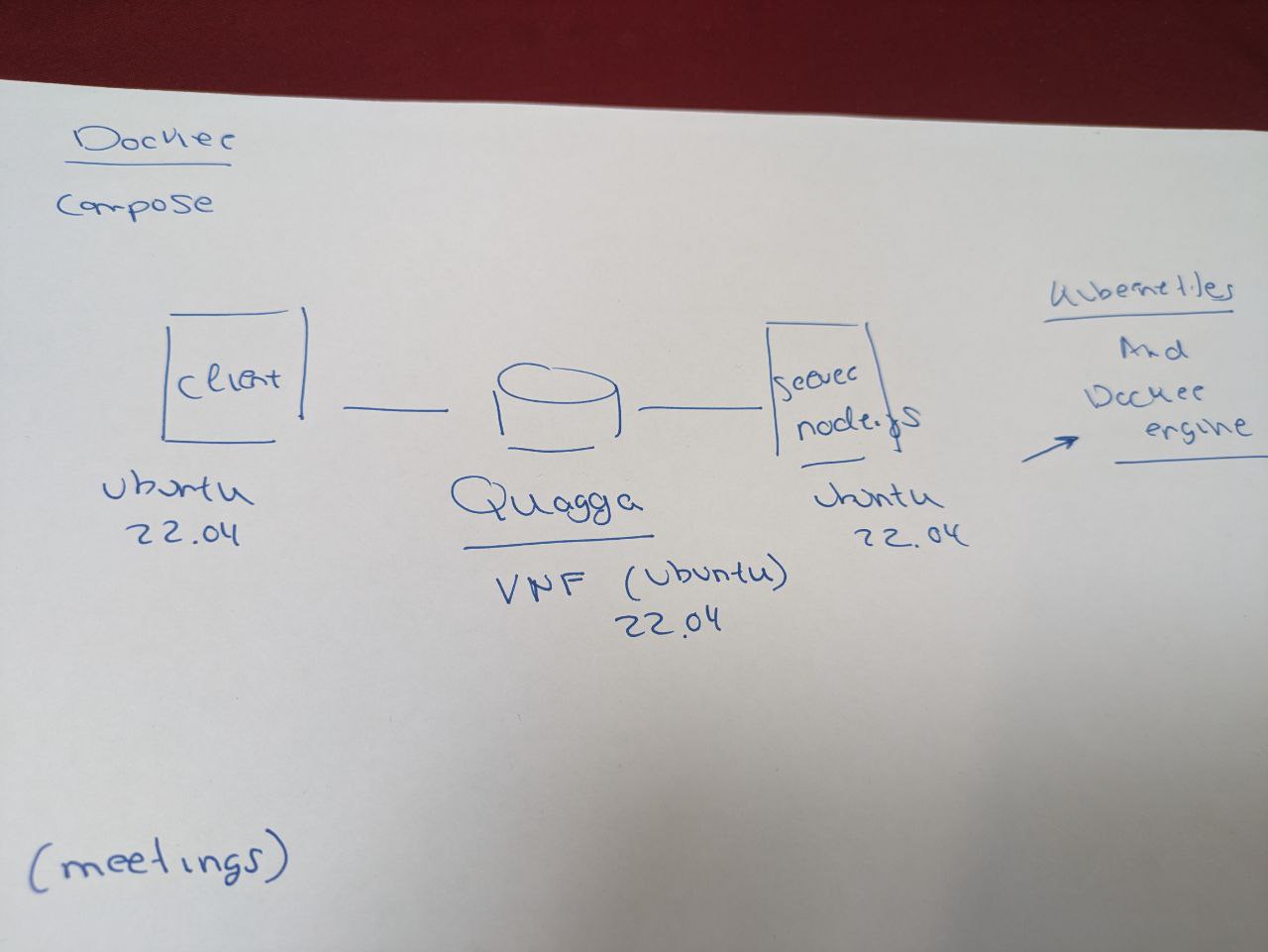
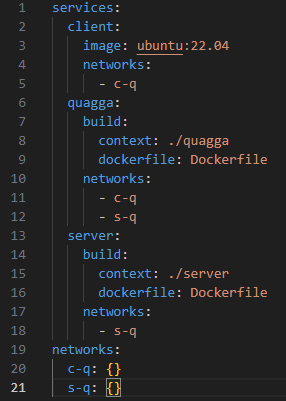
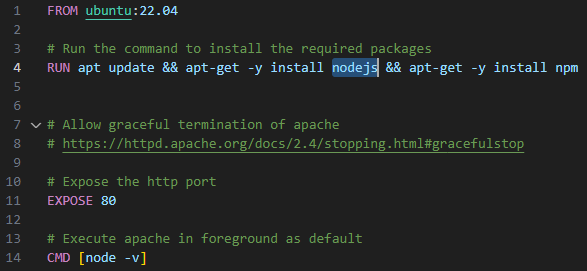
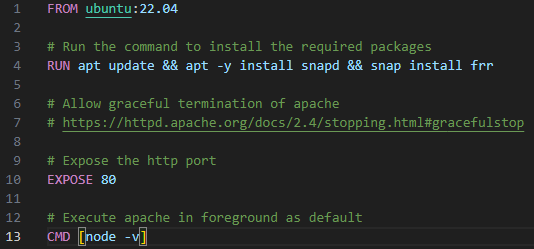
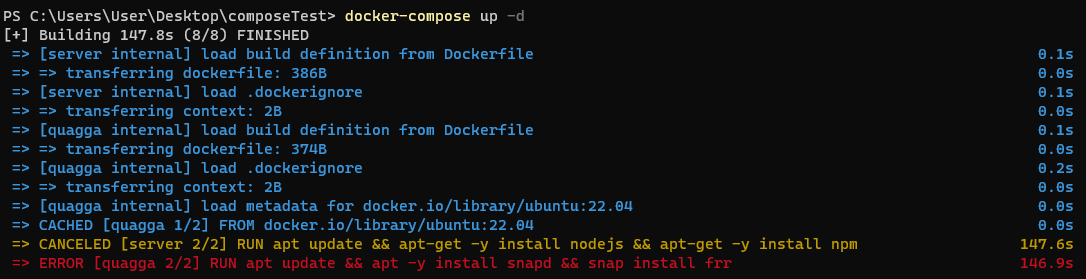
The aim is creating a virtual environment to create a monitoring system or a way to analyse the state of the system.





 server







It seemed that quagga was not available for Ubuntu 22.04 and the installation process it is very time expensive. For this reason, it is taken as hypothesis to switch the image of Ubuntu with other which are lighter.

Comparison between quagga, frr and a new base image.

Quagga is a routing software package that provides TCP/IP based routing services, using protocols such as RIP, OSPF, IS-IS and BGP-4. It also supports IPv6 routing protocols. A system with Quagga installed acts as a dedicated router. With Quagga, your machine exchanges routing information with other routers using routing protocols. Quagga uses this information to update the kernel routing table so that the right data goes to the right place.

It is made from a collection of several daemons that work together to build the routing table. There may be several protocol-specific routing daemons and zebra the kernel routing manager.

Each protocol has its own daemon in charge of managing that routing table. Finally, one more daemon that manages the kernel routing table. It is easy to add a new routing protocol daemons to the entire routing system without affecting any other software. You need to run only the protocol daemon associated with routing protocols in use. Thus, user may run a specific daemon and send routing reports to a central routing console. Each daemon has it’s own configuration file and terminal interface. When you configure a static route, it must be done in zebra configuration file.

FRRouting is a free and open source IRP suite for Linux and Unix platform with its roots planted in Quagga. It offers some improvement for each protocol offered by quagga

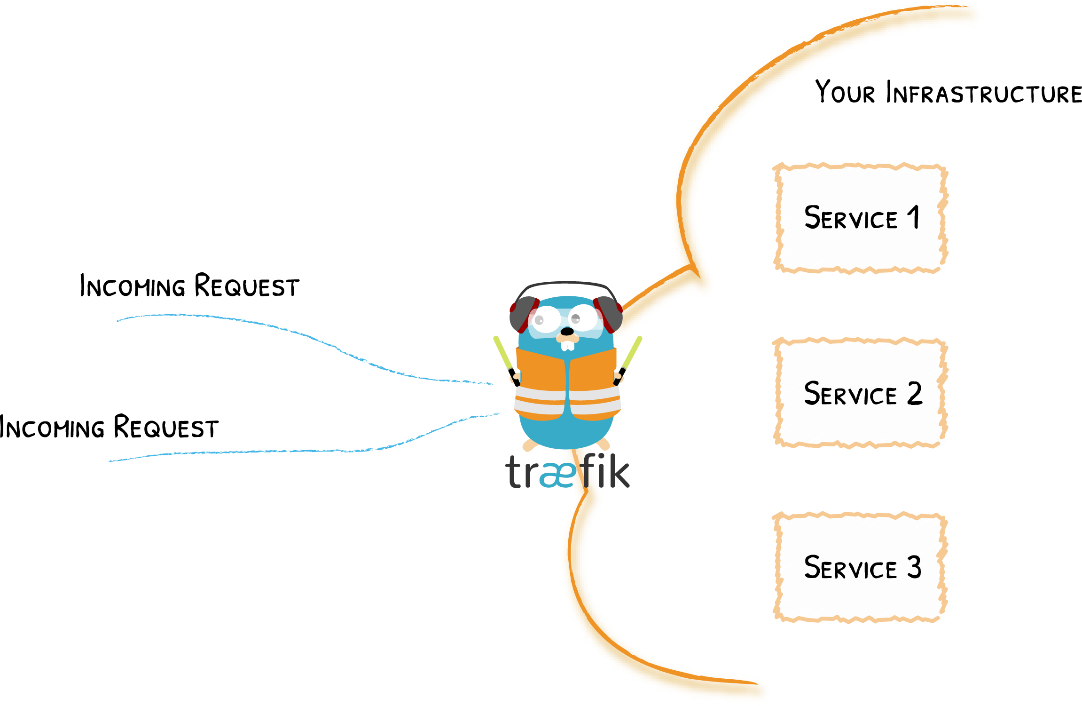
Traefik is an [open-source](https://github.com/traefik/traefik) Edge Router that makes publishing your services a fun and easy experience. It receives requests on behalf of your system and finds out which components are responsible for handling them. With Traefik, there is no need to maintain and synchronize a separate configuration file: everything happens automatically, in real time (no restarts, no connection interruptions).

Traefik is based on the concept of EntryPoints, Routers, Middlewares and Services.

The main features include dynamic configuration, automatic service discovery, and support for multiple backends and protocols.

1. [EntryPoints](https://doc.traefik.io/traefik/routing/entrypoints/): EntryPoints are the network entry points into Traefik. They define the port which will receive the packets, and whether to listen for TCP or UDP.
2. [Routers](https://doc.traefik.io/traefik/routing/routers/): A router is in charge of connecting incoming requests to the services that can handle them.
3. [Middlewares](https://doc.traefik.io/traefik/middlewares/overview/): Attached to the routers, middlewares can modify the requests or responses before they are sent to your service
4. [Services](https://doc.traefik.io/traefik/routing/services/): Services are responsible for configuring how to reach the actual services that will eventually handle the incoming requests.

Traefik is an Edge Router (or reverse Proxy), it means that it's the door to your platform, and that it intercepts and routes every incoming request: it knows all the logic and every [rule](https://doc.traefik.io/traefik/routing/routers/#rule) that determine which services handle which requests



when a service is deployed, Traefik detects it immediately and updates the routing rules in real time. Similarly, when a service is removed from the infrastructure, the corresponding route is deleted accordingly.

Trying to set Traefik I had trouble trying to ping from the client to a server. But this could be related to the fact that the router was not set as forwarding node (to be checked).

FRR seemed not to work. Once configured as show in the documentation and started the service, it required a lot of time to complete the command and once checked if the service was actually working (service frr status), the answer was:

\* Status of watchfrr: running

 \* Status of zebra: FAILED

 \* Status of bgpd: FAILED

 \* Status of ospfd: FAILED

 \* Status of staticd: running

Which means that the daemons failed and are not running. To solve this problem I had to set higher privileges.

Once done this and set the router as forwarding node (to check if it is necessary) the frr daemons started working.

During the studying, I also tried to run GNS3 and it ended in bringing compatibility problems related to WSL.

Comparison between Ubuntu with node.js, alpine+node and apache.

I have to agree with the others, I always use official images if I can and I always select the Alpine version if available. Why use Alpine? Two reasons:

1. Much small images. Ubuntu is 188MB alone. Then you add your app on top of that probably exceeding 200MB. Alpine Linux is only 4MB! After adding my Python runtime and code most of my images are only 52MB. Compare that will almost 200MB of Ubuntu. Smaller images are smaller upload/download and take up less disk space.
2. Smaller attack surface! When you start from Ubuntu, you are adding lots of other services that may be running that are not needed and could be exploited. Alpine Linux has very little to attack so it is inherently more secure.

You don’t need an entire generic OS… all you need is a runtime for your app and nothing more. I always start with Alpine Linux when I’m building my own containers from scratch.

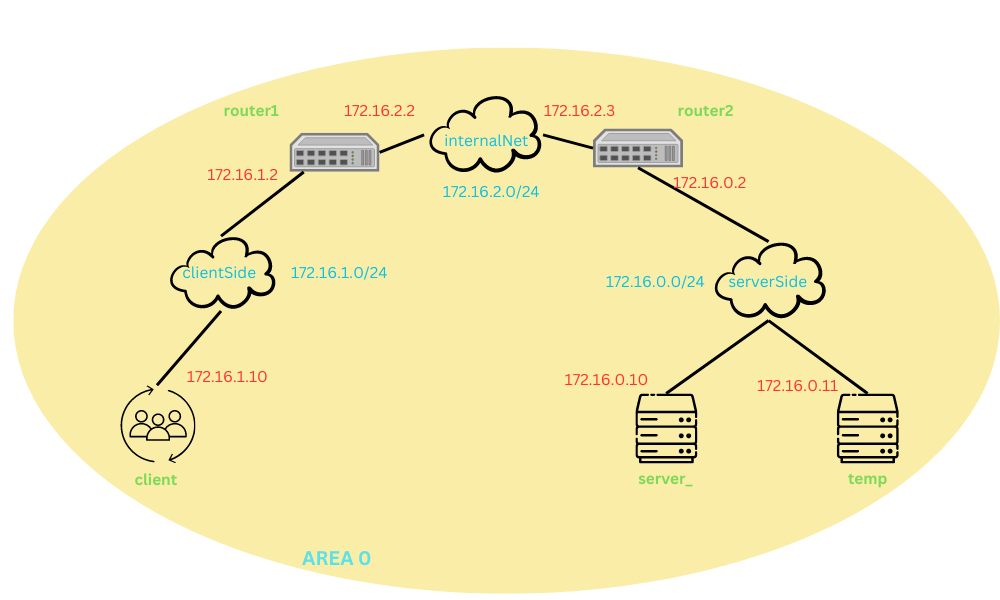
How to use node:

<https://github.com/optimized/docker-node/blob/master/Dockerfile>

https://github.com/nodejs/docker-node/blob/main/README.md#how-to-use-this-image

<https://github.com/nodejs/docker-node>

Since the protocol I decided to use is ospf and a routing protocol make sense if there are more than one router, I decided to add one more but to make it simple by putting everything inside just one area (area 0).



**# Intelligent-forensics-for-the-automatic-anomaly-detection-in-distributed-infrastructures README.md**

The system is composed by a client (ubuntu 22.04 with iputils-ping and iproute2), 2 server (server\_: ubuntu 22.04; temp: ubuntu 22.04 with iputils-ping, iproute2, nodejs, npm and other packet that still have their usefullness to be checked) and 2 routers (ubuntu 22.04 with iputils-ping, iproute2, coreutils, iptables, frr).

The routers have been set t0 work with the protocol ospf. For sake of semplicity, in this phase, the entire architecture have been inserted in a single area (area 0).

![architecture](./images/architecture\_temp2.png)

To run the system:

1. run the docker engine (tested: by running docker desktop)

2. launch the command "docker-compose up -d"

3. access the server temp "docker exec -it intelligent-forensics-temp-1 bash". Add the route to the client network "ip route add 172.16.1.0/24 via 172.16.0.2", in which 172.16.0.2 is the interface of the router connected to the server (router2).

4. in another terminal, access the client "docker exec -it intelligent-forensics-client-1 bash". Add the route to the server network "ip route add 172.16.0.0/24 via 172.16.1.2", in which 172.16.1.2 is the interface of the router connected to the client (router1). Test the connection "ping 172.16.0.11" by pinging the server temp: it should fail:

```console

PING 172.16.0.11 (172.16.0.11) 56(84) bytes of data.

From 172.16.1.2 icmp\_seq=2 Redirect Host(New nexthop: 172.16.1.1)

From 172.16.1.2 icmp\_seq=3 Redirect Host(New nexthop: 172.16.1.1)

From 172.16.1.2 icmp\_seq=4 Redirect Host(New nexthop: 172.16.1.1)

```

5. access the router1 "docker exec -it intelligent-forensics-router1-1 bash". Start frr "service frr start" and check its status "service frr status" is:

```console

 \* Status of watchfrr: running

 \* Status of zebra: running

 \* Status of bgpd: running

 \* Status of ospfd: running

 \* Status of staticd: running

```

which means all the frr daemons chosen are running.

6. access the router2 "docker exec -it intelligent-forensics-router2-1 bash". Start frr "service frr start" and check its status "service frr status" is:

```console

 \* Status of watchfrr: running

 \* Status of zebra: running

 \* Status of bgpd: running

 \* Status of ospfd: running

 \* Status of staticd: running

```

which means all the frr daemons chosen are running.

7. Now it's necessary to configure the protocol for both the router. Enter the vty bash with the command "vtysh". Enter the configuration with "conf t" and then the conf of the router "router ospf". Finally, insert the network connected to the router into the same area with "network <xx.xx.xx.xx/xx> area <x>".

     - router1: "network 172.16.2.0/24 area 0" "network 172.16.1.0/24 area 0"

     - router2: "network 172.16.2.0/24 area 0" "network 172.16.0.0/24 area 0"

     Finish this operation with "end"

After this operation, thanks to the daemon of ospf, the routers are going to exchange their routing tables. Verify it with the command "show ip route" that should give:

- router1:

```console

    K>\* 0.0.0.0/0 [0/0] via 172.16.1.1, eth0, 00:02:52

    O>\* 172.16.0.0/24 [110/20] via 172.16.2.3, eth1, weight 1, 00:00:54

    O   172.16.1.0/24 [110/10] is directly connected, eth0, weight 1, 00:01:29

    C>\* 172.16.1.0/24 is directly connected, eth0, 00:02:52

    O   172.16.2.0/24 [110/10] is directly connected, eth1, weight 1, 00:01:37

    C>\* 172.16.2.0/24 is directly connected, eth1, 00:02:52

```

- router2

```console

    K>\* 0.0.0.0/0 [0/0] via 172.16.2.1, eth0, 00:02:41

    O   172.16.0.0/24 [110/10] is directly connected, eth1, weight 1, 00:01:00

    C>\* 172.16.0.0/24 is directly connected, eth1, 00:02:41

    O>\* 172.16.1.0/24 [110/20] via 172.16.2.2, eth0, weight 1, 00:00:55

    O   172.16.2.0/24 [110/10] is directly connected, eth0, weight 1, 00:01:02

    C>\* 172.16.2.0/24 is directly connected, eth0, 00:02:41

```

8. Exit the vtysh with "exit" and check the route of the host with "ip route show":

- router1:

```console

    default via 172.16.1.1 dev eth0

    172.16.0.0/24 nhid 12 via 172.16.2.3 dev eth1 proto ospf metric 20

    172.16.1.0/24 dev eth0 proto kernel scope link src 172.16.1.2

    172.16.2.0/24 dev eth1 proto kernel scope link src 172.16.2.2

```

- router2:

```console

   default via 172.16.2.1 dev eth0

    172.16.0.0/24 dev eth1 proto kernel scope link src 172.16.0.2

    172.16.1.0/24 nhid 14 via 172.16.2.2 dev eth0 proto ospf metric 20

    172.16.2.0/24 dev eth0 proto kernel scope link src 172.16.2.3

```

9. Now verify that the client can actually ping the server. Enter the client bash and "ping 172.16.0.11" to ping the server temp.

**# advices**

If the images gets modified by changing the dockerfiles, to be effective the image should be mounted again or simply deleted before using again "docker-compose up -d". Fast way is deleting the images.

EXTRA: check if dividing in 2 areas the architecture something change and further configuration are needed. For example it is possible that the border routers to exchange info between them need others protocol like bgp.

DOCKER LOGS: <https://stackoverflow.com/questions/41144589/how-to-redirect-docker-container-logs-to-a-single-file>

FIWARE:

Generated with chatGPT:

FIWARE is an open-source platform designed to facilitate the development and deployment of smart applications and services in various domains, particularly within the context of the Internet of Things (IoT) and smart cities. It provides a set of standardized APIs (Application Programming Interfaces) that enable seamless communication and data exchange between different components and systems.

The main goal of FIWARE is to create a framework that promotes interoperability, scalability, and reusability of smart solutions. It aims to bridge the gap between proprietary technologies and provide a common foundation for developers, businesses, and governments to build smart applications and services more efficiently.

Key features and components of FIWARE include:

1. Context Broker: The Context Broker is a central component of FIWARE that manages and stores context information about devices, sensors, and entities. It allows applications to query and update context data in real-time.

2. NGSI (Next Generation Service Interface) API: This API is used to interact with the Context Broker and access context information.

3. Data Models: FIWARE provides standard data models for various types of entities, making it easier to represent and handle different types of data in a standardized way.

4. IoT Agents: FIWARE offers IoT agents that facilitate the integration of IoT devices and sensors with the platform.

5. Application Mashups: FIWARE enables the development of applications by combining data and services from various sources, fostering a collaborative and integrated approach to development.

6. Publishing and Subscribing: FIWARE supports the publish/subscribe model, allowing applications to subscribe to real-time updates and events from context data sources.

FIWARE is not limited to a specific industry or application area, but it has found significant adoption in various domains, including smart cities, agriculture, transportation, environmental monitoring, and more.

The project is driven by the FIWARE Foundation, a non-profit organization that oversees the development and promotion of the FIWARE platform. It encourages an open and collaborative ecosystem, where developers can contribute to the platform's enhancement and share their applications with the community.

FIWARE is an open-source platform designed to offer a universal way of represents data to facilitate communication between different organisations and individuals. Its standardised API allows the acceleration of the development of smart solutions promoting interoperability, scalability and reusability. It deals mainly with the context of the IoT and smart cities.

Generated with chatGPT:

Yes, within the FIWARE ecosystem, there is a specific component known as "FIWARE Cygnus" that is related to Docker and is used for gaining information from Docker logs and other data sources.

FIWARE Cygnus is an open-source connector service that facilitates the integration of context data from different sources into FIWARE's Context Broker. It is commonly used for handling context data generated by IoT devices, but it can also be employed for other sources like logs from Docker containers.

Cygnus supports multiple persistence sinks, including databases (such as MySQL and MongoDB) and data brokers (such as Apache Kafka). These sinks allow Cygnus to store data from various sources in a format that is easily consumable by other FIWARE components and applications.

To collect information from Docker logs using Cygnus, you would typically set up the Docker container logs as a data source for Cygnus. Cygnus would then process these logs and forward the relevant context data to the FIWARE Context Broker, which acts as a central repository for context information.

It's important to note that the FIWARE ecosystem is dynamic and continuously evolving, so it's always a good idea to check the latest documentation and resources for specific details on how to set up and use Cygnus for Docker log integration. Additionally, other components or plugins may be available or introduced over time to handle Docker logs or other aspects of Docker within the FIWARE framework.

<https://fiware-docker-container-service.readthedocs.io/en/latest/>

<https://fiware-monitoring.readthedocs.io/en/develop/> : <https://github.com/telefonicaid/fiware-monitoring>

## Challenge & Context

In December 2019, the European Commission launched the [European Green Deal,](https://ec.europa.eu/info/strategy/priorities-2019-2024/european-green-deal_en) designing the path for Europe to become the first climate-neutral continent. At the centre of the strategy is the understanding that any ambition for a positive impact on climate and the environment needs to be closely matched by economic measures to assure the well-being of European citizens and the success of its economy. The “electrification” of our societies (together with a reinforced focus on renewable energy sources) is seen as a way to build both a new “industrial revolution” and strong economies across all of Europe.

This can also be seen as a confirmation of **how rapidly the world of energy is changing** and the implications of such changes, not only on the economies and our societies but also on the way we live, do business and interact with the environment.

In this context, **FIWARE technologies** have been used in the last few years **to experiment on solutions to empower the role of “local energy communities”**, also called “prosumers”. Being a portmanteau of the words provider and consumer, a prosumer is an individual who both consumes and produces.  This new category of “prosumers” in the energy marketplace can have a significant impact on the achievement of the climate neutrality objectives set by the European Union and the future economic development of its Member States [**FINESCE**](http://www.finesce.eu/) (Future INtErnet Smart Utility ServiCEs).

Introduced by the 2019 EU Electricity Directive, which requests that legal and commercial barriers to the participation of consumers to the energy markets should be removed,  “local (or “*citizens’*, as they are often called) energy communities” could change the way through which energy is a) produced, b) transmitted, c) distributed and d) used.

Various research initiatives across Europe have been developing and testing architectural models and use cases enabling communities to target energy autarchy, improved flexibility, or developing local energy marketplaces by leveraging FIWARE technology. Several platforms (e.g. [FISMEP](https://fismep.de/)) and services have been validating new use cases, and the added value provided by FIWARE[Context Broker](https://www.fiware.org/developers/) and other FIWARE generic enablers acknowledged by some of the most renowned experts in the development of microgrids services and energy efficiency.

While technological innovation, supported by the FIWARE Community, helps to address resilience, affordability and vulnerabilities within the energy sector, the combined bottom-up pressure from urban authorities and the top-down push from national regulators could, in the near future, be of extreme benefit. This could indeed support the changes deemed necessary to revolutionise the energy sector, fastening and helping scale-up the transition process to new digital infrastructure management and business models.

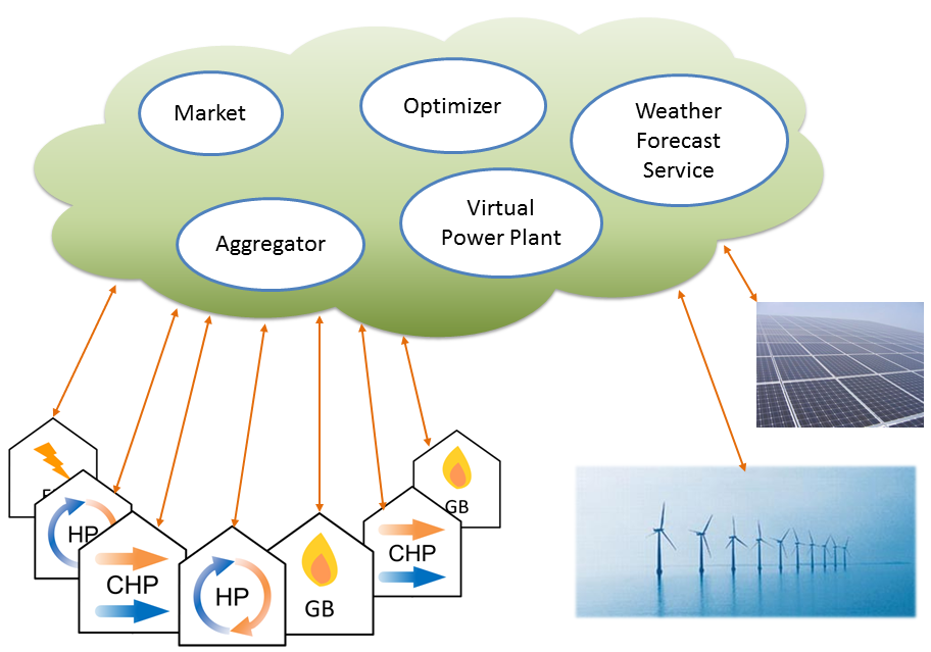
## Solution

The use of [FIWARE generic enablers](https://www.fiware.org/developers/) has been shortening and streamlining the development of solutions to support “microgrid services” and enabling “citizens energy communities” already for time, in particular supporting the development of local energy marketplaces. These generic enablers help implementing monitoring systems for “self-sufficient” energy districts (i.e. implementing concepts of energy autarchy) or integrating systems of different service providers with applications to consume new market services.

One example of such approach is the [**FINESCE**](http://www.finesce.eu/) Marketplace, which involves a consortium of dozens of energy companies, R&D centres, and universities from all over Europe, with the aim of providing an electronic marketplace for electricity closer to new categories of “prosumers”.

In the course of the project, FINESCE partners have recognised that a **common, Open-Source platform architecture** is better suited to the Smart Energy domain than a siloed approach and that FIWARE can act as the basis for such platform.

The revolution represented by the use of an open Smart Energy Platform concept represents the implementation of a new concept of a utilities marketplace driven by services (named by the project consortium “**Utility 4.0”**, as it may represent a new, revolutionary “bottom up” approach and a development of the traditional business model of utilities).

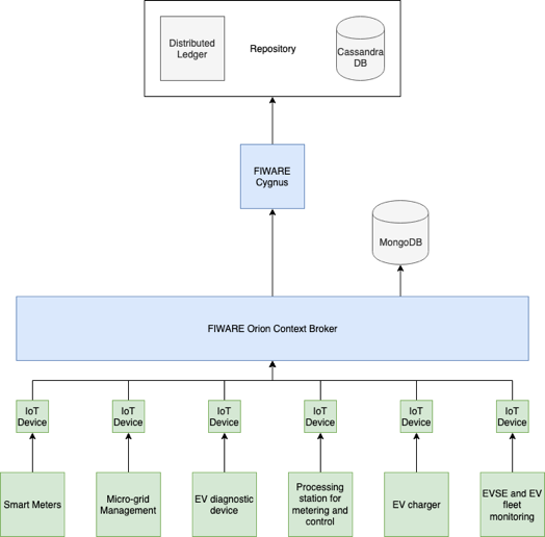
[](https://www.fiware.org/wp-content/uploads/ImpactStories_EnergyMSC_Fig-1.png)

*Figure 1 . Solution*

The active and successful participation of startups to the project has been an effective way of demonstrating how this “service”-based open marketplace, focused on the possibility to integrate new services provided by new categories of prosumers, can better represent the future of a world of energy supply dominated by renewables and clean energy.

## How it works

Continuing our excursus on how FIWARE technologies can enable citizens’ energy communities by leveraging blockchain technology, the **[eDREAM](https://edream-h2020.eu/" \t "_blank)** (Innovative Demand Response Technologies for a decentralized energy ecosystem) Project (funded under the Horizon2020) is developing and validating a secure, decentralised, tamper-proof and transparent pool for energy data. Due to the rapid growth of Distributed Energy Sources (DRES) across the globe, the need for a visionary, decentralised approach towards clean energy production and distribution has been addressed by leveraging on the Context Broker at the centre of these processes.

[](https://www.fiware.org/wp-content/uploads/ImpactStories_EnergyMSC_Fig-2.png)

*Figure 2 . Architecture*

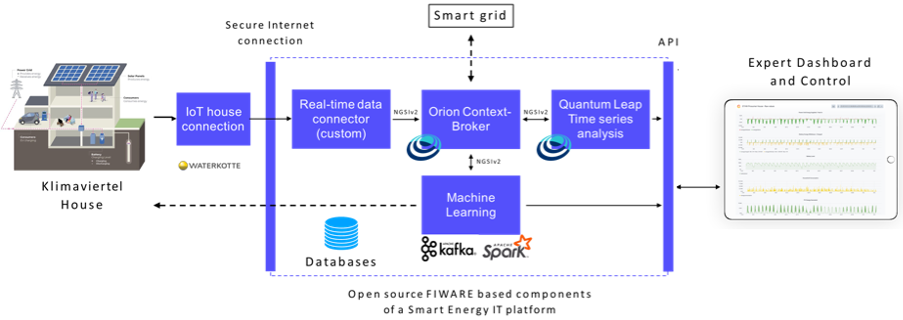
Thanks to a specific component called the **smart-meter aggregator**, energy data are transferred via the Context Broker to retrieve, translate and store the data into “**BigchainDB**” (blockchain database), injecting near real-time data into the big data layer used to develop the clustering techniques for load, profiling and customers segmentation and to achieve the following three objectives:

* near real-time control and supervision of grid management and operation at all voltage levels;
* turn traditional, centralised market approaches and smart grid operations into novel, decentralised and community-driven energy systems;
* introduce new marketplace models, adopting distributed ledger technologies for near-real-time closed loop ecosystems, making optimal use of blockchain-based Demand-Response processes.

Thanks to the use of FIWARE Open Source technologies, the eDream project has allowed developers to rely on effective “community” support and reduced the total cost of the technological infrastructure.

Another important example of how the Context Broker can quickly enable the deployment of an infrastructure to support a local energy community is the [KlimaViertel](https://www.stadtwerke-herne.de/unternehmen/das-klimaviertel" \t "_blank) in the German **city of Herne in North Rhine-Westphalia (NRW)**. This project, associated to an initiative funded by the European Union and the European Regional Development fund for NRW, is the result of a cooperation between **Stadtwerke Herne (the local utility of the city of Herne)**, [Accelogress](https://www.accelogress.com/" \t "_blank)and [Waterkotte](https://www.waterkotte.eu/" \t "_blank) and had the ambition to connect and monitor the energy production and consumption in a real-life living lab composed by a number of energy autarchic buildings.

The architecture, designed to integrate the various data sources,  leverages  FIWARE Context Broker as a data aggregator. In addition, Grafana (a multi-platform open source analytics and interactive visualization web application) has been used to develop a series of dashboards to present a real-time overview of the actual energy production and consumption.

[](https://www.fiware.org/wp-content/uploads/ImpactStories_EnergyMSC_Fig-3.png)

*Figure 3 . Functionalities*

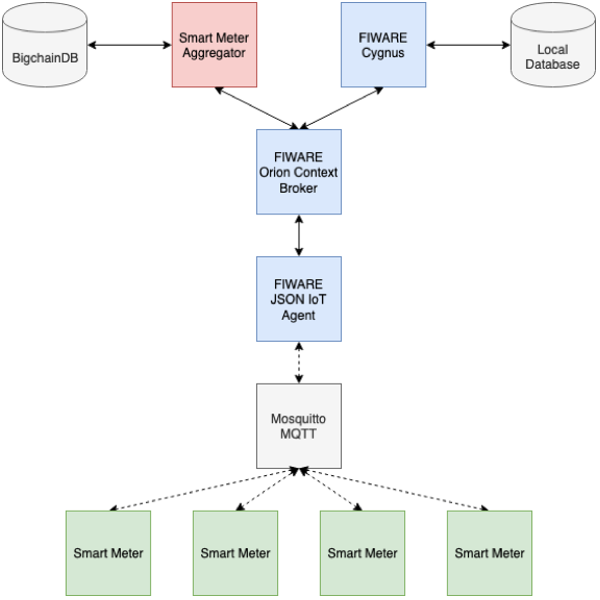
## Benefits & Impact

The use of the open API-based architecture provided by FIWARE has been providing to developers and industry players a number of specific benefits (both in terms of technology advancements and contribution to the exploration of possible new business models), but it has also helped the FIWARE Open Source community to **better tailor** some of the “Generic Enablers” to support the development of energy applications.

In particular, the use of **FIWARE technologies** has been instrumental in assessing the real**impact of “System of Systems”**approaches applied to**local energy communities and**[**marketplaces**](https://www.fiware.org/2020/11/12/fiware-sap-white-paper-is-out/).

In the context of the European project [SOFIE](https://www.sofie-iot.eu/) (which stands for “Secure, Open Federation For Internet Everywhere), a **marketplace, where consumers can search for “clean energy” offers,**as well as **monetise their “spare” electricity,**it has been developed an integration data model coming from a number of different service providers **building new services** on the basis of the collected data streams.

Coordinated by a consortium of enterprises, research centres and universities, project SOFIE is currently experimenting in particular on ways to use **Electrical Vehicles (EV) to decarbonise cities, turning electrical cars into “mobile rechargeable batteries”** (this approach is generally indicated as “vehicle-to-grid”, or V2G), instead of relying exclusively on EV charging stations, with the ultimate goal of building a trustworthy marketplace that **enables new business models** leveraging on data exchange between different parties by setting the security framework for connecting and enabling automatic discoveries of other participating devices. The Context Broker is once again at the centre of the action as it integrates and enhances the interoperability of the different systems involved, creating a standardised platform (“System-OfSystems” approach) to manage, process, and analyse the energy data received.

[](https://www.fiware.org/wp-content/uploads/ImpactStories_EnergyMSC_Fig-4.png)

*Figure 4 . Architecture*

The architecture and the supported use cases of SOFIE are being validated in a pilot involving an **Italian Distribution Systems Operator (DSO – ASM Terni) and a Fleet Manager (Emotion)**, with a user base of circa 100 end users.

## Added value through FIWARE

The use of various FIWARE Generic Enablers to build the foundation for the innovative solutions mentioned above are a clear demonstration of the flexibility and powerful scalability of FIWARE Open Source technologies in addressing the requirements of this **new category of prosumers** operating in the energy marketplace.

The applications needed to implement the revolution of the energy sector based on renewable energy sources and new energy generation technologies, require at their core a powerful “engine” like the FIWARE Context Broker that aggregates data from a variety of sources and packages it in a way that allows utilities and energy operators to work with information that’s already filtered, organised, and properly labelled.

Several research and innovation initiatives have demonstrated how, leveraging on the FIWARE Context Broker, data from power-plants, energy “smart-meters”, weather forecasts, and other sources, can be aggregated and produce the most up-to-date “intel” of the context they are operating into, to energy producers and consumers.

The FIWARE Context Broker offers a range of advanced functionalities, such as updates, queries, registrations, subscriptions and notifications, helping users retrieve, discover and access information as needed, leveraging it to implement new services and supporting innovative business models.

In addition to this, the FIWARE Context Broker, through its “publish and subscribe” system, allows to publish data from different categories of IOT devices and send notifications when any of the monitored conditions change, allowing, at any given time, operators to act quickly according to their fluctuating energy demands, drastically improving the quality of the services provided to the marketplace.

## Next Steps

With the support of the [FIWARE Smart Energy Mission Support Committee](https://www.fiware.org/foundation/mission-support-committees/) (MSC ENERGY), the FIWARE **[iHub IDEASFORUM](https://ideasforum.org/?lang=en" \t "_blank)** (through its Smart Energy Competence Centre), is coordinating the drafting of a white paper channelling various project scenarios for Local Energy Communities, looking at different geographical (villages, small cities, city districts, etc.) and business dimensions (e.g. industry/manufacturing, retail, sport or educations districts), in order to capture the interest of regional and national authorities and build momentum in the request to develop projects to empower this new category of players in the energy market

In addition to this, to assure a tighter integration with national and regional smart grids, the members of the **MSC Smart Energy**of the FIWARE Foundation and the IDEASFORUM Smart Energy Competence Centre will continue to work to establishing a stronger relationship with the working groups at the [Linux Foundation for Energy Program (LFE)](https://www.lfenergy.org/), with particular focus on data architectures, cybersecurity and the use of NGSI-LD to create the integration between different energy systems.

This work is fundamental to enrich the set of tools available for developers in the FIWARE Community in order to assure that new applications developed support “Local Energy Communities”. It also adds to a security by design DNA able to keep them safe through state-of-the-art-security features and principles.

Next steps will be led by the MSC in Smart energy include :

* standardization of Smart Energy services based on FIWARE component;
* evaluation of SAREF data models;
* consolidation of the the relationship between DG energy and Connecting Europe Facility ( [Success Story](https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/2020/10/19/How+Context+Broker+is+helping+Europe+in+its+quest+for+smarter+energy+solutions)was published by Connecting Europe Facility in  November 2020);
* continuous support to the FIWARE Community in the developing solutions to implement new concept of “local energy communities”;
* definition of the FIWARE Architecture for new Smart Energy Marketplace
* to link Smart Energy with e-mobility.
* monitoring of the evolution of Big Data and IoT.

Last, but not least, Smart Energy is a**cost-effective,** **sustainable**and **secure system** in which renewable energy **production**, **infrastructure** and **consumption** are integrated and coordinated through energy services, active users and enabling technologies.

Initiatives and projects that will focus on these specific concepts are following up with FIWARE’s MSC.

<https://docs.fluentd.org/container-deployment/docker-compose>

1. DOCKER LOG: it is possible to access the container docker logs only from the outside and of course, it is possible to move them in a container. However they seem to be empty.

If you do not specify a logging driver, the default is json-file. To find the current default logging driver for the Docker daemon, run docker info and search for Logging Driver. You can use the following command on Linux, macOS, or PowerShell on Windows:

$ docker info --format '{{.LoggingDriver}}'

Or

docker inspect -f '{{.HostConfig.LogConfig.Type}}' <CONTAINER>

json-file

<https://docs.docker.com/config/containers/logging/dual-logging/>

<https://docs.docker.com/config/containers/logging/local/>

1. LINUX LOGS

$ dmesg

is used to examine or control the kernel ring buffer. The default action is to display all messages from the kernel ring buffer

1. HOST LOGS

It is better to access also the logs of the host. To do so it is possible to mount the directory into the container used to manage the logs.

--volume /var/log:/host\_var\_log

Or

docker cp <container\_name\_or\_id>:/var/log/kern.log .

1. FRR

Attempt installing fluentd.

Docker-compose:

router1:

    build:

      context: ./router

      dockerfile: Dockerfile\_router

    #set the network and system priviledges high to make it possible to add static route and set the flag for forwarding to 1. Done for all the containers in this primal phase.

    cap\_add:

      - NET\_ADMIN

    privileged: true

    #links establishes a link to the fluentd service enabling communication with that container

    links:

      - fluentd

    logging:

      - driver: "fluentd"

        options:

          fluentd-address: localhost:24224

          tag: frr.router1

      - driver: "fluentd"

        options:

          fluentd-address: localhost:24224

          tag: container.router1

      - driver: "fluentd"

        options:

          fluentd-address: localhost:24224

          tag: linux.router1

    #put the container in a loop in order to access it with bash

    command:

      - "/bin/sleep"

      - "infinity"

    networks:

      clientSide:

        ipv4\_address: 172.16.1.2

      internalNet:

        ipv4\_address: 172.16.2.2

fluentd:

    image: fluent/fluentd:v1.12-1

    volumes:

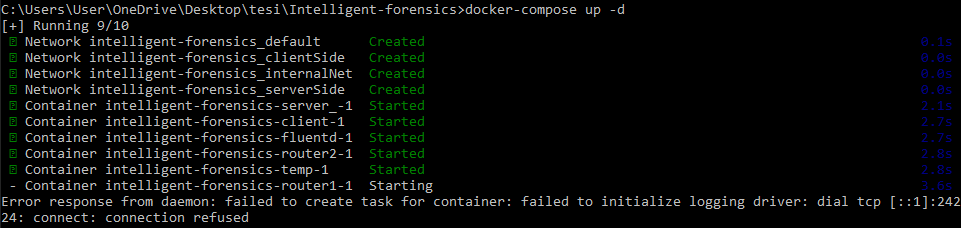
      - ./fluentd:/fluentd/etc

    ports:

      - "24224:24224"

      - "24224:24224/udp"

However it gives this error:



This was due to the fact the router needs to be created after the container fluentd:

Possible solutions:

To use the fluentd driver as the default logging driver, set the log-driver and log-opt keys to appropriate values in the daemon.json file, which is located in /etc/docker/ on Linux hosts or C:\ProgramData\docker\config\daemon.json on Windows Server. For more about +configuring Docker using daemon.json, see +daemon.json.

* The following example sets the log driver to fluentd and sets the fluentd-address option.

{

"log-driver": "fluentd",

"log-opts": {

"fluentd-address": "fluentd:24224"

}

}

Problem to recognize the host. Used a static address

<https://docs.docker.com/config/containers/logging/fluentd/>

* I have facing issue, I have solved using using static ip address.

logging:

driver: fluentd

options:

fluentd-address: 172.24.0.5:24224

* The problem could be the image of fluentd. In the official site it is required to build the image using a dockerfile. This led to the solution

<https://docs.docker.com/config/containers/logging/fluentd/>



Solved with

fluentd-async: 'true'

need to understand why:

chatgpt answ: he addition of the **fluentd-async: 'true'** option likely solved the issue by enabling asynchronous logging when communicating with Fluentd. Let me explain why this might have worked:

Fluentd supports an asynchronous mode of operation where the logging driver doesn't block the main process while sending logs to Fluentd. By default, the Docker logging driver waits for a response from Fluentd before moving on, which can cause delays and potentially lead to timeouts if there are communication issues.

Enabling **fluentd-async: 'true'** instructs the Docker logging driver to operate asynchronously, meaning that it doesn't wait for responses from Fluentd immediately after sending logs. Instead, it continues its main operation, and Fluentd processes logs at its own pace.

This asynchronous behavior can be particularly helpful when dealing with connectivity or responsiveness issues between the Docker container and the Fluentd service. It prevents the Docker container from becoming blocked due to slow communication with Fluentd, and it allows the container to continue its normal operations without waiting for Fluentd's response.

By adding this option, you've essentially allowed the Docker container to send logs to Fluentd in a non-blocking manner, which can help avoid the connection timeout issue you were facing previously.

Remember that asynchronous logging might impact the exact order of log events as they are sent to Fluentd, but it's generally a good trade-off to ensure the stability and responsiveness of your application's containerized environment.

Next step is:

* Undestand better the configuration file of fluentd :

<https://docs.fluentd.org/input>

the basics of the config file: https://docs.fluentd.org/configuration/config-file

* direct the logs of the other container:

Having a lot of trouble finding the logs.

* Maybe syslog can be taken directly by means of fluentd configuration.
* Relating to frr logs need to be activated. Enter vtysh, conf t and run the command: “log file shared-volume/frr/frr.log informational”. There should be already in the folder a file callef frr.log.
* C:\Users\User\AppData\Local\Docker\log\host this is the windows folder for docker logs. Useful ones:
* **com.docker.backend.exe.log** and its rotated files (**com.docker.backend.exe.log.0**, **com.docker.backend.exe.log.1**, etc.) contain logs related to the Docker backend and Docker engine.
* **com.docker.build.exe.stderr.log** and its rotated files contain logs related to Docker container builds.
* **com.docker.dev-envs.exe.stderr.log** and **com.docker.dev-envs.exe.stdout.log** may contain logs related to development environments.
* **com.docker.extensions.exe.log** and its rotated files contain logs related to Docker extensions.
* **com.docker.proxy.exe.log** and its rotated files contain logs related to Docker proxy settings.
* **Docker Desktop.exe.log** contains logs related to Docker Desktop itself.
* **docker.exe.stderr.log** and **docker.exe.stdout.log** contain logs related to Docker CLI commands
* direct different source of log of the same container:

Is it possible to direct the log of the linux os into a mounted volume?

ChatGPT

Yes, it is possible to direct the system logs of a Linux operating system into a mounted volume, including when you're running within a Docker container. System logs are typically stored in the **/var/log/** directory on most Linux distributions.

To achieve this, you can use the technique of creating symbolic links (symlinks) to redirect the log files from their default location to a directory within your mounted volume. Here's how you can do it:

1. **Mount the Volume in Docker:** First, make sure you have properly mounted the volume into your Docker container using the **-v** flag when running the container. For example:

bashCopy code

docker run -v /host/path:/container/path -d your-docker-image

1. **Create Log Directory in the Mounted Volume:** Create a directory within your mounted volume where you want the logs to be stored. For example:

bashCopy code

mkdir /host/path/system\_logs

1. **Create Symbolic Links:** Now, you can create symbolic links from the original log files to the new directory in your mounted volume. For example, if you want to redirect the syslog and kernel logs:

bashCopy code

ln -s /container/path/system\_logs/syslog /var/log/syslog ln -s /container/path/system\_logs/kern.log /var/log/kern.log

Replace **/container/path/** with the actual path to your mounted volume.

1. **Restart Services or Reboot:** Some log files may be actively written to by services or the kernel. To ensure that the changes take effect, you might need to restart services that are responsible for log management (e.g., rsyslog) or simply reboot the system or Docker container.

Keep in mind that redirecting system logs in this manner may require root privileges or superuser access, especially when creating symlinks in system directories like **/var/log/**. Also, be cautious when manipulating log files, as incorrect operations can lead to loss of log data or system instability.

This approach allows you to collect system logs in a centralized location within your mounted volume, making it easier to manage and analyze log data within your Docker container.

How to read the log: inside the container: /fluentd/log/output.log. Here there are 2 files. One of them should be the collection of the logs (right now it is assured just the log of the container fluentd).

To activate fluentd:

In the fluentd container-> fluentd –c fluentd/etc/fluent.conf

Fluentd seems to have problem writing the output. For this reason I set the user as root

In relation to the fluent.conf > position of path (is it host based or container based?)

* In relation to the type of source syslog has its own. To send the logs has been changed the configuration file of syslog. It can be also done for all the other logs (like kern.log). Syslog produces tcp packet but by default fluent takes udp, so I changed it to tcp which is more secure

2023-09-15T12:04:54+00:00 fluent.warn {"host":"172.16.10.10","msg":111,"message":"incoming chunk is broken: host=\"172.16.10.10\" msg=111"}

* The output seemed to cause problem when running fluentd and an output already exists. It cannot obtain a connection with the file. Added append true
* If a container sends a log before the fluentd is active it could raise this error. Has been solved by flushing the buffer every time it starts

2023-09-18 10:44:16 +0000 [info]: #0 fluentd worker is now running worker=0 2023-09-18 10:44:16 +0000 [warn]: #0 failed to flush the buffer. retry\_time=0 next\_retry\_seconds=2023-09-18 10:44:17.857752722 +0000 chunk="6056698d070ca6e1495f40f72c3f572c" error\_class=Errno::ENOENT error="No such file or directory - fstat" 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/buffer/chunk.rb:180:in `copy\_stream' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/buffer/chunk.rb:180:in `block in write\_to' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/buffer/file\_chunk.rb:171:in `open' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/buffer/chunk.rb:179:in `write\_to' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/out\_file.rb:226:in `block in write\_without\_compression' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/out\_file.rb:225:in `open' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/out\_file.rb:225:in `write\_without\_compression' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/out\_file.rb:213:in `call' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/out\_file.rb:213:in `write' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/output.rb:1138:in `try\_flush' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/output.rb:1450:in `flush\_thread\_run' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin/output.rb:462:in `block (2 levels) in start' 2023-09-18 10:44:16 +0000 [warn]: #0 /usr/lib/ruby/gems/2.7.0/gems/fluentd-1.12.4/lib/fluent/plugin\_helper/thread.rb:78:in `block in thread\_create' 2023-09-18 10:44:17 +0000 [warn]: #0 failed to flush the buffer. retry\_time=1 next\_retry\_seconds=2023-09-18 10:44:18 14677335253615061658731/17592186044416000000000 +0000 chunk="6056698d070ca6e1495f40f72c3f572c" error\_class=Errno::ENOENT error="No such file or directory - fstat" 2023-09-18 10:44:17 +0000 [warn]: #0 suppressed same stacktrace 2023-09-18 10:44:18 +0000 [warn]: #0 failed to flush the buffer. retry\_time=2 next\_retry\_seconds=2023-09-18 10:44:20 835427739601591928703/879609302220800000000 +0000 chunk="6056698d070ca6e1495f40f72c3f572c" error\_class=Errno::ENOENT error="No such file or directory - fstat" 2023-09-18 10:44:18 +0000 [warn]: #0 suppressed same stacktrace 2023-09-18 10:44:20 +0000 [warn]: #0 failed to flush the buffer. retry\_time=3 next\_retry\_seconds=2023-09-18 10:44:24 1429385818803029228609/2199023255552000000000 +0000 chunk="6056698d070ca6e1495f40f72c3f572c" error\_class=Errno::ENOENT error="No such file or directory - fstat"

To activate the syslog use syslog-ng > service syslog-ng start

Since the setup operation where too much I decided to reduce them by creating and entrypoint script. To make it work as entrypoint it needs to be copied inside the image. Since there could be many router this is a good idea until the number of router is restricted, because it’s necessary to save the entrypoint for each router. Ofcourse the entrypoints are similar but but different since it change the network address of the neighbours and the name of the file that needs to collect the frr log. To make it possible to execute the script it’s necessary to make the executable with the privileges. Since the scripts have been written in windows the string termination is CR (carriage return) LF (line feed) (\r\n) but the entryscript is required to be in Unix termination (LF), it has been required to convert it and this has been done with notepad++.

It seems that git has changed this when I pushed everything but it has to be checked.

-Docker engine logs: C:\Users\User\AppData\Local\Docker\log\vm -> containerd.log

Useful link: https://docs.docker.com/config/daemon/logs/  
Debugging: daemon.json in C:\Users\User\.docker

-EBPF

-DOCKER SECURITY: https://docs.docker.com/engine/security/

# **Docker security**

There are four major areas to consider when reviewing Docker security:

* the intrinsic security of the kernel and its support for namespaces and cgroups;
* the attack surface of the Docker daemon itself;
* loopholes in the container configuration profile, either by default, or when customized by users.
* the "hardening" security features of the kernel and how they interact with containers.

## **[Kernel namespaces](https://docs.docker.com/engine/security/" \l "kernel-namespaces)**

Docker containers are very similar to LXC containers, and they have similar security features. When you start a container with docker run, behind the scenes Docker creates a set of namespaces and control groups for the container.

**Namespaces provide the first and most straightforward form of isolation**: processes running within a container cannot see, and even less affect, processes running in another container, or in the host system.

**Each container also gets its own network stack**, meaning that a container doesn't get privileged access to the sockets or interfaces of another container. Of course, if the host system is setup accordingly, containers can interact with each other through their respective network interfaces — just like they can interact with external hosts. When you specify public ports for your containers or use [links](https://docs.docker.com/network/links/) then IP traffic is allowed between containers. They can ping each other, send/receive UDP packets, and establish TCP connections, but that can be restricted if necessary. From a network architecture point of view, all containers on a given Docker host are sitting on bridge interfaces. This means that they are just like physical machines connected through a common Ethernet switch; no more, no less.

How mature is the code providing kernel namespaces and private networking? Kernel namespaces were introduced [between kernel version 2.6.15 and 2.6.26open\_in\_new](https://man7.org/linux/man-pages/man7/namespaces.7.html). This means that since July 2008 (date of the 2.6.26 release ), namespace code has been exercised and scrutinized on a large number of production systems. And there is more: the design and inspiration for the namespaces code are even older. Namespaces are actually an effort to reimplement the features of [OpenVZopen\_in\_new](https://en.wikipedia.org/wiki/OpenVZ" \t "_blank) in such a way that they could be merged within the mainstream kernel. And OpenVZ was initially released in 2005, so both the design and the implementation are pretty mature.

## **[Control groups](https://docs.docker.com/engine/security/" \l "control-groups)**

Control Groups are another key component of Linux Containers. They implement resource accounting and limiting. They provide many useful metrics, but they also help ensure that each container gets its fair share of memory, CPU, disk I/O; and, more importantly, that a single container cannot bring the system down by exhausting one of those resources.

So while they do not play a role in preventing one container from accessing or affecting the data and processes of another container, they are essential to fend off some denial-of-service attacks. They are particularly important on multi-tenant platforms, like public and private PaaS, to guarantee a consistent uptime (and performance) even when some applications start to misbehave.

Control Groups have been around for a while as well: the code was started in 2006, and initially merged in kernel 2.6.24.

## **[Docker daemon attack surface](https://docs.docker.com/engine/security/" \l "docker-daemon-attack-surface)**

Running containers (and applications) with Docker implies running the Docker daemon. This daemon requires root privileges unless you opt-in to [Rootless mode](https://docs.docker.com/engine/security/rootless/), and you should therefore be aware of some important details.

First of all, **only trusted users should be allowed to control your Docker daemon**. This is a direct consequence of some powerful Docker features. Specifically, Docker allows you to share a directory between the Docker host and a guest container; and it allows you to do so without limiting the access rights of the container. This means that you can start a container where the /host directory is the / directory on your host; and the container can alter your host filesystem without any restriction. This is similar to how virtualization systems allow filesystem resource sharing. Nothing prevents you from sharing your root filesystem (or even your root block device) with a virtual machine.

This has a strong security implication: for example, if you instrument Docker from a web server to provision containers through an API, you should be even more careful than usual with parameter checking, to make sure that a malicious user cannot pass crafted parameters causing Docker to create arbitrary containers.

For this reason, the REST API endpoint (used by the Docker CLI to communicate with the Docker daemon) changed in Docker 0.5.2, and now uses a UNIX socket instead of a TCP socket bound on 127.0.0.1 (the latter being prone to cross-site request forgery attacks if you happen to run Docker directly on your local machine, outside of a VM). You can then use traditional UNIX permission checks to limit access to the control socket.

You can also expose the REST API over HTTP if you explicitly decide to do so. However, if you do that, be aware of the above mentioned security implications. Note that even if you have a firewall to limit accesses to the REST API endpoint from other hosts in the network, the endpoint can be still accessible from containers, and it can easily result in the privilege escalation. Therefore it is mandatory to secure API endpoints with [HTTPS and certificates](https://docs.docker.com/engine/security/protect-access/). It is also recommended to ensure that it is reachable only from a trusted network or VPN.

You can also use DOCKER\_HOST=ssh://USER@HOST or ssh -L /path/to/docker.sock:/var/run/docker.sock instead if you prefer SSH over TLS.

The daemon is also potentially vulnerable to other inputs, such as image loading from either disk with docker load, or from the network with docker pull. As of Docker 1.3.2, images are now extracted in a chrooted subprocess on Linux/Unix platforms, being the first-step in a wider effort toward privilege separation. As of Docker 1.10.0, all images are stored and accessed by the cryptographic checksums of their contents, limiting the possibility of an attacker causing a collision with an existing image.

Finally, if you run Docker on a server, it is recommended to run exclusively Docker on the server, and move all other services within containers controlled by Docker. Of course, it is fine to keep your favorite admin tools (probably at least an SSH server), as well as existing monitoring/supervision processes, such as NRPE and collectd.

## **[Linux kernel capabilities](https://docs.docker.com/engine/security/" \l "linux-kernel-capabilities)**

By default, Docker starts containers with a restricted set of capabilities. What does that mean?

Capabilities turn the binary "root/non-root" dichotomy into a fine-grained access control system. Processes (like web servers) that just need to bind on a port below 1024 do not need to run as root: they can just be granted the net\_bind\_service capability instead. And there are many other capabilities, for almost all the specific areas where root privileges are usually needed.

This means a lot for container security; let's see why!

Typical servers run several processes as root, including the SSH daemon, cron daemon, logging daemons, kernel modules, network configuration tools, and more. A container is different, because almost all of those tasks are handled by the infrastructure around the container:

* SSH access are typically managed by a single server running on the Docker host;
* cron, when necessary, should run as a user process, dedicated and tailored for the app that needs its scheduling service, rather than as a platform-wide facility;
* log management is also typically handed to Docker, or to third-party services like Loggly or Splunk;
* hardware management is irrelevant, meaning that you never need to run udevd or equivalent daemons within containers;
* network management happens outside of the containers, enforcing separation of concerns as much as possible, meaning that a container should never need to perform ifconfig, route, or ip commands (except when a container is specifically engineered to behave like a router or firewall, of course).

This means that in most cases, containers do not need "real" root privileges at all. And therefore, containers can run with a reduced capability set; meaning that "root" within a container has much less privileges than the real "root". For instance, it is possible to:

* deny all "mount" operations;
* deny access to raw sockets (to prevent packet spoofing);
* deny access to some filesystem operations, like creating new device nodes, changing the owner of files, or altering attributes (including the immutable flag);
* deny module loading;
* and many others.

This means that even if an intruder manages to escalate to root within a container, it is much harder to do serious damage, or to escalate to the host.

This doesn't affect regular web apps, but reduces the vectors of attack by malicious users considerably. By default Docker drops all capabilities except [those neededopen\_in\_new](https://github.com/moby/moby/blob/master/oci/caps/defaults.go#L6-L19), an allowlist instead of a denylist approach. You can see a full list of available capabilities in [Linux manpagesopen\_in\_new](https://man7.org/linux/man-pages/man7/capabilities.7.html).

One primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may provide incomplete isolation, either independently, or when used in combination with kernel vulnerabilities.

Docker supports the addition and removal of capabilities, allowing use of a non-default profile. This may make Docker more secure through capability removal, or less secure through the addition of capabilities. The best practice for users would be to remove all capabilities except those explicitly required for their processes.

## **[Docker Content Trust Signature Verification](https://docs.docker.com/engine/security/" \l "docker-content-trust-signature-verification)**

The Docker Engine can be configured to only run signed images. The Docker Content Trust signature verification feature is built directly into the dockerd binary.  
This is configured in the Dockerd configuration file.

To enable this feature, trustpinning can be configured in daemon.json, whereby only repositories signed with a user-specified root key can be pulled and run.

This feature provides more insight to administrators than previously available with the CLI for enforcing and performing image signature verification.

For more information on configuring Docker Content Trust Signature Verification, go to [Content trust in Docker](https://docs.docker.com/engine/security/trust/).

## **[Other kernel security features](https://docs.docker.com/engine/security/" \l "other-kernel-security-features)**

Capabilities are just one of the many security features provided by modern Linux kernels. It is also possible to leverage existing, well-known systems like TOMOYO, AppArmor, SELinux, GRSEC, etc. with Docker.

While Docker currently only enables capabilities, it doesn't interfere with the other systems. This means that there are many different ways to harden a Docker host. Here are a few examples.

* You can run a kernel with GRSEC and PAX. This adds many safety checks, both at compile-time and run-time; it also defeats many exploits, thanks to techniques like address randomization. It doesn't require Docker-specific configuration, since those security features apply system-wide, independent of containers.
* If your distribution comes with security model templates for Docker containers, you can use them out of the box. For instance, we ship a template that works with AppArmor and Red Hat comes with SELinux policies for Docker. These templates provide an extra safety net (even though it overlaps greatly with capabilities).
* You can define your own policies using your favorite access control mechanism.

Just as you can use third-party tools to augment Docker containers, including special network topologies or shared filesystems, tools exist to harden Docker containers without the need to modify Docker itself.

As of Docker 1.10 User Namespaces are supported directly by the docker daemon. This feature allows for the root user in a container to be mapped to a non uid-0 user outside the container, which can help to mitigate the risks of container breakout. This facility is available but not enabled by default.

Refer to the [daemon command](https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-user-namespace-options) in the command line reference for more information on this feature. Additional information on the implementation of User Namespaces in Docker can be found in [this blog postopen\_in\_new](https://integratedcode.us/2015/10/13/user-namespaces-have-arrived-in-docker/).

## **[Conclusions](https://docs.docker.com/engine/security/" \l "conclusions)**

Docker containers are, by default, quite secure; especially if you run your processes as non-privileged users inside the container.

You can add an extra layer of safety by enabling AppArmor, SELinux, GRSEC, or another appropriate hardening system.

If you think of ways to make docker more secure, we welcome feature requests, pull requests, or comments on the Docker community forums.

<https://www.youtube.com/watch?v=Gp0-7oVOtPw&ab_channel=ThatDevOpsGuy>

<https://github.com/patoi/springboot-fluentd-logging-example/tree/master>

<https://www.logicmonitor.com/blog/how-to-analyze-logs-using-artificial-intelligence>

Different routing algorithms for Fat-tree topologies have been proposed, namely BGP in the data center

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8855699