mF2C Documentation

Release 1.0 (2018-03-21)

mF2C

Sep 30, 2019

Contents

Welcome to the mF2C documentation!

CHAPTER 1

Introduction

1.1 mF2C

The mF2C project brings together relevant industry and academic players in the cloud arena, aimed at designing an open, secure, decentralized, multi-stakeholder management framework for F2C computing, including novel programming models, privacy and security, data storage techniques, service creation, brokerage solutions, SLA policies, and resource orchestration methods

CHAPTER 2

Developer Guide

This is the developer guide

2.1 Cloud Agent

The mF2C System will be supported by at least one cloud agent, which is managed by the mF2C consortium, and where the non-distributed critical software and data administration capabilities will be located.

2.1.1 Requirements

The mF2C Cloud Agent should have the following mininum hardware requirements:

- 2GB of RAM
- 20GB of disk space
- 2 (v)CPU cores

In terms of software, the cloud agent should sit on a CentOS or Ubuntu VM, and have:

- Docker CE 17.12.0+
- Docker Compose 1.18.0+
- Git

2.2 Installation

2.2.1 Installing the mF2C System

For Linux only

1. download and run the Linux installation script (**docker-compose required**)

```
git clone https://github.com/mF2C/mF2C.git
cd mF2C/docker-compose
sudo ./install.sh
```

To check the status of the mF2C system:

To stop the agent:

Installing a Leader

Prerequisites for the discovery module:

A device with a wireless card that supports "master mode" (i.e. that can act as an access point). You can check whether your card supports master mode by running the following command, looking for the "Supported interface modes". You should find "AP" in the list (i.e. Master mode) :

```
sudo iw list
```

Then, the installation script should be run as follows to start the agent with the "leader" role:

```
sudo ./install.sh -L
```

As far as the discovery module is concerned, the installation script grabs the name of the wireless interface to be used. It then makes sure the discovery container is run with the –cap-add=NET_ADMIN, since network admin capabilities are needed to access the wireless interface of the host machine. It also programmatically associates the physical wireless interface to the newly created container. Note that Discovery is attached to the host network.

Prerequisites for the data management module:

This module is responsible for transparently replicating the necessary data from children to their leader so that the leader has a global view of its cluster. This allows the different components in the leader to forget about data transfers and replicas, and access all the data in the cluster as if it was only in the leader.

To achieve this behaviour, you should modify the *.env* file adding the IP addresses of this leader's children as follows:

```
CHILDREN_DC=host1:port1;host2:port2;...;hostn:portn
```

Installing a regular agent

By default, the installation script will start a normal agent.

```
sudo ./install.sh
```

Prerequisites for the data management module:

This module is responsible for transparently replicating the necessary data from children to their leader so that the leader has a global view of its cluster.

To achieve this behaviour, you should modify the *.env* file adding the IP address of this agent's leader as follows:

```
LEADER_DC=host:port
```

2.2.2 Installing the mF2C Cloud Agent

1. install Docker, by following the instructions at https://docs.docker.com/install/

2. make sure Docker Compose is also installed (https://docs.docker.com/compose/install/)
3. install *git*:

```
# assuming Ubuntu
apt-get update
apt-get install -y git
```

4. (recommended) use the */opt* directory as working directory:

```
cd /opt
```

5. clone the main mF2C repository:

```
git clone https://github.com/mF2C/mF2C
```

6. go in and choose the right distribution - **docker-compose-cloud**

```
cd mF2C/docker-compose-cloud
```

7. using the version 3 Compose file in this folder, deploy the mF2C cloud agent core engine:

```
docker-compose -f docker-compose-core.yml -p mf2c up
```

8. note that step 7. will only deploy the core services for mF2C. To deploy the remaining services, make sure to add the proper credentials to **.env** and run:

```
docker-compose -f docker-compose-components.yml -p mf2c up
```

The full installation might take a few minutes, depending on the user's local Docker images and network connection

Container Monitoring

To add container monitoring simply run:

```
docker run --volume=/:/rootfs:ro \
    --volume=/var/run:/var/run:rw \
    --volume=/sys:/sys:ro \
    --volume=/var/lib/docker/:/var/lib/docker:ro \
    --volume=/dev/disk/:/dev/disk:ro \
    --publish=8080:8080 --detach=true \
    --name=cadvisor google/cadvisor:latest
```

Note that this monitoring page will be publicly available in port 8080.

2.2.3 Updating Components

with docker-compose

If the mF2C agent has been installed with Docker Compose, then to update a single component without having to re-deploy the full stack, simply run:

```
docker-compose -f <yml_file> -p mf2c up -d <service_name>
```

2.2.4 Use of the Certificate Authority server

The Certification Authority (https://github.com/mF2C/certauth) is a JAVA Jersey ReST application deployed on a Tomcat container. It provides 8 different certification CA endpoints. The CAs are independent components that exist independently to the mF2C Agents and fog clusters as well as the CAU middleware. The different CAs provide X.509 certificates to uniquely identify mF2C Agents and infrastructure components within mF2C. The CAU middleware interacts with the unstrusted CA services over HTTPS to request certificates for candidate Agents. Trusted infrastructure components need to obtain a certificate and the associate RSA private key from the appropriate trusted CA service.

The how-to documentation at https://github.com/mF2C/certauth/blob/master/src/main/resources/vanilla-ca-howto.pdf provides detailed information on the list and usage of the certification service endpoints.

Requirements

A host VM with 4GB of memory, 15GB of disk as a minimum and running Centos 7.4 and Docker 18.03. The VM is hosted on the Tiscali Engineering Openstack.

Domain names and DNS

The DNS name is registered and published by Tiscali Engineering. Contact Antonio for assistance.

Installation requirements and procedures

The server is deployed as a Docker container. Refer to the how-to documentation (https://github.com/mF2C/certauth/blob/master/src/main/resources/vanilla-ca-howto.pdf) on the installation requirements and steps to build and deploy the CA application. Please note that you need to have access to the Engineering box and the items listed in the following section to deploy the application.

CA certificates, private keys and Tomcat scripts

These are available from the 'CA credentials' and 'tomcat' folders at https://repository.atosresearch.eu/owncloud/index.php/apps/files/?dir=%2FmF2C%2FWorking%20Folders%2FWP5%20PoC%20integration%2FCA

Cau-client component

Description: This component is a JAVA application. It supports the Agent Discovery and Authentication process. It is triggered by the policy block via TCP-IP to kick off the agent authentication process. It starts by establishing a TLS connection via TCP-IP to the regional CAU to request an Agent certificate. After successfully obtained the signed certificate, it performs a TLS handshake via TCP-IP with the Leader Agent's CAU to exchange keys to secure future communication.

Installation: The component is installed by running the mF2C docker-compose.yml.

Configuration: The cau-client listens on port 46065 for the policy block trigger. This value is fixed for the IT1 demo. You also need to tell cau-client where the regional CAU and the leader agent CAU are located. This is done by amending the cau-client block in the docker-compose.yml file, providing values to the CAU_URL and LCAU_URL environemnt variables. For example:

environment:

- CAU_URL=10.0.0.129:46400
- LCAU_URL=10.0.0.129:46410

2.2.5 Use of the SLA Management component

The SLA Management is a lightweight implementation of an SLA system, inspired by the WS-Agreement standard. It features (i) a REST interface to manage agreements, (ii) a background agreement assessment.

To make use of the SLA Management on IT-1, you must install an agreement in the system. This agreement will be detected by the GUI when creating a service instance and will be associated to it.

An agreement is represented by a simple JSON structure. Below is the default agreement that you should install. This agreement will check that the service operations are executed in less than one second. Modify the constraint to allow different time threshold.

```
{
   "name": "*",
   "details":{
       "id": "2018-000234",
       "type": "agreement",
       "name": "*",
       "provider": { "id": "mf2c", "name": "mF2C Platform" },
       "client": { "id": "a-client", "name": "A client" },
       "creation": "2018-01-16T17:09:45Z",
       "expiration": "2020-01-17T17:09:45Z",
       "guarantees": [
           {
               "name": "*",
               "constraint": "[execution_time] < 1000"
           }
       ]
   }
}
```

To install the agreement, type the following command, where $CIMI_URL is the URL of the CIMI server in the leader agent.

```
curl -X POST -d @agreement.json -H"Content-type:application/json" $CIMI_URL/api/
↪agreement
```

Advance usage

The LifecycleManager is responsible, on a service instance creation, to generate an agreement and to start its assessment. At the moment, the agreement generation is not available. For this reason, you must install an agreement as explained above, which will be utilized when creating services using the GUI. If you plan to have different SLAs for the different services, an agreement must be manually created on CIMI for each service instance that needs to have an SLA. In this case, you must also modify the fields .name and .details.name of the agreement to match the name of an installed service. Install as many agreements as service kinds you want to observe.

Currently, the assessment only is able to evaluate execution_time metrics, which are retrieved from the service-operation-report resource. The Distributed Execution Runtime (DER) stores instances of this resource when completing an operation. Any non-DER service instance can store the appropriate service-operation-report to have its agreement evaluated. For DER service instances, the guarantee name must match the operation names.

The steps to evaluate an agreement for a service instance are:

1. Create an sla-agreement CIMI resource using the excerpt above as template. Add as many guarantees as operations you need to observe, and set the guarantee name to the COMPSs name of the operation (qualified class name '.' method name). Take note of the agreement ID auto generated by CIMI.

2. Start the service instance through the Lifecycle Manager passing the agreement ID as parameter. The Lifecycle Manager also starts the agreement assessment. Alternatively, you can manually update the agreement field of an existing service instance and update the status field to “started” of the corresponding agreement resource.
3. Once the service is started, instances of the sla-violation resource are created if any guarantee term is not fulfilled.

2.2.6 Check QoS provider

Before to check the QoS of a specific service, some previous steps are required.

1. Submit an Agreement:

```
  cat >agreement.json <<EOF
  {
  "name": "AGREEMENT 1",
  "state": "started",
  "details":{
      "id": "agreement",
      "type": "agreement",
      "name": "AGREEMENT 1",
      "provider": { "id": "mf2c", "name": "mF2C Platform" },
      "client": { "id": "c02", "name": "A client" },
      "creation": "2018-01-16T17:09:45.01Z",
      "expiration": "2019-01-17T17:09:45.01Z",
      "guarantees": [
            {
              "name": "TestGuarantee",
              "constraint": "execution_time < 10.0"
            }
            ]
    }
}
  EOF
  curl -XPOST -k https://cimi/api/agreement -d @agreement.json -H "Content-type:␣
↪application/json" -H 'slipstream-authn-info: super ADMIN'
```

2. Submit a Service Instance specifying the <*service-id*> and the <*agreement-id*>:

```
  cat >service-instance.json <<EOF
  {
  "service" : "service/<service-id>",
  "status" : "not-defined",
  "agreement" : "agreement/<agreement-id>",
  "agents" : [ {
    "agent" : {
      "href" : "agent/default-value"
    },
    "allow" : true,
    "ports" : [ 46100, 46101, 46102, 46103 ],
    "status" : "not-defined",
    "agent_param" : "not-defined",
    "url" : "192.168.252.41",
    "container_id" : "-",
    "master_compss" : true,
    "num_cpus" : 7
  } ],
```

(continues on next page)

(continued from previous page)

```
  "user" : "testuser"
}
  EOF
  curl -XPOST -k https://cimi/api/service-instance -d @service-instance.json -H
↪"Content-type: application/json" -H 'slipstream-authn-info: super ADMIN'
```

3. Submit a Service Operation Report specifying the <service-instance-id>:

```
  cat >service-operation-report.json <<EOF
  {
    "serviceInstance": {"href": "service-instance/<service-instance-id>"},
    "operation": "TestGuarantee",
    "execution_time": 50.0
}
  EOF
  curl -XPOST -k https://cimi/api/service-operation-report -d @service-operation-
↪report.json -H "Content-type: application/json" -H 'slipstream-authn-info: super␣
↪ADMIN'
```

Finally, check the QoS of a service instance specifying the id:

```
curl -XGET http://service-manager:46200/api/service-management/qos/<service-instance-
↪id>
```

As a result of the operation, the service instance will be returned.

2.3 Testing

2.3.1 Manual Testing

CIMI

(these instructions have not been tested in Windows)

CIMI will be running over HTTPS (through Traefik). Since this is a development and testing environment, we'll use a special CIMI HTTP header (`slipstream-authn-info:internal ADMIN`) to bypass user authentication and authorization, by impersonating *admin*. Let's also assume that the TLS certificates in-use were self-signed, thus we'll need `-k`. Finally, for creating and modifying resources, the expected payload is always JSON, so we'll need the HTTP header `content-type:application/json`).

For simplicity, let's setup the 4 possible operations in CIMI:

CREATE

```
alias mf2c-curl-post="curl -XPOST -k -H 'slipstream-authn-info:internal ADMIN' -H
↪'content-type:application/json' "
```

READ

```
alias mf2c-curl-get="curl -XGET -k -H 'slipstream-authn-info:internal ADMIN' "
```

UPDATE

```
alias mf2c-curl-put="curl -XPUT -k -H 'slipstream-authn-info:internal ADMIN' -H
↪'content-type:application/json' "
```

DELETE

```
alias mf2c-curl-delete="curl -XDELETE -k -H 'slipstream-authn-info:internal ADMIN' "
```

Let's also assume the test CIMI server is running at *localhost*.

Cloud Entry Point

When CIMI is ready, the *cloud-entry-point* should be available:

```
mf2c-curl-read https://localhost/api/cloud-entry-point
```

Adding a new user

If the SMTP configuration is enabled, you shall receive a user validation email (check the SPAM folder).

```
curl -XPOST -k -H 'content-type:application/json' https://localhost/api/user -d '''
{
    "userTemplate": {
        "href": "user-template/self-registration",
        "password": "testpassword",
        "passwordRepeat" : "testpassword",
        "emailAddress": "your_email@",
        "username": "testuser"
    }
}'''
```

Login

You **must have** validated the user before you can login. To login, simply create a session.

```
curl -XPOST -k -H 'content-type:application/json' https://localhost/api/session --
↪cookie-jar ~/cookies -b ~/cookies -d '''
{
    "sessionTemplate": {
        "href": "session-template/internal",
        "username": "testuser",
        "password": "testpassword"
    }
}'''
```

Get a collection of any resources

To retrieve all the records of a certain resource type, simply do:

```
mf2c-curl-read https://localhost/api/<resourceName>
```

where *resourceName* is something like *user*, *service*, *device*, etc.

Filter a collection of resources

To filter for a specific set of resources, use CIMI's filtering grammar. Example:

```
mf2c-curl-read 'https://cimi/api/<resourceName>?$filter=<AttrName>="<Value>"&$filter=
↪<AttrName2><=<Value2>&$orderby=<AttrName3>:desc'
```

Get a specific resource

To get a specific resource, use its unique ID:

```
mf2c-curl-read https://localhost/api/<resourceName>/<uuid>
```

Create a new service

Example with only required fields:

```
mf2c-curl-post https://localhost/api/service -d '''
{
   "name": "compss-hello-world",
   "exec": "mf2c/compss-test:it2",
   "exec_type": "compss",
   "agent_type": "normal"
}'''
```

Example with all optional fields:

```
mf2c-curl-post https://localhost/api/service -d '''
{
    "name": "compss-hello-world",
    "description": "Hello World Service",
    "exec": "mf2c/compss-test:it2",
    "exec_type": "compss",
    "exec_ports": [8080],
    "agent_type": "normal",
    "num_agents": 2,
    "cpu_arch": "x86-64",
    "os": "linux",
    "memory_min": 1000,
    "storage_min": 100,
    "disk": 100,
    "req_resource": ["Location"],
    "opt_resource": ["SenseHat"]
}'''
```

Create a service instance

```
mf2c-curl-post https://localhost/api/service-instance -d '''
{
   "user": "testuser",
       "device_id": "3dfe332d-dbd6-49c0-9788-56457a6d781b",
```

(continues on next page)

(continued from previous page)

```
        "device_ip": "192.169.1.41",
        "parent_device_id": "11fe332d-dbd6-49c0-9788-56457a6d78cc",
        "parent_device_ip": "192.169.252.42",
        "service": "a5fe332d-dbd6-4ff0-9788-56457a6d7813",
        "agreement": "15fe311d-dbd6-4ff0-9711-56457a6d7819",
        "status": "waiting",
        "service_type": "swarm",
        "agents": [
                {"compss_app_id": "523242342121", "url": "192.168.1.41", "ports":␣
↪[8081], "container_id": "10asd673f", "status": "waiting",
                        "device_id": "3dfe332d-dbd6-49c0-9788-56457a6d781b", "allow":␣
↪true, "master_compss": true, "app_type": "swarm"},
                {"compss_app_id": "", "url": "192.168.1.42", "ports": [8081],
↪"container_id": "99asd673f", "status": "waiting",
                        "device_id": "3dfe332d-d556-49c0-9788-56457a6d7889", "allow":␣
↪true, "master_compss": false, "app_type": "swarm"}
        ]
}'''
```

Create a “sharing-model” record

```
mf2c-curl-post https://localhost/api/sharing-model -d '''
{
  "user_id": "user/testuser2",
  "device_id": "device/c749fcbb-651d-4ae6-877a-125e372398a4",
  "gps_allowed": false,
  "max_cpu_usage": 3,
  "max_memory_usage": 3,
  "max_storage_usage": 3,
  "max_bandwidth_usage": 3,
  "battery_limit": 50
}'''
```

Create a user profile

```
mf2c-curl-post https://localhost/api/user-profile -d '''
{
  "user_id": "user/testuser2",
  "device_id": "device/c749fcbb-651d-4ae6-877a-125e372398a4",
  "service_consumer": true,
  "resource_contributor": true,
  "max_apps": 1
}'''
```

Create an service level agreement

```
mf2c-curl-post https://localhost/api/agreement -d '''
{
    "id": "a02",
    "name": "Agreement 02",
```

(continues on next page)

(continued from previous page)

```
    "state": "stopped",
    "details":{
        "id": "a02",
        "type": "agreement",
        "name": "Agreement 02",
        "provider": { "id": "mf2c", "name": "mF2C Platform" },
        "client": { "id": "c02", "name": "A client" },
        "creation": "2018-01-16T17:09:45.0Z",
        "expiration": "2019-01-17T17:09:45.0Z",
        "guarantees": [
            {
                "name": "TestGuarantee",
                "constraint": "[test_value] < 10"
            }
        ]
    }
}'''
```

Create an SLA violation

```
mf2c-curl-post https://localhost/api/sla-violation -d '''
{
    "guarantee" : "TestGuarantee",
    "datetime" : "2018-04-11T10:39:51.527008088Z",
    "agreement_id" : {"href": "agreement/4e529393-f659-44d6-9c8b-b0589132599b"},
    "constraint": "var1 < 100 and var2 > 100",
    "values": { "var1": 101, "var2": 100 }
}'''
```

Add a new device

```
mf2c-curl-post https://localhost/api/device -d '''
{
"deviceID":
↪"fd97ac4cf865e108c143c57428f742022f38653f1f4c4166938a3154d7b5818967fd27dae6422a2b1da1ceb8dc9d25f358
↪",
"isLeader": false,
"os": "Linux-4.15.0-45-generic-x86_64-with-debian-9.7",
"arch": "x86_64",
"cpuManufacturer": "Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz",
"physicalCores": 4,
"logicalCores": 8,
"cpuClockSpeed": "1.8000 GHz",
"memory": 7873.7734375,
"storage": 195865.0234375,
"agentType": "Fog agent",
"networkingStandards": "['eth0', 'lo']",
"hwloc": "/bin/sh: 1: hwloc-ls: not found\n",
"cpuinfo": "xml info CPU"
}'''
```

Add the device-dynamic info

```
mf2c-curl-post https://localhost/api/device-dynamic -d '''
{
    "device": {"href": "device/f14de9c3-9221-4f51-84bf-b3836bad601a"},
    "ramFree": 3060.19140625,
    "ramFreePercent": 38.9,
    "storageFree": 168181.26171875,
    "storageFreePercent": 90.5,
    "cpuFreePercent": 79.5,
    "powerRemainingStatus": "39.74431818181818",
    "powerRemainingStatusSeconds": "BatteryTime.POWER_TIME_UNLIMITED",
    "powerPlugged": true,
    "ethernetAddress": "[snicaddr(family=<AddressFamily.AF_INET: 2>, address='172.18.
↪0.14', netmask='255.255.0.0', broadcast='172.18.255.255', ptp=None),␣
↪snicaddr(family=<AddressFamily.AF_PACKET: 17>, address='02:42:ac:12:00:0e',␣
↪netmask=None, broadcast='ff:ff:ff:ff:ff:ff', ptp=None)]",
    "wifiAddress": "Empty",
    "ethernetThroughputInfo": ["13178", "8956", "18", "68", "0", "0", "0", "0"],
    "wifiThroughputInfo": ["E", "m", "p", "t", "y"],
    "actuatorInfo": "Please check your actuator connection",
    "sensors": [{"sensorType": "Temperature", "sensorModel": "DHT22",
↪"sensorConnection": "{\"baudRate\": 5600}"}]
}'''
```

Create a fog area

```
mf2c-curl-post https://localhost/api/fog-area -d '''
{
    "leaderDevice": {"href": "device/123refegh"},
    "numDevices": 10,
    "ramTotal": 56789.90,
    "ramMax": 4569.34,
    "ramMin": 1478.34,
    "storageTotal": 120003456798.23456,
    "storageMax": 345678000.23456,
    "storageMin": 3456789.248,
    "avgProcessingCapacityPercent": 88.6,
    "cpuMaxPercent": 98.2,
    "cpuMinPercent": 56.7,
    "avgPhysicalCores": 4,
    "physicalCoresMax": 6,
    "physicalCoresMin":  2,
    "avgLogicalCores" : 4,
    "logicalCoresMax": 6,
    "logicalCoresMin": 2,
    "powerRemainingMax": "Device has unlimited power source",
    "powerRemainingMin": "88.2"
}'''
```

Add the service operation report

```
mf2c-curl-post https://localhost/api/service-operation-report -d '''
{
    "serviceInstance": {"href": "service-instance/asasdasd"},
    "operation": "newMethod",
    "execution_time": 123.32
}'''
```

2.4 Credentials (API)

This section describes how to create new users, sessions, and API keys, all through the CIMI API.

Note: remeber to use *-k* in all API requests below, if you are working with a test deployment without a green server certificate.

2.4.1 Create a user

Once the mF2C system is up and running,

1. create a new user by doing:

```
curl -XPOST -H "Content-type: application/json" \
    https://<server>/api/user -d @addRegularUser.json
```

where *addRegularUser.json* is

```
{
    "userTemplate": {
        "href": "user-template/self-registration",
        "password": "testpassword",
        "passwordRepeat" : "testpassword",
        "emailAddress": "your@email.com",
        "username": "testuser"
    }
}
```

2. check your email for a user validation email (if working with a test deployment of mF2C, check the Spam folder). Once you find the email, copy the validation URL and paste it in the browser, looking like this "*https://<server>/api/callback/a3a9b6d9-5229-455a-b31b-87fa2b950159/execute*"

3. once the user is validated, you can create a session and login:

```
curl -XPOST -H 'content-type: application/json' \
    https://<server>/api/session -d @regularUser.json \
    --cookie-jar ~/cookies -b ~/cookies -sS
```

where *regularUser.json* is

```
{
    "sessionTemplate": {
        "href": "session-template/internal",
        "username": "testuser",
        "password": "testpassword"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

note that ~/cookie expire by default after 1 day.

You are now logged in.

2.4.2 Generate an API key for a user

API keys are a safer way to have robots (scripts) interacting with the API on behalf of a user, since the same user can issue multiple API keys, and every one of them can be revoked without interfering with the original user access.

Basically CIMI distinguishes between internal logins and api_key logins, even though they might be associated with the same user account.

Before using API keys, create the session-template for it (only do it once, and if it doesn't exist yet):

```
curl -XPOST -H content-type:application/json -d '
    {
    "method": "api-key",
    "instance": "api-key",

    "name" : "Login with API Key and Secret",
    "description" : "Authentication with API Key and Secret",
    "group" : "Login with API Key and Secret",

    "key" : "key",
    "secret" : "secret",

    "acl": {
                "owner": {"principal": "ADMIN",
                        "type":      "ROLE"},
                "rules": [{"principal": "ADMIN",
                            "type":      "ROLE",
                            "right":     "ALL"},
                        {"principal": "ANON",
                            "type":      "ROLE",
                            "right":     "VIEW"},
                        {"principal": "USER",
                            "type":      "ROLE",
                            "right":     "VIEW"}]
            }
}' https://<server>/api/session-template -H 'slipstream-authn-info: super ADMIN'
```

Then, to create an API key, do the following:

1. login (like demonstrated in step 3. of the previous section)

```
curl -XPOST -H 'content-type: application/json' \
    https://<server>/api/session -d @regularUser.json \
    --cookie-jar ~/cookies -b ~/cookies -sS
```

2. generate an API key and secret

```
curl -XPOST -H 'content-type: application/json' \
    https://localhost/api/credential -d @generateAPIKey.json \
    --cookie-jar ~/cookies -b ~/cookies -sS
```

where *generateAPIKey.json* is something like

```
{
    "credentialTemplate": {
        "href": "credential-template/generate-api-key",
        "ttl": 0
    }
}
```

3. you'll get a server response similar to

```
{
    "status" : 201,
    "message" : "credential/4f8b8f66-2e15-4570-a14e-f9d3582425ad created",
    "resource-id" : "credential/4f8b8f66-2e15-4570-a14e-f9d3582425ad",
    "secretKey" : "nehrHa.V9Ppzb.vHf4BG.5vxv3j.DzLtqb"
}
```

4. save the "resource-id" and "secretKey"

```
export CIMI_API_KEY=credential/4f8b8f66-2e15-4570-a14e-f9d3582425ad
export CIMI_API_SECRET=nehrHa.V9Ppzb.vHf4BG.5vxv3j.DzLtqb
```

5. create another session login, with regularUserAPIKey.json:

```
cat >regularUserAPIKey.json <<EOF
{
    "sessionTemplate": {
        "href": "session-template/api-key",
        "key": "$CIMI_API_KEY",
        "secret": "$CIMI_API_SECRET"
    }
}
EOF

curl -XPOST -H 'content-type: application/json' \
https://<server>/api/session -d @regularUserAPIKey.json \
--cookie-jar ~/cookies -b ~/cookies -sS
```

And now you are logged in using an API key and secret instead of your internal user credentials.