Middleware Testing Environment Guide

5G-ERA Middleware

Abstract: This document shows the working procedure given the instructions on how to configure the testing environment for running and testing the 5G-ERA Middleware.

Furthermore, it provides a brief description of the role of the Middleware in the 5G-ERA project and specifies detailed instructions on how to install and configure the testing environment. It also contains the required files for the configuration of the Kubernetes cluster to accommodate the functionality needed by the middleware. This document shows that the API is accessible and simple containers can be deployed via REST request, and how to login to the Middleware, which will be shown in the Testing Scenarios. The document also has all the required information for the AWS Container registry that is needed to instantiate the Middleware. The further deployment of publisher-subscriber will be updated in the next version. Additionally, the deployment of distributed object detection and machine learning will be completed by the end of the next month.

Overall concept:

5G-ERA Middleware is the link between vertical applications managed by ROS and 5G infrastructure managed by OSM. It realises the 5G-ERA intent-based network using cloud-native design.

The Middleware is instantiated in the Edge Machines, and it allows the Robot to request the instantiation of the cloud-native resources that will support the execution of the task.

The main components of the Middleware are listed below:

- Gateway
- Identity service
- Action Planner
- Resource Planner
- Orchestrator

Each instance of the Middleware is connected to the Redis Cluster that shares the information between all the instances of the Middleware running.

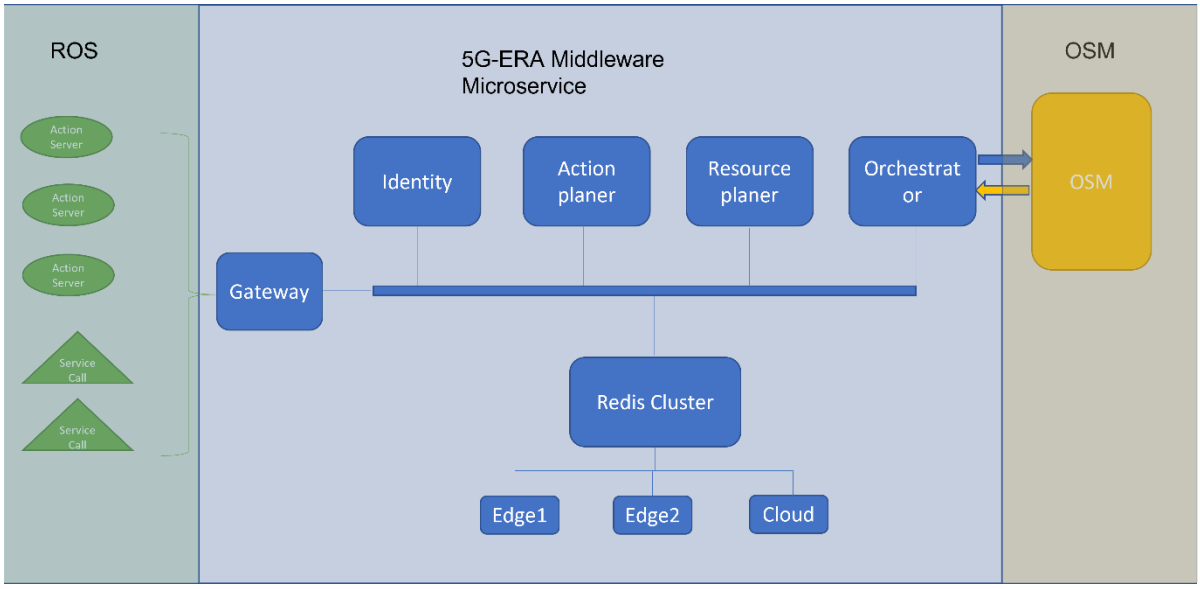

Figure 1 – Middleware structure

The 5G-ERA Middleware is a bridge among three separated layers of networking. It connects the **ROS Vertical layer network** with the **Kubernetes network layer** and the **5G-ERA testbed network**.

As presented in the below Figure, each of the layers has a different responsibility. The Vertical Layer Network is responsible for the management of the ROS applications and their communication with the Robot. The Resource Layer Network is responsible for the Kubernetes network that ROS applications are running on. The last layer is the Resource Enablement Layer Network. It is responsible for the deployment of the Edge Machines with Cloud instances used by the system and the dedicated network slices.

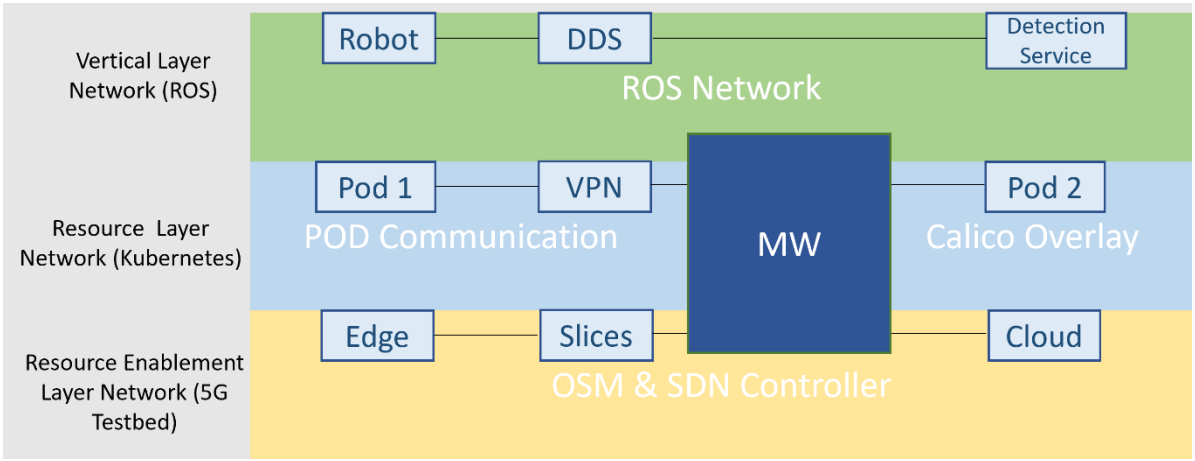

Figure 2 - Middleware between network layers

The 5G-ERA Middleware is responsible for the planning and life cycle management of the ROS services running in the Kubernetes environment. It deploys and monitors the resources requested by the robot for the execution of the specified task. The deployment and the orchestration applies to the resources, planning and lifecycle management of ROS Services. It deploys the resources in the Kubernetes network that runs on the dedicated hardware or network slices provided by the OSM and the SDN controllers.

Current version

The latest version of the 5G-ERA Middleware is available at the GitHub repository under this link. The Middleware is active development so the specification and available functionality will grow. This is the instruction for the pre-release version for D4.1 and D4.2 system integration and internal testing.

As this document presents the installation and configuration of the testing environment, the development environment preparation is described in the official 5G-ERA Middleware repository. For the instructions on how to prepare the environment for the ROS Action-Server development follow instructions in the middleware-action-server repository.

Testing Environment Installation Instruction

This section presents detailed steps needed for the preparation of the testing environment. It explains the installation and configuration process of the Middleware.

The testing environment is based on the **Microk8s**, the minimal Kubernetes installation. It assumes the use of the **Ubuntu 20.04** operating system. For other Linux distributions or other operating systems, the instructions may vary, so check the official guides for installing the respective software.

Recommended hardware:

Processor: 8 logical cores or above.

Ram: 16GB or above.

Storage: 100GB fee space.

Docker Engine installation

The first step is to install Docker Engine. Docker engine is used for the verification of the credentials and pulling the containers from the private AWS registry that Middleware uses.

To install the Docker Engine, refer to Docker's official website on how to install it on the Ubuntu distro.

For easier access to the Docker CLI execute the post-installation steps.

AWS CLI installation

After the installation of the Docker, the AWS CLI must be installed. It is used to authenticate the computer and the Docker client for access to the private container registries.

To install the AWS CLI, follow the official guide on the installation process.

After finishing, the installation process of AWS CLI it is time to configure the AWS CLI with the IAM account that has access to the services needed to run the Middleware.

Use the following command to configure the AWS CLI:

$ aws configure

During the execution of the command supply the AWS Access Key ID and AWS Secret Access Key and the default region to be used. The required container registry is in the **eu-west-1** region. You can also specify the default output format for the CLI. The example is shown in the Figure below.

```
bartosz@5GERA3:/mnt/c/Works/other/orchestrator-> aws configure
AWS Access Key ID [****************FSKH]: AKIAVXYBVN7H3ID4JCZJ
AWS Secret Access Key [****************mgGP]: rE5THSr0qqKQn5lKJr6cvUohu8Vsp3yKEE7hjeFJ
Default region name [eu-west1]: eu-west-1
Default output format [yaml]: yaml
```

Figure 3 - aws configure command example

Authenticate docker with AWS credentials

After the Docker and the AWS CLI are installed, use the following command to authenticate your device against the AWS cloud.

$ aws ecr get-login-password --region eu-west-1 | docker login --username AWS --password-stdin 394603622351.dkr.ecr.eu-west-1.amazonaws.com

After executing this command, you should be able to pull the images from the private AWS registry. It can be verified using the following command:

$ sudo docker pull 394603622351.dkr.ecr.eu-west-1.amazonaws.com/5g-era-redis

This will download the 5g-era-redis image that hosts the Redis cache.

Note!

The token obtained using the *aws ecr get-login-password* will expire after a few hours and re-running the command will be necessary, as Microk8s does not provide support for the credential-helpers. In case when errors occur during the logging into the *ecr*, the deletion of the *.docker/config.json* file usually helps.

Install kubectl

The kubectl is the command-line tool that allows communication and management of the Kubernetes cluster. To install it use the preferred way on the official guide.

Install Microk8s

Microk8s is the minimal Kubernetes installation that can be used on the local computer. It will be used to run the Middleware. Install it with the command:

$ sudo snap install microk8s –classic

$ sudo usermod -a -G microk8s $USER

$ sudo chown -f -R $USER ~/.kube

After the installation is finished copy the configuration file of the Microk8s to the .kube/config file so the kubectl command can communicate with the newly installed cluster.

$ microk8s config > ~/.kube/config

Afterwards, validate the connection to the cluster with the command

$ kubectl get all -n kube-system

Afterwards, the additional modules for the microk8s must be installed:

$ sudo microk8s enable dns multus metallb ingress

It enables the DNS on the cluster as well as the Load Balancer and Multus network card. During the installation, the program will ask for the range of the IP addresses for the Load Balancer. Provide desired range, it can be a default one.

Configure Microk8s access to Docker credentials

After the successful installation of the Microk8s, it must be configured for access to the private AWS Registry. For this, the Microk8s must be stopped.

$ microk8s stop

In the next step create the link to the Docker credentials for the Microk8s:

*$ sudo ln -s ~/.docker/config.json *
/var/snap/microk8s/common/var/lib/kubelet/

Next, start the Microk8s

$ microk8s start

Afterwards, Microk8s is operational and can clone the images from the private repositories.

Microk8s cluster configuration

After the Microk8s is installed and the kubectl command has access to the cluster, it is time to configure the cluster so the middleware can be deployed and function correctly inside of it.

For this purpose, the Service Account with the correct permissions is needed. The Service Account will give the necessary permissions for the Middleware access to the Kubernetes API and to manage the resources as a part of its functionality.

Middleware namespace

As the whole middleware operates in the middleware k8s namespace, it is required to create it before the launch of the service. To do so, use the command:

$ kubectl create namespace middleware

The files required for the execution of the cluster configuration are provided with this instruction. Create the Service Account with the following command:

$ kubectl apply -f orchestrator_service_account.yaml

The next step is to create the Cluster Role, which specifies the permissions needed for the proper functioning of the Middleware. Cluster Role specifies the permissions to get, watch, list create and delete resources in the Middleware namespace. It affects the pods, services, deployments, namespaces, and replica sets in the cluster. To apply for the Cluster Role, use the following command:

$ kubectl apply -f orchestrator_cluster_role.yaml

The last step to configuring the Kubernetes cluster is to bind the Cluster Role to the Service Account. For this the Cluster Role Binding is necessary. To create it, use the following command:

$ kubectl apply -f orchestrator_cluster_role_binding.yaml

Middleware Deployment Instruction

The last step is to prepare the deployment script for the middleware. In the orchestrator_deployment.yaml file there are environment variables that must be set for the correct work of the Orchestrator. The needed variables are:

1. AWS_IMAGE_REGISTRY – contains the address of the registry in which the Middleware images are stored
2. REDIS_HOSTNAME – hostname of the REDIS server that has the Middleware data
3. REDIS_PORT – port on which REDIS operates
4. REDIS_INTERFACE_ADDRESS – address on which the API client for the REDIS in the middleware operates. Defaults to http://redis-interface-api

After all the values are set, the Middleware can be deployed. Start with the deployment of the service for the Orchestrator:

$ kubectl apply –n middleware –f orchestrator_service.yaml

Afterwards, deploy the Orchestrator Deployment:

$ kubectl apply –n middleware –f orchestrator_deployment.yaml

The containers will be downloaded, and the Orchestrator will deploy the rest of the Middleware deployments and services required to function correctly.

Verification of Middleware Deployment

To check and monitor the status of the deployment of the Middleware services use the following command

$ watch -c kubectl get all -n middleware

It will monitor the status of all the services deployed in the middleware namespace. The following objects should be deployed:

5. Orchestrator
6. Redis interface
7. Gateway
8. Task planner
9. Resource planner

Each of these services is represented by the pod, service, deployment and replica set in the Kubernetes environment. With the deployment of the Orchestrator, the other services are deployed automatically. The process of their deployment may take a while depending on the internet connection that machine has. If only the Orchestrator is visible with the status of the pod as Container Creating, it needs additional time to download the application. After the deployment of the Orchestrator, soon the other components should begin their deployment. The result should look like the one presented in Figure 4 - Deployed middleware.

```
pod/orchestrator-api-78688cd8d7-sg7fk       1/1     Running   0             58s
pod/redis-interface-api-86fc7f6b-gxt5t      1/1     Running   0             49s
pod/gateway-5fcfd874f8-tnq9w                1/1     Running   0             49s
pod/task-planner-api-6f495d6b65-xjlnj       1/1     Running   0             49s
pod/resource-planner-api-6977dc8ddc-rbsl5   1/1     Running   0             49s

NAME                              TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)                      AGE
service/orchestrator-api          ClusterIP      10.152.183.184   <none>          80/TCP,433/TCP               76s
service/gateway                   LoadBalancer   10.152.183.51    192.168.1.105   80:32075/TCP,443:30405/TCP   49s
service/redis-interface-api       ClusterIP      10.152.183.129   <none>          80/TCP,443/TCP               49s
service/resource-planner-api      ClusterIP      10.152.183.126   <none>          80/TCP,443/TCP               49s
service/task-planner-api          ClusterIP      10.152.183.98    <none>          80/TCP,443/TCP               49s

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/orchestrator-api           1/1     1            1           58s
deployment.apps/redis-interface-api        1/1     1            1           49s
deployment.apps/gateway                    1/1     1            1           49s
deployment.apps/task-planner-api           1/1     1            1           49s
deployment.apps/resource-planner-api       1/1     1            1           49s

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.apps/orchestrator-api-78688cd8d7         1         1         1       58s
replicaset.apps/redis-interface-api-86fc7f6b        1         1         1       49s
replicaset.apps/gateway-5fcfd874f8                  1         1         1       49s
replicaset.apps/task-planner-api-6f495d6b65         1         1         1       49s
replicaset.apps/resource-planner-api-6977dc8ddc     1         1         1       49s
```

Figure 4 - Deployed middleware

If there are errors during the deployment of the orchestrator, then check if you correctly configured access to the AWS registry.

In case there are any errors during the deployment of the Gateway and Redis interface, check if the firewall does not block access to the Redis server.

After the deployment is complete the gateway should be accessible through the IP address specified in the EXTERNAL-PI column. In case the IP address is not working use the following command to redirect the traffic from the specified port on the *localhost* to the gateway:

$ kubectl port-forward -n middleware service/gateway 5000:80

This command will port forward the traffic from port 5000 to port 80 in the service. The middleware will be now accessible under the following address:

http://localhost:5000

Testing Scenarios

The below scenarios will showcase how to access the example services of the Middleware using its REST API. In the tutorial, the REST calls will be made using the tool Postman. Other methods of calling the REST API can be used. For the Visualization purpose, the postman is used. For this, the Middleware must be running, so make sure that the previous steps have been completed.

Log in to the Middleware

The first step is to log in to the middleware. It is required to be able to access the functionality of the Middleware.

First, we have to create a new request in the Postman. Specify the HTTP Method to be POST and the address to *http://localhost:5047/Login*. In the headers, add the header with the key “Content-Type” and value “application/json”. It is needed to correctly parse the body of the request. Next, paste the following into the raw body of the method:

```
{
    "Id":"ffd8a2e3-cdd6-4ddf-aed9-56bc20371f5a",
    "Password":"5g-era"
}
```

These are credentials that will be used to authenticate the example user in the Middleware. Afterwards, press the “Send” button. In the response, the token should be present as in Figure 5 - Login to the Middleware.

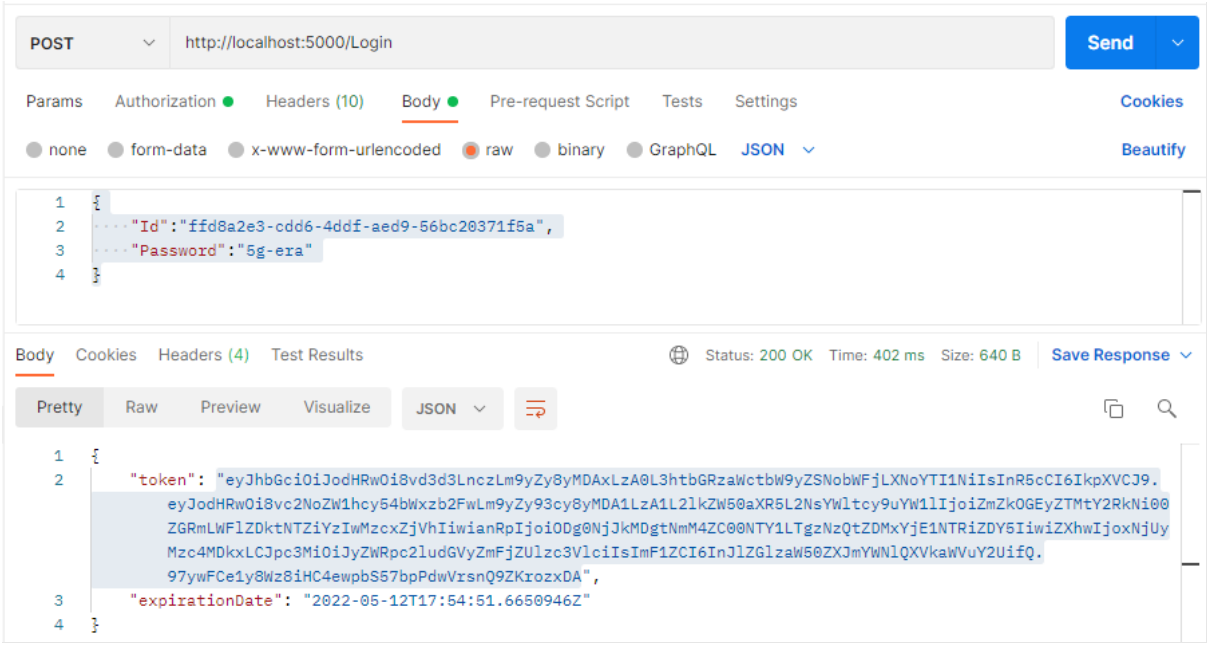

Figure 5 - Login to the Middleware

When the token is retrieved and the status code of the request is “200 OK“. The task has been completed.

Retrieving data from the middleware

The first step is to copy the token retrieved in the previous step. Use it with every following request. To set it properly in Postman, go to the Authorization tab in the request view, specify the Authorization type to “Bearer token” and paste the content of the token to the box on the right-hand side.

As a test, we will retrieve all the actions present in the Middleware. Specify the HTTP Method to “GET” and the address to “localhost:5000/Data/Action”. The body of the request should remain empty. Remember to specify the token for the new request!

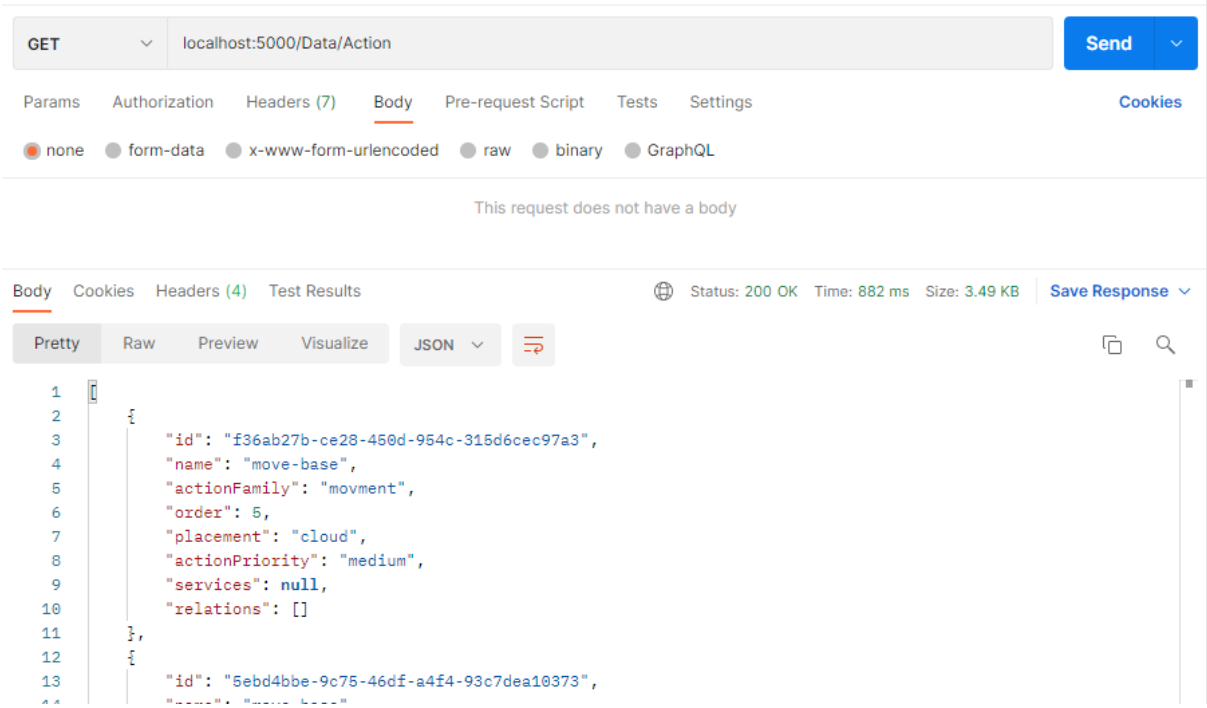

Figure 6 - Get all actions request

The request should look like in the Figure above. After pressing the “Send” button the list of the action definitions should be returned in the bottom panel. Congratulations, now you can access the data in the Middleware above.

The more detailed information on what data can be retrieved from the Middleware see the next chapter

Deployment of the example service

With the access to the Middleware configured, now it is time to deploy the example service. This demonstration will deploy the Nginx web server into the Kubernetes cluster.

Create a new request with the HTTP Method as “POST” that will call the http://localhost:5000/Task/Plan address. Paste the following to the body of the method. This identifier of the task will allow us the deployment of the Nginx service.

```
{
  "TaskId": "11071d4d-d1ae-4e55-8de2-e562c6078277"
}
```

The request definition should look like in Figure 7 - Deployed task instance.

When the “Send” button is pressed, the request is sent to the Middleware. When the Middleware receives it, it identifies the task and the actions needed to execute it. With the actions identified the task information is passed to the task planner that will identify the resources needed for the correct execution of the task. In the last step, the information about the requested task is sent to the Orchestrator, which will deploy the required resources to complete the action. After the deployment is complete, the information will be returned and the whole task definition will be given back to the user, including the address through which the deployed application can be accessed.

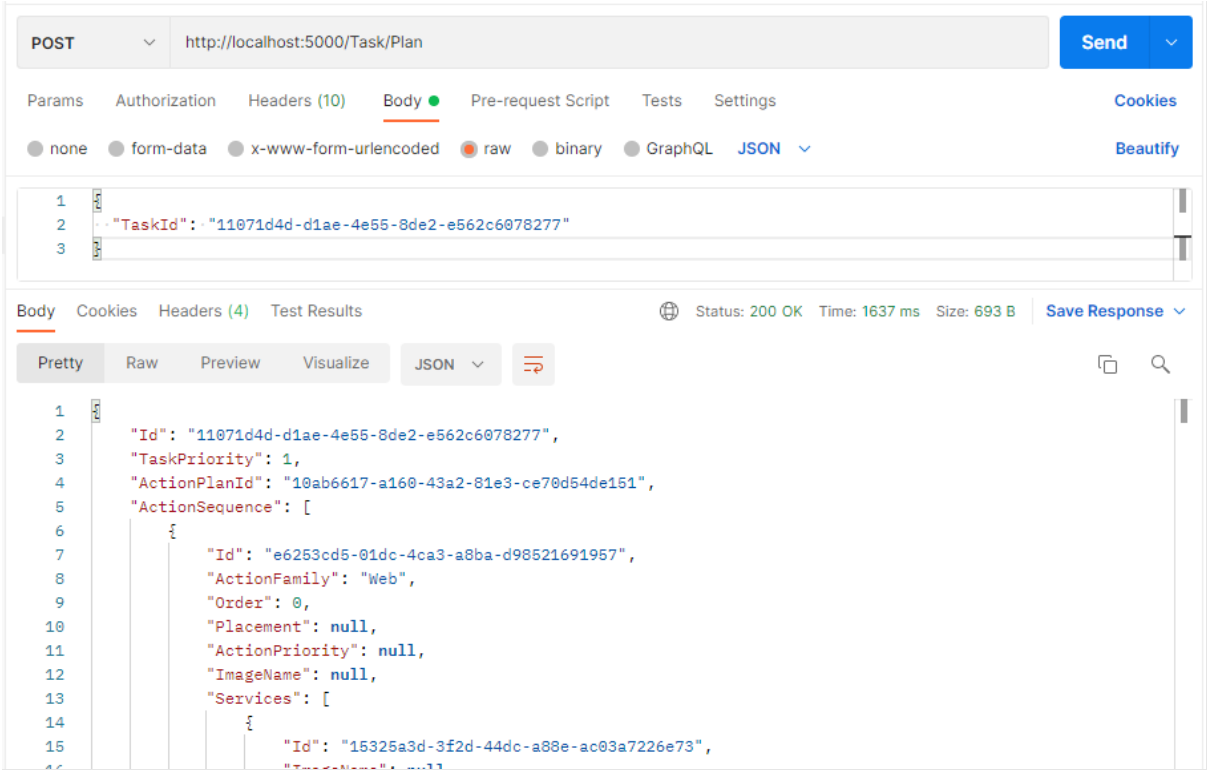

Figure 7 - Deployed task instance

Additionally, the newly deployed Nginx service can be seen among the other Middleware services in the Kubernetes cluster, as shown in the Figure below.

```
Every 2.0s: kubectl get all -n middleware

NAME                                        READY   STATUS    RESTARTS       AGE
pod/curl                                    1/1     Running   13 (89m ago)   28d
pod/orchestrator-api-78688cd8d7-pm8mr       1/1     Running   0              33m
pod/gateway-5fcfd874f8-2kvm9                1/1     Running   0              33m
pod/resource-planner-api-6977dc8ddc-625nh   1/1     Running   0              33m
pod/redis-interface-api-86fc7f6b-s7xlg      1/1     Running   0              33m
pod/task-planner-api-6f495d6b65-2ld2b       1/1     Running   0              33m
pod/nginx-deployment-9456bbbf9-hbbh8        1/1     Running   0              26m

NAME                                TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)                       AGE
service/orchestrator-api            ClusterIP      10.152.183.62    <none>          80/TCP,433/TCP                34m
service/gateway                     LoadBalancer   10.152.183.26    192.168.1.105   80:30302/TCP,443:31669/TCP    33m
service/redis-interface-api         ClusterIP      10.152.183.61    <none>          80/TCP,443/TCP                33m
service/resource-planner-api        ClusterIP      10.152.183.112   <none>          80/TCP,443/TCP                33m
service/task-planner-api            ClusterIP      10.152.183.88    <none>          80/TCP,443/TCP                33m
service/nginx                       LoadBalancer   10.152.183.28    192.168.1.106   80:31166/TCP,433:32441/TCP    26m

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/orchestrator-api           1/1     1            1           33m
deployment.apps/gateway                    1/1     1            1           33m
deployment.apps/resource-planner-api       1/1     1            1           33m
deployment.apps/redis-interface-api        1/1     1            1           33m
deployment.apps/task-planner-api           1/1     1            1           33m
deployment.apps/nginx-deployment           1/1     1            1           26m

NAME                                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/orchestrator-api-78688cd8d7           1         1         1       33m
replicaset.apps/gateway-5fcfd874f8                    1         1         1       33m
replicaset.apps/resource-planner-api-6977dc8ddc       1         1         1       33m
replicaset.apps/redis-interface-api-86fc7f6b          1         1         1       33m
replicaset.apps/task-planner-api-6f495d6b65           1         1         1       33m
replicaset.apps/nginx-deployment-9456bbbf9            1         1         1       26m
```

Figure 8 - Deployed Nginx service

After the service is deployed, it should be accessible from the web browser using the IP address or using the “kubectl port-forward” command. The working Nginx is seen in the Figure below.

Figure 9 - Accessed Nginx from the web

Available endpoints for data retrieval

The 5G-ERA Middleware allows its users to retrieve, insert and update data from/into the Redis storage. More precisely, the API is sending the request from the client to the Redis system and sending the Redis system response back to the client.

The endpoint classes that handle this functionality through HTTP Methods are the controller classes. When a request is made to manipulate the data, such as a request to retrieve data from the database, an HttpGet request will be executed and will return the data to the client.

The RedisInterface API allows full CRUD operations (GET, POST, PUT, PATCH, DELETE) on all data models, which are stored in JSON format, and on the graph.

The CRUD operations which are implemented in the Action, Cloud, ContainerImage, Edge, Instance, Policy, Robot and Task controllers are described in the table below. For the latest up to date information on the structure of the models, please visit the documentation in the GitHub repository.

The first column of the table below represents the HTTP method, that must be used to access the specific endpoint. The “**Endpoint Path**” column specifies the path that has to be used, and the “**Use Case**” column gives a brief explanation of what the endpoint does.

Remember! When calling the endpoints replace names in the ”{}” with the actual values.

Method	Endpoint Path	Use Case
GET	Data/Action	Get all the Action entities.
POST	Data/Action	Add a new Action entity.
GET	Data/Action/{id}	Get an Action entity by id.
PATCH	Data/Action/{id}	Partially update an existing Action entity.
DELETE	Data/Action/{id}	Delete an Action entity for the given id.
POST	Data/Action/AddRelation	Create a new relation

GET	Data/Action/relation/{name}	Get relation by name
GET	Data/Action/relations/{firstName}/{secondName}	Get multiple relations by names
GET	Data/Action/plan	Get all the plans
POST	Data/Action/plan	Create a new plan
GET	Data/Action/plan/{id}	Get plan by id
DELETE	Data/Action/plan/{id}	Delete plan by id
PUT	Data/Action/plan/{id}	Update an existing plan
GET	Data/Cloud	Get all the Cloud entities.
POST	Data/Cloud	Add a new Cloud entity.
GET	Data/Cloud/{id}	Get a Cloud entity by id.
PATCH	Data/Cloud/{id}	Partially update a cloud entity
DELETE	Data/Cloud/{id}	Delete a Cloud entity for the given id.
POST	Data/Cloud/AddRelation	Create a new relation
GET	Data/Cloud/relation/{name}	Get relation by name
GET	Data/Cloud/relations/{firstName}/{secondName}	Get multiple relations by names
GET	Data/ContainerImage	Get all the ContainerImage entities.
POST	Data/ContainerImage	Add a new ContainerImage entity.
GET	Data/ContainerImage/{id}	Get a ContainerImage entity by id.
PATCH	Data/ContainerImage/{id}	Partially update a ContainerImage entity.
DELETE	Data/ContainerImage/{id}	Delete a ContainerImage entity.
POST	Data/ContainerImage/AddRelation	Create a new relation
GET	Data/ContainerImage/relation/{name}	Get relation by name
GET	Data/ContainerImage/relations/{firstName}/{secondName}	Get multiple relations by names
GET	Data/ContainerImage/instance/{id}	Get the ContainerImage associated with an Instance
GET	Data/Edge	Get all the Edge entities.
POST	Data/Edge	Add a new Edge entity.
GET	Data/Edge/{id}	Get an Edge entity by id.
DELETE	Data/Edge/{id}	Delete an Edge entity for the given id.
POST	Data/Edge/AddRelation	Create a new relation
GET	Data/Edge/relation/{name}	Get relation by name
GET	Data/Edge/relations/{firstName}/{secondName}	Get multiple relations by names
GET	Data/Instance	Get all the Instance entities.
POST	Data /Instance	Add a new Instance entity.
GET	Data /Instance/{id}	Get an Instance entity by id.
PATCH	Data /Instance/{id}	Partially update an Instance entity.

DELETE	Data /Instance/{id}	Delete an Instance entity.
POST	Data/Instance/AddRelation	Create a new relation
GET	Data/Instance/relation/{name}	Get relation by name
GET	Data/Instance/relations/{firstName}/{secondName}	Get multiple relations by names
GET	Data/Policy/{id}	Get a Policy entity by id.
PATCH	Data/Policy/{id}	Partially update a Policy entity
GET	Data/Policy	Get all the Policy entities.
GET	Data/Policy/current	Get active policies.
GET	Data/Robot	Get all the Robot entities.
POST	Data/Robot	Add a new Robot entity.
GET	Data/Robot/{id}	Get a Robot entity by id.
PATCH	Data/Robot/{id}	Partially update a Robot entity
DELETE	Data/Robot/{id}	Delete a Robot entity for the given id.
POST	Data/Robot/AddRelation	Create a new relation
GET	Data/Robot/relation/{name}	Get relation by name
GET	Data/Robot/relations/{firstName}/{secondName}	Get multiple relations by names
GET	Data/Task	Get all the Task entities.
POST	Data/Task	Add a new Task entity.
GET	Data/Task/{id}	Get a Task entity by id.
PATCH	Data/Task/{id}	Partially update a Task entity
DELETE	Data/Task/{id}	Delete a Task entity for the given id.
POST	Data/Task/AddRelation	Create a new relation
GET	Data/Task/relation/{name}	Get relation by name
GET	Data/Task/relations/{firstName}/{secondName}	Get multiple relations by names
POST	Data/Task/ImportTask	Imports the complete task definition for the user to incorporate services to be used in the middleware.

Table 1 - Redis CRUD operations

To execute one of the above-mentioned methods, follow the procedure of creating new requests as presented in the Testing Scenarios. Create a new request and specify the HTTP method, address and body if needed.

Interface for ROS Action Server / Action Client

The ROS action server is still a work in progress and will be updated shortly.

Appendix: 5G-ERA NetApp Definition

Existing NetApp/Distributed Solutions	
NetApp for general 5G Testbed: • VNF (Virtual Network Functions) • KNF (Kubernetes Network Functions) managed by the OSM, ready to be deployed as NS (Network Services)	Preferred by network application developers and teleoperators **Pro:** Excellent scalability **Con:** Only for pre-defined procedures (purely based on the information model). cannot handle complicated semantic relationships (life cycle management, task planning etc.). A very high learning curve for the application developer. **Example:** cannot solve MEC switch over the problem on its own, cannot be used by robot developers directly
Distributed Kubernetes Applications • Containerized object detection services (Demonstrated by BUT in Feb 2022)	Preferred by the vertical application developer **Pro:** Can fulfil complicated semantic relationships using cloud-native design **Con:** Require pre-configured Kubernetes with static resources, hence limited scalability Onboarding problems (Hard to be integrated into existing 5G testbed) **Example:** Required networks and Kubernetes clusters must be preconfigured before deployment

5G-ERA NetApps

5G-ERA NetApp solutions	
Distributed Kubernetes Applications + 5G-ERA Middleware + OSM (KNF &VNF)	• Fulfil the complicated semantic relationship required by the vertical and the scalability required by the networking development at the same time • Onboarding will be achieved by middleware and OSM • Semantic, to be realised by cloud-native application and middleware • Reference 5G-ERA NetApp will be demonstrated by an object detection End-to-End demo (To be released in June 2022) • To be further extended and optimised for PPDR, logistics and Manufacturing respectively