

★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



# FastAPI, Prometheus, Grafana, Graylog in docker-compose example



Denis Gr

Follow

13 min read · Nov 14, 2023



95



graylog



Hello, everyone. I would like to explore an example of a simple application using FastAPI, React, Prometheus for collecting basic application metrics, and Graylog for log aggregation. Let's take a closer look at our application.

In the `src` folder, you'll find the backend of our application, which includes CRUD operations for creating, reading, updating, and deleting some items.

The Dockerfile is based on the Alpine image for Python 3.9.1:

```
# Use the official Python 3.9.1 Alpine image as the base image
FROM python:3.9.1-alpine

# Set the working directory inside the container
WORKDIR /usr/src/app

# Prevent Python from writing bytecode files and force unbuffered mode for better
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

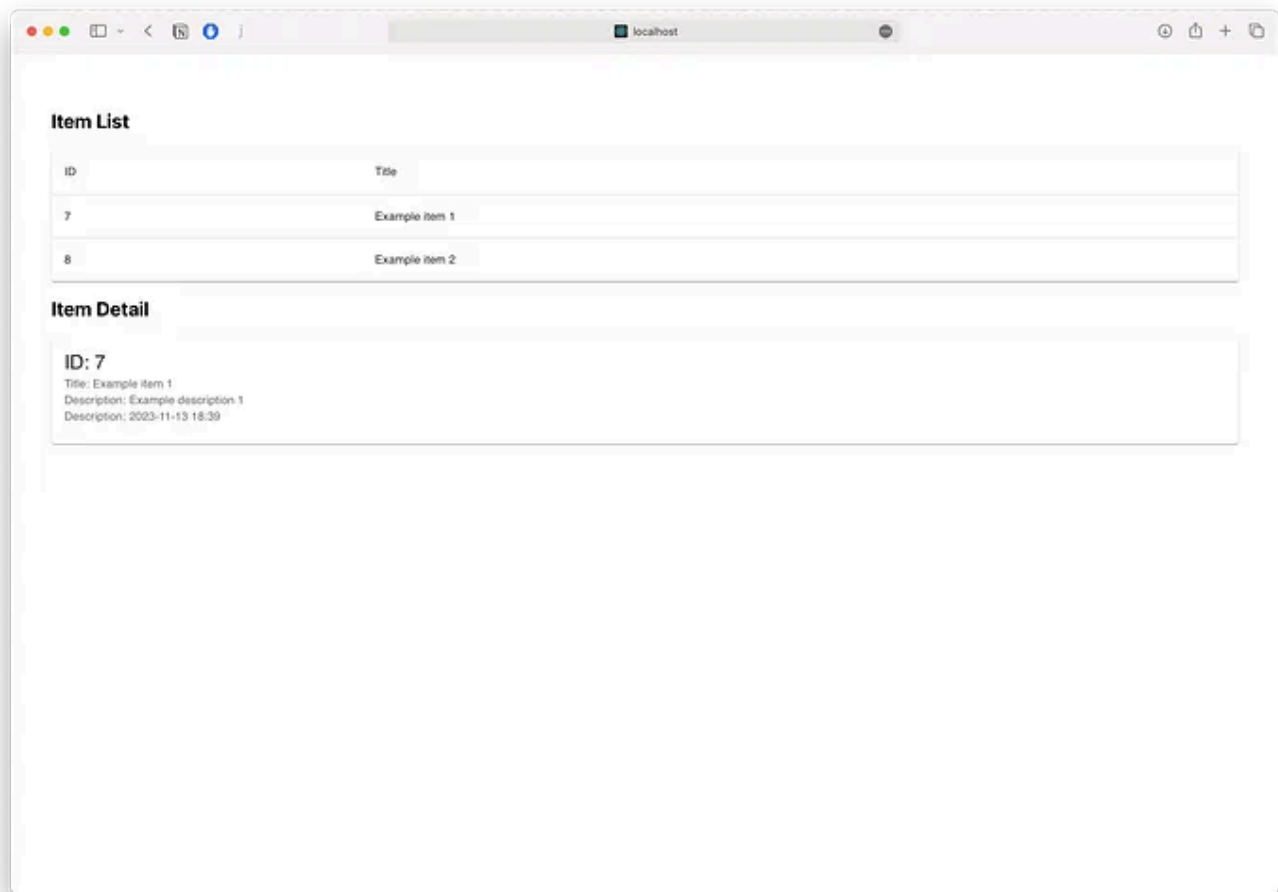
# Copy the requirements.txt file to the working directory
COPY ./requirements.txt /usr/src/app/requirements.txt

# Install build dependencies, including libraries and development tools
RUN set -eux \
    && apk add --no-cache --virtual .build-deps build-base \
    libressl-dev libffi-dev gcc musl-dev python3-dev \
    postgresql-dev \
    && pip install --upgrade pip setuptools wheel \
    && pip install -r /usr/src/app/requirements.txt \
    && rm -rf /root/.cache/pip

# Copy the rest of the application code to the working directory
COPY . /usr/src/app/
```

In the `frontend` folder, there's a React frontend application displaying our created items with the ability for detailed viewing. It will be available at

<http://localhost:8081/> after the compose cluster is up.



<http://localhost:8081>

The multi-stage Dockerfile for the React application:

```
# Development stage: Use Node.js 18 Alpine as the base image, set the working directory
FROM node:18-alpine AS develop-stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

# Build stage: Use the development stage as a base, run npm build to generate production-ready code
FROM develop-stage AS build-stage
RUN npm run build
```

```
# Production stage: Use Nginx 1.15.7 Alpine as the base image, copy the built as
FROM nginx:1.15.7-alpine AS production-stage
COPY --from=build-stage /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

At the root of our application, you can find the Docker Compose file:

```
# Docker Compose version
version: "3.3"

# Define services for the application

# Service for the web application
services:
  web:
    # Build the web service from the source directory
    build: ./src
    # Command to run the web service with Uvicorn
    command: uvicorn app.main:app --reload --workers 1 --host 0.0.0.0 --port ${APP_PORT}
    # Mount the local source directory into the container
    volumes:
      - ./src:/usr/src/app/
    # Expose ports for the web service
    ports:
      - "8000:${APP_PORT}"
    # Environment variables for the web service
    environment:
      - DATABASE_USER=${DATABASE_USER}
      - DATABASE_PASSWORD=${DATABASE_PASSWORD}
      - DATABASE_NAME=${DATABASE_NAME}
      - DATABASE_HOST=${DATABASE_HOST}
      - DATABASE_PORT=${DATABASE_PORT}
      - APP_PORT=${APP_PORT}
      - GRAYLOG_HOST=${GRAYLOG_HOST}
      - GRAYLOG_PORT_UDP=${GRAYLOG_PORT_UDP}
    # Dependencies for the web service
    depends_on:
      db:
        condition: service_started
      graylog:
        condition: service_healthy
```

```
# Service for the PostgreSQL database
db:
  # Use the official PostgreSQL 13.1 Alpine image
  image: postgres:13.1-alpine
  # Mount a volume for persistent data storage
  volumes:
    - ./postgres_data:/var/lib/postgresql/data/
  # Environment variables for the database
  environment:
    - POSTGRES_USER=${DATABASE_USER}
    - POSTGRES_PASSWORD=${DATABASE_PASSWORD}
    - POSTGRES_DB=${DATABASE_NAME}
  # Expose ports for the database
  ports:
    - "5432:${DATABASE_PORT}"

# Service for the frontend application
frontend:
  # Build the frontend service from the frontend directory
  build: ./frontend
  # Environment variables for the frontend
  environment:
    - APP_HOST=${APP_HOST}
    - APP_PORT=${APP_PORT}
  # Expose ports for the frontend
  ports:
    - '8081:80'

# Service for Prometheus
prometheus:
  # Use the official Prometheus image
  image: prom/prometheus
  # Expose ports for Prometheus
  ports:
    - "9090:9090"
  # Mount the local Prometheus configuration file
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
  # Command to start Prometheus with the specified configuration file
  command:
    - '--config.file=/etc/prometheus/prometheus.yml'

# Service for Grafana
grafana:
  # Use the official Grafana image
  image: grafana/grafana
  # Set environment variables for Grafana
  environment:
    - GF_SECURITY_ADMIN_USER=${GRAFANA_ADMIN_USER}
    - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_ADMIN_PASSWORD}
```

```
- GF_USERS_ALLOW_SIGN_UP=false
# Expose ports for Grafana
ports:
  - "3000:3000"
# Mount a volume for persistent data storage
volumes:
  - grafana_data:/var/lib/grafana

# Service for MongoDB
mongo:
  # Use the official MongoDB image
  image: mongo:3
  # Mount a volume for persistent data storage
  volumes:
    - mongodb_data:/data/db

# Service for Elasticsearch
elasticsearch:
  # Use the official Elasticsearch 6.8.10 image
  image: docker.elastic.co/elasticsearch/elasticsearch-oss:6.8.10
  # Set environment variables for Elasticsearch
  environment:
    - http.host=0.0.0.0
    - transport.host=localhost
    - network.host=0.0.0.0
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
  # Set ulimits for Elasticsearch
  ulimits:
    memlock:
      soft: -1
      hard: -1
  # Deployment configuration for Elasticsearch
  deploy:
    resources:
      limits:
        memory: 1g

# Service for Graylog
graylog:
  # Use the official Graylog 3.3 image
  image: graylog/graylog:3.3
  # Set environment variables for Graylog
  environment:
    - GRAYLOG_PASSWORD_SECRET=${GRAYLOG_PASSWORD}
    - GRAYLOG_ROOT_PASSWORD_SHA2=${GRAYLOG_PASSWORD_SHA}
    - GRAYLOG_HTTP_EXTERNAL_URI=http://127.0.0.1:9000/
  # Dependencies for Graylog
  depends_on:
    - mongo
    - elasticsearch
```

```
# Expose ports for Graylog
ports:
  # Graylog web interface and REST API
  - 9000:9000
  # Syslog TCP
  - 1514:1514
  # Syslog UDP
  - 1514:1514/udp
  # GELF TCP
  - 12201:12201
  # GELF UDP
  - 12201:12201/udp

# Define volumes for persistent data storage
volumes:
  postgres_data:
  mongodb_data:
  vue-client:
  grafana_data:

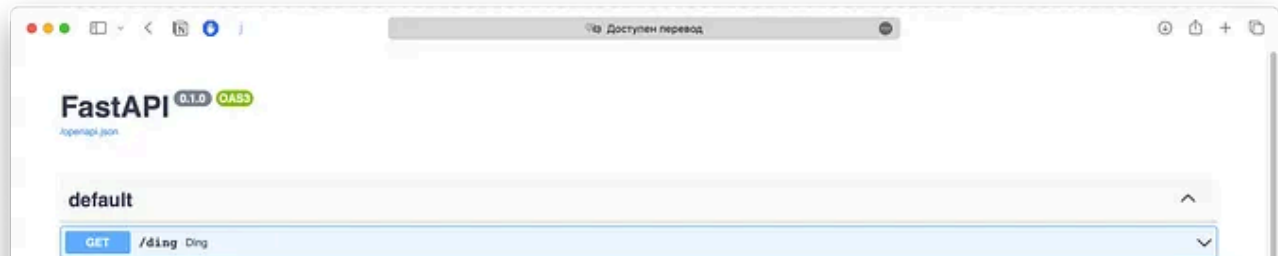
# Define default network
networks:
  default:
```

To run our application, use the command:

```
docker compose up -d
```

This will bring up our compose cluster, with the `-d` flag allowing containers to work in the background.

Our backend is deployed at <http://localhost:8000/>, and Swagger is available at <http://localhost:8000/docs>

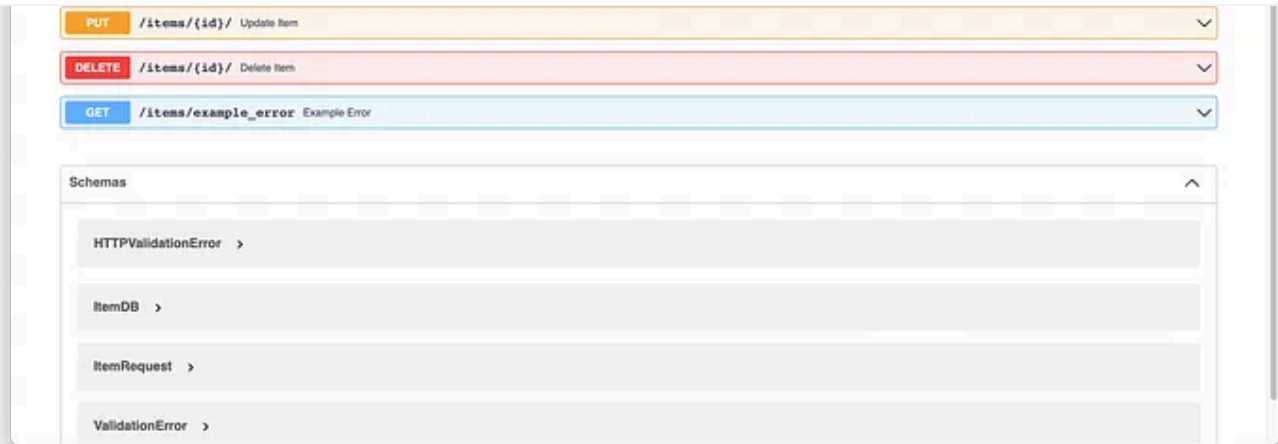


Open in app ↗

Medium

Search

Write



<http://localhost:8000/docs>

## Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit originally developed by SoundCloud. It is designed for reliability and scalability and is part of the Cloud Native Computing Foundation (CNCF).

Key features of Prometheus include:

1. Multi-dimensional Data Model: Prometheus stores time-series data, allowing measurements to be tagged by key-value pairs. This enables efficient querying and filtering.
2. PromQL: Prometheus Query Language (PromQL) is a powerful query language used for querying and processing the collected metrics. It



allows users to perform complex queries and aggregations.

3. **Pull-Based Model:** Prometheus follows a pull-based model where it scrapes metrics data from instrumented jobs at regular intervals. This approach simplifies the architecture and allows for dynamic service discovery.
4. **Service Discovery:** Prometheus supports service discovery mechanisms, enabling it to automatically discover and monitor new services as they come online in dynamic environments.
5. **Alerting:** Prometheus has a built-in alerting system that allows users to define alert rules based on the collected metrics. When conditions specified in the rules are met, alerts are triggered.
6. **Exporters:** Prometheus relies on exporters, which are small applications that expose metrics in a format that Prometheus can understand. There are various exporters available for different types of systems and applications.
7. **Integration with Grafana:** Prometheus is often used in conjunction with Grafana, a popular open-source analytics and monitoring platform. Grafana allows users to create dashboards and visualizations based on data collected by Prometheus.
8. **Community and Ecosystem:** Prometheus has a vibrant and active open-source community. It is widely used in the container orchestration space, particularly with Kubernetes, and integrates well with other cloud-native technologies.

Prometheus is a versatile tool used for monitoring the performance and health of systems, applications, and services in a distributed and dynamic environment.



In this format, Prometheus expects metrics from our application. Naturally, basic metrics have been added here. We can initialize our custom metrics separately to gather the specific information we need. An example is available on [Github](#)

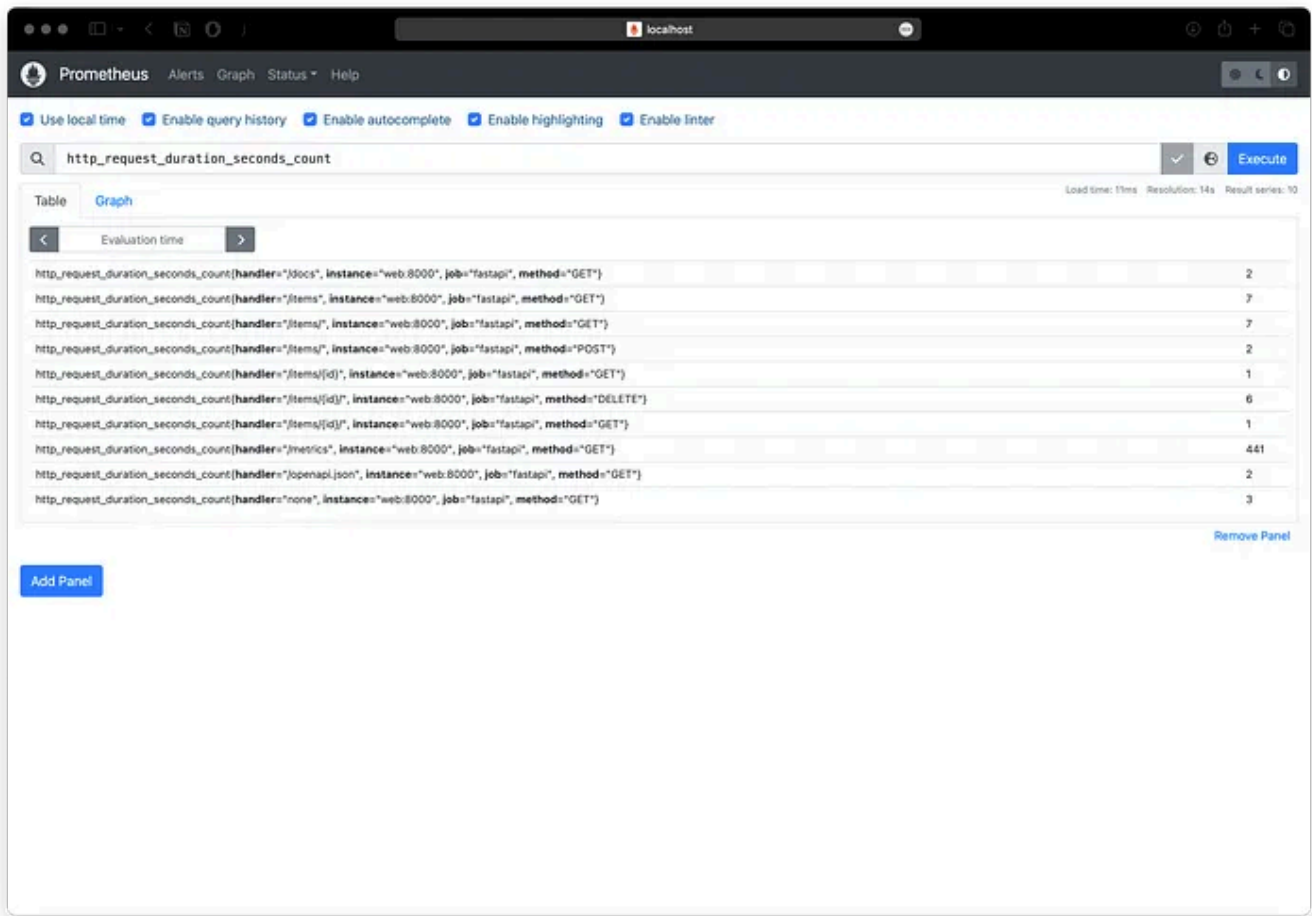
The Prometheus configuration file ( `prometheus.yml` ) in our application:

```
# Global configuration settings for Prometheus
global:
  # Set the interval at which Prometheus scrapes targets (15 seconds)
  scrape_interval: 15s
  # Set the timeout for individual scrape requests (10 seconds)
  scrape_timeout: 10s
  # Set the interval at which rules are evaluated (15 seconds)
  evaluation_interval: 15s

# Alerting configuration for Prometheus
alerting:
  alertmanagers:
    # Configure Alertmanager with HTTP settings
    - follow_redirects: true
      enable_http2: true
      scheme: http
      timeout: 10s
      api_version: v2
    # Configure static targets for Alertmanager
    static_configs:
      - targets: []

# Scrape configurations for Prometheus, defining jobs and their settings
scrape_configs:
  # Job configuration for 'fastapi'
  - job_name: 'fastapi'
    # Set the interval at which Prometheus scrapes targets for this job (10 seconds)
    scrape_interval: 10s
    # Set the metrics path for this job
    metrics_path: /metrics
    # Define static targets for this job
    static_configs:
      - targets: ['web:8000']
```

The Prometheus UI is available at <http://localhost:9090>.



Example select for `http_request_duration_seconds_count`

In this screenshot, we see data for the `http_request_duration_seconds_count` metric, but these data alone are not very intuitive. It would be great to visualize them in a more appealing way. This is where Grafana comes in handy.

## Grafana

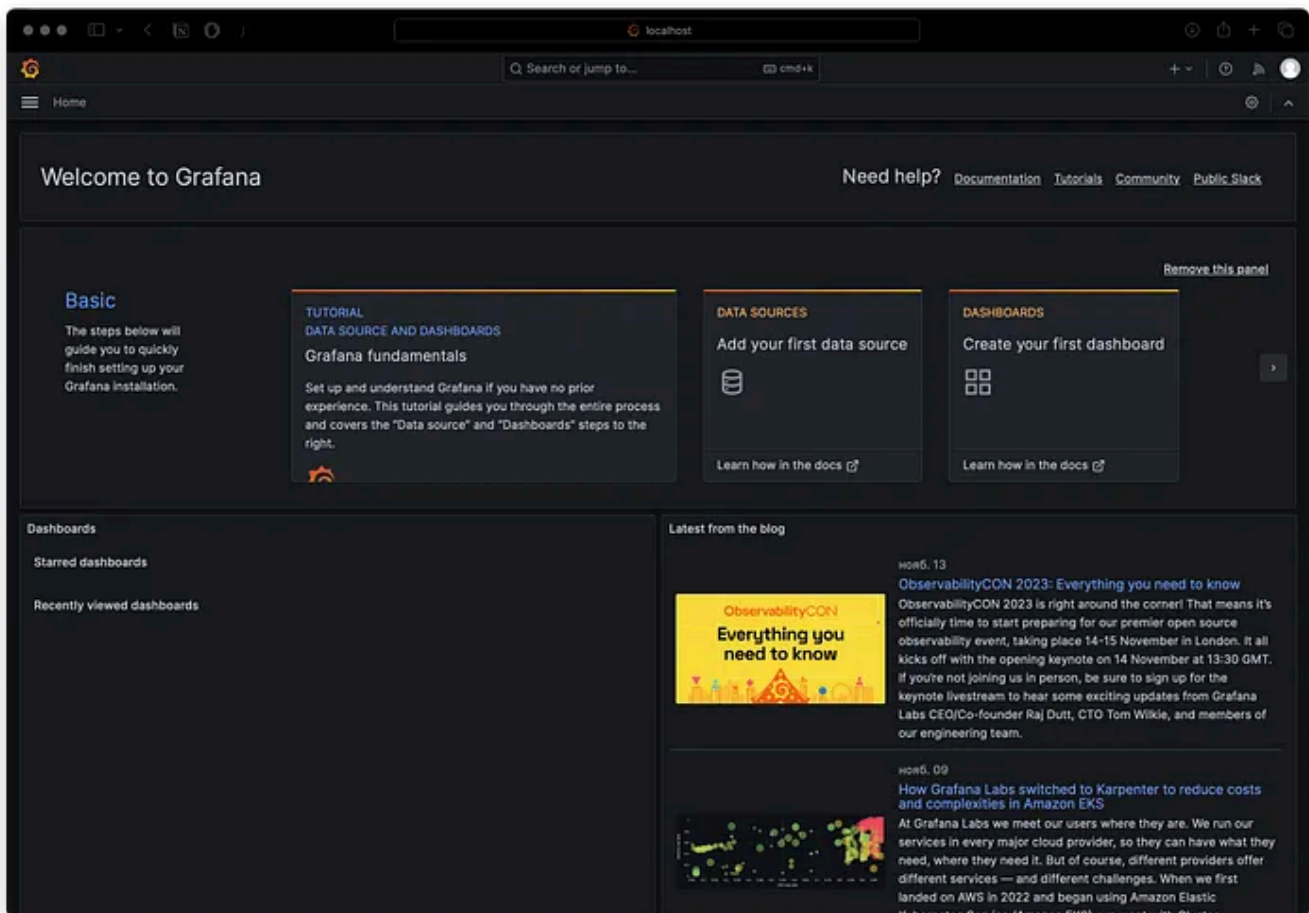
Grafana is an open-source platform for monitoring and observability. It provides a powerful and flexible framework for creating, exploring, and sharing dashboards with data from various sources, including databases, cloud services, and other monitoring tools.

## Key features of Grafana include:

1. **Data Visualization:** Grafana allows users to create interactive and customizable dashboards to visualize time-series data. It supports a wide range of data sources, including popular databases, cloud platforms, and monitoring systems.
2. **Multiple Data Sources:** Grafana can integrate with various data sources, such as Prometheus, InfluxDB, Elasticsearch, MySQL, PostgreSQL, and many others. This flexibility makes it a central hub for monitoring and analytics.
3. **Alerting:** Grafana includes an alerting system that allows users to define and receive alerts based on specified conditions. Users can set up alert rules and receive notifications through various channels, such as email, Slack, or other messaging services.
4. **Plugins:** Grafana supports a plugin architecture, enabling the community to develop and share plugins to extend functionality. This allows users to integrate Grafana with additional data sources, visualization options, and other features.
5. **User-Friendly Interface:** Grafana provides an intuitive and user-friendly interface for building dashboards. Users can customize panels, queries, and visualizations without extensive technical knowledge.
6. **Community and Ecosystem:** Grafana has a large and active community, contributing to its ecosystem with plugins, extensions, and support. The platform is widely adopted in the DevOps and monitoring communities.
7. **Cross-Platform Support:** Grafana can run on various operating systems and is often deployed in containerized environments, making it easy to integrate into different infrastructure setups.

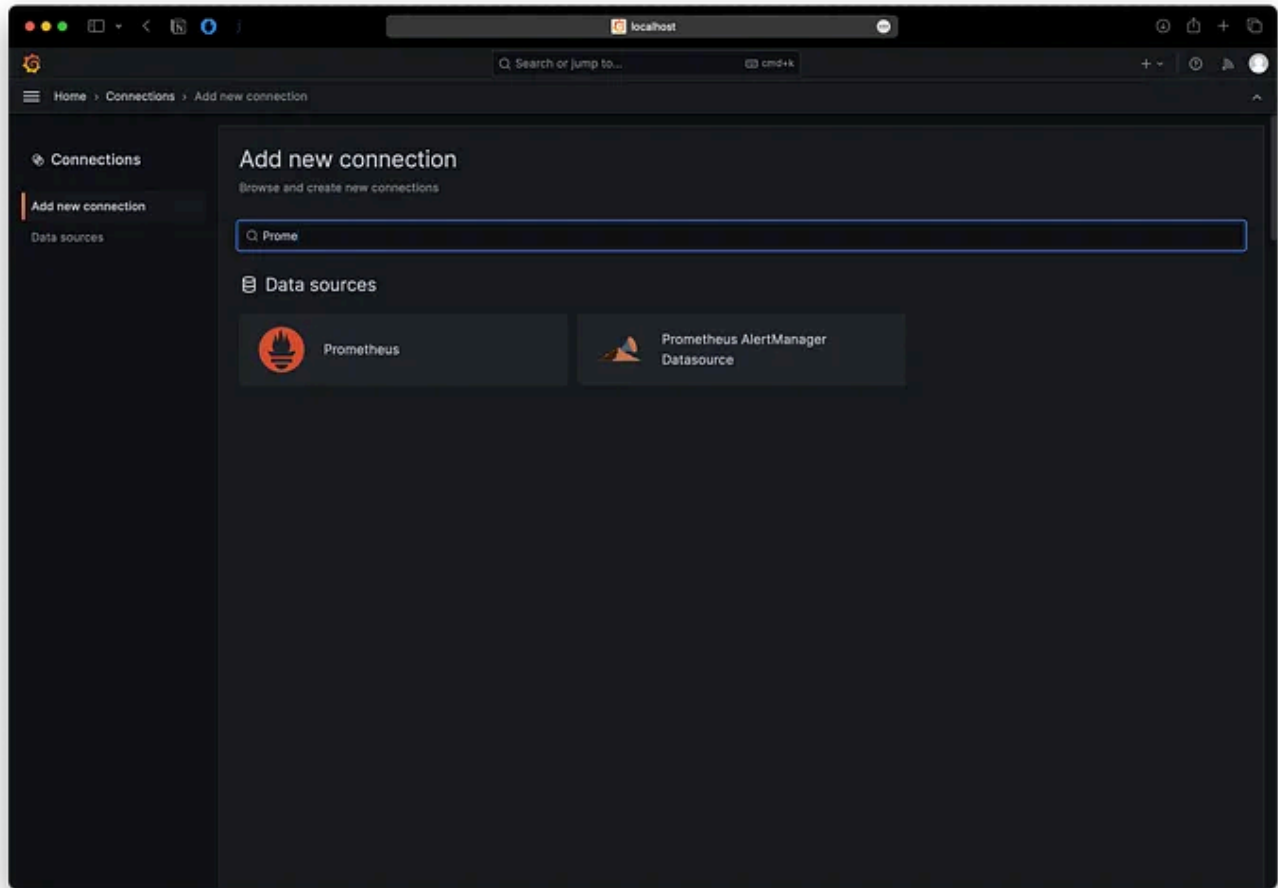
Overall, Grafana is a powerful tool for monitoring and observability, allowing users to gain insights into the performance and health of their systems through visually appealing and customizable dashboards.

Grafana is available at <http://localhost:3000>, the login and password should be configured in the `.env` file using variables `GRAFANA_ADMIN_USER` and `GRAFANA_ADMIN_PASSWORD`



To configure Grafana to collect data from Prometheus:

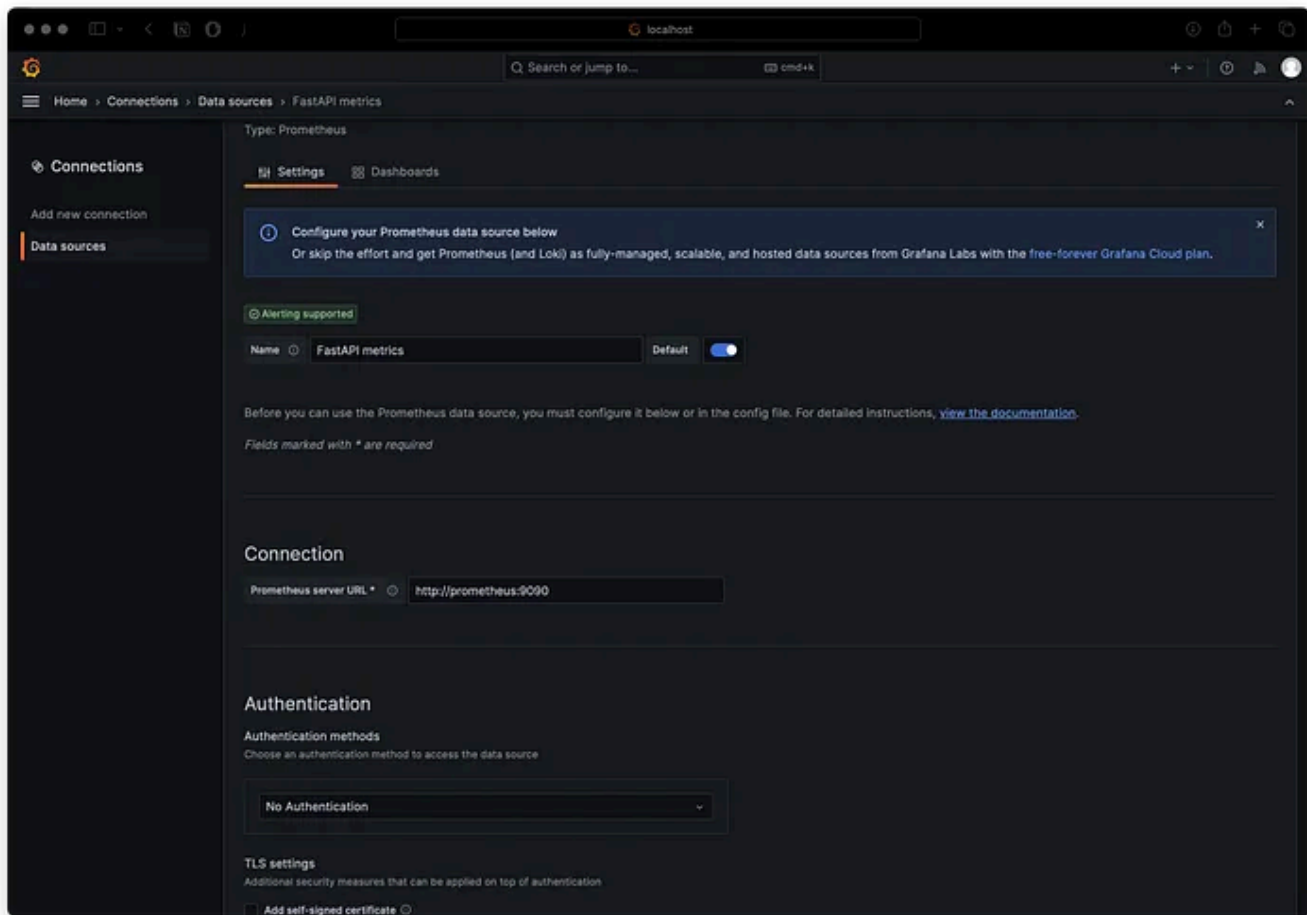
Go to Connections, search for Prometheus, and add a new datasource.



search for Prometheus

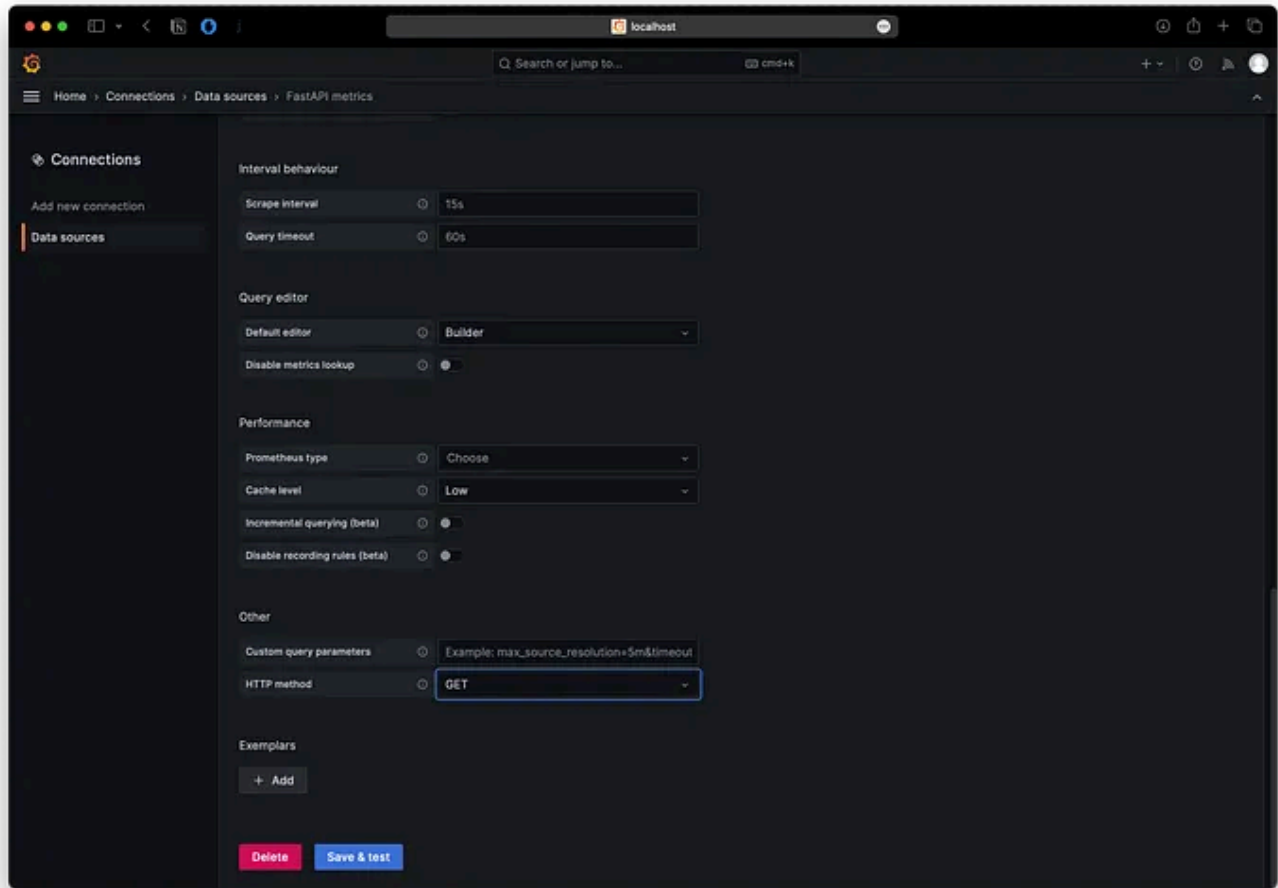
Configure the datasource with the following:

- Name: FastAPI metrics
- Prometheus server URL: <http://prometheus:9090> (use container name in the compose)
- HTTP method: GET



Prometheus server URL: <http://prometheus:9090>

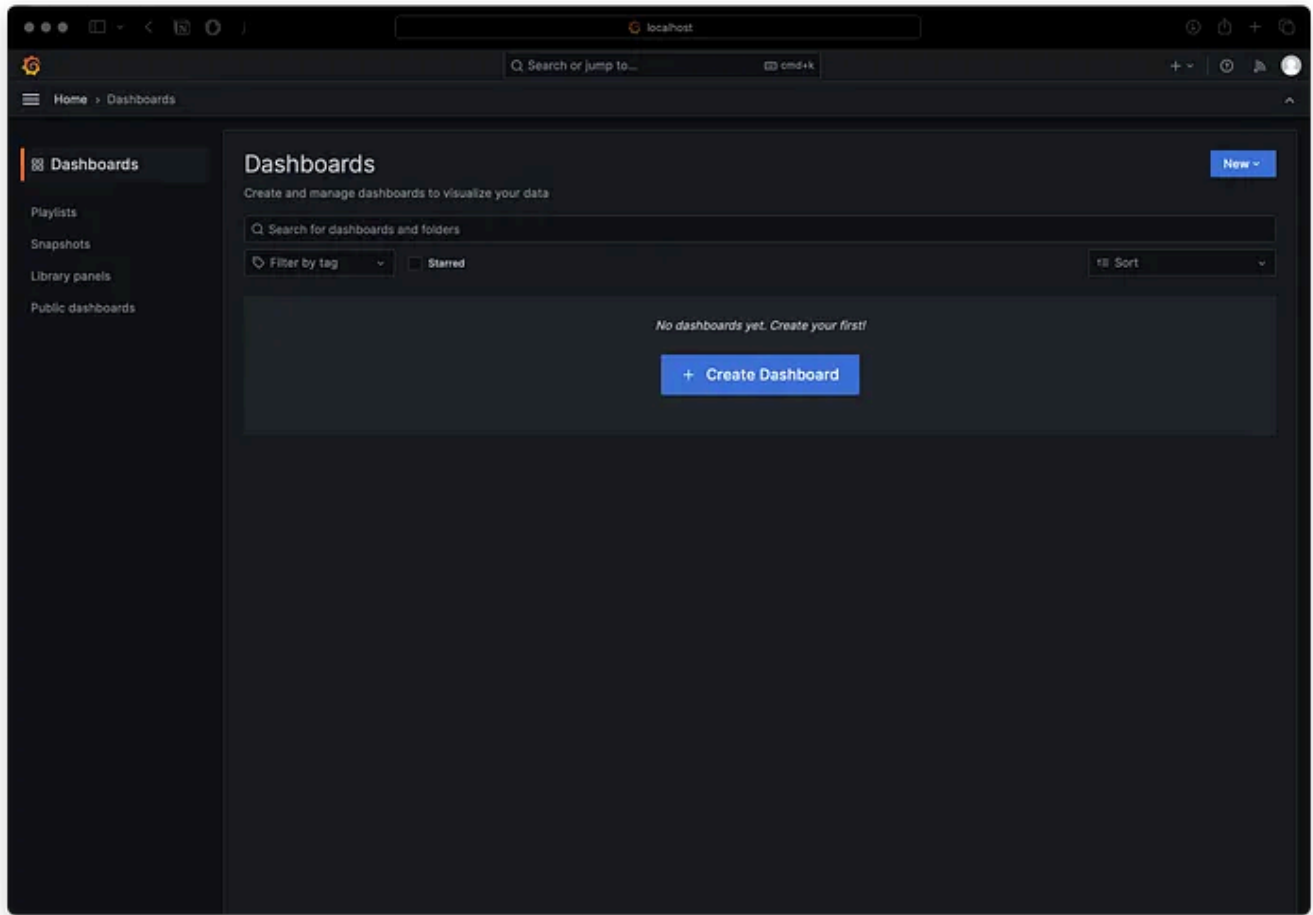




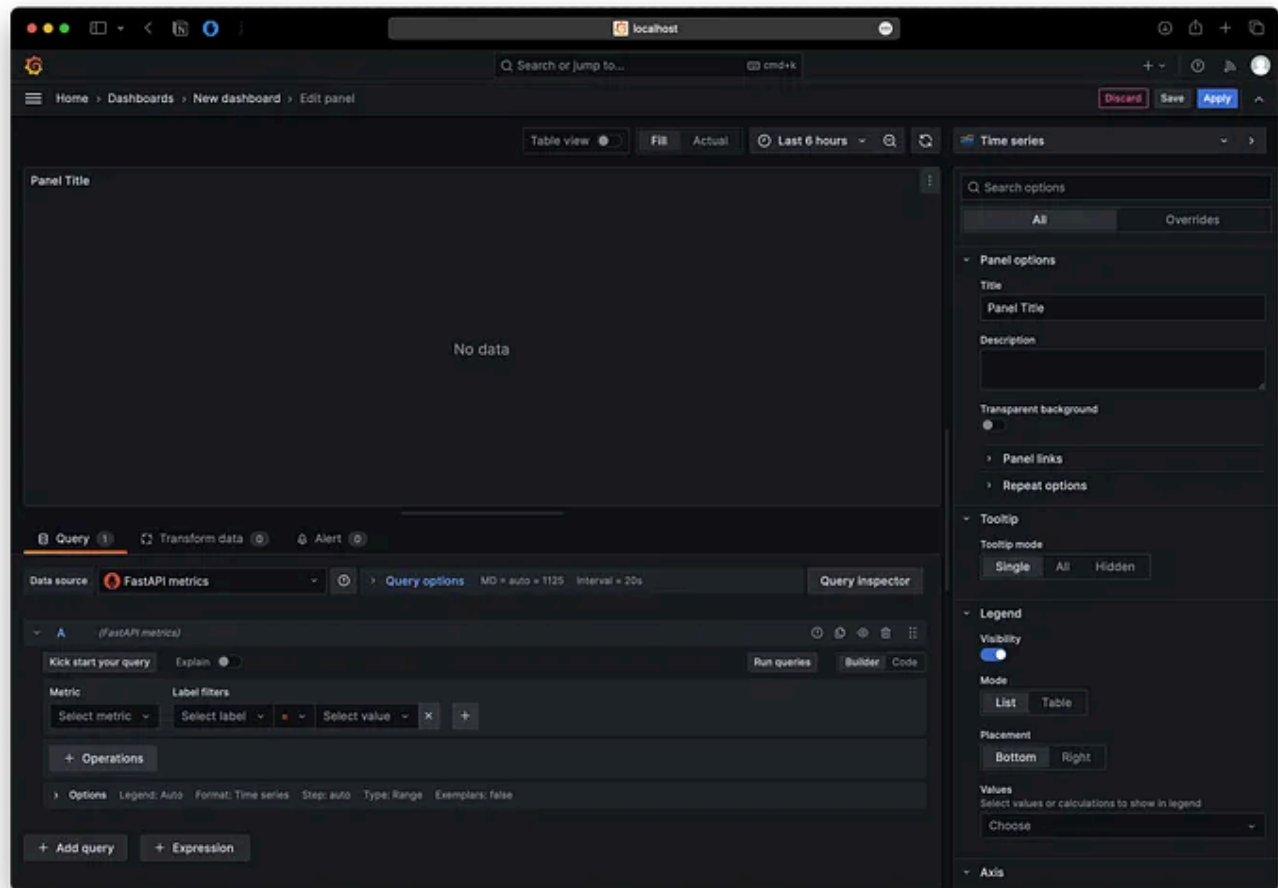
HTTP method: GET

Push “Save & test” and see “Successfully queried the Prometheus API.”

Next, go to Dashboards and click on ‘Create Dashboard’ -> ‘Add visualization.’

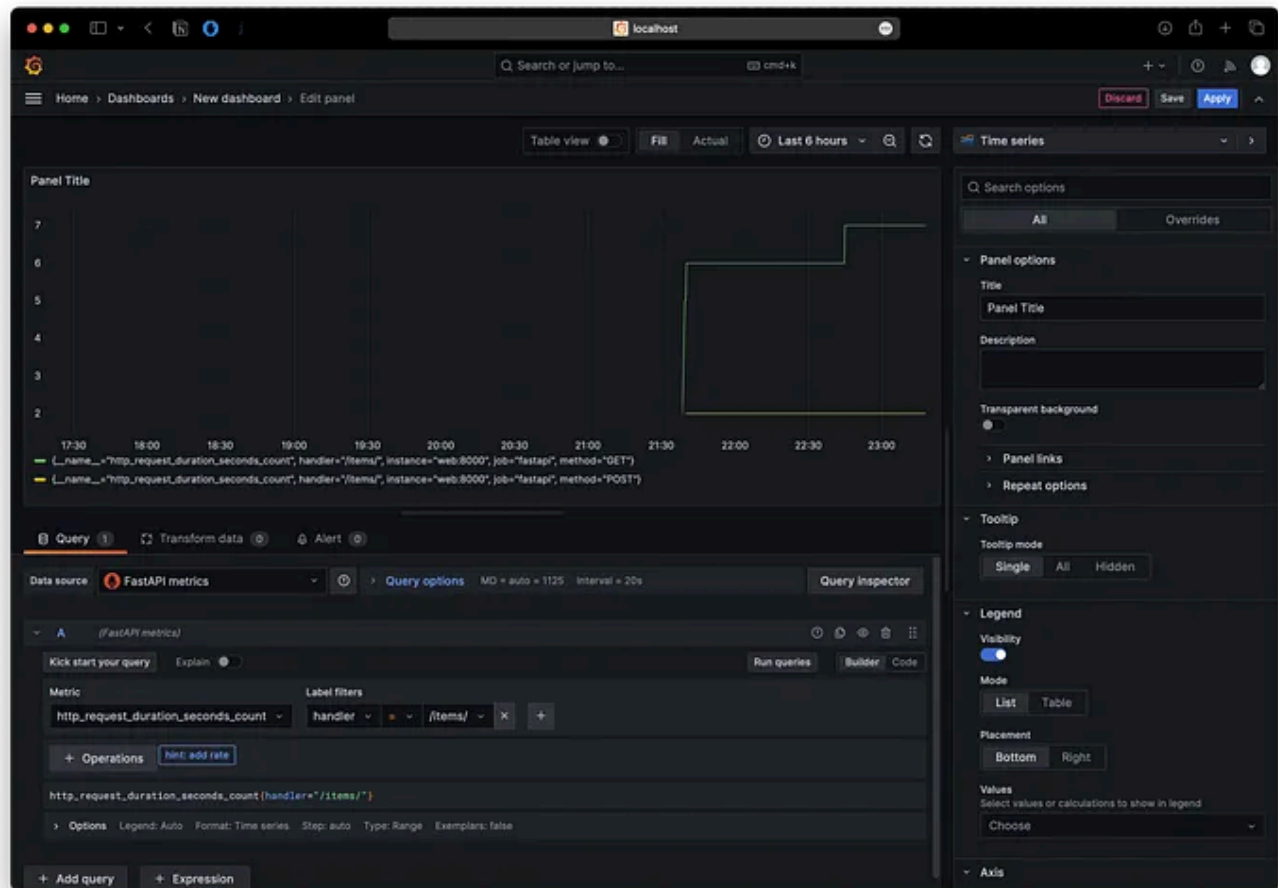


Choose the source — our added FastAPI metrics, and you’ll see an empty dashboard. Let’s add an example visualization for one of the metrics.

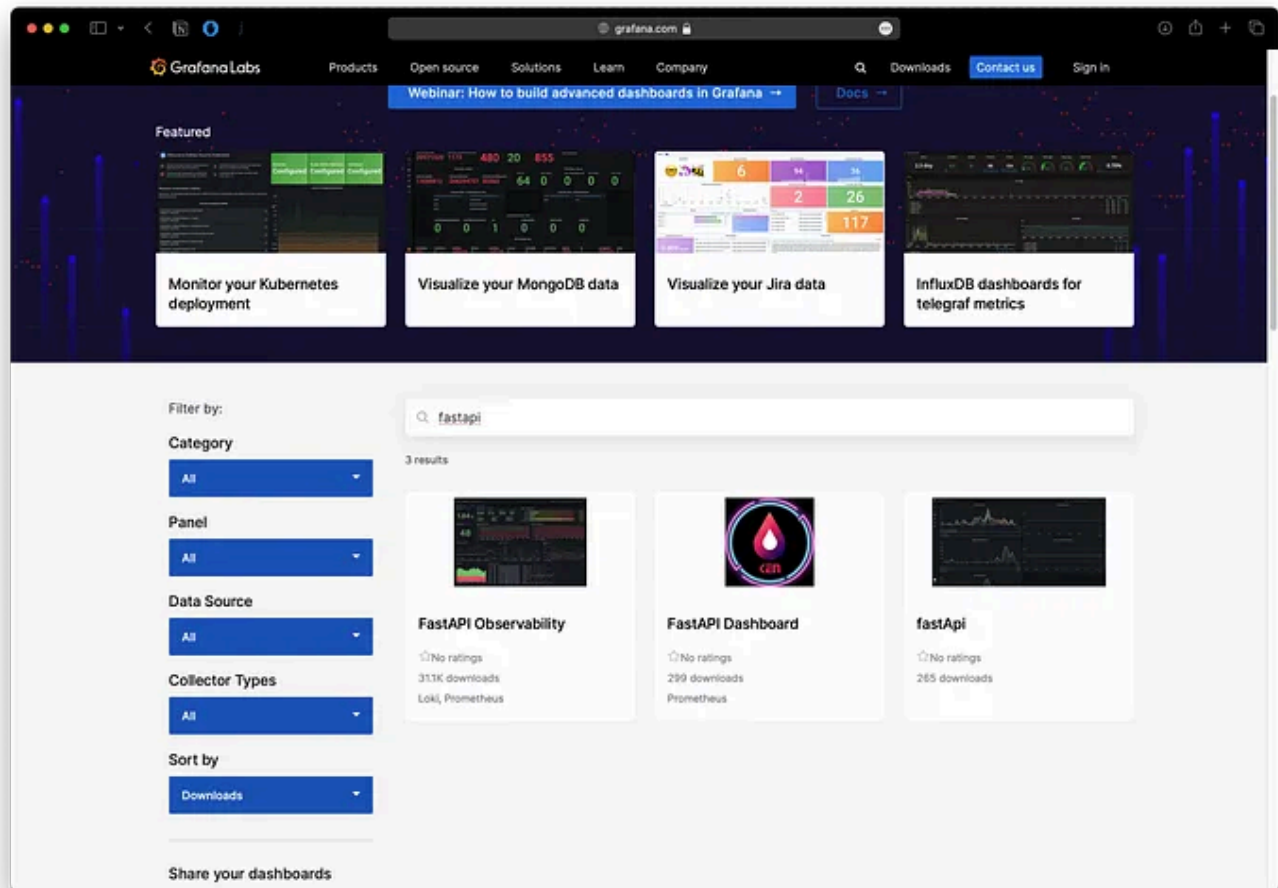


To begin, interact with our application so that metrics are collected from various endpoints. The most convenient way to do this is through the Swagger interface at <http://localhost:8000/docs>. After that, let's try visualizing the response duration metric.

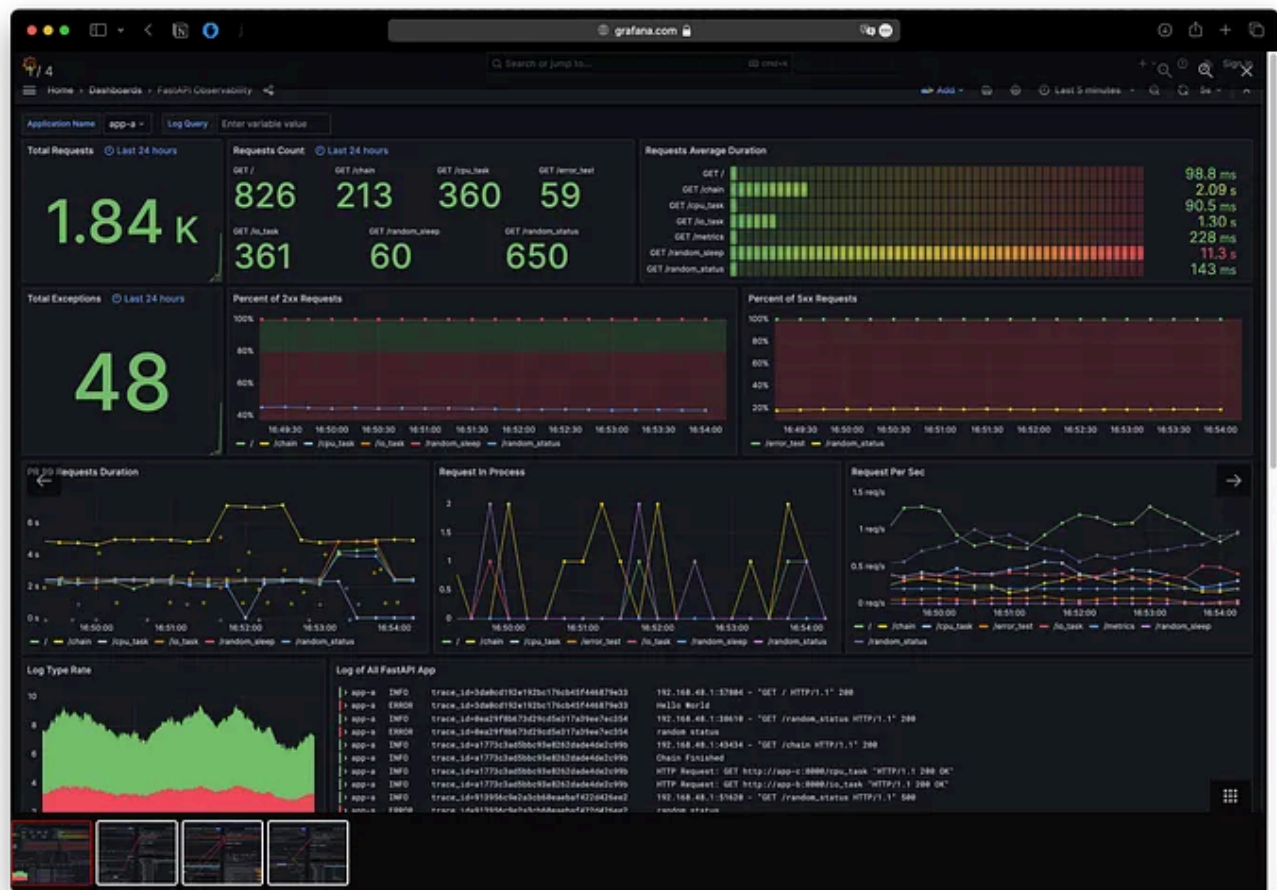
In Metrics, select `http_request_duration_seconds_count`, set the label to 'method,' select the value to '/items/' and click 'Run queries.'



This is how we've added a simple metric. I would like to note that in the Grafana store, there are ready-made dashboards that you only need to configure correctly.”



<https://grafana.com/grafana/dashboards/?search=fastapi>



## Graylog

Graylog is an open-source log management platform that enables organizations to collect, index, and analyze log data from various sources in real-time. It provides a centralized and scalable solution for handling log data, allowing users to gain insights into system activities, troubleshoot issues, and monitor the health of their applications and infrastructure.

Key features of Graylog include:

1. **Log Collection:** Graylog can collect logs from a wide range of sources, including applications, servers, network devices, and more.
2. **Log Processing:** It processes and normalizes log data, making it easier to analyze and search through the collected information.

3. **Search and Analysis:** Graylog offers a powerful search and query interface that allows users to quickly find relevant log entries based on specific criteria.
4. **Dashboards:** Users can create customizable dashboards with visualizations to gain insights into log data trends and patterns.
5. **Alerting:** Graylog provides alerting capabilities, allowing users to define conditions based on log data and receive notifications when specific events occur.
6. **Scalability:** It is designed to scale horizontally, making it suitable for handling large volumes of log data in enterprise environments.
7. **Integration:** Graylog integrates with various data sources, including Elasticsearch for data storage, and it supports plugins for extending functionality.
8. **Security:** It includes features for securing log data, such as access controls, encryption, and authentication mechanisms.

Graylog is commonly used in DevOps and IT operations for centralized log management and analysis. It plays a crucial role in monitoring, troubleshooting, and maintaining the reliability of complex systems and applications.

In the Docker Compose file for launching Graylog, the necessary dependencies in the form of Elasticsearch and MongoDB have been added from official Graylog site.

```
mongo:  
  image: mongo:3
```

```
volumes:
  - mongodb_data:/data/db

elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch-oss:6.8.10
  environment:
    - http.host=0.0.0.0
    - transport.host=localhost
    - network.host=0.0.0.0
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
  ulimits:
    memlock:
      soft: -1
      hard: -1
  deploy:
    resources:
      limits:
        memory: 1g

graylog:
  image: graylog/graylog:3.3
  environment:
    - GRAYLOG_PASSWORD_SECRET=${GRAYLOG_PASSWORD}
    - GRAYLOG_ROOT_PASSWORD_SHA2=${GRAYLOG_PASSWORD_SHA}
    - GRAYLOG_HTTP_EXTERNAL_URI=http://127.0.0.1:9000/
  depends_on:
    - mongo
    - elasticsearch
  ports:
    # Graylog web interface and REST API
    - 9000:9000
    # Syslog TCP
    - 1514:1514
    # Syslog UDP
    - 1514:1514/udp
    # GELF TCP
    - 12201:12201
    # GELF UDP
    - 12201:12201/udp
```

A dependency has also been added for our web container to wait until the Graylog container is ready to accept connections.



```
depends_on:  
  ...  
  graylog:  
    condition: service_healthy
```

Graylog uses Elasticsearch and MongoDB as dependencies for specific roles in its infrastructure:

### Elasticsearch:

- **Distributed Data Store:** Elasticsearch provides a powerful and distributed mechanism for storing and searching data. Graylog utilizes Elasticsearch for storing and indexing logs and metadata, ensuring fast and efficient searches.
- **Search and Query Support:** Elasticsearch comes with a powerful query language, allowing Graylog to perform complex search queries and data aggregations. This is crucial for analyzing and extracting information from large volumes of logs.
- **Scalability:** Elasticsearch scales horizontally with ease, making it suitable for handling substantial amounts of log data efficiently.

### MongoDB:

- **Storage for Configuration Data and Metadata:** MongoDB is used to store configuration data and metadata required for Graylog's operation. This may include information about user settings, log processing rules, and other configuration parameters.
- **Ease of Use:** MongoDB is a flexible and user-friendly NoSQL database, providing fast access to data and ease of development.

The general principle behind using Elasticsearch and MongoDB is to leverage each for its specific strengths: Elasticsearch for storing and searching events, and MongoDB for storing configurations and metadata. This ensures an efficient and scalable solution for log management within the Graylog infrastructure.

In our application, Graylog is accessible at <http://localhost:9000>. Additionally, we need to add variables GRAYLOG\_PASSWORD and GRAYLOG\_PASSWORD\_SHA to the .env file. You can obtain GRAYLOG\_PASSWORD\_SHA from your password using an online tool like <https://string-o-matic.com/sha256> or with a Python script:

```
import hashlib

text = 'Some text'

m = hashlib.sha256(text.encode('UTF-8'))
print(m.hexdigest())
```

In our application, some changes are also required to send logs to Graylog:

In src/app/config.py, we import GelfUdpHandler from [pygelf](#)

```
from pygelf import GelfUdpHandler

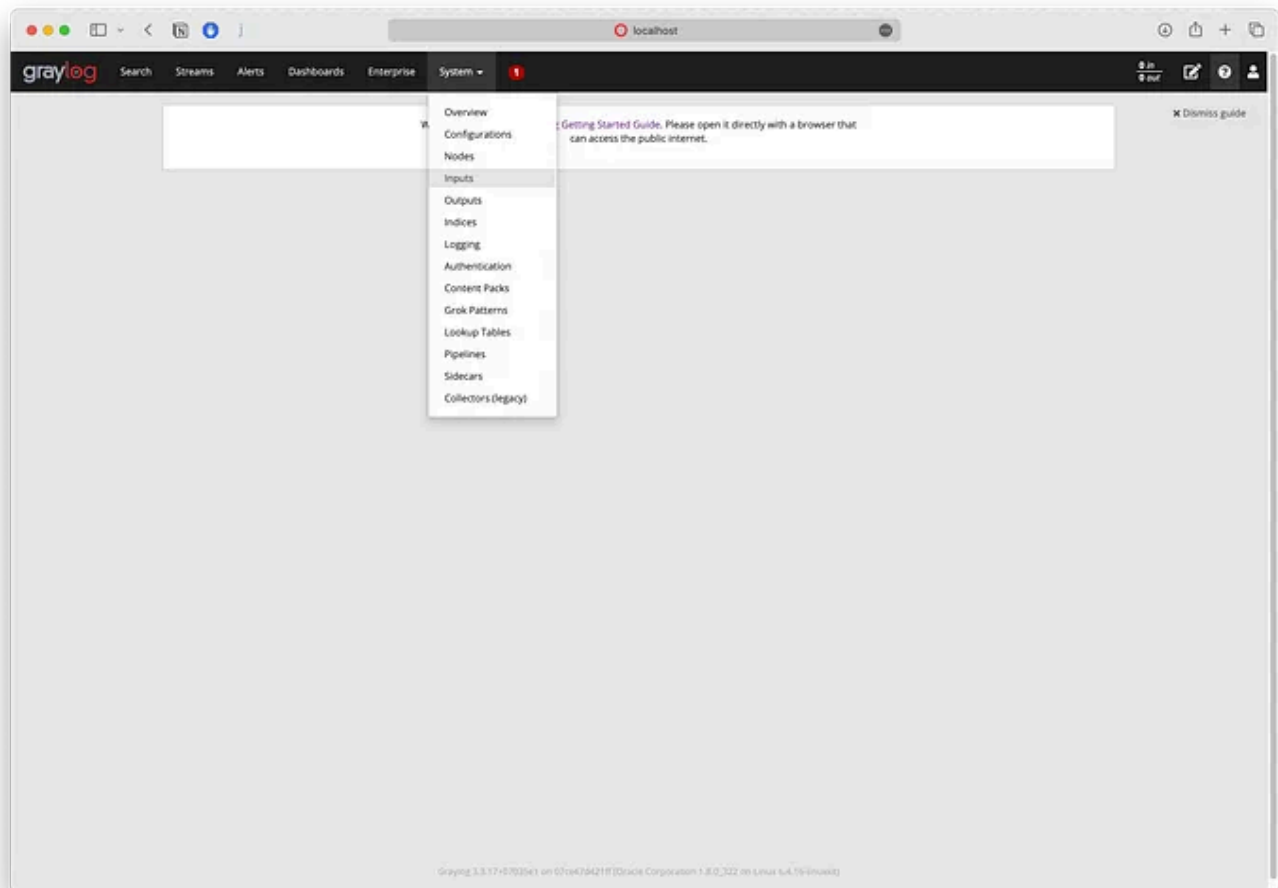
class ContextFilter(logging.Filter):
    def filter(self, record):
        record.request_id = str(uuid.uuid4())
        return True

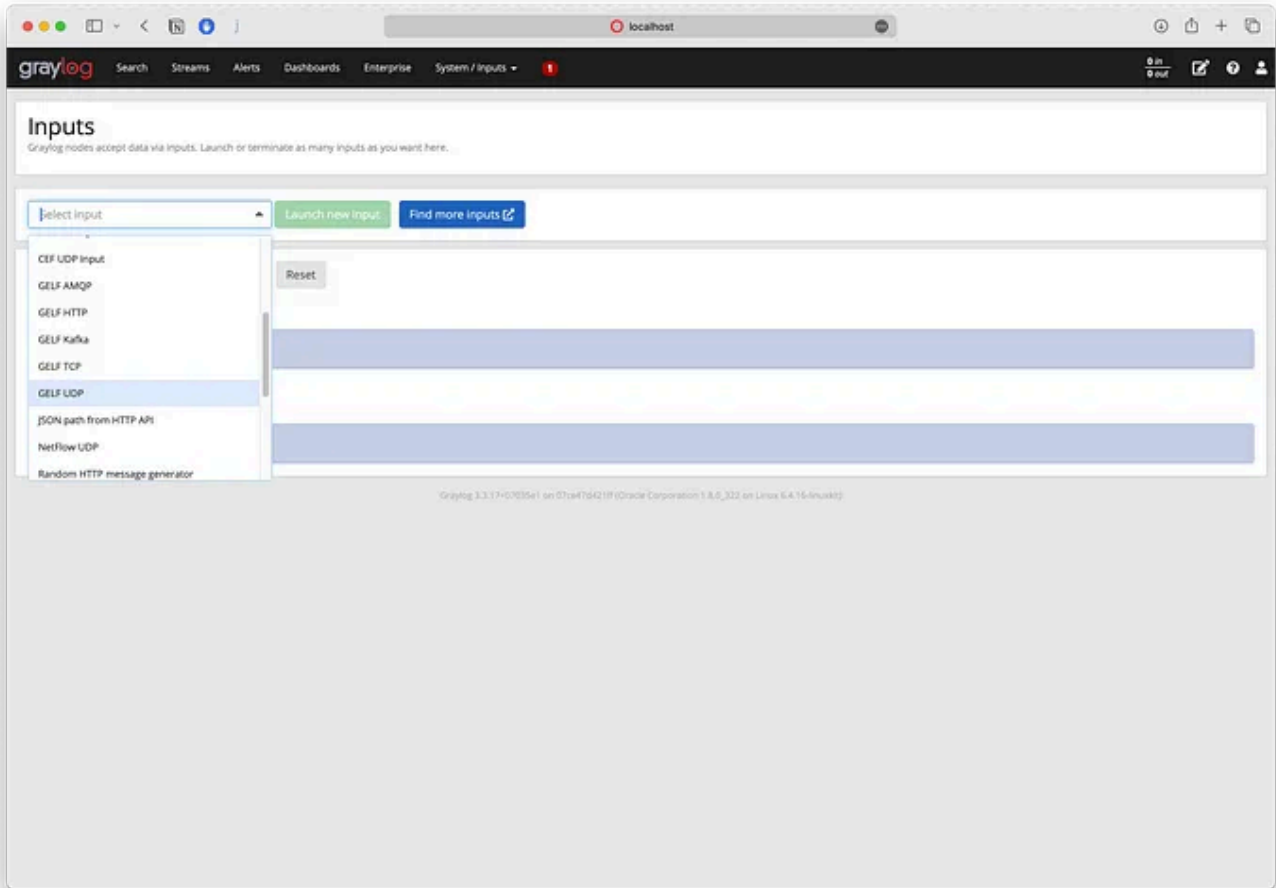
graylog_host = os.getenv("GRAYLOG_HOST")
graylog_port_udp = os.getenv("GRAYLOG_PORT_UDP")
```

```
if graylog_host and graylog_port_udp:  
    handler = GelfUdpHandler(host=graylog_host, port=int(graylog_port_udp), incl  
    logger.addHandler(handler)  
    logger.addFilter(ContextFilter())
```

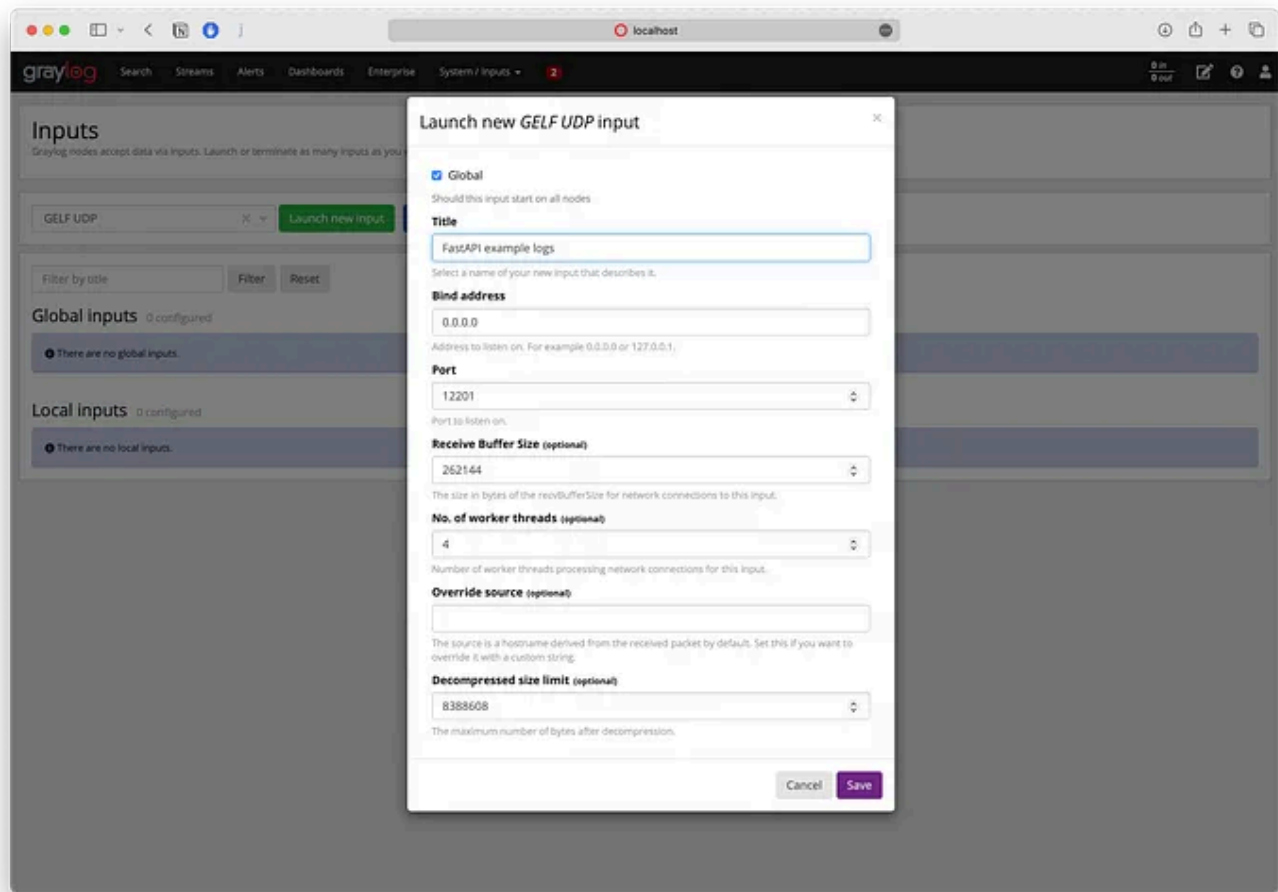
Here, we retrieve the GRAYLOG\_HOST and GRAYLOG\_PORT\_UDP variables from the environment. If they are present, we configure the connection to Graylog with the GELF (Graylog Extended Log Format) format. The ContextFilter class adds the request\_id to each log entry.

Got to UI Graylog and configure date input.

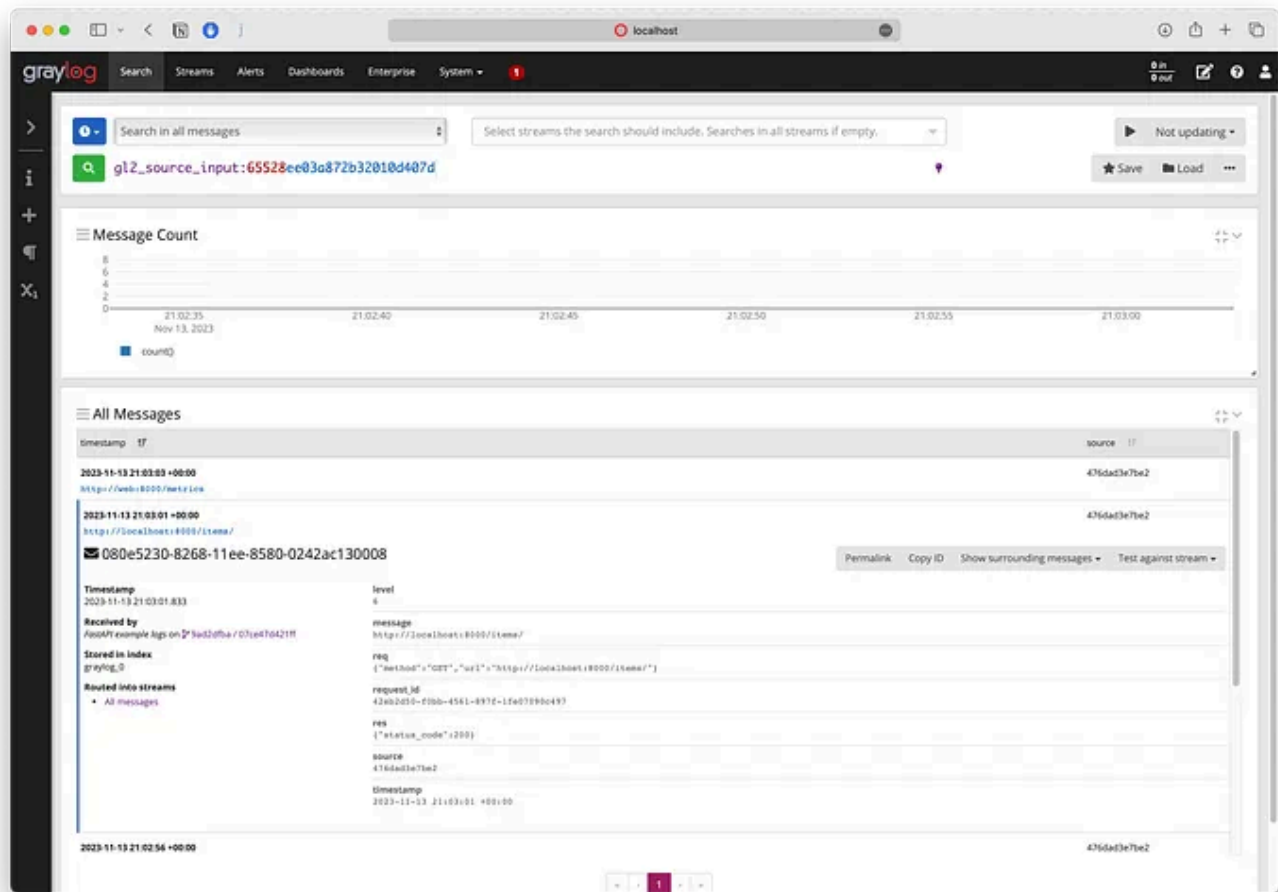




Input - GELF UDP



Write some name for input and select global



Result

On the “Search” page in Graylog, you can perform various actions with your logs for analysis and extracting useful information. Here are several actions you can take:

### Search and Filtering:

- Use the search field to enter queries. Graylog supports a powerful query language that allows you to specify search conditions precisely.
- Apply filters to narrow down the search scope. Click on the “Add Filter” button and choose parameters by which you want to filter logs.

### Selecting a Time Range:

- Use time selectors to specify the time range you are interested in. It can be a specific period or a relative interval, such as “Last hour.”

### Log Analysis:

- Graylog provides a convenient interface for log analysis. You can see various log fields and information about events.
- Use the information provided in the interface to identify issues, analyze performance, and track actions in the system.

### Viewing Detailed Information:

- Clicking on a specific event allows you to view detailed information about the log, including all extracted fields, timestamps, log levels, and other metadata.

### Exporting Results:

- You can export search results to various formats such as CSV or JSON. This is useful if you need to share results or perform additional analysis outside of Graylog.

### Log Actions:

- Graylog allows you to perform additional actions with logs, such as adding tags, creating alerts, performance analysis, and more.

### Using Dashboards:

- If you have pre-created dashboards, navigate to the “Dashboards” section to visualize data in a convenient format.

These features will help you efficiently analyze and work with your logs to identify issues, monitor performance, and ensure the stable operation of your system.

## Conclusion:

In this comprehensive guide, we have successfully set up a FastAPI application integrated with Prometheus, Grafana, and Graylog using Docker Compose. This powerful combination provides us with robust monitoring and logging capabilities, ensuring the reliability and performance of our web application.

## Invitation to Collaborate:

We welcome contributions and enhancements to [this project on GitHub](#). If you have ideas, improvements, or additional features you'd like to see, please feel free to fork the repository and submit pull requests. Together, we can make this example even more valuable for the developer community.

Happy coding!

[Backend](#)[Python](#)[Fastapi](#)[Docker](#)[Grafana](#)

Written by Denis Gr

5 Followers · 1 Following

[Follow](#)



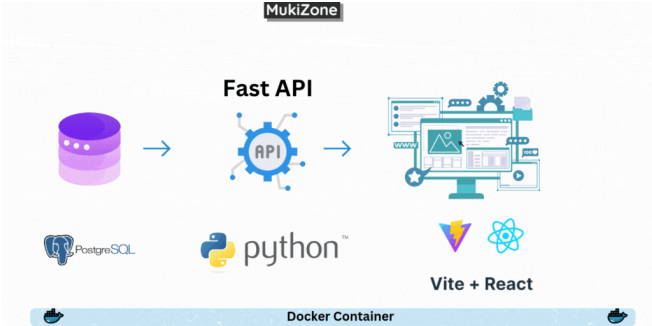
# No responses yet



Fahad

What are your thoughts?

# Recommended from Medium



Mukesh Vast

## Building a basic Tracker with Python FastAPI, PostgreSQL and...

Spin up a tracker in no time.



Philip Wawazi


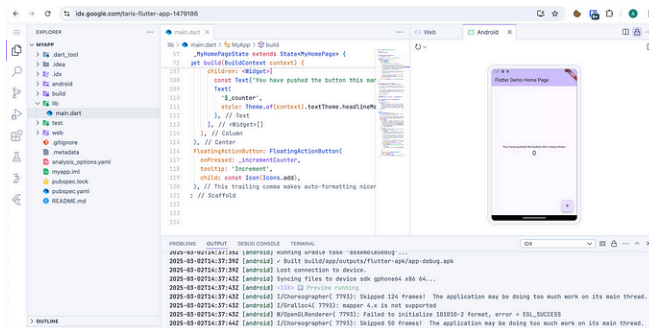
## How to Log and Monitor a Django REST Framework (DRF) API with...

As developers, we often deploy APIs and trust that our error handling will surface any issue...

★ Feb 18 🤝 2



Feb 1 🤝 5

 In Level Up Coding by Gautam Bharadwaj

## How I Simplified Kubernetes Monitoring Using Prometheus an...

Deploy Once, Monitor Everything: Your Unified K8s Monitoring Stack

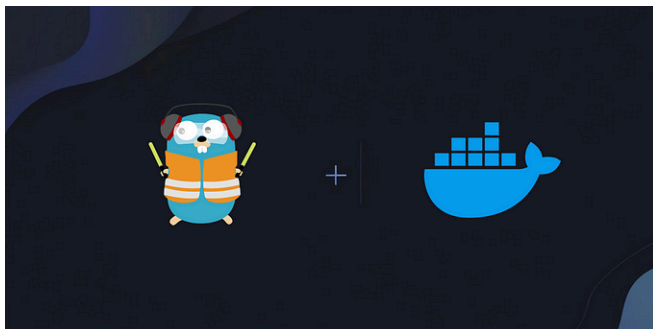
★ 3d ago 🤝 373

 In Coding Beauty by Tari Ibaba

## This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.

★ Mar 12 🤝 4.8K 💬 276

 Prateek Jain

## Why I Replaced NGINX with Traefik in My Docker Compose Setup

For a long time, NGINX was my go-to solution for setting up reverse proxies. Whether it wa...

★ 6d ago 🤝 550 💬 8

 In Python in Plain English by Vijay

## How to Build a Lightning-Fast API with FastAPI and AsyncIO

Learn how I optimized my FastAPI app with AsyncIO to handle high traffic and boost...

★ 3d ago 🤝 83 💬 1



See more recommendations