Cloudmesh REST Interface for Virtual Clusters

Gregor von Laszewski[1,*], Fugang Wang[1], and Badi Abdhul-Wahid[1]

[1] School of Informatics and Computing, Bloomington, IN 47408, U.S.A.
[*] Corresponding authors: laszewski@gmal.com

Draft v0.0.1, April 18, 2017

This document summarizes a number of objects that are instrumental for the interaction with Clouds, Containers, and HPC systems to manage virtual clusters. TBD

Keywords: CLoudmesh, REST, NIST

https://github.com/cloudmesh/rest/tree/master/docs

CONTENTS

1. INTRODUCTION

In this document we summarize elementary objects that are important to for the NBDRA.

1.1. Design by Example

To accelerate discussion among the team we use an approach to define objects and its interfaces by example. These examples are than taken in a later version of the document and a schema is generated from it. The schema will be added in its complete form to the appendix B. While focusing first on examples it allows us to speed up our design and simplifies discussions of the objects and interfaces eliminating getting lost in complex syntactical specifications. The process and specifications used in this document will also allow us to automatically create a implementation of the objects that can be integrated into a reference architecture as provided by for example the cloudmesh client and rest project [?].

An example object will demonstrate our approach. The following object defines a JSON object representing a user.

Listing 1.1: User profile

```
{
  "profile": {
    "description": "The Profile of a user",
    "uuid": "jshdjkdh...",
    "context:": "resource",
    "email": "laszewski@gmail.com",
    "firstname": "Gregor",
    "lastname": "von Laszewski",
    "username": "gregor"
  }
}
```

Such an object can be transformed to a schema specification while introspecting the types of the original example. The resulting schema object follows the Cerberus [?] specification and looks for our object as follows:

```
profile = {
   'description': { 'type': 'string'},
   'email': { 'type': 'email' },
   'firstname': { 'type': 'string'},
   'lastname': { 'type': 'string' },
   'username': { 'type': 'string' }
}
```

As mentioned before, the AppendixB will list the schema that is automatically created from the definitions.

1.2. Tools to Create the Specifications

The tools to create the schema and object are all available opensource and are hosted on github. It includes the following repositories:

cloudmesh.common
https://github.com/cloudmesh/cloudmesh.common

cloudmesh.cmd5
https://github.com/cloudmesh/cloudmesh.cmd5

cloudmesh.rest
https://github.com/cloudmesh/cloudmesh.rest

cloudmesh/evegenie
https://github.com/cloudmesh/evegenie

1.3. Installation of the Tools

The current best way to install the tools is from source. A convenient shell script conducting the install is located at:
TBD

Once we have stabilized the code the package will be available from pypi and can be installed as follows:

```
pip install cloudmesh.rest
pip install cloudmesh.evengine
```

1.4. Document Creation

It is assumed that you have installed all the tools. TO create the document you can simply do

```
git clone https://github.com/cloudmesh/cloudmesh.rest
cd cloudmesh.rest/docs
make
```

This will produce in that directory a file called object.pdf containing this document.

1.5. Conversion to Word

We found that it is inconvenient for the developers to maintain this document in Microsoft Word as typically is done for other documents. This is because the majority of the information contains specifications that are directly integrated in a reference implementation, as well as that the current majority of contributors are developers. We would hope that editorial staff provides direct help to improve this document, which even can be done through the github Web interface and does not require any access either to the tools mentioned above or the availability of LaTeX.

The files are located at:

- https://github.com/cloudmesh/cloudmesh.rest/tree/master/docs

1.6. Interface Compliancy

Due to the extensibility of our interfaces it is important to introduce a terminology that allows us to define interface compliancy. We define it as follows

Full Compliance: These are reference implementations that provide full compliance to the objects defined in this document. A version number will be added to assure the snapshot in time of the objects is associated with the version. This reference implementation will implement all objects.

Partially Compliance: These are reference implementations that provide partial compliance to the objects defined in this document. A version number will be added to assure the snapshot in time of the objects is associated with the version. This reference implementation will implement a partial list of the objects. A document is accompanied that lists all objects defined, but also lists the objects that are not defined by the reference architecture.

Full and extended Compliance: These are interfaces that in addition to the full compliance also introduce additional interfaces and extend them.

2. USER AND PROFILE

In a multiuser environment we need a simple mechanism of associating objects and data to a particular person or group. While we do not want to replace with our efforts more elaborate solutions such as proposed by eduPerson (http://software.internet2.edu/eduperson/internet2-mace-dir-eduperson-201602.html) or others [?], we need a very simple way of distinguishing users. Therefore we have introduced a number of simple objects including a profile and a user.

2.1. Profile

A profile is simple the most elementary information to distinguish a user profile. It contains name and e-mail information. It may have an optional uuid and/or use a unique e-mail to distinguish a user.

Listing 2.1: User profile

```
{
  "profile": {
    "description": "The Profile of a user",
    "uuid": "jshdjkdh...",
    "context:": "resource",
    "email": "laszewski@gmail.com",
    "firstname": "Gregor",
    "lastname": "von Laszewski",
    "username": "gregor"
  }
}
```

2.2. User

In contrast to the profile a user contains additional attributs that define the role of the user within the system.

Listing 2.2: user

```
{
  "user": {
    "uuid": "jshdjkdh...",
    "context:": "resource",
    "email": "laszewski@gmail.com",
    "firstname": "Gregor",
    "lastname": "von Laszewski",
    "username": "gregor",
    "roles": ["admin", "user"]
  }
}
```

2.3. Organization

An important concept in many applications is the management of a roup of users in a virtual organization. This can be achieved through two concepts. First, it can be achieved while useing the profile and user resources itself as they contain the ability to manage multiple users as part of the REST interface. The second concept is to create a virtual organization that lists all users of this virtual organization. The third concept is to introduce groups and roles either as part of the user definition or as part of a simple list similar to the organization

Listing 2.3: user

```
{
  "organization": {
      "users": [objectid:user],
  }
}
```

2.4. Group/Role

Listing 2.4: role

```
{
    "group": {
      "name": "users",
      "description": "This group contains all
 ↪  users",
      "users": [objectid:user],
  }
}
```

Listing 2.5: role

```
{
    "role": {
      "name": "editor",
      "description": "This role contains all
↪   editors",
      "users": [objectid:user],
  }
}
```

3. DATA

Data for Big Data applications are delivered through data providers. They can be either local providers contributed by a user or distributed data providers that refer to data on the internet. At this time we focus on an elementary set of abstractions related to data providers that offer us to utilize variables, files, virtual data directories, data streams, and data filters.

Files are used to represent information collected within the context of classical files in an operating system.

Streams are services that offer the consumer a stream of data. Streams may allow the initiation of filters to reduce the amount of data requested by the consumer Stream Filters operate in streams or on files converting them to streams

Batch Filters operate on streams and on files while working in the background and delivering as output Files

Virtual directories and non-virtual directories are collection of files that organize them. For our initial purpose the distinction between virtual and non-virtual directories is non-essential and we will focus on abstracting all directories to be virtual. This could mean that the files are physically hosted on different disks. However, it is important to note that virtual data directories can hold more than files, they can also contain data streams and data filters.

3.1. Var

Listing 3.1: var

```
{
  "var": {
    "name": "string",
    "value": "string"
  }
}
```

3.2. Default

Listing 3.2: default

```
{
  "default": {
    "value": "string",
    "name": "string",
    "context": "string  - defines teh context
↪   of the default (user, cloud, ...)"
```

```
  }
}
```

3.3. File

A file is a computer resource allowing to store data that is being processed. The interface to a file provides the mechanism to appropriately locate a file in a distributed system. Identification include the name, and andpoint, the checksum and the size. Additional parameters such as the lasst access time could be stored also. As such the Interface only describes the location of the file

The **file** object has *name*, *endpoint* (location), *size* in GB, MB, Byte, *checksum* for integrity check, and last *accessed* timestamp.

Listing 3.3: file

```
{
    "file": {
        "name": "report.dat",
        "endpoint":
↪   "file://gregor@machine.edu:/data/report.dat",
        "checksum":
↪  {"md5":"8c324f12047dc2254b74031b8f029ad0"},
        "accessed": "1.1.2017:05:00:00:EST",
        "created": "1.1.2017:05:00:00:EST",
        "modified": "1.1.2017:05:00:00:EST",
        "size": ["GB", "Byte"]
  }
}
```

3.4. File Alias

A file could have one alias or even multiple ones.

Listing 3.4: file alias

```
{
    "file_alias": {
        "alias": "report-alias.dat",
        "name": "report.dat"
  }
}
```

3.5. Replica

In many distributed systems, it is of importance that a file can be replicated among different systems in order to provide faster access. It is important to provide a mechanism that allows to trace the pedigree of the file while pointing to its original source

Listing 3.5: replica

```
{
  "replica": {
    "name": "replica_report.dat",
    "replica": "report.dat",
    "endpoint":
↪   "file://gregor@machine.edu:/data/replica_report.dat",
    "checksum": {
        "md5":
↪   "8c324f12047dc2254b74031b8f029ad0"
    },
    "accessed": "1.1.2017:05:00:00:EST",
    "size": [
      "GB",
      "Byte"
    ]
  }
}
```

3.6. Virtual Directory

A collection of files or replicas. A virtual directory can contain an number of entities cincluding files, streams, and other virtual directories as part of a collection. The element in the collection can either be defined by uuid or by name.

Listing 3.6: virtual directory

```
{
  "virtual_directory": {
    "name": "data",
    "endpoint": "http://.../data/",
    "protocol": "http",
    "collection": [
      "report.dat",
      "file2"
    ]
  }
}
```

3.7. Database

A **database** could have a name, an *endpoint* (e.g., host:port), and protocol used (e.g., SQL, mongo, etc.).

Listing 3.7: database

```
{
  "database": {
    "name": "data",
    "endpoint": "http://.../data/",
    "protocol": "mongo"
  }
}
```

3.8. Stream

<Description>
Stream source
Stream subscriber
Values: Rate Limit Redirect (virtual/alias) Pipe Filters
Functions: list Subscribe Unsubscribe Get

4. IAAS

4.1. Openstack

4.1.1. Openstack Flavor

Listing 4.1: openstack flavor

```
{
  "openstack_flavor": {
    "os_flv_disabled": "string",
    "uuid": "string",
    "os_flv_ext_data": "string",
    "ram": "string",
    "os_flavor_acces": "string",
    "vcpus": "string",
    "swap": "string",
    "rxtx_factor": "string",
    "disk": "string"
  }
}
```

4.1.2. Openstack Image

Listing 4.2: openstack image

```
{
  "openstack_image": {
    "status": "string",
    "username": "string",
    "updated": "string",
    "uuid": "string",
    "created": "string",
    "minDisk": "string",
    "progress": "string",
    "minRam": "string",
    "os_image_size": "string",
    "metadata": {
      "image_location": "string",
      "image_state": "string",
      "description": "string",
      "kernel_id": "string",
      "instance_type_id": "string",
      "ramdisk_id": "string",
      "instance_type_name": "string",
      "instance_type_rxtx_factor": "string",
      "instance_type_vcpus": "string",
      "user_id": "string",
      "base_image_ref": "string",
      "instance_uuid": "string",
      "instance_type_memory_mb": "string",
      "instance_type_swap": "string",
      "image_type": "string",
      "instance_type_ephemeral_gb": "string",
      "instance_type_root_gb": "string",
      "network_allocated": "string",
      "instance_type_flavorid": "string",
      "owner_id": "string"
    }
  }
}
```

4.1.3. Openstack Vm

Listing 4.3: openstack vm

```
{
  "openstack_vm": {
    "username": "string",
    "vm_state": "string",
    "updated": "string",
    "hostId": "string",
    "availability_zone": "string",
    "terminated_at": "string",
    "image": "string",
    "floating_ip": "string",
    "diskConfig": "string",
    "key": "string",
    "flavor__id": "string",
    "user_id": "string",
    "flavor": "string",
    "static_ip": "string",
    "security_groups": "string",
    "volumes_attached": "string",
    "task_state": "string",
    "group": "string",
    "uuid": "string",
    "created": "string",
    "tenant_id": "string",
    "accessIPv4": "string",
    "accessIPv6": "string",
    "status": "string",
    "power_state": "string",
    "progress": "string",
    "image__id": "string",
    "launched_at": "string",
    "config_drive": "string"
  }
}
```

4.2. Azure

4.2.1. *Azure Size*

The size description of an azure vm

Listing 4.4: azure-size

```
{
  "azure-size": {
    "_uuid": "None",
    "name": "D14 Faster Compute Instance",
    "extra": {
      "cores": 16,
      "max_data_disks": 32
    },
    "price": 1.6261,
    "ram": 114688,
    "driver": "libcloud",
    "bandwidth": "None",
    "disk": 127,
    "id": "Standard_D14"
  }
}
```

4.2.2. *Azure Image*

Listing 4.5: azure-image

```
{
  "_uuid": "None",
  "driver": "libcloud",
  "extra": {
    "affinity_group": "",
    "category":  "Public",
    "description":
      "Linux VM image with
    ↪ coreclr-x64-beta5-11624 installed to
    ↪ /opt/dnx. This image is based on Ubuntu
    ↪ 14.04 LTS, with prerequisites of CoreCLR
    ↪ installed. It also contains PartsUnlimited
    ↪ demo app which runs on the installed
    ↪ coreclr. The demo app is installed to
    ↪ /opt/demo. To run the demo, please type
    ↪ the command /opt/demo/Kestrel in a
    ↪ terminal window. The website is listening
    ↪ on port 5004. Please enable or map a
    ↪ endpoint of HTTP port 5004 for your azure
    ↪ VM.",
    "location": "East Asia;Southeast
    ↪ Asia;Australia East;Australia
    ↪ Southeast;Brazil South;North Europe;West
    ↪ Europe;Japan East;Japan West;Central
    ↪ US;East US;East US 2; North Central
    ↪ US;South Central US;West US",
    "media_link": "",
    "os": "Linux",
    "vm_image": "False"
  },
  "id": "03f55de797f546a1b29d1....",
  "name": "CoreCLR x64 Beta5 (11624) with
  ↪ PartsUnlimited Demo App on Ubuntu Server
  ↪ 14.04 LTS"
}
```

4.2.3. *Azure Vm*

An Azure virtual machine

Listing 4.6: azure-vm

```
{
  "vm-azure": {
    "username": "string",
    "status": "string",
    "deployment_slot": "string",
    "cloud_service": "string",
    "image": "string",
    "floating_ip": "string",
    "image_name": "string",
    "key": "string",
    "flavor": "string",
    "resource_location": "string",
    "disk_name": "string",
    "private_ips": "string",
    "group": "string",
    "uuid": "string",
    "dns_name": "string",
    "instance_size": "string",
    "instance_name": "string",
    "public_ips": "string",
    "media_link": "string"
  }
}
```

5. HPC

5.1. Batch Job

Listing 5.1: batchjob

```
{
  "batchjob": {
    "output_file": "string",
    "group": "string",
    "job_id": "string",
    "script": "string, the batch job script",
    "cmd": "string, executes the cmd, if None
 ↪  path is used",
    "queue": "string",
    "cluster": "string",
    "time": "string",
    "path": "string, path of the batchjob, if
 ↪  non cmd is used",
    "nodes": "string",
    "dir": "string"
  }
}
```

6. VIRTUAL CLUSTER

6.1. Cluster

The cluster object has name, label, endpoint and provider. The *endpoint* defines.... The *provider* defines the nature of the cluster, e.g., its a virtual cluster on an openstack cloud, or from AWS, or a bare-metal cluster.

Listing 6.1: cluster

```
{
  "cluster": {
    "label": "c0",
    "endpoint": {
      "passwd": "secret",
      "url": "https"
    },
    "name": "myCLuster",
    "provider": [
      "openstack",
      "aws",
      "azure",
      "eucalyptus"
    ]
  }
}
```

6.2. Compute Resource

An important concept for big data analysis it the representation of a compute resource on which we execute the analysis. We define a compute resource by name and by endpoint. A compute resource is an abstract concept and can be instantiated through virtual machines, containers, or bare metal resources. This is defined by the “kind” of the compute resource

compute_resource object has attribute *endpoint* which specifies ... The *kind* could be *baremetal* or *VC*.

Listing 6.2: compute resource

```
{
  "compute_resource": {
    "name": "Compute1",
    "endpoint": "http://.../cluster/",
    "kind": "baremetal"
  }
}
```

6.3. Computer

This defines a **computer** object. A computer has name, label, IP address. It also listed the relevant specs such as memory, disk size, etc.

Listing 6.3: computer

```
{
  "computer": {
    "ip": "127.0.0.1",
    "name": "myComputer",
    "memoryGB": 16,
    "label": "server-001"
  }
}
```

6.4. node

A node is composed of multiple components:

1. Metadata such as the `name` or `owner`.
2. Physical properties such as `cores` or `memory`.
3. Configuration guidance such as `create_external_ip`, `security_groups`, or `users`.

The metadata is associated with the node on the provider end (if supported) as well as in the database. Certain parts of the metadata (such as `owner`) can be used to implement access control. Physical properties are relevant for the initial allocation of the node. Other configuration parameters control and further provisioning.

In the above, after allocation, the node is configured with a user called `hello` who is part of the `wheel` group whose account can be accessed with several SSH identities whose public keys are provided (in `authorized_keys`).

Additionally, three ssh keys are generated on the node for the `hello` user. The first uses the `ed25519` cryptographic method with a password read in from a GPG-encrypted file on the Command and Control node. The second is a 4098-bit RSA key also password-protected from the GPG-encrypted file. The third key is copied to the remote node from an encrypted file on the Command and Control node.

This definition also provides a security group to control access to the node from the wide-area-network. In this case all ingress and egress TCP and UDP traffic is allowed provided they are to ports 22 (SSH), 443 (SSL), and 80 and 8080 (web).

Listing 6.4: node-new

```
{
  "node_new": {
    "authorized_keys": [
        "ssh-rsa AAAA...",
        "ssh-ed25519 AAAA...",
        "...etc"
      ],
      "name": "example-001",
      "external_ip": "",
      "loginuser": "root",
      "create_external_ip": true,
      "internal_ip": "",
      "memory": 2048,
      "owner": "",
      "cores": 2,
      "users": {
        "name": "hello",
        "groups": [
          "wheel"
        ]
      },
      "disk": 80,
      "security_groups": [
        {
          "ingress": "0.0.0.0/32",
          "egress": "0.0.0.0/32",
          "ports": [
            22,
            443,
            80,
            8080
          ],
          "protocols": [
            "tcp",
            "udp"
          ]
        }
      ],
      "ssh_keys": [
        {
          "to": ".ssh/id_rsa",
          "password": {
            "decrypt": "gpg",
            "from": "yaml",
            "file": "secrets.yml.gpg",
            "key": "users.hello.ssh[0]"
          },
          "method": "ed25519",
          "ssh_keygen": true
        },
        {
          "to": ".ssh/testing",
          "password": {
            "decrypt": "gpg",
            "from": "yaml",
            "file": "secrets.yml.gpg",
            "key": "users.hello.ssh[1]"
          },
          "bits": 4098,
          "method": "rsa",
          "ssh_keygen": true
        },
        {
          "decrypt": "gpg",
          "from":
   ↪    "secrets/ssh/hello/copied.gpg",
          "ssh_keygen": false,
          "to": ".ssh/copied"
        }
      ]
    }
  }
```

6.5. Virtual Cluster

A virtual cluster is an agglomeration of virtual compute nodes that constitute the cluster. Nodes can be assembled to be baremetal, virtual machines, and containers. A virtual cluster contains a number of virtual compute nodes.

Listing 6.5: virtual cluster

```
{
  "virtual_cluster": {
    "name": "myvc",
    "frontend": "objectid:virtual_machine",
    "nodes": [
      "objectid:virtual_machine"
    ]
  }
}
```

6.6. Virtual Compute node

Listing 6.6: virtual compute node

```
{
  "virtual_compute_node": {
    "name": "data",
    "endpoint": "http://.../cluster/",
    "metadata": {
      "experiment": "exp-001"
    },
    "image": "Ubuntu-16.04",
    "ip": [
      "TBD"
    ],
    "flavor": "TBD",
    "status": "TBD"
  }
}
```

6.7. Virtual Machine

Virtual Machine Virtual machines are an emulation of a computer system. We are maintaining a very basic set of information. It is expected that through the endpoint the virtual machine can be introspected and more detailed information can be retrieved.

Listing 6.7: virtual machine

```
{
  "virtual_machine" :{
    "name": "vm1",
    "ncpu": 2,
    "RAM": "4G",
      "disk": "40G",
      "nics": ["objectid:nic"
      ],
      "OS": "Ubuntu-16.04",
      "loginuser": "ubuntu",
      "status": "active",
      "metadata":{
      },
      "authorized_keys": [
        "objectid:sshkey"
      ]
    }
  }
```

6.8. Mesos

Listing 6.8: mesos

```
{
  "mesos-docker": {
    "instances": 1,
    "container": {
      "docker": {
        "credential": {
          "secret": "my-secret",
          "principal": "my-principal"
        },
        "image": "mesosphere/inky"
      },
      "type": "MESOS"
    },
    "mem": 16.0,
    "args": [
      "argument"
    ],
    "cpus": 0.2,
    "id": "mesos-docker"
  }
}
```

7. CONTAINERS

7.1. Container

This defines **container** object.

Listing 7.1: container

```
{
    "container": {
        "name": "container1",
        "endpoint": "http://.../container/",
        "ip": "127.0.0.1",
        "label": "server-001",
        "memoryGB": 16
    }
}
```

7.2. Kubernetes

Listing 7.2: kubernetes

```
{
  "kubernetes": {
    "kind": "List",
    "items": [
      {
        "kind": "None",
        "metadata": {
          "name": "127.0.0.1"
        },
        "status": {
          "capacity": {
            "cpu": "4"
          },
          "addresses": [
            {
              "type": "LegacyHostIP",
              "address": "127.0.0.1"
            }
          ]
        }
      },
      {
        "kind": "None",
        "metadata": {
          "name": "127.0.0.2"
        },
        "status": {
          "capacity": {
            "cpu": "8"
          },
          "addresses": [
            {
              "type": "LegacyHostIP",
              "address": "127.0.0.2"
            },
            {
              "type": "another",
              "address": "127.0.0.3"
            }
          ]
        }
      }
    ],
    "users": [
      {
        "name": "myself",
        "user": "gregor"
      },
      {
        "name": "e2e",
        "user": {
          "username": "admin",
          "password": "secret"
        }
      }
    ]
  }
}
```

8. DEPLOYMENT

8.1. Deployment

A **deployment** consists of the resource *cluster*, the location *provider*, e.g., AWS, OpenStack, etc., and software *stack* to be deployed (e.g., hadoop, spark).

Listing 8.1: deployment

```
{
    "deployment": {
        "cluster": [{ "name": "myCluster"},
                    { "id" : "cm-0001"}
                   ],
        "stack": {
            "layers": [
                "zookeeper",
                "hadoop",
                "spark",
                "postgresql"
            ],
            "parameters": {
                "hadoop": {
↪   "zookeeper.quorum": [ "IP", "IP", "IP"]
                        }
            }
        }
    }
}
```

9. MAPREDUCE

9.1. Hadoop

A **hadoop** definition defines which *deployer* to be used, the *parameters* of the deployment, and the system packages as *requires*. For each requirement, it could have attributes such as the library origin, version, etc.

Listing 9.1: hadoop

```
{
  "hadoop": {
    "deployers": {
      "ansible":
↪   "git://github.com/cloudmesh_roles/hadoop"
    },
    "requires": {
      "java": {
        "implementation": "OpenJDK",
        "version": "1.8",
        "zookeeper": "TBD",
        "supervisord": "TBD"
      }
    },
    "parameters": {
      "num_resourcemanagers": 1,
      "num_namenodes": 1,
      "use_yarn": false,
      "use_hdfs": true,
      "num_datanodes": 1,
      "num_historyservers": 1,
      "num_journalnodes": 1
    }
  }
}
```

9.2. Mapreduce

This defines a **mapreduce** deployment with its layered components.

Listing 9.2: mapreduce

```
{
  "mapreduce": {
      "layers": [
          "hadoop"
      ],
      "hdfs_datanode": "IP ADDRESS",
      "java": {
          "platform": "OpenJDK",
          "version": "1.8"
      },
      "supervisord": "",
      "hdfs_namenode": "IP ADDRESS",
      "zookeeper": "IP ADDRESS",
      "yarn_historyserver": "IP ADDRESS",
      "hdfs_journalnode": "IP ADDRESS",
      "yarn_resourcemanager": "IP ADDRESS"
  }
}
```

10. SECURITY

10.1. Key

Listing 10.1: key

```
{
  "sshkey": {
    "comment": "string",
    "source": "string",
    "uri": "string",
    "value": "ssh-rsa",
    "fingerprint": "string, unique"
  }
}
```

11. MICROSERVICE

11.1. Microservice

introduce registry we can register many things to it latency provide example on how to use each of them, not just the object definition example

necessity of local direct attached storage. Mimd model to storage Kubernetis, mesos can not spin up ? Takes time to spin them up and coordinate them. While setting up environment takes more thsn using the microservice, so we must make sure that the micorservices are used sufficiently to offset spinup cost.

limitation of resource capacity such as networking.

Benchmarking to find out thing about service level agreement to access the

A system could be composed of from various microservices, and this defines each of them.

Listing 11.1: microservice

```
{
"microservice" :{
    "name": "ms1",
    "endpoint": "http://.../ms/",
    "function": "microservice spec"
    }
}
```

11.2. Reservation

Listing 11.2: reservation

```
{
  "reservation": {
    "hosts": "string",
    "description": "string",
    "start_time": [
      "date",
      "time"
    ],
    "end_time": [
      "date",
      "time"
    ]
  }
}
```

12. NETWORK

We are looking for volunteers to contribute here.

A. SCHEMA COMMAND

B. SCHEMA

TBD

C. CONTRIBUTING

We invite you to contribute to this paper and its discussion to improve it. Improvements can be done with pull requests. We suggest you do *small* individual changes to a single section and object rather than large changes as this allows us to integrate the changes individually and comment on your contribution via github.

Once contributed we will appropriately acknoledge you either as contributor or author. Please discuss with us how we best acknowledge you.

D. USING THE CLOUDMESH REST SERVICE

Components are written as YAML markup in files in the `resources/samples` directory.

For example:

Listing D.1: profile

```
{
  "profile": {
    "description": "The Profile of a user",
    "uuid": "jshdjkdh...",
    "context:": "resource",
    "email": "laszewski@gmail.com",
    "firstname": "Gregor",
    "lastname": "von Laszewski",
    "username": "gregor"
  }
}
```

D.1. Element Definition

Each resource should have a `description` entry to act as documentation. The documentation should be formated as reStructuredText. For example:

D.2. Yaml

```
entry = yaml.read('''
profile:
  description: |
    A user profile that specifies general information
    about the user
  email: laszewski@gmail.com, required
  firtsname: Gregor, required
  lastname: von Laszewski, required
  height: 180
'''}
```

D.3. Cerberus

```
schema = {
'profile': {
  'description': {'type': 'string'}
  'email':       {'type': 'string', 'required': True}
  'firtsname':   {'type': 'string', 'required': True}
  'lastname':    {'type': 'string', 'required': True}
  'height':      {'type': 'float'}
}
```

E. MONGOENGINE

```
class profile(Document):
    description = StringField()
    email = EmailField(required=True)
    firstname = StringField(required=True)
    lastname = StringField(required=True)
    height = FloatField(max_length=50)
```

F. CLOUDMESH NOTATION

```
profile:
    description: string
    email: email, required
    firstname: string, required
    lastname: string, required
    height: flat, max=10

proposed command

cms schema FILENAME --format=mongo -o OUTPUT
cms schema FILENAME --format=cerberus -o OUTPUT
cms schema FILENAME --format=yaml -o OUTPUT

  reads FILENAME in cloudmesh notation and returns format


cms schema FILENAME --input=evegenie -o OUTPUT
   reads eavegene example and create settings for eve
```

F.1. Defining Elements for the REST Service

To manage a large number of elements defined in our REST service easily, we manage them trhough definitions in yaml files. To generate the appropriate settings file for the rest service, we can use teh following command:

```
cms admin elements <directory> <out.json>
```

where

- `<directory>`: directory where the YAML definitions reside
- `<out.json>`: path to the combined definition

For example, to generate a file called all.json that integrates all yml objects defined in the directory `resources/samples` you can use the following command:

```
cms elements resources/samples all.json
```

F.2. DOIT

cms schema spec2tex resources/specification resources/tex

F.3. Generating service

With evegenie installed, the generated JSON file from the above step is processed to create the stub REST service definitions.

G. ABC

README.rst

H. CLOUDMESH REST

H.1. Prerequistis

- mongo instalation
- eve instalation
- cloudmesh cmd5
- cloudmesh rest

H.1.1. Install Mongo on OSX

```
brew update
brew install mongodb

# brew install mongodb --with-openssl
```

H.1.2. Install Mongo on OSX

ASSIGNMET TO STUDENTS, PROVIDE PULL REQUEST WITH INSTRUCTIONS

H.2. Introduction

With the cloudmesh REST framework it is easy to create REST services while defining the resources in the service easily with examples. The service is based on eve and the examples are defined in yml to be converted to json and from json with evegenie into a valid eve settings file.

Thus oyou can eother wite your examples in yaml or in json. The resources are individually specified in a directory. The directory can contain multiple resource files. We recomment that for each resource you define your own file. Conversion of the specifications can be achieved with the schema command.

H.3. Yaml Specification

Let us first introduce you to a yaml specification. Let us assume that your yaml file is called profile.yaml and located in a directory called 'example':

```
profile:
  description: The Profile of a user
  email: laszewski@gmail.com
  firstname: Gregor
  lastname: von Laszewski
  username: gregor
```

As eve takes json objects as default we need to convert it first to json. This is achieved wih:

```
cd example
cms schema convert profile.yml profile.json
```

This will provide the json file profile.json as Listed in the next section

H.4. Json Specification

A valid json resource specification looks like this:

```
{
  "profile": {
    "description": "The Profile of a user",
    "email": "laszewski@gmail.com",
    "firstname": "Gregor",
    "lastname": "von Laszewski",
    "username": "gregor"
  }
}
```

H.5. Conversion to Eve Settings

The json files in the ~/sample directory need now to be converted to a valid eve schema. This is achieved with tow commands. First, we must concatenate all json specified resource examples into a single json file. We do this with:

```
cms schema cat . all.json
```

As we assume you are in the samples directory, we use a . for the current location of the directory that containes the samples. Next, we need to convert it to the settings file. THis can be achieved with the convert program when you specify a json file:

```
cms schema convert all.json
```

THe result will be a eve configuration file that you can use to start an eve service. The file is called all.settings.py

H.5.1. Managing Mongo

Next you need to start the mongo service with

```
cms admin mongo start
```

You can look at the status and information about the service with :

```
cms admin mongo info
cms admin mongo status
```

If you need to stop the service you can use:

```
cms admin mongo stop
```

H.5.2. Manageing Eve

Now it is time to start the REST service. THis is done in a separate window with the following commands:

```
cms admin settings all.settings.json
cms admin rest start
```

The first command coppies the settings file to

~/cloudmesh/eve/settings.py

This file is than used by the start action to start the eve service. Please make sure that you execute this command in a separate window, as for debugging purpposes you will be able to monitor this way interactions with this service

Testing - OLD ^^^^^^^ :

```
make setup    # install mongo and eve
make install  # installs the code and integrates it into cmd5
make deploy
make test
```

classes lessons rest.rst

I. REST WITH EVE

I.1. Overview of REST

REST stands for REpresentational State Transfer. REST is an architecture style for designing networked applications. It is based on stateless, client-server, cacheable communications protocol. Although not based on http, in most cases, the HTTP protocol is used. In contrast to what some others write or say, REST is not a *standard*.

RESTful applications use HTTP requests to:

- post data: while creating and/or updating it,
- read data: while making queries, and
- delete data.

Hence REST uses HTTP for the four CRUD operations:

- Create
- Read
- Update
- Delete

As part of the HTTP protocol we have methods such as GET, PUT, POST, and DELETE. These methods can than be used to implement a REST service. As REST introduces collections and items we need to implement the CRUD functions for them. The semantics is explained in the Table illustrationg how to implement them with HTTP methods.

Source: https://en.wikipedia.org/wiki/Representational_state_transfer

I.2. REST and eve

Now that we have outlined the basic functionality that we need, we lke to introduce you to Eve that makes this process rather trivial. We will provide you with an implementation example that showcases that we can create REST services without writing a single line of code. The code for this is located at https://github.com/cloudmesh/rest

This code will have a master branch but will also have a dev branch in which we will add gradually more objects. Objects in the dev branch will include:

- virtual directories
- virtual clusters
- job sequences
- inventories

;You may want to check our active development work in the dev branch. However for the purpose of this class the master branch will be sufficient.

I.2.1. Installation

First we havt to install mongodb. The instalation will depend on your operating system. For the use of the rest service it is not important to integrate mongodb into the system upon reboot, which is focus of many online documents. However, for us it is better if we can start and stop the services explicitly for now.

On ubuntu, you need to do the following steps:

```
TO BE CONTRIBUTED BY THE STUDENTS OF THE CLASS as homework
```

On windows 10, you need to do the following steps:

```
TO BE CONTRIBUTED BY THE STUDENTS OF THE CLASS as homework, if
elect Windows 10. YOu could be using the online documentation
provided by starting it on Windows, or rinning it in a docker c
```

On OSX you can use homebrew and install it with:

```
brew update
brew install mongodb
```

In future we may want to add ssl authentication in which case you may
need to install it as follows:

brew install mongodb –with-openssl

I.2.2. Starting the service

We have provided a convenient Makefile that currently only works for OSX. It will be easy for you to adapt it to Linux. Certainly you can look at the targes in the makefile and replicate them one by one. Improtaht targest are deploy and test.

When using the makefile you can start the services with:

```
make deploy
```

IT will start two terminals. IN one you will see the mongo service, in the other you will see the eve service. The eve service will take a file called sample.settings.py that is base on sample.json for the start of the eve service. The mongo servide is configured in suc a wahy that it only accepts incimming connections from the local host which will be sufficent fpr our case. The mongo data is written into the $USER/.cloudmesh directory, so make sure it exists.

To test the services you can say:

```
make test
```

YOu will se a number of json text been written to the screen.

I.3. Creating your own objects

The example demonstrated how easy it is to create a mongodb and an eve rest service. Now lets use this example to creat your own. FOr this we have modified a tool called evegenie to install it onto your system.

The original documentation for evegenie is located at:

- http://evegenie.readthedocs.io/en/latest/

However, we have improved evegenie while providing a commandline tool based on it. The improved code is located at:

- https://github.com/cloudmesh/evegenie

You clone it and install on your system as follows:

```
cd ~/github
git clone https://github.com/cloudmesh/evegenie
cd evegenie
python setup.py install
pip install .
```

This shoudl install in your system evegenie. YOu can verify this by typing:

```
which evegenie
```

If you see the path evegenie is installed. With evegenie installed its usaage is simple:

```
$ evegenie

Usage:
    evegenie --help
    evegenie FILENAME
```

It takes a json file as input and writes out a settings file for the use in eve. Lets assume the file is called sample.json, than the settings file will be called sample.settings.py. Having the evegenie programm will allow us to generate the settings files easily. You can include them into your project and leverage the Makefile targets to start the services in your project. In case you generate new objects, make sure you rerun evegenie, kill all previous windows in whcih you run eve and mongo and restart. In case of changes to objects that you have designed and run previously, you need to also delete the mongod database.

I.4. Towards cmd5 extensions to manage eve and mongo

Naturally it is of advantage to have in cms administration commands to manage mongo and eve from cmd instead of targets in the Makefile. Hence, we **propose** that the class develops such an extension. We will create in the repository the extension called admin and hobe that students through collaborative work and pull requests complete such an admin command.

The proposed command is located at:

- https://github.com/cloudmesh/rest/blob/master/cloudmesh/ext/command/admin.py

It will be up to the class to implement such a command. Please coordinate with each other.

The implementation based on what we provided in the Make file seems straight forward. A great extensinion is to load the objects definitions or eve e.g. settings.py not from the class, but forma place in .cloudmesh. I propose to place the file at:

```
.cloudmesh/db/settings.py
```

the location of this file is used whne the Service class is initialized with None. Prior to starting the service the file needs to be copied there. This could be achived with a set commad. classes lesson python cmd5.rst

J. CMD5

CMD is a very useful package in python to create command line shells. However it does not allow the dynamic integration of newly defined commands. Furthermore, addition to cmd need to be done within the same source tree. To simplify developping commands by a number of people and to have a dynamic plugin mechnism, we developed cmd5. It is a rewrite on our ealier effords in cloudmesh and cmd3.

J.1. Resources

The source code for cmd5 is located in github:

- https://github.com/cloudmesh/cmd5

Installation from source ———————

We recommend that you use a virtualenv either with virtualenv or pyenv. This can be either achieved vor virtualenv with:

```
virtualenv ~/ENV2
```

or for pyenv, with:

```
pyenev virtualenv 2.7.13 ENV2
```

Now you need to get two source directories. We assume yo place them in ~/github:

```
mkdir ~/github
cd ~/github

git clone https://github.com/cloudmesh/common.git
git clone https://github.com/cloudmesh/cmd5.git
git clone https://github.com/cloudmesh/extbar.git

cd ~/github/common
python setup.py install
pip install .

cd ~/github/cmd5
python setup.py install
pip install .

cd ~/github/extbar
python setup.py install
pip install .
```

The cmd5 repository contains the shell, while the extbar directory contains the sample to add the dynamic commands foo and bar.

J.2. Execution

To run the shell you can activate it with the cms command. cms stands for cloudmesh shell:

```
(ENV2) $ cms
```

It will print the banner and enter the shell:

```
+-------------------------------------------------------+
|   ____ _                 _                     _      |
|  / ___| | ___  _   _  __| |_ __ ___   ___  ___| |__   |
| | |   | |/ _ \| | | |/ _‘ | ’_ ‘ _ \ / _ \/ __| ’_ \  |
| | |___| | (_) | |_| | (_| | | | | | |  __/\__ \ | | | |
|  \____|_|\___/ \__,_|\__,_|_| |_| |_|\___||___/_| |_| |
+-------------------------------------------------------+
|                  Cloudmesh CMD5 Shell                 |
+-------------------------------------------------------+

cms>
```

To see the list of commands you can say

cms> help

To see the manula page for a specific command, please use:

```
help COMMANDNAME
```

J.3. Create your own Extension

One of the most important features of CMD5 is its ability to extend it with new commands. This is done via packaged name spaces. This is defined in the setup.py file of your enhancement. The best way to create an enhancement is to take a look at the code in

- https://github.com/cloudmesh/extbar.git

Simply copy the code and modify the bar and foo commands to fit yor needs.

make sure you are not copying the .git directory. Thus we recommend that you copy it explicitly file by file or directory by directory

It is important that all objects are defined in the command itself and that no global variables be use in order to allow each shell command to stand alone. Naturally you should develop API libraries outside of the cloudmesh shell command and reuse them in order to keep the command code as small as possible. We place the command in:

```
cloudmsesh/ext/command/COMMANDNAME.py
```

An example for the bar command is presented at:

- https://github.com/cloudmesh/extbar/blob/master/cloudmesh/ext/command/bar.py

It shows how simple the command definition is (bar.py):

```
from __future__ import print_function
from cloudmesh.shell.command import command
from cloudmesh.shell.command import PluginCommand

class BarCommand(PluginCommand):

    @command
    def do_bar(self, args, arguments):
        """
        ::
          Usage:
                command -f FILE
                command FILE
                command list
          This command does some useful things.
          Arguments:
              FILE   a file name
          Options:
              -f      specify the file
        """
        print(arguments)
```

An important difference to other CMD solutions is that our commands can leverage (besides the standrad definition), docopts as a way to define the manual page. This allows us to use arguments as dict and use simple if conditions to interpret the command. Using docopts has the advantage that contributors are forced to think about the command and its options and document them from the start. Previously we used not to use docopts and argparse was used. However we noticed that for some contributions the lead to commands that were either not properly documented or the developers delivered ambiguous commands that resulted in confusion and wrong ussage by the users. Hence, we do recommend that you use docopts.

The transformation is enabled by the @command decorator that takes also the manual page and creates a proper help message for the shell automatically. Thus there is no need to introduce a sepaarte help method as would normally be needed in CMD.

J.4. Excersise

CMD5.1: Install cmd5 on your computer.

CMD5.2: Write a new command with your firstname as the command name.

CMD5.3: Write a new command and experiment with docopt syntax and argument interpretation of the dict with if conditions.

CMD5.4: If you have useful extensions that you like us to add by default, please work with us.