Maestro

Command Line Interfaces

Developer’s Guide

July 2020

M3DG-15

Version 1.5

Contents

PREFACE

ABOUT THIS GUIDE

The guide describes the non-GUI interfaces and kits (API, SDK, CLI) implemented for the Maestro3 application.

AUDIENCE

The guide is targeted on the developers and architects, working with Maestro3 application with the command line tools.

THE STRUCTURE OF THIS GUIDE

The guide consists of the following sections:

1. Introduction to Maestro3 provides the general overview of the solution, its main capabilities and tools.
2. Maestro3 CLI explains how to install, configure, and use Maestro3 command line inteface.
3. M3 SDK describes the main points of M3 SDK configuration and algorhythms.
4. Support and DevOps Tools describes tools and approaches used for Maestro3 support and DevOps operations.
5. m3admin provides the information about the Maestro3 administrative functionality tool.
6. Lambda API shows how to develop lambda-based plugins and extensions for working with the Maestro3 dynamic UI.

1 INTRODUCTION TO MAESTRO3

Maestro3 application – is a fault-tolerant system for managing distributed hyper-converged virtual infrastructures. The system is based on event-driven architecture leveraging AMQP protocol (Advanced Message Queuing Protocol), powered by RabbitMQ software application for working with message queuing.

1.1 MAESTRO3 MAIN FEATURES AND CAPABILITIES

Maestro3 is a Cloud Management solution which enables effective, unified, and controllable self-service access to hybrid virtual infrastructures, based on both public and private clouds.

The solution provides users with role-based access to both web and native mobile applications (for Android and Apple). It includes a wide range of events audit, optimization, costs, and billing reports to enable high transparency, accountability, and pro-active optimization for your resources in Cloud.

Maestro3 purpose and functionality is based on the three pillars, each allowing to establish a high level of virtual infrastructures provisioning, monitoring, and audit:

- **Orchestration, provisioning, brokerage**: Maestro3 provides a single entry point for creating, reviewing, and managing virtual resources hosted at one or several Cloud providers, both public or private, in a unified way.
- **Costs visibility, audit, reduction**: Maestro3 includes a set of tools enabling costs visibility, review and control in a convenient unified way for all supported cloud providers.
- **Governance, risk management, compliance (GRC):** With access control, inventory, monitoring, automation, and other tools, Maestro3 is able to effectively support your solutions for modern enterprise-level demands and challenges in GRC.

Maestro3 has several interfaces for communicating with different types of users:

- **Maestro3 User Interface** is a web and mobile application providing users with access to Maestro3 capabilities. The main feature of Maestro3 UI is usability and visibility which give any cloud user an easy and comprehensible access to his/her cloud resources regardless of the expertise level and working position of this user.
- **Maestro3 SDK** is a programming interface to enable automation and integration with Maestro3 user capabilities.
- **Maestro3 CLI** is intended to perform basic commands via remote command line by sending server requests without the need to install 3rd party utilities.
- **Maestro3 Admin Interface** – a CLI tool available only to Maestro3 administrative users and allowing Maestro3 configuration (clouds, tenants, users, etc.). The main purpose of the Maestro3 Admin is to provide the highest possible security level for the Maestro3 application.
- **Dynamic inventory** is a tool that allows using Ansible native management tools.
- **Terraform provider** is a tool that allows using Terraform native management tools.

Maestro3 is a complex system, using a big set of components, facing specific needs and tasks of the application.

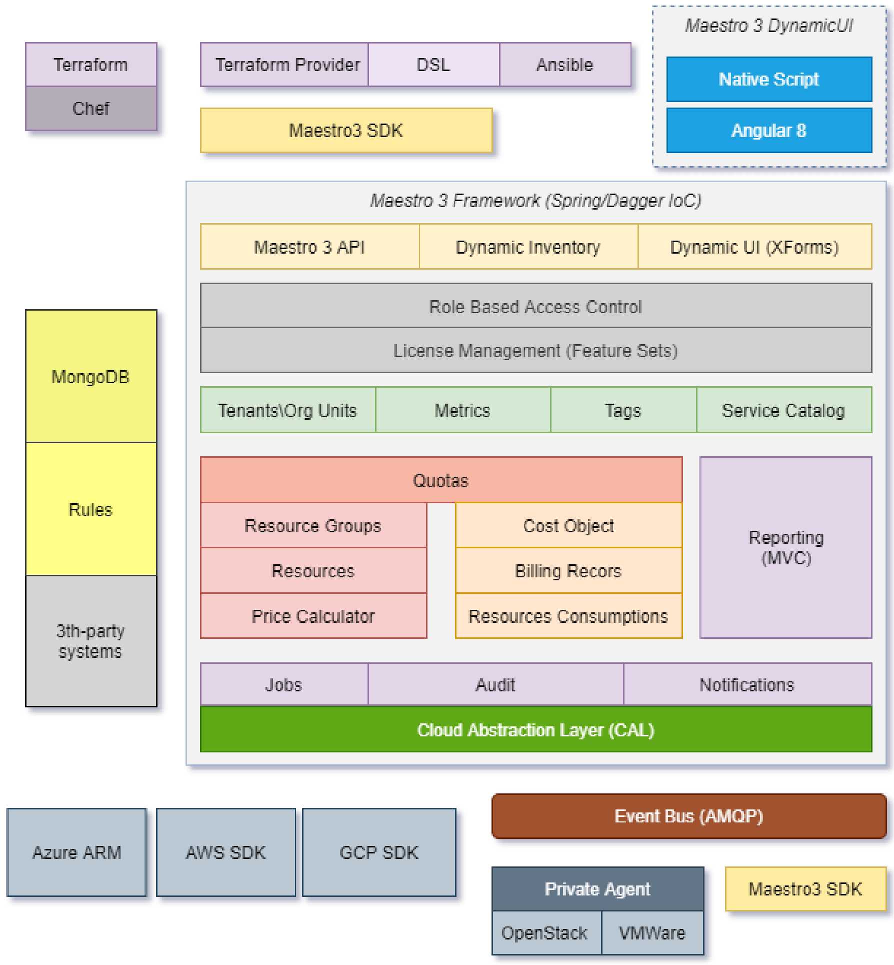


Figure 1 – Maestro3 components scheme

In this document, we focus on the command line tools used for working with such Maestro3 components as Maestro3 API, Maestro3 dynamic UI, Maestro3 SDK, dynamic UI forms, Terraform, Chef, and Ansible.

Complex structure of Maestro3 components requires the multi-level system of development tools. Maestro3 provides these development tools, namely

- **Maestro3 Application Programming Interface (API)** as a part of API-first approach,
- **Maestro SaaS Software Developer's Kit (SDK)** implemented using JSON-RPC as a Remote Procedure Call (RPC) protocol,
- **Maestro3 Command Line Interface (CLI)** as a unified tool to manage your Maestro services and resources. Allows working with the whole Maestro3 functionality without applying the Maestro3 user interface.

1.2 API-FIRST APPROACH

As part of the Maestro3 system, we follow the concept of API First.

Such a concept implies that at the beginning of any application development phase a certain API should be created. Then, on top of this API, we develop additional necessary modules.

In such a way the Maestro3 API was implemented, providing Maestro3 SDK as an additional module on top of it.

1.3 MAESTRO3 SDK

Initially the Maestro3 functionality is supported in the Maestro3 SDK, and only then it is implemented on the UI. Thus, SDK is the keystone for all Maestro3 functionality.

Maestro3 SDK is based on JSON RPC protocol – a remote procedure call protocol encoded in JSON. The protocol is similar to XML-RPC and defines a few data types and commands. JSON-RPC allows for notifications (data sent to the server that does not require a response) and for multiple calls to be sent to the server which may be answered out of order.

Based on this protocol, a simple and flexible working SDK architecture was created.

The user, working with Maestro3 SDK, contains a client. The client contains some managers. Managers contain some methods. The client is a ready-made entity that can do anything. Each of managers is responsible for a certain task, i.e. a certain manager for reporting, a manager for a private cloud, a manager for working with Terraform, etc.

In case a completely new functionality should be added in the system, a new manager is created. This manager is responsible for a specific new function. The methods required for performing this function are included to this manager as well.

The JSON RPC protocol is used in such a case, allowing to rapidly expand the functionality. Maestro3 SDK collects the necessary data and sends it to a server via HTTP or RabbitMQ protocols (both are supported in Maestro3 SDK). The server processes these requests and returns responses.

Security of data transfer is ensured by the AES encryption.

1.4 ANSIBLE AND DYNAMIC INVENTORY

Ansible is an automation system for provisioning and configuring infrastructure, allowing to install software on virtual machines. It can also manage large clusters of VMs on different cloud providers.

Maestro3 provides the possibility to generate a dynamically assembled list of virtual machines via API that is used by Ansible DI via self-service.

Ansible Dynamic inventory is a tool allowing setting up configuration for several instances. It requires a list of instances to be configured. Having the necessary list, Ansible can start working with it to configure the given instances.

Together with other Maestro3 features, the product allows solving task which can be covered by Ansible Tower.

2 MAESTRO3 CLI

Maestro3 Command Line Interface (CLI) is a unified tool to manage your Maestro services and resources. It allows working with multiple cloud resources by means of one simple tool that is easy to use and configure and includes the functionality for creating and using scripts.

2.1 INSTALLATION AND CONFIGURATION

2.1.1 Prerequisites

To install m3 cli assumed, you need Python3.6+, pip and virtualenv installed.

2.1.2 Installation

This is the m3 cli installation command:

```
pip install https://m3-cli-distribution.s3.eu-central-1.amazonaws.com/m3-1.0.0.zip
```

this command allows installing m3 cli from the git source:

```
pip install git+https://git.epam.com/epmc-eoos/m3-cli.git@master
```

Access to epmc-eoos/m3-cli repo is also required.

2.1.3 Configuration

To use m3 cli, you must set these environment variables:

- M3SDK_ACCESS_KEY: Specifies an M3 access key associated with Maestro3 user,
- M3SDK_SECRET_KEY: Specifies the secret key associated with the access key. This is essentially the "password" for the access key.

To get your credentials, you must login to Maestro3 and perform these steps:

1. Switch to **My Account** page.
2. Start the **SDK Keys** wizard.
3. Specify access key name and expiration time for the credentials.
4. Obtain your credentials and export access key to environment variable M3SDK_ACCESS_KEY, and secret key to M3SDK_SECRET_KEY

These environment variables could be used to override default m3 cli values:

- M3SDK_ADDRESS specifies the address of Maestro3 environment. Default value: actual production environment.
- M3SDK_COMMANDS_FILE specifies the path to the file containing the actual group, commands, and parameters definitions. Default value: internal application path.

2.2 USING MAESTRO3 CLI

Maestro3 CLI is an easy-to-use tool innately understandable for users who have any expertise of working with command line tools.

All Maestro3 CLI commands have this structure:

```
m3 <group> <command> [parameters]
```

The parameters are of two kind – **required** and **optional**. Required parameters must be specified in order for the command to be executed. Optional parameters ae necessary to get more specific command output. If you specify a wrong parameter, the command is not executed, and an error is returned.

You can get the specific information about all Maestro3 commands by the **help** command.

The synopsis for all commands shows its parameters and their usage. Required parameters are marked with the star (*) character.

2.3 MAESTRO CLI: INFORMATION FOR DEVELOPERS

The Maestro3 CLI tool is designed to provide dynamic command line interface based on commands configuration declared in **commands_def.json** file.

The sequence diagram is displayed below.

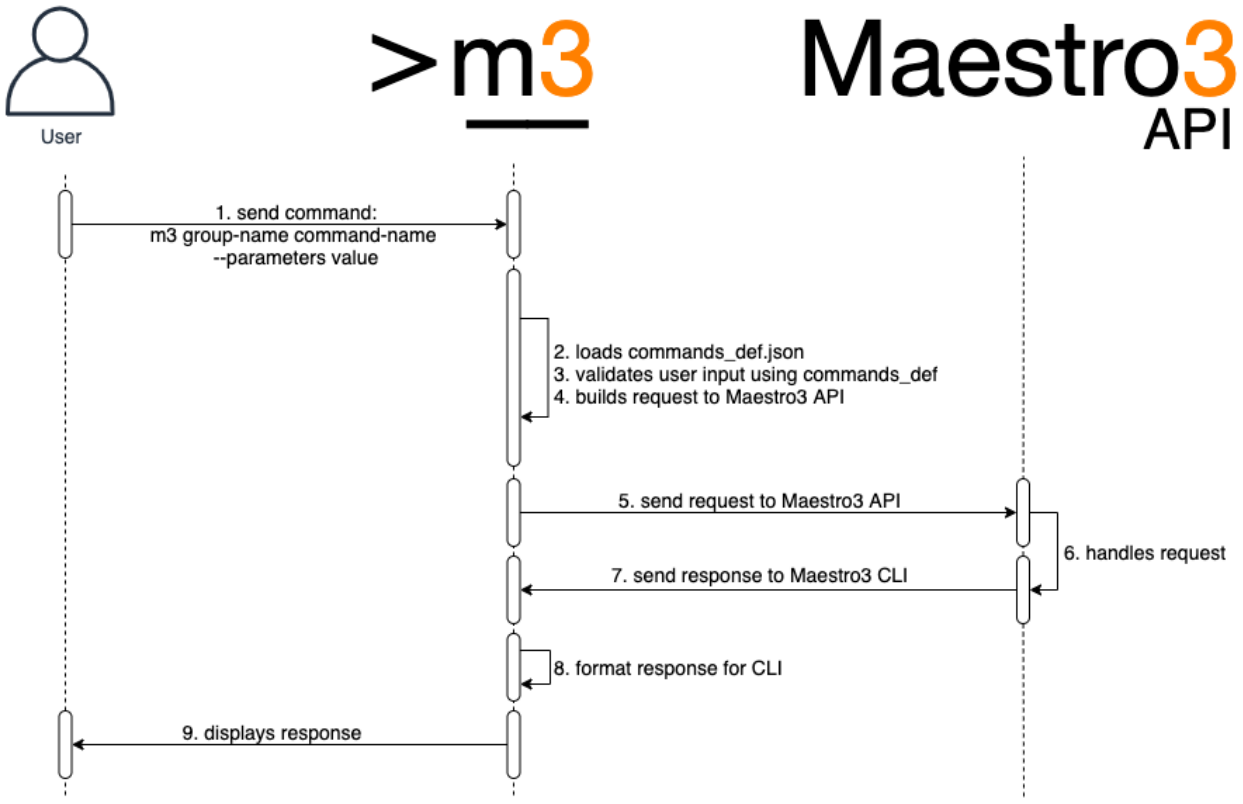

User
>m3
Maestro3 API
1. send command: m3 group-name command-name --parameters value
2. loads commands_def.json
3. validates user input using commands_def
4. builds request to Maestro3 API
5. send request to Maestro3 API
6. handles request
7. send response to Maestro3 CLI
8. format response for CLI
9. displays response

Figure 2 – Maestro3 CLI structure

2.3.1 Commands definition file

The commands definition file (**commands_def.json**) is a file that contains set of groups, commands in the groups and parameters of a commands that could be executed via CLI.

Here is an example of how commands and command groups are decomposed:

1. The root object `groups` encloses all group declarations.
2. `groups-name-1` is a name of the first group defined in a file. It is used in CLI like `m3 groups-name-1`. Each group object contains `help` and `commands` attributes.
3. `command-name-1` is a name of the first command defined in a group. This is used in CLI like `m3 groups-name-1 command-name-1`. Each group object contains `help` and `parameters` attributes.
4. `parameter-1` is a name of the first parameter defined in a command. This is used in CLI like `m3 groups-name-1 command-name-1 --parameter-1 <parameter_value>`. Each parameter object contains `help` and `required` attributes. The `required` property set to true means that the parameter is required.

```
{
  "groups": {
    "group-name-1": {
      "help": "This is an example help for the group \"group-name-1\"",
      "commands": {
        "command-name-1": {
          "help": "This is an example help for a command \"command-name-1\"",
          "parameters": {
            "parameter-1": {
              "help": "Test help for the parameter \"parameter-1\"",
              "required": true
            }
          }
        },
        "command-name-2": {
          "help": "This is an example help for a command \"command-name-2\"",
          "parameters": {
            "parameter-1": {
              "help": "",
              "required": true
            }
          }
        }
      }
    },
    "group-name-2": {
      "help": "This is an example help for the group \"group-name-2\"",
      "commands": {}
    }
  }
}
```

2.3.2 CLI update flow

1. Update **commands_def.json** file with new group/commands/parameters that correspond to the current release.
2. Update the version attribute in `setup.py`.
3. Update **CHANGELOG.md** with changes made to commands configuration.

2.3.3 Delivery flow

1. Change directory to the project root folder. File `setup.py` should be in the same folder as you are
2. Prepare the dist with the command: `python setup.py sdist --formats=zip`.
3. Login to epmc-eoos AWS account.
4. Update the file **m3-<version>.zip** in S3 bucket named `m3-cli-distribution`.

3 M3 SDK

Maestro SaaS Software Developer's Kit (SDK) has been implemented using JSON-RPC as a Remote Procedure Call (RPC) protocol. It uses JSON for serialization and has been chosen due to its simplicity, lightweight and cleanness, among other advantages.

The protocol works by sending a request to a server. The client is typically a software application, intended to call a single method of a remote system. Multiple input parameters can be passed to a remote method as an array or an object, whereas the method itself can return multiple output data as well.

A remote method is invoked by sending a request to a remote service using HTTP or a TCP/IP socket (starting from version 2.0). When using HTTP, the content-type can be defined as application/json.

All transfer types are single objects, serialized using JSON. A request is a call to a specific method provided by a remote system. It must contain three certain properties:

- **method** – a string with the name of the method to be invoked
- **params** – an array of objects to be passed as parameters to the defined method
- **id** – a value of any type, which is used to match the response with the request it is replying to.

The receiver of the request must reply with a valid response to all received requests. A response must contain the properties mentioned below.

- **result** – the data returned by the invoked method. If an error occurred while invoking the method, this value must be null.
- **error** – a specified Error code if there was an error invoking the method, otherwise null.
- **id** – the id of the request it is responding to.

Since there are situations where no response is needed or even desired, notifications were introduced. A notification is similar to a request except for the id, which is not needed because no response will be returned. In this case the id property should be omitted or be null.

There are different ways to implement API for the application. For example, REST-based API means that each unique URL is a representation of some object or resource. A user can get the contents of that object using an **HTTP GET** method, to delete it, then use **POST**, **PUT**, or **DELETE** to modify the object (in practice most of the services use **POST** for this). **REST** is great for public-facing APIs, intended for use by other developers. They can be designed in accordance with common standards, as to not require a lot of preexisting knowledge about the service that is going to be used.

Our main goal was to develop a lightweight, well-specified interface that does not have direct access to data but performs a remote call of Orchestration functionality.

These are advantages of *JSON-RPC*:

- **Unicode** – both *JSON* and *JSON-RPC* support *Unicode* out-of-the-box,
- **Transport-independent** – *JSON-RPC* can be used with any transport socket: TCP/IP, HTTP, HTTPS, etc.,
- **Direct support of Null/None**,
- **Support of named/keyword parameters**,
- **Built-in request-response matching** ("id"-field),
- **Error codes**: ranked and well specified, covering a wide spectrum of possible exceptions.
- **Notifications**.

All procedure calls are strictly atomic and return a well specified, determined result. Clients are not required to know procedure names and the specific order of arguments, because the specifics of JSON-RPC is hidden within our implementation to make the SDK more convenient for use.

3.1 CONFIGURATION

3.1.1 Maven Configuration for Maestro JAVA SDK

Add the following within the <dependencies> section of your POM:

```
    <dependencies>
         <dependency>
            <groupId>com.m3.sdk</groupId>
            <artifactId>m3-sdk</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
```

The artifacts are available via EPAM Maven Repository.

```
<repositories>
        <repository>
          <id>artifactory.epam.com</id>
          <name>artifactory.epam.com-releases</name>
          <url>http://artifactory.epam.com/artifactory/EPM-CIT</url>
        </repository>
    </repositories>
```

3.1.2 Entry Point

The entry point for Maestro3 Java SDK is **M3Sdk**. It can be reached by the following address: **com.m3.sdk**.

3.1.3 Typical Working Scenario

1. Create a client

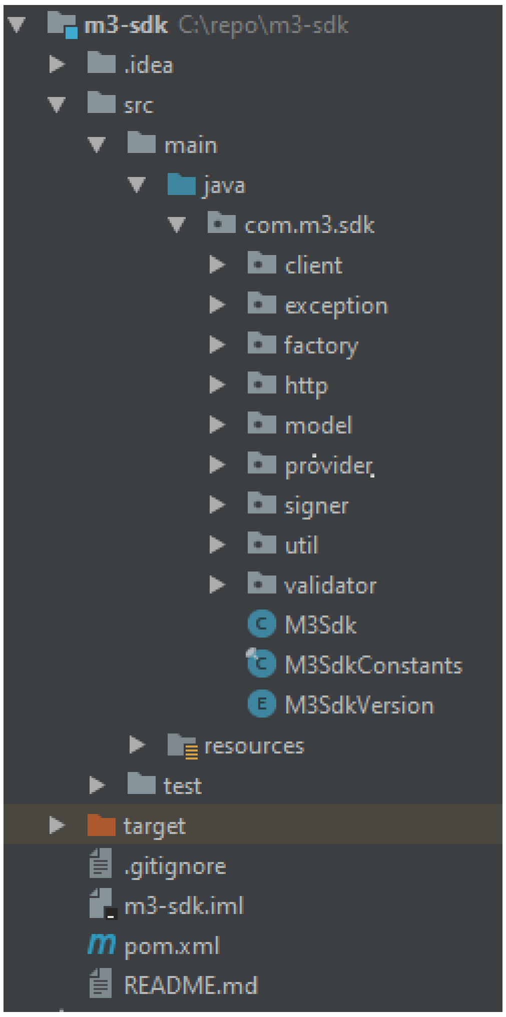
```
IM3client client =
M3Sdk.client(serverContextProvider, clientContextProvider, credentialsProvider, access
KeyProvider);
```

2. Execute command

```
M3ApiResult result = client.<commandName>(
                "parameter1",
                "parameter2",
                 …);
```

3.1.4 Maestro3 SDK Structure

This is the structure of Maestro3 SDK, which is available in the project repo.



```
▼ m3-sdk C:\repo\m3-sdk
  ▶ .idea
  ▼ src
    ▼ main
      ▼ java
        ▼ com.m3.sdk
          ▶ client
          ▶ exception
          ▶ factory
          ▶ http
          ▶ model
          ▶ provider
          ▶ signer
          ▶ util
          ▶ validator
            M3Sdk
            M3SdkConstants
            M3SdkVersion
      ▶ resources
    ▶ test
  ▶ target
    .gitignore
    m3-sdk.iml
    pom.xml
    README.md
```

Figure 3 – Maestro Java SDK structure

3.2 AUTHORIZATION ALGORITHM

The SDK client for Maestro API uses three headers to provide authorization information into the server.

- **Maestro-Authentication** - the request signature generated by M3 SDK based on Maestro-accessKey provided by SDK client.
- **Maestro-request-identifier** - the client identifier provided by the SDK Client and used for the security purpose.
- **Maestro-accessKey** - the identity that should be registered in EC2 parameters store and used to verify the client. To register the identity, please address the M3 admin team.

The authorization is performed according to the following scenario:

1. Get the long representation of the request date from <Maestro-date> header:

```
long date = new Date().getTime();
```

2. Construct SecretKeySpec from the secretKey and date

```
byte[] bytes = Charset.forName("UTF-
8").encode(String.format("%s%s", secretKey, date)).array();
SecretKeySpec secretKeySpec = new SecretKeySpec(bytes, "HmacSHA256");
```

3. Generate the sign:

```
Mac mac = Mac.getInstance("HmacSHA256");
Mac.init(secretKeySpec);
byte[] message = mac.doFinal(String.format("M3-
POST:%s:%s", accessKey,date).getBytes());

StringBuilder builder = new StringBuilder();
for (final byte element : message) {
  builder.append(Integer.toString((element & 0xff) + 0x100, 16));
}
String sign = builder.toString();
```

4. The new String(sign) is <Maestro-Authentication> header

On the Server side, it is needed to get the same signature and compare it with the Client's signature.

Java SDK sample:

If you use Maestro3 Java SDK, all of the above is performed inside M3Client class.

```
M3Sdk.client(new M3StaticServerContextProvider("serverUrl"),
new M3StaticClientContextProvider(new M3ClientContext("clientIdentifier")),
   new M3StaticCredentialsProvider("accessKey","secretKey"),
new M3StaticAccessKeyProvider("accessKey"))
```

Also, you can replace static providers with your own implementations of IM3ServerContextProvider, IM3ClientContextProvider, etc.

4 SUPPORT AND DEVOPS TOOLS

4.1 TERRAFORM AND TERRAFORM PROVIDER

The main IaC toos taken on board by Maestro3 is Terraform by HashiCorp – a cross-platform solution which allows managing complex infrastructures hosted in multiple clouds. Terraform is a tool for building, changing, and versioning infrastructure in a safe and efficient manner.

Terraform is built on a plugin-based architecture, enabling developers to extend Terraform by writing new plugins or compiling modified versions of existing ones.

The main principle of this tool is in Infrastructure as a Code approach, i.e. infrastructure parameters are described in a template, in a JSON and HCL formats. Both formats are supported by Maestro3.

4.1.1 Integration with Terraform

Maestro3 integration with Terraform provides its users with ability to:

- store templates in AWS S3 and in GitHub and upload templates from the repository
- automatically react to updates in GitHub (auto plan and auto apply)
- plan and apply templates
- review and validate logs
- lock templates for further modification, planning or applying
- share and re-use templates

M3 supports the following Terraform functionality:

- ability to upload templates through the UI
- possibility to edit templates via Maestro3 UI, it is possible to receive information about the resources that were deployed and monitor the logic of the template
- imposition of limitations for specific users, i.e. for users and for regions.

.

To register the GitHub based template on Maestro3, the following parameters should be specified:

- **source type** (GitHub) – only after this, the necessary input field will appear,
- your GitHub **username** and **password**,
- **URI** of the relevant GitHub repository and its **branch**,
- **folder** where your Terraform templates are stored,
- (optional) Terraform variables and their values to be used as default ones when the template is applied.

You must also select what actions Maestro3 will take once the template is updated in the GitHub repository (None, Auto plan, or Auto apply):

Once registered, the both GitHub and AWS-based Terraform templates can be reviewed in the **Catalog** page with any version of Maestro3 Orchestrator. Terraform templates stored in GitHub are updated and validated automatically once they are changed in the repository – you do not need to do it specifically for Maestro3.

Terraform allows to automate all changed and provide versioning of the infrastructure.

Terraform CLI is a free tool and can be used by command line utility.

The table below lists the available stack templates actions and their basic details:

Action	Description
Upload	Upload a stack template to Maestro3
Plan	Create execution plan and estimate the results of template application.
Apply	Apply the stack template on infrastructure
Remove	Deletes the stack template from Maestro3
Lock	Locks the template from other users, so that they cannot modify or remove it
View execution log	Display the execution logs of the template
Download logs	Download template execution or planning logs in a zip archive

4.1.2 Terraform Provider for Maestro3

Terraform is a framework for configuration, but one Terraform template cannot be written in a unified way so that it could be used for different cloud providers (the template for AWS will not work if you deploy it in Azure). Maestro3 provides its own custom tool - Terraform-Provider that gives the users possibility to work with the provided API so that they can deploy the infrastructure with a single template on any cloud provider supported by Maestro3, including private OpenStack region.

Working with Terraform Provider for OpenStack

Requirements:

The following software should be installed before you use Terraform:

- Terraform 0.12+.
- Go 1.13 (to build the provider plugin).

To start using the provider, do the following:

1. To build the Terraform Provider plugin, run:

```
#linux
go build -o terraform-provider-m3_v0.2
#windows
go build -o terraform-provider-m3_v0.2.exe
```

2. Move the plugin to the user plugins directory (you can find more details here):

```
#linux
mv terraform-provider-m3_v0.2 ~/.terraform.d/plugins
#windows
move terraform-provider-m3_v0.2.exe %APPDATA%\terraform.d\plugins
```

3. Specify the provider settings:

```
provider "m3" {
      url = "http://ip:port/maestro/api/V3"
      access_key = "access_key"
      secret_key = "secret_key"
      user_identifier = "user_identifier"
}
```

The current implementation supports the limited scope of operations, but the existing configuration is enough to run a VM and create an image.

Below, you can find an example of the provider usage by one of our colleagues who contributed the feature:

These parameters are enough to run an instance:

```
resource "m3_instance" "my-server" {
      image = "CentOS7_64-bit"
      instance_name = "test_name"
      region = "EPAM-OPENSTACK-3"
      tenant_name = "EPMC-EOOS"
      shape = "MINI"
      key_name = "sshkey"
}
```

These parameters are enough to create an image:

```
resource "m3_image" "my-image" {
      tenant_name = "EPMC-EOOS"
      region_name = "EPAM-OPENSTACK-3"
      image_name = "ImageFromTf"
      source_instance_id = "ecs00100019F"
      description = "Here is image desccription"
}
```

4.2 CHEF

Chef is an open source automation tool that allows to quickly set up and configure the necessary infrastructure, and to change the configuration if needed. System configuration files that describe how Chef manages server applications are called **recipes**. Several recipes grouped together constitute a **cookbook**.

Currently this functionality is implemented and working in a test mode on the AWS cloud, supporting Jenkins and Artifactory automation systems.

Functionality was tested on AWS cloud using the following parameters:

OS	Shape	Role
CentOs7	Medium	artifactory-acs
Ubuntu16	Medium	jenkins_epc

All profiles have some requirements to instance OS/Shape.
In the current implementation, validation for these requirements is not supported, so some combination for OS/Profile/Shape may not be working.

4.2.1 Configuring Chef

See an example of configuring Chef in Maestro3 below:

1. A region should be activated with parameters, set specifically for working with Chef. To get an access to such a region or to configure Chef for the already activated project, send a specified request to the Maestro3 support team.

   Thus, when launching an instance, this region should be specified in a **Region** field together with other parameters for running the instance.

2. Select the **Set chef configuration** checkbox and specify the **Chef Profile**. The profile defines the configuration that should be installed on the instance.
3. On the screen you can review information about the Chef Profile, which was specified on a previous step.
4. The **Autoconfiguration** tab is added to the **Content View** section of the specified instance in AWS cloud. It contains useful information about chef role (not available at the moment), state and chef server. As a result, the installation of the software selected depending on the Chef Profile was successfully performed on the instance.
5. In order to access and interact with the interfaces of the installed software (in this case - artifactory and Jenkins), perform the following actions:
   a. Copy the public DNS name of the instance;
   b. Open the User Interface of the launched application, using the copied DNS name:
      - For Jenkins: <DNS_NAME>:8080
      - For artifactory: < DNS_NAME > :8081/artifactory.

4.2.2 Work Principles

When a software is being installed on an instance, an init script is added to it. The instance gets registered in Maestro3. After that, communication between the Chef server, Maestro3 server, and a Chef client is established.

See the scheme for an init script used by Chef below:

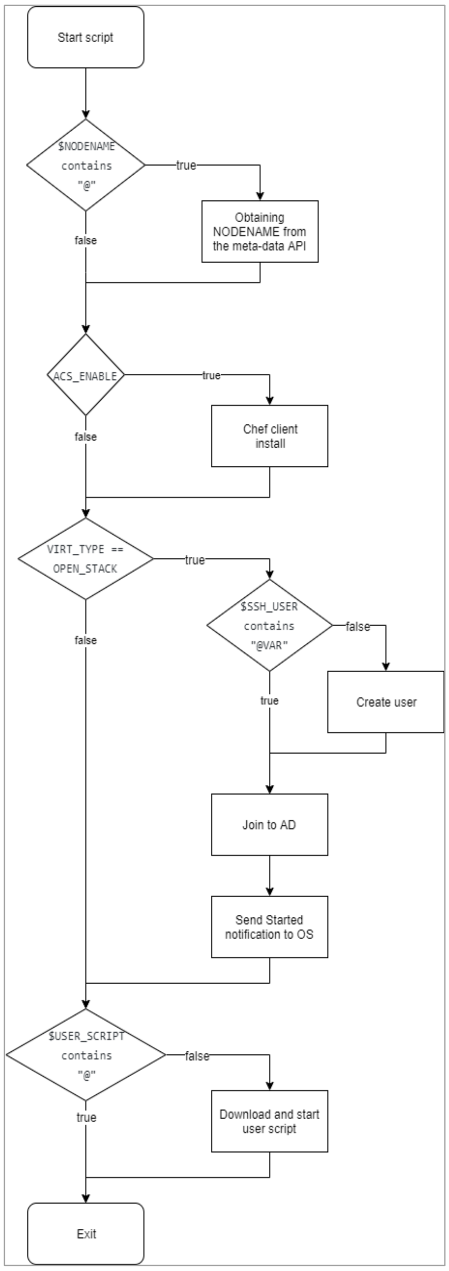


Figure 4 – Init script used by Chef

The table below lists required variables for the init script.

Variable	Placeholder	Description	Example
CONF_URL	@VAR_EP_ORCH_IP	URL endpoint to Orchestrator	https://config.cloud.epam.com/orchestration-dev
orch_url	@VAR_CONFIG_URL	URL endpoint to Orchestrator (when VAR_ACS_ENABLE=false)	https://qa.cloud.epam.com
VIRT_TYPE	@VAR_VIRT_TYPE	Virtualization type	OPEN_STACK/AZURE/AWS/GOOGLE
NODENAME	@VAR_NODENAME	instanceId	ecs00012345678
ACS_ENABLE	@VAR_ACS_ENABLE	autoConfigurationDisabledType from Projects	true/false
CHEF_PROJECT	@VAR_PROJECT_CHEF	Dedicated Chef server is used	true/false
CHEF_SRV	@VAR_CHEF_SERVER	Chef server fqdn	acs.cloud.epam.com
CHEF_ENV	@VAR_CHEF_ENV	Chef environment	development/production
CHEF_ORG	@VAR_CHEF_ORG_NAME	Chef organization name	epam-dev/epam-qa/epam
SSH_USER	@VAR_SSH_USER	Default OS username	ubuntu/admin/centos
USER_SCRIPT	@VAR_USER_SCRIPT	String containing links to user scripts and launch parameters, if used	/api/files/0aca25d4-ffc8-46c0-907a-0dc4f21cd98a/user-init.sh:fuu#bar
NOTIF_URL	@VAR_NOTIF_URL	Used as a signal that the VM has entered the running condition	https://qa.cloud.epam.com/api/openstack/notification/running
OS_CHECKSUM	@VAR_NOTIF_CHECKSUM	Used together with NOTIF_URL	SXVZSjl0czlJbENFSmxpdXFnSk0wUlBo

4.3 ANSIBLE

As an automation system for provisioning and configuring infrastructure, Ansible allows installing software on virtual machines and manage large clusters of them on different cloud providers.

A distinctive feature of Ansible is that the system has a server, which does not contain the client side, i.e. you do not need a client to be installed on a virtual machine. Ansible accesses virtual machines via SSH, and is intended for working with Linux machines, though it is possible to use it for Windows as well. Thus, a Linux server has access to a virtual machine via SSH and performs pre-defined tasks there.

The tasks which should be performed on a virtual machine are described in playbooks. A playbook is structured on lambda, which describe conditions of the necessary tasks (similar to a template or a stack).

Ansible also works with **host groups** where hosts are described/registered. The host group defines virtual machines that Ansible should affect. Grouping is necessary to define the scope of work, i.e. a playbook is launched for a certain group.

Every host group is created **manually** in a JSON format and contains the following:

- name of the host group
- DNS names or IP addresses of the VMs to perform automation configuration on.

For example, to install a new Java software version on a group of Jenkins slaves, you can define the conditions in a playbook. Running a playbook will make Ansible go to all hosts and perform the same playbook on them.

Besides, Ansible includes the **Dynamic Inventory** feature that enables retrieving information from dynamic sources. In this case, dynamic groups are created on the fly using cloud providers (Maestro3). You can configure dynamic groups to any criteria - keys, security groups, tags.

Maestro3 works with dynamic groups, their management, and provisioning of these groups to users.

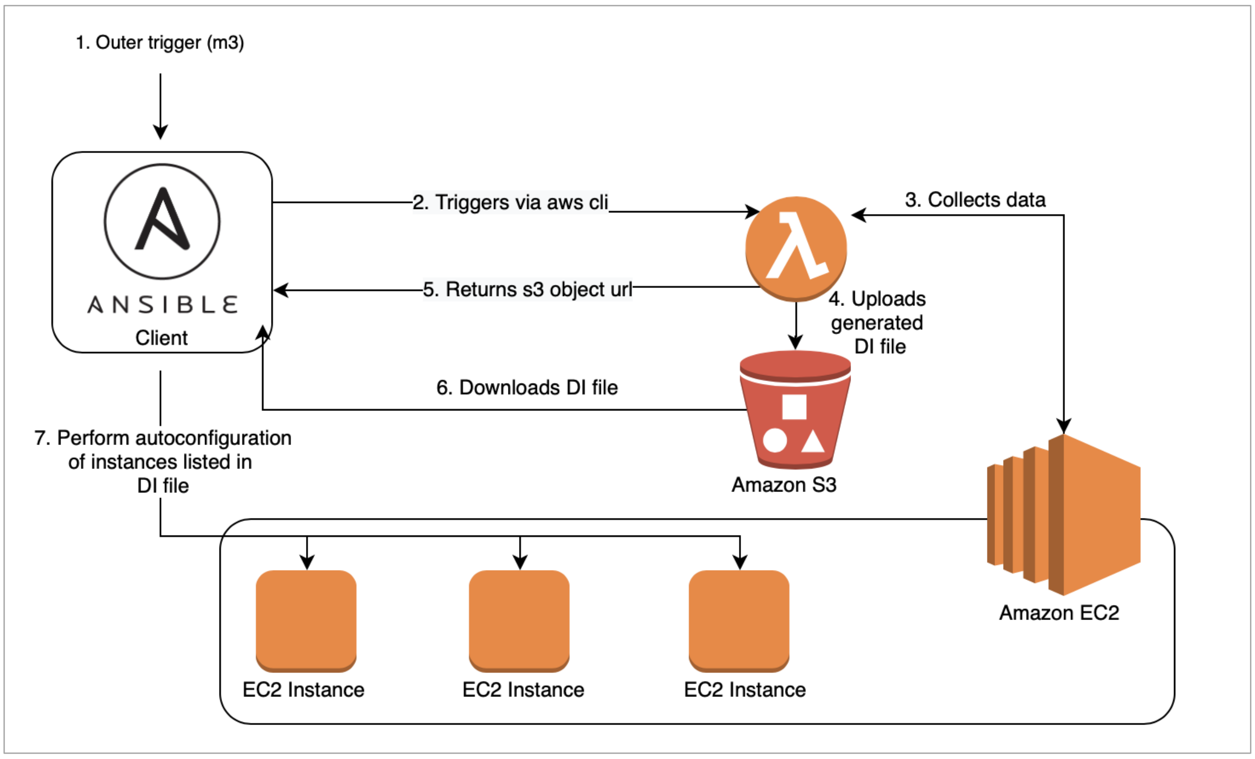


Figure 5 – Ansible and Dynamic Inventory flow

In Maestro3 solution, Ansible is introduced via the respective wizard on the Main Dashboard.

The wizard allows to generate an Ansible client for the selected region and tenant. The result will be provided as a downloadable zip, containing the dynamic inventory files.

To start working with Ansible client, select Ansible wizard, and perform the required steps. For a detailed information, please see Annex A.

After you perform the necessary steps, you obtain a downloadable .zip file. The zip file contains a configuration file and a Python script with lambda that configures a dynamic inventory file.

The **dynamic inventory file** is a list of virtual machines, including their IPs, DNS addresses and all other VM properties grouped by parameters, e.g. tags, keys, owners, security groups etc.

To make the Python script work, correct it according to your parameters. After that, launch the script using CLI. It will start sending the parameters, created during the generation of this Ansible client, to lambda.

Knowing the parameters, lambda describes virtual instances in a specified tenant and region. Lambda sends a request to a TCP service for obtaining a list of all VMs. Upon obtaining the data, lambda groups it according to the specified keys, loads the dynamic inventory file into S3 and send a link to this file to Ansible Client.

Resign URL, a temporary link to the file stored in S3 is signed with the credentials of this lambda and expires after 15 minutes on creation. If it is not obtained by the client within 15 minutes, the file becomes unavailable, as the infrastructure is updated every 15 minutes.

Upon loading the file, Ansible starts to analyze the file and defines what software should be configured on which VMs. Information about the software is obtained from the playbook, the list of VMs to be configured is obtained from the Dynamic Inventory file.

The list of VMs with properties grouped by virtual keys, is provided to Ansible.

Ansible is the most popular open source automation tool on GitHub. It is an automation tool used to configure systems, deploy software, and orchestrate more advanced tasks such as continuous deployments or zero downtime rolling updates.

5 M3ADMIN

m3admin is a tool that provides all administrative functionality that Maestro3 environment requires to operate successfully. There is a set of commands, divided into groups, which allow Maestro3 environment administrator to configure the environment to operate in a way that a customer requested.

The m3admin tool can be used via command line interface and via API.

5.1 ACCESSING M3ADMIN CLI

By default, m3admin tool is available via web by the following link: https://admin.maestro3.tools/.

The access is restricted and available to authorized users only.

You can use it manually by logging in with provided credentials, or automatically via http requests.

5.2 M3ADMIN CLI COMMANDS USAGE

Each m3admin command call consists of the following elements:

```
m3admin [group] [subgroup][command] [arguments]
```

- **Group**: defines the general area of the command application (e.g. '**aws'** – commands related to AWS, '**billing'** – commands related to billing configuration, etc.).
- **Subgroup**: defines a specific area within some groups (e.g., '**environment setting**' – the **setting** subgroup within the **environment** group allows to manage environment settings).
- **Command:** the actual command string defining the action to be performed (e.g., '**add_default_ tenant_group**').
- **Arguments:** define the specific object of the command and/or the values to be set for it.

Below, you can find an example of a command call in m3admin. The command will delete tenant-related resources in a tenant's AWS account.

```
m3admin aws deactivate --bundle_name ${BUNDLE_NAME}
```

For each of the groups and commands, you can use the **–help** parameter that will display all available options with comments.

Detailed descriptions of the m3admin commands are in the **Admin CLI User Guide**.

5.3 USING M3ADMIN VIA API

The Maestro3 Administration API (further referred to as m3admin API) is an API that provides the possibility to configure Maestro3 installation. The functionality of m3admin API is similar to the m3admin CLI.

There is a restriction mechanism that will limit access to some command groups (like AWS, Azure, OS, etc.). This restriction functionality is fully configurable.

5.3.1 Accessing m3admin API

The m3admin API is secured with SSL encryption (HTTPS protocol is used).

User credentials are required to interact with m3admin API. Basic Authentication is used as an authentication method.

The access to API can be checked using API testing tool like Postman, Insomnia or curl.

Curl example:

```
curl --request GET --url https://${host}/ --header 'authorization: Basic
${credentials}'
```

Variable credentials in the example above are represented by a placeholder for credentials, encoded in Base64 in the following format: *username:credentials.*

m3admin API credentials are provided by Maestro3 Support Team by request.

5.3.2 Checking m3admin API Information

${host}/ GET call returns API summary which contains the following items:

Response parameter	Description
api_version	The current version of m3admin API
message	The summary message
available_groups	The dataset with currently available groups, commands in these groups, and their parameters.

The example is displayed in the code snippet below:

```
{
 "available_groups": {
  "azure": [
   {
    "name": "validate_vm_counters",
    "description": "Checks if counters values correspond to real instances
quantity",
    "parameters": [
     {
      "name": "tenant_name",
      "required": true,
      "description": "Tenant name"
     }
    ],
    "path": "/azure/validate_vm_counters",
    "method": "POST"
   },
   ...
  ],
  "aws": [...]
 },
 "api_version": "2.4.67",
 "message": "This is the Maestro3 administration tool API. To request support, please
contact Maestro3 Support Team"
}
```

5.3.3 Sending Requests via m3admin API

The m3admin API suggests a similar interface to m3admin CLI with the following mapping of commands:

```
CLI command:
m3admin azure validate_vm_counters --tenant_name TENANT_NAME
API command using `curl`:
curl --request POST \
--url https://${host}/azure/validate_vm_counters \
--header 'authorization: Basic b25zaGE6YWFjOWEyOTktY2EHjKYhjkGJKMDItM2NhNTc3NDdlZmFm'
\
--header 'content-type: application/json' \
--data '{
"tenant_name": "EPMC-EOOS"
}'
```

The m3admin API validates incoming requests. In case of parameters mismatch, the tool will describe the

The example below shows the parameter mismatch response:

```
HTTP/1.1 400 Bad Request
Content-Type: text/html; charset=UTF-8
```

```
{
 "status": "FAILED",
 "required_parameters": [
  {
   "name": "tenant_name",
   "required": true,
   "description": "Tenant name"
  }
 ],
 "message": "Not all required parameters specified"
}
```

6 LAMBDA API

Lambda API allows managing instances, storage volumes, tags, and getting reports and metrics information by addressing the respective Lambda functions that enable Maestro3 functionality.

Lambda API gives Maestro3 users the tool for developing their plugins and extensions for working with Maestro dynamic UI.

Below we will describe Lambda API based on Maestro3 API and provide auto generated JS SDK as an example. Maestro3 API offers a possibility to use other languages as well.

6.1 ENTRY POINT

The entry point for Maestro3 API is **MaestroAPIClient**. The pattern for the entry point is as follows:

```
<API G id>.execute-api.<aws region>.amazonaws.com
```

6.2 TYPICAL WORKING SCENARIO

6.2.1 Prerequisites

For the JavaScript SDK to work, your APIs need to support CORS. The Amazon API Gateway developer guide explains how to setup CORS for an endpoint. The generated SDK depends on third-party libraries. Include all of the scripts in your webpage

```
<script type="text/javascript" src="lib/axios/dist/axios.standalone.js"></script>
<script type="text/javascript" src="lib/CryptoJS/rollups/hmac-sha256.js"></script>
<script type="text/javascript" src="lib/CryptoJS/rollups/sha256.js"></script>
<script type="text/javascript" src="lib/CryptoJS/components/hmac.js"></script>
<script type="text/javascript" src="lib/CryptoJS/components/enc-base64.js"></script>
<script type="text/javascript" src="lib/url-template/url-template.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/sigV4Client.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/apiGatewayClient.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/simpleHttpClient.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/utils.js"></script>
<script type="text/javascript" src="apigClient.js"></script>
```

6.2.2 Using the SDK

To initialize the most basic form of the SDK:

```
var apigClient = apigClientFactory.newClient();
```

Calls to an API take the form outlined below. Each API call returns a promise, that invokes either a success or failure callback.

```
var params = {
    //This is where any header, path, or querystring request params go. The key is the
parameter named as defined in the API
    param0: '',
    param1: ''
};
var body = {
    //This is where you define the body of the request
};
var additionalParams = {
    //If there are any unmodeled query parameters or headers that need to be sent with
the request you can add them here
    headers: {
        param0: '',
        param1: ''
    },
    queryParams: {
        param0: '',
        param1: ''
    }
};

apigClient.methodName(params, body, additionalParams)
    .then(function(result){
        //This is where you would put a success callback
    }).catch( function(result){
        //This is where you would put an error callback
    }
);
```

6.2.3 Using AWS IAM for Authorization

Maestro SDK is protected with the AWS IAM role. To create a client for work with SDK methods, we use temporary credentials got during authentication.

To initialize the SDK with AWS credentials, use the code below. Note, if you use credentials all requests to the API will be signed. This means you will have to set the appropriate CORS accept-* headers for each request.

```
var apigClient = apigClientFactory.newClient({
    accessKey: 'ACCESS_KEY',
    secretKey: 'SECRET_KEY',
    sessionToken: 'SESSION_TOKEN', //OPTIONAL: If you are using temporary credentials
you must include the session token
    region: 'us-east-1' // OPTIONAL: The region where the API is deployed (e.g. eu-
west-1, us-west-2). Defaults to us-east-1.
});
```

Usage example

```
client['googleMetricsGraphGet']({'availabilityZone': 'eu-central-1a', 'instanceId':
'i-1234ffbd569e61234', 'region': 'AWS-EU-CENTRAL-1', 'range': 'WEEK', 'tenant': 'AWS-
EPMC-EOOS', 'metricName': 'CPU_UTILIZATION'}).then((data) => / *processing code* /));
```

6.3 AUTHENTICATION ALGORITHM

6.3.1 Authentication with Corporate SSO

Maestro3 API can be integrated with any corporate SSO. So, after running the respective command to login with SSO, an authentication algorithm is as follows:

1. The request is sent to the API through the lambda_maestro_auth.
2. API connects to the corporate SSO and requests the information about the projects and position level of the user.
3. The lambda_maestro_auth generates the token with the following parameters:

   - Temporary user credentials for AWS IAM user (expire in 1 hour)
   - Project/ projects
   - Position level

When the user passes the authentication, the temporary credentials are shown as HTTP Headers in each user's request to Maestro3 API.

6.3.2 Authentication with Google

Another way to authenticate a user is to use Google authentication process. Maestro3 API is integrated with Google and provides the Google OAuth authentication. After running the respective command, Maestro3 API provides a user with the permissions to use the default tenants configured on Maestro environment. The temporary credentials are sent for the access to Maestro3 API.

6.4 HTTP HEADER

This header is generated automatically for any generated AWS API Gateway SDK is placed at the beginning of each HTTP request.

In case you need to work without provided API Gateway generated SDK, please see the detailed information in the Using Temporary Security Credentials section on the AWS web-site.

```
authority:
plkjp4b6q2.execute-api.eu-central-1.amazonaws.com
:method:
POST
:path:
/prod/components/toolbar
:scheme:
https
accept:
application/json
accept-encoding:
gzip, deflate, br
accept-language:
ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7,uk;q=0.6
authorization:
AWS4-HMAC-SHA256 Credential=ASIAIWAFQAJWQSDILX7A/20180427/eu-central-1/execute-
api/aws4_request, SignedHeaders=accept;content-type;host;x-amz-date,
Signature=deda30b25b388da12fceaad93c5deb68381ff21ef51d34e84160e7ab2074c6c4
content-length:
39
content-type:
```

```
application/json
origin:
https://maestro3.tools
referer:
https://maestro3.tools/home
user-agent:
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36
x-amz-date:
20180427T125923Z
x-amz-security-token:
Ago0000Z2luEGEaDGV1LWNlbnRyYWwtMSKAAmHALTkHEWXb7Hw0SlgEcg/C7WWmBHiTQlk/vHDLmhmRy9ylh5lk33S3xTv
T3cp4FH2p+dMl90nJRyV60V0RA6NHEnZ672h5+9GIujxeumvwqErodQo+Y8gOM7UvJ3QL3EenxI4fA/M1wqKFvfC2s3M7z
vP5UBGwiLkZ6OYSUNTXXBdRz2tUB6cgC/RLLhgyR+TKoWpDiLhLdTp1bNBGLmBP1B7Mx5rqbuqEYO0xKJGsaS6NUls8s8n
IXYHvdWMIAm91sfM1pnOTgaJjRmfPnoyLfZlSfid48B7fAbEDxmNPCM4FS7/1PCGxs0Bw1+oI189AfoUUQeZ9fVg+z/8Br
b4q+AQIpv//////////ARAAGgw1NDI4MjY4MDEwNjMiDIElOI52o/A9kEMT9SrMBF++PemrRVwqZXp5PV8lXKzieEJmaGW
TfiXIjVJZeI5majcF3r0DN0DnHUQqxj+gUpRl2T5Beq7N0BFx2OL2Klo934TN1MHXg0g9cBpHZpID55R+PUyIXgqe1z1NG
yxdP6+wKUS/2L9VXLYKUFT06twSE5iHSCLtN01ZnUOBVpb6G1pZuhqrIOJt0Rlugn3SFvL6Mse6lzWgQRo7Jq49d8N4GG4
bdDUUfrydPYaBFdgW9SS9/kKoRdfEVrAeJAZrLOMQpRIJJBG3kxR8SAWVmSRaXoa8o9jC6kh36m2ieGoYMNu8igpn0T9Zx
E4XYdfuFn21RzlN2HCJY7m50fvv6TUJ9Gl7/NEux6ySQ3rUTCXSL/XiylhII9cJe5L5xq+/mCdHgEZfkW7Ar2SxwB799ua
ASQ3pdEfKSeRf181HjtlRqCUTnD4NB+KYz9448/KB3swMA8BG9kdht2bPQc1jtnZHOQuiPAFbF3iSbris41Hf/X9NEZ4NW
AtfVnzYLgWbJySPR8NjP66oFwgXzuOwYMEtX9IeJlqK5Ru82B8ZbnnnWcmQq54hPyBgnHMP8snYjUjkgRkdi5RKqPO413N
PGmIFXXK7W7QDkr9Kkp0Uhdv7J15D7SloZZR6wTKo/q9BmKXoFPC6uI/kmaMByLfsga+PzB6xTZ0DSju9Sh92RSQ12iWma
bKqUc6mMHhGgvnaAukzAOv2uXFe+HpCIDWjuhWAdYPXb6tZkif7inWIz2QWgDVtFpTRfSSRQNLrIx50Ayt32Ze1b801geq
9FDCQvYzXBQ==
```

You can find the detailed description of the HTTP methods, API resource identifiers, JS SDK methods, and related Lambda functions in the **Lambda API Developer’s Guide**.

ANNEX A – ANSIBLE CLIENT

ANSIBLE CLIENT

Ansible is the most popular open source automation tool on GitHub. It is an automation tool used to configure systems, deploy software, and orchestrate more advanced tasks such as continuous deployments or zero downtime rolling updates.

In Maestro3 solution, Ansible is introduced via the respective wizard on the Main Dashboard.

The wizard allows to generate an Ansible client for the selected region and tenant. The result will be provided as a downloadable zip, containing the dynamic inventory files.

DOWNLOADING ANSIBLE CLIENT

In order to download Ansible Client perform the following steps:

1. Start the **Ansible Client** wizard from the main Maestro3 Dashboard.
2. Specify tenant and region for which you want to generate the Ansible Client parameters and click the **Next** button.
3. In the next window you can review the properties of the file to be downloaded. Click the **Download** button to proceed.
4. The downloaded package will be stored in the **Downloads** folder of your computer.

EXECUTING ANSIBLE CLIENT

To configure Ansible Client in order to get the inventory file, follow the instructions provided in the **readme.md** file from the downloaded package.

The following software should be installed on the machine where the Ansible Client will be executed:

- Python3 v3.7.1 or later
- pip v19.3.1 or later
- virtualenv v16.1.0 or later

In the current revision of Ansible Client (3.2.100.46) only the guide for Linux systems is present in the readme.md file. The flow of work for Linux and Windows is described below in this instruction.

For Linux

1. Navigate to AnsibleClient folder using terminal
2. Create virtualenv using the command
   `virtualenv -p python3 .venv`

Your OS may have no configured python3 alias. To configure it properly please follow the instructions by link: https://realpython.com/installing-python/ .

3. Activate virtualenv with the command `source .venv/bin/activate`
4. Install requirements from the requirements.txt file with command
   `pip install -r requirements.txt`
5. Edit client.config file by providing AWS credentials from the environment.

These credentials should have permissions to invoke lambda function ('lambda:InvokeFunction').

```
#Maestro3 ansible client config
#Wed Feb 19 10:33:54 UTC 2020
aws_secret_access_key=******************************
clientId=******************************
cache.max.age=300
aws_access_key_id=******************************
lambda.name=autoconf_ansible_meta
lambda.region=eu-central-1
region=AWS-AP-NORTHEAST
tenant=AWS-EPMC-ACM3
```

6. Save the file.
7. Execute the command `python m3_inventory.py`
8. This result will be returned:

```
(.venv) ➜  m3-AnsibleClient-d20c python m3_inventory.py
Ansible DI file has been created by path /Users/user/Downloads/m3-
AnsibleClient-d20c/AWS-EPMC-ACM3.json
```

9. Your Ansible inventory file is ready and is stored in a file located by path returned in output by m3_inventory.py script. In the example this is `/Users/user/Downloads/m3-AnsibleClient-d20c/AWS-EPMC-ACM3.json`.

For Windows

1. Navigate to AnsibleClient folder using cmd.
2. Check the version of the python. Execute 'python':

```
(venv) C:\Users\Daryna_kozub\Desktop\m3-AnsibleClient-ffe7>python
Python 3.7.6 (tags/v3.7.6:43364a7ae0, Dec 19 2019, 00:42:30) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3. If the version of python is higher than 3.7.0 everything is ok. If the version is less, install Python3.7+ and use the alias of python3.7 while creating the virtualenv using the command `virtualenv -p python3.7 venv`
4. Create virtualenv with the command `virtualenv venv`
5. Activate virtualenv using the command `.\venv\Scripts\activate.bat`
6. Install requirements using the command `pip install -r requirements.txt`
7. Edit client.config file by providing AWS credentials from the environment. These credentials should have permission to invoke lambda function ('lambda:InvokeFunction').
8. Save the file.
9. Execute m3_inventory.py script using the command `python m3_inventory.py`

```
(venv) C:\Users\Daryna_kozub\Desktop\m3-AnsibleClient-ffe7>python
m3_inventory.py
Ansible DI file has been created by path C:\Users\Daryna_kozub\Desktop\m3-
AnsibleClient-ffe7/AWS-EPMC-ACM3.json
```

10. The result has been created and saved to **AWS-EPMC-ACM3.json** file.

TABLE OF FIGURES

VERSION HISTORY

Version	Date	Summary
1.5	July 11, 2020	Added information about Ansible and Chef
1.4	May 18, 2020	Added information about the Terraform provider
1.3	March 2, 2019	Reviewed and updated the SDK section
1.2	December 5, 2019	Added information about Admin CLI
1.1	July 05, 2019	Added information about Lambda API
1.0	April 1, 2019	Initially published