4

Deployment Guides

This section aims to explain how to deploy software within the i3-MARKET Backplane instances.

4.1 Artifact Deployment Guides

The target audience are the i3-MARKET project developers who are participating in the development and deployment of the i3-MARKET Backplane.

The i3-MARKET operative considers four possible deployment scenarios, categorized into manual and automatized deployments. These scenarios are the following:

- Manual deployment scenario one (MDS1)
- Automatized deployment scenario with Ansible (ADS1)
- Automatized deployment scenario with Ansible and GitHub CI/CD (ADS2)
- Automatized deployment scenario with Docker Compose (ADS3)

Considering an i3-MARKET user role perspective, the main roles involved in the different deployment scenarios are:

- i3M root instance admin
- i3M SW developer
- i3M third-party SW admin
- i3M pilot instance admin

Table 4.1 provides the mapping between the i3-MARKET user roles and the previously listed deployment scenarios.

The following subsections describe in detail each identified deployment scenario.

Table 4.1 Deployment scenarios and i3M user roles mapping.

Deployment scenario/user role	i3M root instance admin	i3M SW developer	i3M third-party SW admin	i3M pilot instance admin
MDS1	✗	✓	✓	✓
ADS1	✓	✗	✓	✓
ADS2	✗	✓	✗	✓
ADS3	✓	✓	✗	✓

4.2 MDS1: Manual Deployment

The manual deployment scenario one (MDS1) is based on accessing the physical resources by establishing an SSH connection. Once the physical resource is accessed, the user proceeds with the SW deployment manually. An overview of MDS1 is provided in Figure 4.1. The actors involved in these scenarios are i3M SW developer and i3M third-party SW admin; see Figure 4.1.

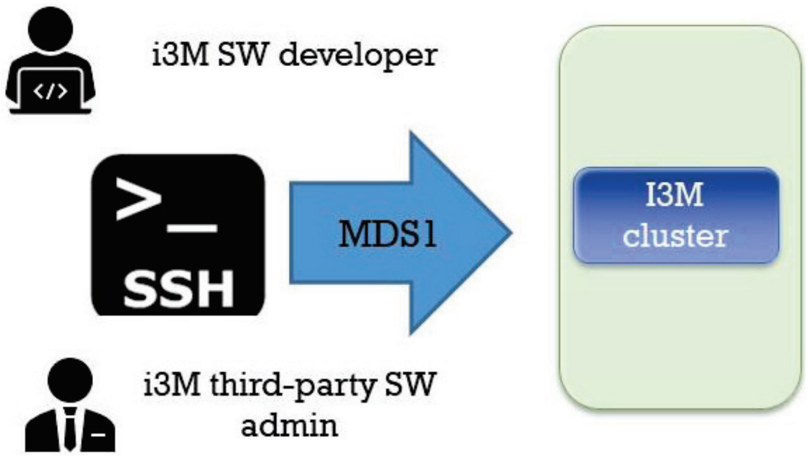


Figure 4.1 MDS1.

4.3 ADS1: Automatized Deployment with Ansible Scenario One

Automated deployment scenario one (ADS1) is based on the provision of a set of Ansible playbooks containing deployment recipes. Playbooks are one of the core features of Ansible and tell Ansible what to execute. They are like a to-do list for Ansible that contains a list of tasks. Playbooks contain the steps that the user wants to execute on a concrete physical resource, and they are run sequentially. From an operative point of view, actors involved in this scenario must cover the following deployment workflow:

1) Create an Ansible template (playbook) with concrete deployment instructions using the physical resources specified in Section 4.3.
2) Start an Ansible job by instantiating the playbook template provided in step 1.

An overview of ADS1 is provided in Figure 4.2. The actors involved in this scenario are i3M IT admin and i3M third-party SW admin.

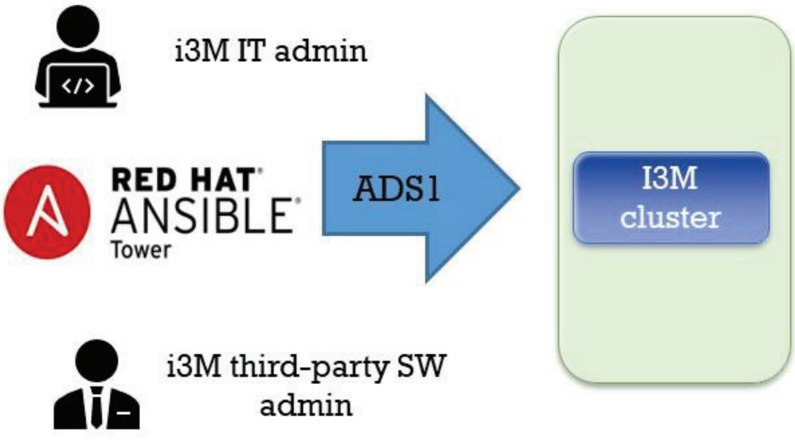
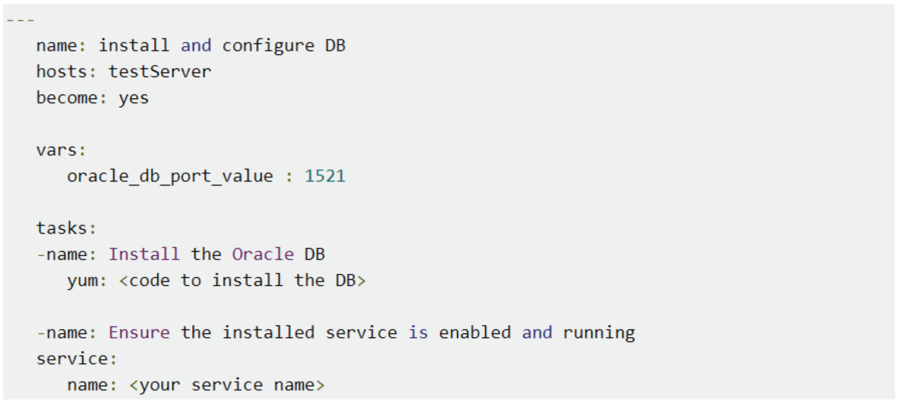


Figure 4.2 ADS1.

Finally, Figure 4.3 contains a playbook example showing the main structure in terms of tags to be included in i3-MARKET playbooks, which are: name, hosts, vars, and tasks.



```
---
  name: install and configure DB
  hosts: testServer
  become: yes

  vars:
     oracle_db_port_value : 1521

  tasks:
  -name: Install the Oracle DB
     yum: <code to install the DB>

  -name: Ensure the installed service is enabled and running
  service:
     name: <your service name>
```

Figure 4.3 Ansible playbook example.

4.4 ADS2: Automated Deployment with Ansible and CI/CD GitHub Pipelines Two

Automatized deployment scenario two (ADS2) is based on the provision of CI/CD pipelines with Ansible and GitHub. The only actor involved in this scenario is i3-MARKET SW developer. The goal to reach in current deployment scenario should be aligned with i3-MARKET DevOps strategy and based on the provision of an Ansible Tower CI/CD architecture.

An overview of ADS2 is provided in Figure 4.4. The only actor involved in this scenario is i3M SW developer.

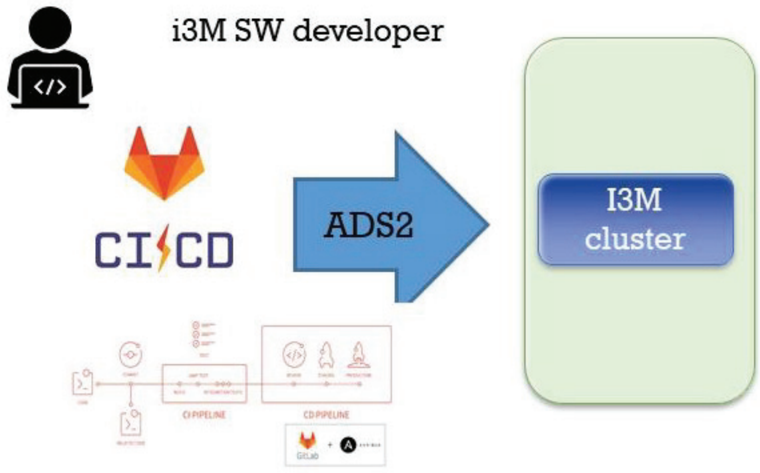


Figure 4.4 ADS2.

The goal to reach in current deployment scenario should be aligned with i3-MARKET DevOps strategy [3] and based on the provision of an Ansible Tower CI/CD architecture.

Considering the approach presented in [4], Figure 4.5 illustrates what we should build to support CI/CD in i3-MARKET using Ansible and GitHub.

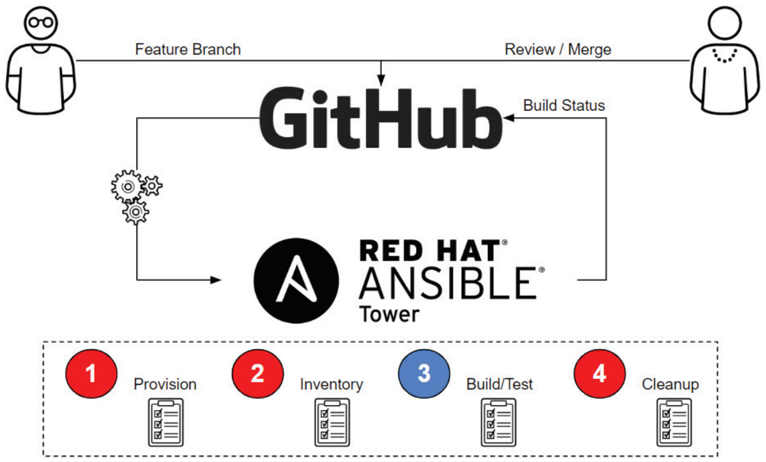


Figure 4.5 i3-MARKET CI/CD with Ansible and GitHub.

As is well known, the main purpose of CI is of course to protect the master branch so that it always compiles. The only way to do this is to check the code in another branch (like a function branch), test that code, review the code, and only merge it with the master once all tests pass. The architecture above achieves exactly that and does so with a very simplified approach that leverages Ansible Tower as our CI engine. For the CD part, only a few additional workflows would be needed to implement artifacts generated by the CI process in dev -> test -> production. Using this architecture, one could use the GitHub versions to store artifacts. GitHub has the ability to trigger a webhook when the latest version is updated, which in turn could trigger an Ansible Tower CD workflow.

4.5 ADS3: Automated Deployment with Docker Compose

The last way of automatizing the deployments on i3-MARKET is by means of Docker Compose[1]. After the last release of the deployment strategy adopted by i3-MARKET of having *N* decentralized i3-MARKET instances + 1 master

[1] https://docs.docker.com/compose/

i3-MARKET instance for centralizing some services, a deployment for supporting the installation of an i3-MARKET instance (a decentralized node) has been created based on Docker Compose. This Docker Compose is used for deploying and managing multiple Docker containers, each of them containing different core and decentralized services developed by i3-MARKET.

This mechanism will allow any marketplace to deploy an i3-MARKET "pilot environment" in order to be part and interact with the i3-MARKET ecosystem. Therefore, ADS3 becomes the most useful deployment strategy for supporting i3-MARKET pilots in the deployment of those i3-MARKET services, which need to be decentralized and installed in the pilot premises. These services are (see more details in Table 2.6): "backplane" (Backplane API component), "tokenizer" + "pricing-manager" (Monetization component), "sdk-ref-impl" (SDK-RI component), "web-ri" + "mongo_web-ri" (Web-RI), "oidc-provider-app" + "oidc-provider-db" (Service-centric authentication component), "vc-service" (User-centric authentication component), semantic-engine + semantic-engine-db (Semantic engine component), data_access (Data access component), auditable-accounting (Auditable accounting component), besu (Blockchain network pilot node + RocksDB instance), cockroachdb-node (Distributed storage component), conflict-resolver-service (Conflict resolution component), rating (Rating component), and "keycloak" (Security server component).

In terms of the Docker Compose file definition, a set of ".env.*component*" files has been created for storing config information relative to the deployment of each of the services contained in the Docker Compose file. For a first idea of the compose file, see below in Table 4.2 the header as reference of it.

Table 4.2 i3m-pilots-docker-compose.yml.

```
version: '3'
services:
  backplane:
    container_name: backplane
    image: "XX.XX.XX.XX:XXXX/backplane:${BACKPLANE_VERSION}"
    restart: unless-stopped
    ports:
      - 3000:3000
    env_file: .env.backplane
    networks:
     - i3m-net
    healthcheck:
      test: "exit 0"
  tokenizer:
```

```
  image: registry.gitlab.com/i3-market/code/wp3/t3.3/nodejs-tokenization-treasury-api:${TOKENIZER_VERSION}
  container_name: tokenizer
  ports:
     - 3001:3001
  env_file: .env.tokenizer
  restart: unless-stopped
  networks:
    - i3m-net
  depends_on:
    besu:
        condition: service_healthy
    postgres:
        condition: service_healthy
sdk-ri:
  image: registry.gitlab.com/i3-market/code/sdk/i3m-sdk-reference-implementation/sdk-ri:${SDKRI_VERSION}
  container_name: sdk-ref-impl
  restart: unless-stopped
  env_file: .env.sdk-ri
  ports:
     - 8181:8080
  networks:
    - i3m-net
  depends_on:
    backplane:
        condition: service_healthy
  command: java -jar /usr/local/jetty/start.jar
  healthcheck:
     test: "exit 0"
web-ri:
  image: registry.gitlab.com/i3-market/code/web-ri/web-ri:${WEB_RI_VERSION}
  container_name: web-ri
  ports:
    - 5300:3000
  env_file: .env.web-ri
  restart: unless-stopped
  networks:
    - i3m-net
  depends_on:
    - mongo_web-ri
  healthcheck:
    test: "exit 0"
mongo_web-ri:
  image: mongo:${MONGO_WEBRI_VERSION}
  container_name: mongo_web-ri
  ports:
```

```
  - 27017:27017
restart: unless-stopped
env_file: .env.web-ri
networks:
  - i3m-net
command: --quiet --setParameter logLevel=0
```

Besides installing the decentralized services by means of the Docker Compose file, the administrator of the pilot infrastructure must install a wallet.

4.6 Tagging Releases Strategy

i3-MARKET has evolved into a complex system where a large number of pieces must interact together for a comprehensive and integrated performance. Therefore, the different versions released by each single component/microservice should be managed and controlled to avoid incompatibilities in the deployments.

A strategy based on tagging and a compatibility matrix has been defined to deal with the release's compatibility.

Thus, every version released by a component is formatted as MAJOR.MINOR.PATCH tag, and each part changes according to the following rules.

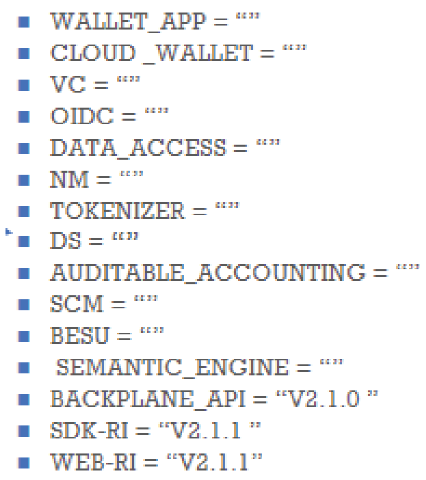
We increment:

- MAJOR when breaking backward compatibility;
- MINOR when adding a new feature which does not break compatibility;
- PATCH when fixing a bug without breaking compatibility.

On the other hand, a matrix including the "microservice name", "microservice version", and a vector of dependencies with other components (and its compatible version) has been defined.

4.7 Deployment Process

At the deployment time, each artifact/service must include in the associated git project a requirements.txt file providing values in the "USES" columns; for example, see the requirement.txt for semantic engine in Figure 4.6.



- WALLET_APP = “”
- CLOUD _WALLET = “”
- VC = “”
- OIDC = “”
- DATA_ACCESS = “”
- NM = “”
- TOKENIZER = “”
- DS = “”
- AUDITABLE_ACCOUNTING = “”
- SCM = “”
- BESU = “”
- SEMANTIC_ENGINE = “”
- BACKPLANE_API = “V2.1.0 ”
- SDK-RI = “V2.1.1 ”
- WEB-RI = “V2.1.1”

Figure 4.6 Requirement.txt for semantic engine repository.

4.7.1 Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define the services and their dependencies in a YAML file and run them with a single command. Docker Compose is especially useful for complex applications that require multiple containers, such as web applications that use a database and a web server.

The Docker Compose file defines the services, networks, and volumes for the application. Each service is defined with its own Docker image, command, environment variables, ports, and volumes. Dependencies between services can be specified using network connections, and shared volumes can be defined to allow data to be shared between containers.

Docker Compose can be used to orchestrate the deployment of containers in a local development environment or in a production environment. It can be used with Docker Swarm to deploy multi-node applications, and it can be integrated with other tools such as Jenkins or GitLab CI/CD for continuous integration and continuous deployment.

Using Docker Compose can provide many benefits for your Docker-based applications, including the following.

1) **Simplified deployment:** Docker Compose makes it easy to deploy multi-container applications with a single command.

2) **Improved scalability:** By defining services and their dependencies, Docker Compose allows you to scale individual components of your application as needed.
3) **Consistent environments:** Docker Compose ensures that all services in your application run in a consistent environment, regardless of the host system.
4) **Easy testing:** Docker Compose makes it easy to spin up test environments with the same configuration as your production environment.
5) **Better collaboration:** By defining the application configuration in a YAML file, Docker Compose makes it easy to share and collaborate on configurations with other team members.

Docker Compose is a powerful tool for defining and deploying multi-container Docker applications. It simplifies the deployment process and allows you to scale your applications with ease, while also ensuring consistency across environments and enabling collaboration between team members.

4.7.2 Technical Requirements

The technical requirements for using Docker Compose include:

1) **Docker Engine:** Docker Compose requires Docker Engine to be installed and running on the host system. Docker Engine is a container runtime that allows you to build, run, and manage Docker containers.
2) **YAML file:** Docker Compose uses a YAML file to define the services, networks, and volumes for the application. The YAML file should be named docker-compose.yml and should be located in the root directory of the application.
3) **Docker images:** Docker Compose uses Docker images to create containers for each service in the application. Docker images can be obtained from Docker Hub, a public registry of Docker images, or from a private registry.
4) **Network connections:** Services in the application may need to communicate with each other over the network. Docker Compose uses Docker networks to create isolated network environments for each application.
5) **Volumes:** Docker Compose allows you to define volumes to share data between containers and persist data beyond the life of a container. Volumes can be defined as local host directories or as named volumes.

6) **Environment variables:** Docker Compose allows you to define environment variables for each service in the application. Environment variables can be used to configure the behaviour of the container at runtime.
7) **Compose CLI:** Docker Compose can be run from the command line using the Compose CLI. The Compose CLI allows you to start, stop, and manage Docker Compose applications.

Docker Compose requires a basic understanding of Docker and containerization concepts, as well as familiarity with YAML syntax. It is recommended to have a solid understanding of Docker Engine before using Docker Compose, as it relies heavily on Docker Engine functionality.

4.7.3 Specification and configurations

The specification and configurations of Docker Compose are defined in a YAML file named "docker-compose.yml". This file consists of several sections that define the services, networks, and volumes for the application.

1) **Version:** The version section specifies the version of the Compose file format to use. The latest version is version 3.9, but earlier versions may be used depending on the Docker Engine version being used.
2) **Services:** The services section defines the individual services that make up the application. Each service is defined as a separate block, with its own image, environment variables, ports, volumes, and other configuration options.
3) **Networks:** The networks section defines the networks that the services use to communicate with each other. By default, Docker Compose creates a network for the application, but additional networks can be defined as needed.
4) **Volumes:** The volumes section defines the volumes that are used by the services to store persistent data. Volumes can be defined as named volumes or as host directories.
5) **Environment variables:** The environment section defines environment variables that are passed to the services. Environment variables can be used to configure the behaviour of the container at runtime.
6) **Deploy:** The deploy section specifies additional deployment options for the services, such as the number of replicas, placement constraints, and resource limits.

7) **External services:** The external_services section is used to define services that are provided by external sources, such as a load balancer or a database that is not part of the Docker Compose application.

These sections can be further configured with various options, such as image pull policies, container restart policies, logging options, and more.

4.7.4 Deployment

This Docker Compose is used for deploying and managing multiple docker containers, each of them containing different core and decentralized services developed by i3-MARKET. Therefore, ADS3 becomes the most useful deployment strategy for supporting i3-MARKET pilots in the deployment of those i3-MARKET services, which need to be decentralized and installed in the pilot premises. It is a practical guide that makes use of the automated deployment based on Docker Compose (ADS3).

The required steps are:

1) **Clone i3-MARKET deployment repository:**

Execute the following command:

```
git clone https://i3m-hackathon-user:userX@github.com/i3-Market-V2-Public-Repository/Support---Deployment-Tools.git
```

2) **Login into i3-MARKET Nexus and GitLab:**

Execute the following two commands:

```
docker login -u i3m-hackathon -p i3m-hackathon X.X.X.X:XXXX


docker login -u i3m-hackathon-user -p userX registry.gitlab.com
```

3) **Execute docker compose:**

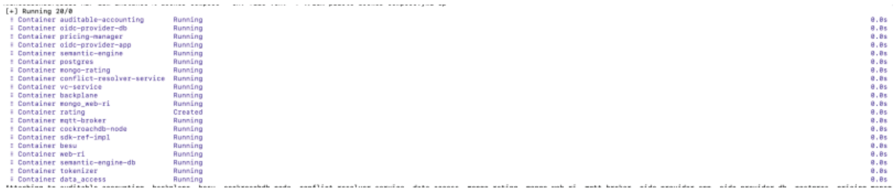
Go to your cloned_dir/docker-compose/i3m-instance and execute the following command:

```
docker-compose --env-file .env -f .\i3m-pilots-docker-compose.yml up
```

To stop services:

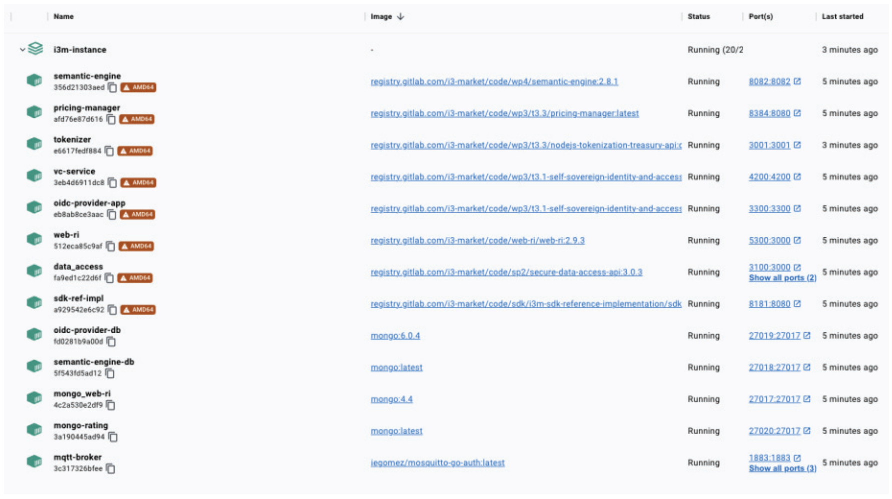
```
docker-compose --env-file .env -f .\i3m-pilots-docker-compose.yml down
```

To verify that all services are up and running:



```
[+] Running 20/0
 ⠿ Container auditable-accounting       Running                                    0.0s
 ⠿ Container oidc-provider-db           Running                                    0.0s
 ⠿ Container pricing-manager            Running                                    0.0s
 ⠿ Container oidc-provider-app          Running                                    0.0s
 ⠿ Container semantic-engine            Running                                    0.0s
 ⠿ Container postgres                   Running                                    0.0s
 ⠿ Container mongo-rating               Running                                    0.0s
 ⠿ Container conflict-resolver-service  Running                                    0.0s
 ⠿ Container vc-service                 Running                                    0.0s
 ⠿ Container backplane                  Running                                    0.0s
 ⠿ Container mongo_web-ri               Running                                    0.0s
 ⠿ Container rating                     Created                                    0.0s
 ⠿ Container mqtt-broker                Running                                    0.0s
 ⠿ Container cockroachdb-node           Running                                    0.0s
 ⠿ Container sdk-ref-impl               Running                                    0.0s
 ⠿ Container besu                       Running                                    0.0s
 ⠿ Container web-ri                     Running                                    0.0s
 ⠿ Container semantic-engine-db         Running                                    0.0s
 ⠿ Container tokenizer                  Running                                    0.0s
 ⠿ Container data_access                Running                                    0.0s
```

If you have Docker Desktop installed, you can view all running containers under the "i3m-instance" as shown in the following image:



Name	Image	Status	Port(s)	Last started
i3m-instance	-	Running (20/2		3 minutes ago
semantic-engine 356d21303aed AMD64	registry.gitlab.com/i3-market/code/wp4/semantic-engine:2.8.1	Running	8082:8082	5 minutes ago
pricing-manager afd76e87d616 AMD64	registry.gitlab.com/i3-market/code/wp3/t3.3/pricing-manager:latest	Running	8384:8080	5 minutes ago
tokenizer e6617fedf884 AMD64	registry.gitlab.com/i3-market/code/wp3/t3.3/nodejs-tokenization-treasury-api:c	Running	3001:3001	3 minutes ago
vc-service 3eb4d6911dc8 AMD64	registry.gitlab.com/i3-market/code/wp3/t3.1-self-sovereign-identity-and-acces:	Running	4200:4200	5 minutes ago
oidc-provider-app eb8ab8ce3aac AMD64	registry.gitlab.com/i3-market/code/wp3/t3.1-self-sovereign-identity-and-acces:	Running	3300:3300	5 minutes ago
web-ri 512eca85c9af AMD64	registry.gitlab.com/i3-market/code/web-ri/web-ri:2.9.3	Running	5300:3000	5 minutes ago
data_access fa9ed1c22d6f AMD64	registry.gitlab.com/i3-market/code/sp2/secure-data-access-api:3.0.3	Running	3100:3000 Show all ports (2)	5 minutes ago
sdk-ref-impl a929542e6c92 AMD64	registry.gitlab.com/i3-market/code/sdk/i3m-sdk-reference-implementation/sdk	Running	8181:8080	5 minutes ago
oidc-provider-db fd0281b9a00d	mongo:6.0.4	Running	27019:27017	5 minutes ago
semantic-engine-db 5f543fd5ad12	mongo:latest	Running	27018:27017	5 minutes ago
mongo_web-ri 4c2a530e2df9	mongo:4.4	Running	27017:27017	5 minutes ago
mongo-rating 3a190445ad94	mongo:latest	Running	27020:27017	5 minutes ago
mqtt-broker 3c317326bfee	iegomez/mosquitto-go-auth:latest	Running	1883:1883 Show all ports (3)	5 minutes ago