Docker Hub username and password.

```
docker login
```

3. **Tag Your Image:** After logging in, tag your local image with the repository name on Docker Hub.

```
docker tag yourusername/imagename:tag yourusername/imagename:tag
```

Replace yourusername, imagename, and tag with the same values you used when building the image.

4. **Push Your Image:** Finally, use the docker push command to push your image to Docker Hub.

```
docker push yourusername/imagename:tag
```

This command will upload your image to Docker Hub, making it available for others to use and download.

See docker hub container registry for my image → m169-application-image

How to reuse a container image

Now that you know how to push your image on to a docker repository

1. **Pull the Image:** Use the docker pull command to download the desired container image from a container registry such as Docker Hub. For example:

```
docker pull image_name:tag
```

Replace image_name with the name of the container image you want to reuse and tag with the specific version or tag you wish to use.

2. **Create a Container:** Once the image is downloaded, you can create a new container instance based on that image using the docker run command. For example:

```
docker run --name my_container -d image_name:tag
```

Replace my_container with the desired name for your container instance. This command will create a new container based on the specified image.

Also don’t forget to adhere to the provided manual/documentation of the image, if available. There are also multiple options you can use for example **-e** for environment this sets a environment variable that will configure the container beforehand often used for credentials.

3. **Start the Container:** If the container is not running after creation, you can start it using the docker start command:

```
docker start my_container
```

Replace my_container with the name of your container instance.

4. **Interact with the Container:** Once the container is running, you can interact with it as needed. This might involve running commands within the container or accessing services provided by the container.
5. **Stop and Remove the Container (Optional):** When you're done using the container, you can stop and remove it using the docker stop and docker rm commands, respectively:

```
docker stop my_container
docker rm my_container
```

This will stop and remove the container instance from your system.

Monitoring and Logging

To monitor the individual containers its recommended to use a third party application, even though docker itself has monitoring capabilities its best and also more convenient to use a third party monitoring tool, like Prometheus or Grafana.

Monitoring specific values

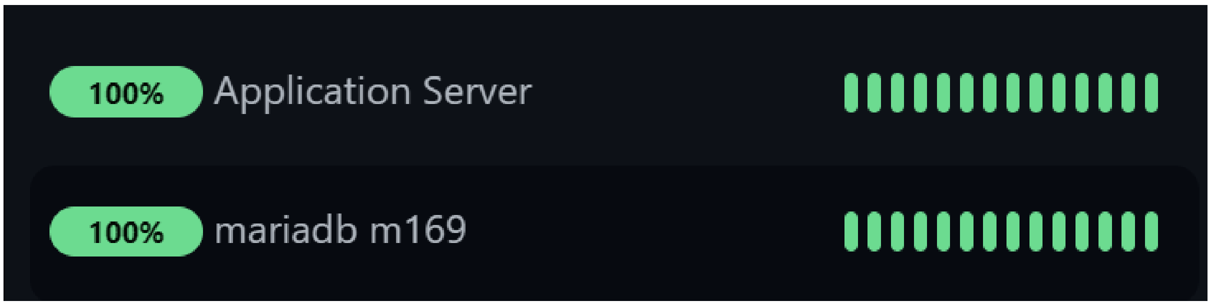
If you want to monitor a object in general its best to have values like, RAM usage, CPU usage or traffic load, however since we don't have a lot of traffic I will focus on the standard values CPU and RAM usage.

Defining which monitoring tool I want to use

Specifying which monitoring tool one wants to use is a crucial step in setting up the monitoring tool. I have asked around and a schoolmate of mine told me to use Uptime Kuma, because it is easy to use and very easy to install.

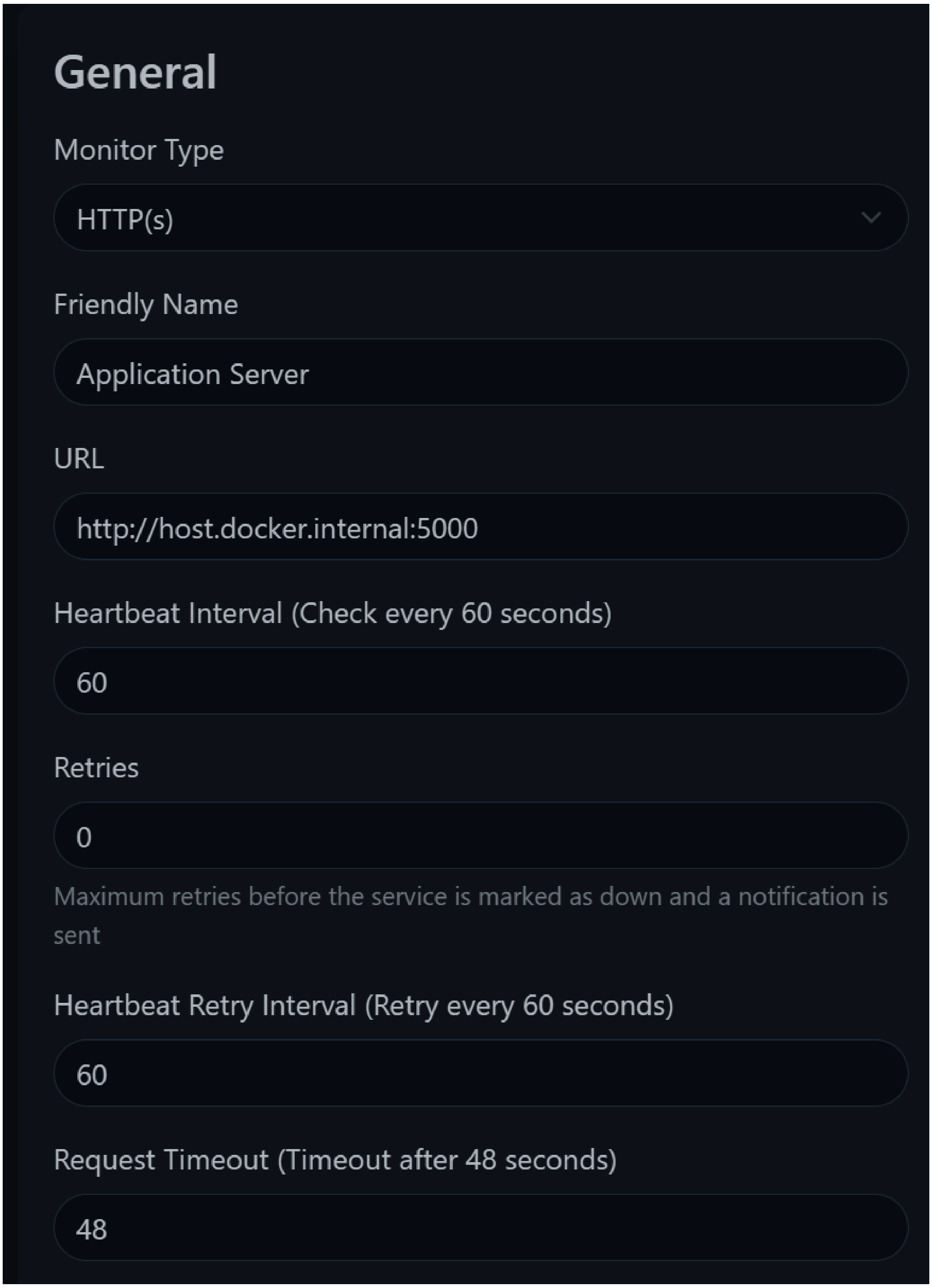
It also spots impressive monitoring capabilities and has a clean UI which I value.

Create the monitoring



I added two monitors to uptime kuma, one should monitor my application server by checking if the web interface is up and the other one checks if the database is intact by making continuous Selects statements of the database.

Monitor: Application server



General

Monitor Type

HTTP(s)

Friendly Name

Application Server

URL

http://host.docker.internal:5000

Heartbeat Interval (Check every 60 seconds)

60

Retries

0

Maximum retries before the service is marked as down and a notification is sent

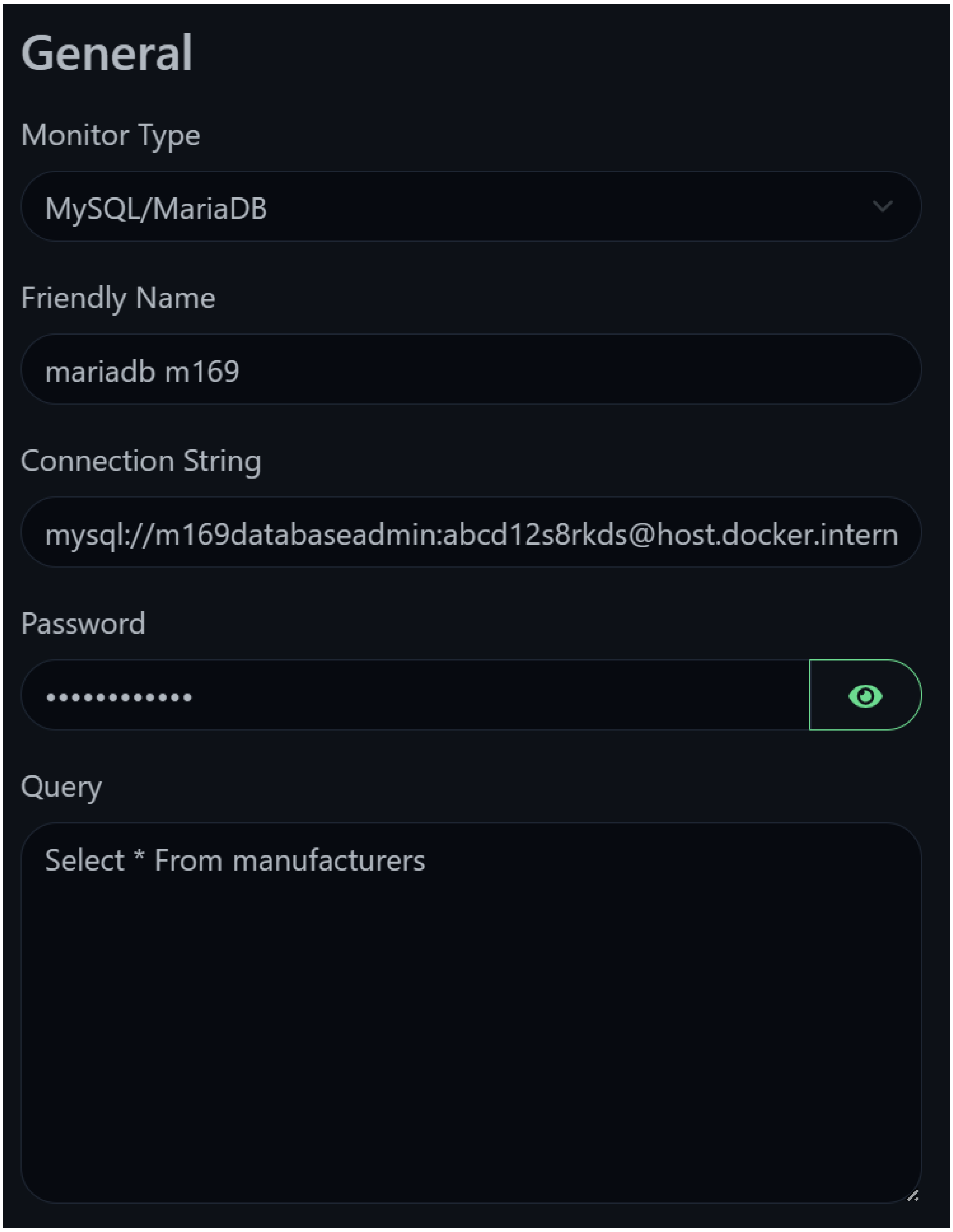
Heartbeat Retry Interval (Retry every 60 seconds)

60

Request Timeout (Timeout after 48 seconds)

48

Monitor: mariadb



General

Monitor Type

MySQL/MariaDB

Friendly Name

mariadb m169

Connection String

mysql://m169databaseadmin:abcd12s8rkds@host.docker.intern

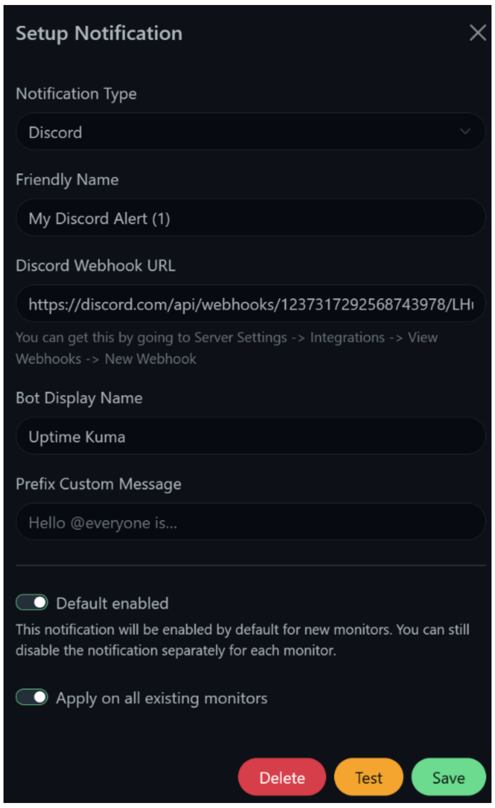
Password

••••••••••••

Query

```
Select * From manufacturers
```

Notification if something went wrong


Setup Notification

Notification Type: Discord

Friendly Name: My Discord Alert (1)

Discord Webhook URL: https://discord.com/api/webhooks/1237317292568743978/LH

You can get this by going to Server Settings -> Integrations -> View Webhooks -> New Webhook

Bot Display Name: Uptime Kuma

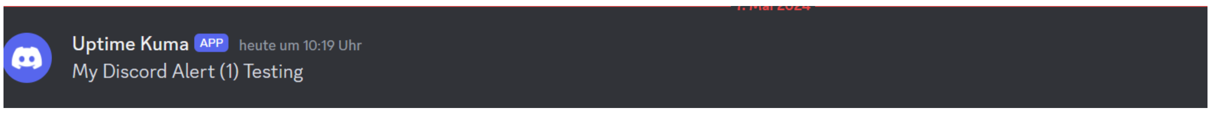
Prefix Custom Message: Hello @everyone is...

Default enabled

This notification will be enabled by default for new monitors. You can still disable the notification separately for each monitor.

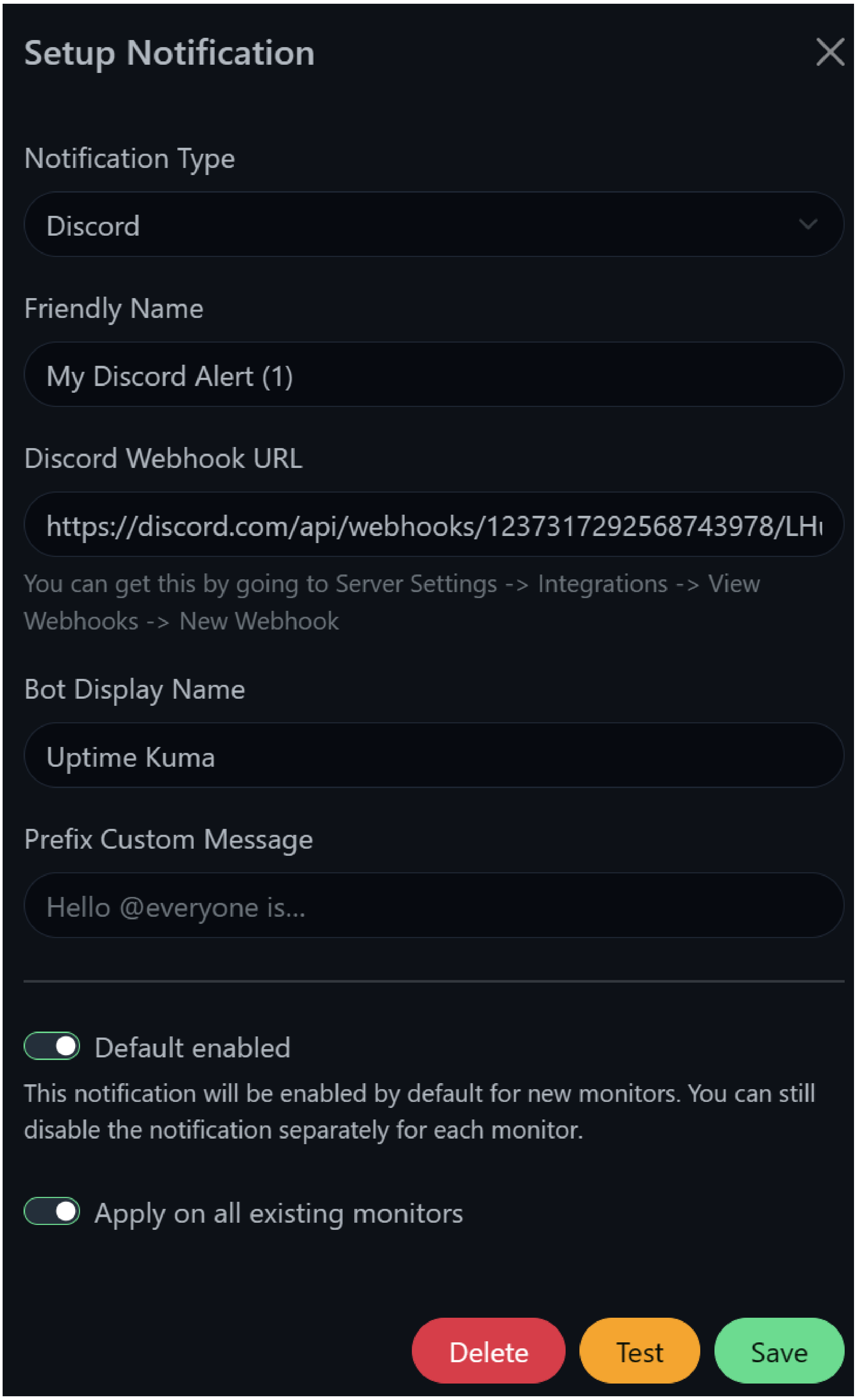
Apply on all existing monitors

I created an notification that if something went wrong or is going wrong it would notify me on my discord server.


Uptime Kuma APP heute um 10:19 Uhr

My Discord Alert (1) Testing

A automated test text message.


Setup Notification

Notification Type

Discord

Friendly Name

My Discord Alert (1)

Discord Webhook URL

https://discord.com/api/webhooks/1237317292568743978/LH

You can get this by going to Server Settings -> Integrations -> View Webhooks -> New Webhook

Bot Display Name

Uptime Kuma

Prefix Custom Message

Hello @everyone is...

Default enabled

This notification will be enabled by default for new monitors. You can still disable the notification separately for each monitor.

Apply on all existing monitors