MaestroNG Documentation

Release 0.7.5

Maxime Petazzoni

Oct 23, 2020

Table of contents

CHAPTER 1

Introduction

MaestroNG is an orchestrator of Docker-based, multi-hosts environments. Working from a description of your environment, MaestroNG offers service-level and container-level controls that rely on Maestro's understanding of your declared service dependencies, and placement of your containers in your fleet of hosts.

Maestro aims at being simple to use whether you are controlling a few containers in a local virtual machine, or hundreds of containers spread across as many hosts.

The orchestration features of Maestro obviously rely on the collaboration of the Docker containers that you are controlling with Maestro. Maestro basically takes care of two things:

1. Controlling the start (and stop) order of services during environment bring up and tear down according to the defined dependencies between services.
2. Passing extra environment variables to each container to pass all the information it may need to operate in that environment, in particular information about its dependencies.

The most common way to integrate your application with Maestro is to make your container's entrypoint a simple Python init script that acts as the glue between Maestro, the information that it passes through the container's environment, and your application. To make this easier to write and put together, Maestro provides a set of *Maestro guest functions* that know how to grok this environment information.

CHAPTER 2

User Guide

This part of the documentation focuses on step-by-step instructions for getting the most out of MaestroNG.

2.1 Usage

Once installed, Maestro is available both as a library through the `maestro` package and as an executable. To run Maestro, simply execute `maestro`. Note that if you didn't install Maestro system-wide, you can still run it with the same commands as long as your `PYTHONPATH` contains the path to your `maestro-ng` repository clone and using `python -m maestro ...`.

```
$ maestro -h
usage: maestro [-h] [-f FILE] [-v]
               {status,pull,start,stop,restart,logs,deptree} ...

Maestro, Docker container orchestrator.

positional arguments:
  {status,pull,start,stop,restart,logs,deptree}
    status              display container status
    pull                pull images from repository
    start               start services and containers
    stop                stop services and containers
    kill                kill services and containers
    restart             restart services and containers
    logs                show logs from a container
    deptree             show the dependency tree
    complete            shell auto-completion helper

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  read environment description from FILE (use - for
                        stdin, defaults to ./maestro.yaml)
  -v, --version         show program version and exit
```

You can then get help on each individual command with:

```
$ maestro start -h
usage: maestro start [-h] [-c LIMIT] [-d] [-i] [-r] [thing [thing ...]]

Start services and containers

positional arguments:
  thing                 container(s) or service(s) to display

optional arguments:
  -h, --help            show this help message and exit
  -c LIMIT, --concurrency LIMIT
                        limit how many containers can be acted on at the same
                        time to LIMIT
  -d, --with-dependencies
                        include dependencies
  -i, --ignore-dependencies
                        ignore dependency order
  -r, --refresh-images  force refresh of container images from registry
```

By default, Maestro will read the environment description configuration from the `maestro.yaml` file in the current directory. You can override this with the `-f` flag to specify the path to the environment configuration file. Additionally, you can use `-` to read the configuration from `stdin`. The following commands are identical:

```
$ maestro status
$ maestro -f maestro.yaml status
$ maestro -f - status < maestro.yaml
```

The first positional argument is a command you want Maestro to execute. The available commands are `status`, `start`, `stop`, `restart`, `logs` and `deptree`. They should all be self-explanatory.

Most commands operate on one or more "things", which can be services or instances, by name. When passing service names, Maestro will automatically expand those to their corresponding list of instances. The `logs` command is the only one that operates on strictly one container instance.

2.1.1 Impact of defined dependencies on orchestration order

One of the main features of Maestro is its understand of dependencies between services. When Maestro carries out an orchestration action, dependencies are always considered unless the `-i | --ignore-dependencies` flag is passed.

But Maestro will only respect the dependencies to other services and containers that the current orchestration action includes. If you want Maestro to automatically include the dependencies of the services or containers you want to act on in the orchestration that will be carried out, you must pass the `-d | --with-dependencies` flag!

For example, assuming we have two services, ZooKeeper (`zookeeper`) and Kafka (`kafka`), and that Kafka depends on ZooKeeper:

```
# Starts Kafka and only Kafka:
$ maestro start kafka

# Starts ZooKeeper, then Kafka:
$ maestro start -d kafka
# Which is equivalent to:
$ maestro start kafka zookeeper
```

(continues on next page)

(continued from previous page)

```
# Starts ZooKeeper and Kafka at the same time (includes dependencies but
# ignores dependency order constraints):
$ maestro start -d -i kafka
```

2.2 Environment description

The environment is described using YAML. The description of the environment follows a specific, versioned *schema*. The default schema version, if not specified, is version 1. The most recent version of the schema, described by this documentation, is version 2. To declare what schema version to use for your YAML environment description file, use the following header:

```
__maestro:
  schema: 2
```

2.2.1 Structure

The environment is named, and composed of three main mandatory sections: *registries*, *ships*, and *services*.

1. The *registries* section defines authentication credentials that Maestro can use to pull Docker images for your *services*. If you only pull public images, this section may remain empty.
2. The *ships* section describes hosts that will run the Docker containers.
3. The *services* section defines which services make up the environment, the dependencies between them, and instances of your services that you want to run.

Here's the outline:

```
__maestro:
  schema: 2

name: demo
registries:
  # Auth credentials for each registry that needs them (see below)
ship_defaults:
  # defaults for some of the ship attributes (see below)
ships:
  # Ship definitions (see below)
services:
  # Service definitions (see below)
```

2.2.2 Metadata

The `__maestro` section contains information that helps Maestro understand your environment description. In particular, the `schema` version specifies what version of the YAML "schema" Maestro should use when parsing the environment description. This provides an easier upgrade path when Maestro introduces backwards incompatible changes.

2.2.3 Registries

The *registries* section defines the Docker registries that Maestro can pull images from and the authentication credentials needed to access them (see *Working with image registries*). For each registry, you must provide the full registry URL, a `username`, a `password`, and possibly `email`. For example:

```
registries:
  my-private-registry:
    registry: https://my-private-registry/v1/
    username: maestro
    password: secret
    email: maestro-robot@domain.com
```

2.2.4 Ship defaults

The ship defaults sections specify certain ship attribute defaults, like `timeout`, `docker_port`, `api_version` or `ssh_timeout`.

```
ship_defaults:
  timeout: 60
```

2.2.5 Ships

A *ship* is a host that runs your Docker containers. They have names (which don't need to match their DNS resolvable host name) and IP addressses/hostnames (`ip`). They may also define:

- `api_version`: The API version of the Docker daemon. If you set it to `auto`, the version is automatically retrieved from the Docker daemon and the latest available version is used.
- `docker_port`: A custom port, used if the Docker daemon doesn't listen on the default port of 2375.
- `endpoint`: The Docker daemon endpoint address. Override this if the address of the machine is not the one you want to use to interact with the Docker daemon running there (for example via a private network). Defaults to the ship's `ip` parameter.
- `ssh_tunnel`: An SSH tunnel to secure the communication with the target Docker daemon (especially if you don't want the Docker daemon to listen on anything else than `localhost`, and rely on SSH key-based authentication instead). Here again, if the `endpoint` parameter is specified, it will be used as the target host for the SSH connection.
- `socket_path`: If the Docker daemon is listening on a unix domain socket in the local filesystem, you can specify `socket_path` to connect to it directly. This is useful when the Docker daemon is running locally.

```
ships:
  vm1.ore1: {ip: c414.ore1.domain.com}
  vm2.ore2: {ip: c415.ore2.domain.com, docker_port: 4243}
  vm3.ore3:
    ip: c416.ore3.domain.com
    endpoint: c416.corp.domain.com
    docker_port: 4243
    ssh_tunnel:
      user: ops
      key: {{ env.HOME }}/.ssh/id_dsa
      port: 22 # That's the default
```

You can also connect to a Docker daemon secured by TLS. Note that if you want to use verification, you have to give the IP (or something that is resolvable inside the container) as IP, and the name in the server certificate as endpoint.

Not using verification works too (just don't mention `tls_verify` and `tls_ca_cert`), but a warning from inside `urllib3` will make Maestro's output unreadable.

In the example below, "docker1" is the CN in the server certificate. All certificates and keys have been created as explained in https://docs.docker.com/articles/https/

```
ships:
    docker1:
        ip: 172.17.42.1
        endpoint: docker1
        tls: true
        tls_verify: true
        tls_ca_cert: ca.pem
        tls_key: key.pem
        tls_cert: cert.pem
```

2.2.6 Services

Services have a name (used for commands that act on specific services instead of the whole environment and in dependency declarations), a Docker image (`image`), and a description of each instance of that service (under `instances`). Services may also define:

- `envfile`: Filename, or list of filenames, of Docker environment files that will apply to all of that service's instances. File names are relative to the Maestro environment YAML file's location;
- `env`: Environment variables that will apply to all of that service's instances. `env` values take precedence over the contents of ‘‘envfile‘‘s;
- `omit`: If `true`, excludes the service from non-specific actions (when Maestro is executed without a list of services or containers as arguments);
- `requires` and `wants_info`: Define hard and soft dependencies (see *Defining dependencies*);
- `lifecycle`: Service instances' lifecycle state checks, which Maestro uses to confirm a service instance correctly started or stopped (see *Lifecycle checks*);
- `limits`: Set container limits at service scope. All service instances would inherit these limits;
- `ports`: Set container ports at service scope. All service instances would inherit these ports;

Here's an example of a simple service with a single instance:

```
services:
  hello:
    image: ubuntu
    limits:
      memory: 10m
      cpu: 1
    ports:
      server: 4848
    envfile:
      - hello-base.env
      - hello-extras.env
    instances:
      hello1:
        ports:
```

(continues on next page)

(continued from previous page)

```
        client: 4242
      command: "while true ; do echo 'Hello, world!' | nc -l 0.0.0.0 4242 ; done"
```

2.2.7 Service instances

Each instance must, at minimum, define the *ship* its container will be placed on (by name). Additionally, each instance may define:

- `image`, to override the service-level image repository name, if needed (useful for canary deployments for example);
- `ports`, a dictionary of port mappings, as a map of `<port name>: <port or port mapping spec>` (see *Port mapping* for port spec syntax);
- `lifecycle`, for lifecycle state checks, which Maestro uses to confirm a service correctly started or stopped (see *Lifecycle checks*);
- `volumes`, for container volume mappings, as a map of `<source from host>: <destination in container>`. Each target can also be specified as a map `{target: <destination>, mode: <mode>}`. `mode` defaults to `rw` for read-write, but can be any combination of comma-separated mode flags, like `ro`, `Z` or `z,rw`;
- `container_volumes`, a path, or list of paths inside the container to be used as container-only volumes with no host bind-mount. This is mostly used for data-containers;
- `volumes_from`, a container or list of containers running on the same _ship_ to get volumes from. This is useful to get the volumes of a data-container into an application container;
- `envfile`: Filename, or list of filenames, of Docker environment files for this container. File names are relative to the Maestro environment YAML file's location;
- `env`, for environment variables, as a map of `<variable name>: <value>` (variables defined at the instance level override variables defined at the service level). `env` values take precedence over ‘‘envfiles‘‘s;
- `privileged`, a boolean specifying whether the container should run in privileged mode or not (defaults to `false`);
- `cap_add`, Linux capabilities to add to the container (see the documentation for docker run;
- `cap_drop`, Linux capabilities to drop from the container;
- `extra_hosts`, a map of custom hostnames to IP addresses that will be added to the `/etc/hosts` for the container. Example: `<hostname>: <ip address>`. You can also define extra hosts by reference to other *ships* defined in the Maestro environment with: `<hostname>: {ship: <ship-name>}`. Note that the ship *must* be defined with an IP address (as opposed to a FQDN) for this to work in the containers' host file;
- `stop_timeout`, the number of seconds Docker will wait between sending `SIGTERM` and `SIGKILL` (defaults to 10);
- `limits`:
  - `memory`, the memory limit of the container (in bytes, or with one of the `k`, `m` or `g` suffixes, also valid in uppercase);
  - `cpu`, the number of CPU shares (relative weight) allocated to the container;
  - `swap`, the swap limit of the container (in bytes, or with one of the `k`, `m` or `g` suffixes, also valid in uppercase);
- `log_driver`, one of the supported log drivers, e.g. syslog or json-file;

- `log_opt`, a set of key value pairs that provide additional logging parameters. E.g. the syslog-address to redirect syslog output to another address;
- `command`, to specify or override the command executed by the container;
- `net`, to specify the container's network mode (one of `bridge` – the default, `host`, `container:<name|id>` or `none` to disable networking altogether);
- `restart`, to specify the restart policy (see *Restart policy*);
- `dns`, to specify one (as a single IP address) or more DNS servers (as a list) to be declared inside the container;
- `security_opt`, to specify additional security options to customize container labels, apparmor profiles, etc.
- `ulimits`, to override the default ulimits for a container. You can either specify a single limit as an integer or soft/hard limits as a mapping.
- `username`, to set the name of the user under which the container's processes will run.
- `labels`, a list or a map (dictionary) of labels to set on the container.

For example:

```
services:
  zookeeper:
    image: zookeeper:3.4.5
    instances:
      zk-1:
        ship: vm1.ore1
        ports: {client: 2181, peer: 2888, leader_election: 3888}
        privileged: true
        volumes:
          /data/zookeeper: /var/lib/zookeeper
        limits:
          memory: 1g
          cpu: 2
        labels:
          - no-relocate
      zk-2:
        ship: vm2.ore1
        ports: {client: 2181, peer: 2888, leader_election: 3888}
        lifecycle:
          running: [{type: tcp, port: client}]
        volumes:
          /data/zookeeper: /var/lib/zookeeper
        limits:
          memory: 1g
          cpu: 2
        labels:
          - no-relocate
    lifecycle:
      running: [{type: tcp, port: client}]
  kafka:
    image: kafka:latest
    requires: [ zookeeper ]
    envfile: kafka.env
    instances:
      kafka-broker:
        ship: vm2.ore1
        ports: {broker: 9092}
        volumes:
```

(continues on next page)

(continued from previous page)

```
          /data/kafka: /var/lib/kafka
          /etc/locatime:
            target: /etc/localtime
            mode: ro
        env:
          BROKER_ID: 0
        stop_timeout: 2
        limits:
          memory: 5G
          swap: 200m
          cpu: 10
        dns: [ 8.8.8.8, 8.8.4.4 ]
        net: host
        restart:
          name: on-failure
          maximum_retry_count: 3
        ulimits:
          nproc: 65535
          nofile:
            soft: 1024
            hard: 1024
    lifecycle:
      running: [{type: tcp, port: broker}]
```

2.3 How Maestro orchestrates and service auto-configuration

The orchestration performed by Maestro is two-fold. The first part is providing a way for each container to learn about the environment they evolve into, to discover about their peers and/or the container instances of other services in their environment. The second part is by controlling the start/stop sequence of services and their containers, taking service dependencies into account.

With inspiration from Docker's _links_ feature, Maestro utilizes environment variables to pass information down to each container. Each container is guaranteed to get the following environment variables:

- `MAESTRO_ENVIRONMENT_NAME`: the name of the Maestro environment (as defined by the top-level `name` property on the environment).
- `DOCKER_IMAGE`: the full name of the image this container is started from.
- `DOCKER_TAG`: the tag of the image this container is started from.
- `SERVICE_NAME`: the friendly name of the service the container is an instance of. Note that it is possible to have multiple clusters of the same kind of application by giving them distinct friendly names.
- `CONTAINER_NAME`: the friendly name of the instance, which is also used as the name of the container itself. This will also be the visible hostname from inside the container.
- `CONTAINER_HOST_ADDRESS`: the external IP address of the host of the container. This can be used as the "advertised" address when services use dynamic service discovery techniques.

Then, for each container of each service that the container depends on (see *Defining dependencies*), a set of environment variables is added:

- `<SERVICE_NAME>_<CONTAINER_NAME>_HOST`: the external IP address of the host of the container, which is the address the application inside the container can be reached with accross the network.

- For each port declared by the dependent container, a `<SERVICE_NAME>_<CONTAINER_NAME>_<PORT_NAME>_PORT` environment variable, containing the external, addressable port number, is provided.

Each container of a service also gets these two variables for each instance of that service so it knows about its peers. It also gets the following variable for each port defined:

- `<SERVICE_NAME>_<CONTAINER_NAME>_<PORT_NAME>_INTERNAL_PORT`, containing the exposed (internal) port number that is, most likely, only reachable from inside the container and usually the port the application running in the container wants to bind to.

With all this information available in the container's environment, each container can then easily know about its surroundings and the other services it might need to talk to. It then becomes really easy to bridge the gap between the information Maestro provides to the container via its environment and the application you want to run inside the container.

You could, of course, have your application directly read the environment variables pushed in by Maestro. But that would tie your application logic to Maestro, a specific orchestration system; you do not want that. Instead, you can write a *startup script* that will inspect the environment and generate a configuration file for your application (or pass in command-line flags).

To make this easier, Maestro provides a set of helper functions available in its `maestro.guestutils` module. The recommended (or easiest) way to build this startup script is to write it in Python, and have the Maestro package installed in your container.

2.4 Defining dependencies

Services can depend on each other (circular dependencies are not supported though). This dependency tree instructs Maestro to start and stop the services in an order that will respect these dependencies. Dependent services are started before services that depend on them, and conversly leaves of the dependency tree are stopped before the services they depend on so that at no point in time a service may run without its dependencies – unless this was forced by the user with the `-o` flag of course.

You can define dependencies by listing the names of dependent service in `requires`:

```
services:
  mysql:
    image: mysql
    instances:
      mysql-server-1: { ... }

  web:
    image: nginx
    requires: [ mysql ]
    instances:
      www-1: { ... }
```

Defining a dependency also makes Maestro inject environment variables into the instances of these services that describe where the instances of the services it depends on can be found (similarly to Docker links). See *How Maestro orchestrates and service auto-configuration* for all the details on these environment variables.

You can also define "soft" dependencies that do not impact the start/stop orders but that still make Maestro inject these variables. This can be useful if you know your application gracefully handles its dependencies not being present at start time, through reconnects and retries for examples. Defining soft dependencies is done via the `wants_info` entry:

```
services:
  mysql:
    image: mysql
    instances:
      mysql-server-1: { ... }

  web:
    image: nginx
    wants_info: [ mysql ]
    instances:
      www-1: { ... }
```

2.5 Port mapping

Maestro supports several syntaxes for specifying port mappings. Unless the syntax supports and/or specifies otherwise, Maestro will make the following assumptions:

- the exposed and external ports are the same (*exposed* means the port bound to inside the container, *external* means the port mapped by Docker on the host to the port inside the container);
- the protocol is TCP (`/tcp`);
- the external port is bound on all host interfaces using the `0.0.0.0` address.

The simplest form is a single numeric value, which maps the given TCP port from the container to all interfaces of the host on that same port:

```
# 25/tcp -> 0.0.0.0:25/tcp
ports: {smtp: 25}
```

If you want UDP, you can specify so:

```
# 53/udp -> 0.0.0.0:53/udp
ports: {dns: 53/udp}
```

If you want a different external port, you can specify a mapping by separating the two port numbers by a colon:

```
# 25/tcp -> 0.0.0.0:2525/tcp
ports: {smtp: "25:2525"}
```

Similarly, specifying the protocol (they should match!):

```
# 53/udp -> 0.0.0.0:5353/udp
ports: {dns: "53/udp:5353/udp"}
```

You can also use the dictionary form for any of these:

```
ports:
  # 25/tcp -> 0.0.0.0:25/tcp
  smtp:
    exposed: 25
    external: 25

  # 53/udp -> 0.0.0.0:5353/udp
  dns:
```

(continues on next page)

(continued from previous page)

```
    exposed: 53/udp
    external: 5353/udp
```

If you need to bind to a specific interface or IP address on the host, you need to use the dictionary form:

```
# 25/tcp -> 192.168.10.2:25/tcp
ports:
  smtp:
    exposed: 25
    external: [ 192.168.10.2, 25 ]


  # 53/udp -> 192.168.10.2:5353/udp
  dns:
    exposed: 53/udp
    external: [ 192.168.10.2, 5353/udp ]
```

Port ranges can be specified using the usual syntax. It's also possible to expose a container's port as a random port within a given host port range:

```
ports:
  direct: 1234-1236:1234-1236
  random: 1234:1234-1236
```

Note that YAML supports references, which means you don't have to repeat your *ship*'s IP address if you do something like this:

```
ship:
  demo: {ip: &demoip 192.168.10.2, docker_port: 4243}

services:
  ...
    ports:
      smtp:
        exposed: 25/tcp
        external: [ *demoip, 25/tcp ]
```

2.5.1 Port mappings and named ports

When services depend on each other, they most likely need to communicate. If service B depends on service A, service B needs to be configured with information on how to reach service A (its host and port).

Even though Docker can provide inter-container networking, in a multi-host environment this is not possible. Maestro also needs to keep in mind that not all hosting and cloud providers provide advanced networking features like multicast or bridged frames. This is why Maestro makes the choice of always using the host's external IP address and relies on traditional layer 3 communication between containers.

There is no performance hit from this, even when two containers on the same host communicate, and it enables inter-host communication in a more generic way regardless of where the two containers are located. Of course, it is up to you to make sure that the hosts in your environment can communicate with each other.

Note that even though Maestro allows for fully customizable port mappings from the container to the host, it is usually recommended to use the same port number inside and outside the container. It makes it slightly easier for troubleshooting, and some services (Cassandra is one example) assume that all their nodes use the same port(s), so the port they know about inside the container may need to be the external port they use to connect to one of their peers.

One of the downsides of this approach is that if you run multiple instances of the same service on the same host, you need to manually make sure they don't use the same ports, through their configuration, when that's possible.

Finally, Maestro uses *named* ports, where each port your configure for each service instance is named. This name is the name used by the instance container to find out how it should be configured and on which port(s) it needs to listen, but it's also the name used for each port exposed through environment variables to other containers. This way, a dependent service can know the address of a remote service, and the specific port number of a desired endpoint. For example, service depending on ZooKeeper would be looking for its `client` port.

2.6 Volume bindings

Volume bindings are specified in a way similar to `docker-py` and Docker's expected format, and the `mode` (read-only 'ro', or read-write 'rw') can be specified for each binding if needed. Volume bindings default to being read-write.

```
volumes:
  # This will be a read-write binding
  /on/the/host: /inside/the/container

  # This will be a read-only binding
  /also/on/the/host/:
    target: /inside/the/container/too
    mode: ro
```

Note that it is currently not possible to bind-mount the same host location into two distinct places inside the container as this is not supported by `docker-py` (it's a dictionary keyed on the host location).

Container-only volumes can be specified with the `container_volumes` setting on each instance, as a path or list of paths:

container_volumes:

- /inside/the/container/1
- /inside/the/container/2

Finally, you can get the volumes of one or more containers into a container with the `volumes_from` feature of Docker, as long as the containers run on the same ship:

```
# other1 and other2 run on the same ship as this container
volumes_from: [ other1, other2 ]
```

2.7 Lifecycle checks

When controlling containers (your service instances), Maestro can perform additional checks to confirm that the service reached the desired lifecycle state, in addition to looking at the state of the container itself. A common use-case of this is to check for a given service port to become available to confirm that the application correctly started and is accepting connections.

Maestro can also execute checks before starting or stopping containers, to confirm that the requested start or stop action should indeed take place.

When starting containers, Maestro will first execute the `pre-start` lifecycle checks before starting the container. It will then execute all the lifecycle checks for the `running` target state; all must pass for the instance to be considered correctly up and running.

Similarly, before stopping a container, Maestro will first execute the `pre-stop` lifecycle checks. After stopping the container, Maestro will execute all the `stopped` target state checks.

Checks can be defined via the `lifecycle` dictionary at the service level, or for each declared instance of a service. If both are present, all defined checks are executed; container-level lifecycle checks don't override service-level checks, they add to them.

```
services:
  zookeeper:
    image: zookeeper:3.4.5
    lifecycle:
      running:
        - {type: tcp, port: client, max_wait: 10}
    instances:
      zk1:
        ship: host1
        ports: {client: 2181}
        lifecycle:
          running:
            - {type: exec, cmd: "python ./ruok.py", attempts: 10}
```

In the example above, Maestro will first perform a TCP port ping on the `client` port; when that succeeds, it will execute the hypothetical `ruok.py` script, which we can image as sending the `ruok` command to the ZooKeeper instance, expecting the `imok` response back to declare the service healthy and operational.

2.7.1 TCP port pinging

TCP port pinging (`type: tcp`) makes Maestro attempt to connect to the configured port (by name), once per second until it succeeds or the `max_wait` value is reached (defaults to 300 seconds).

Assuming your instance declares a `client` named port, you can make Maestro wait up to 10 seconds for this port to become available by doing the following:

```
type: tcp
port: client
max_wait: 10
```

2.7.2 HTTP request

This check (`type: http`) makes Maestro execute web requests to a target, once per second until it succeeds or the `max_wait` value is reached (defaults to 300 seconds).

Assuming your instance declares a `admin` named port that runs a webserver, you can make Maestro wait up to 10 seconds for an HTTP request to this port for the default path "/" to succeed by doing the following:

```
type: http
port: web
max_wait: 10
```

Options:

- `port`, named port for an instance or explicit numbered port
- `host`, IP or resolvable hostname (defaults to ship.ip)
- **`match_regex`, regular expression to test response against (defaults** to checking for HTTP 200 response code)

- **path, path (including querystring) to use for request (defaults to /)**
- scheme, request scheme (defaults to http)
- method, HTTP method (defaults to GET)
- **max_wait, max number of seconds to wait for a successful response** (defaults to 300)
- **requests_options, additional dictionary of options passed directly** to python's requests.request() method (e.g. verify=False to disable certificate validation)

2.7.3 Script execution

Script execution (type: exec) makes Maestro execute the given command, using the return code to denote the success or failure of the test (a return code of zero indicates success, as per the Unix convention). The command is executed a certain number of attempts (defaulting to 180), with a delay between each attempt of 1 second. For example:

```
type: exec
command: "python my_cool_script.py"
attempts: 30
```

The command's execution environment is extended with the same environment that your running container would have, which means it contains all the environment information about the container's configuration, ports, dependencies, etc. You can then use Maestro guest utility functions to easily grok that information from the environment (in Python). See *How Maestro orchestrates and service auto-configuration* and *Maestro guest functions* for more information.

Note that the current working directory is never changed by Maestro directly; paths to your scripts will be resolved from wherever you run Maestro, not from where the environment YAML file lives.

2.7.4 Remote script execution

Remote script execution (type: rexec) makes Maestro execute the given script inside of running container, using the script output to denote the success or failure of the test. Same as Script execution, the script is executed a certain number of attempts (defaulting to 180), with a delay between each attempt of 1 second. For example:

```
type: rexec
command: "python my_cool_remote_script_in_container.py"
attempts: 30
```

Also, the script's execution environment is extended with the same environment that running container would have, which means it contains all the environment information about the container's configuration, ports, dependencies, etc. You can then use Maestro guest utility functions to easily grok that information from the environment (in Python). See *How Maestro orchestrates and service auto-configuration* and *Maestro guest functions* for more information.

Note that minimal api_version is '1.16'. See *Environment description* for more information.

2.8 Restart policy

Since version 1.2 docker allows to define the restart policy of a container when it stops. The available policies are:

- restart: no, the default. The container is not restarted;
- **restart: always: the container is _always_ restarted, regardless** of its exit code;
- **restart: on-failure: the container is restarted if it exits with a** non-zero exit code.

You can also specify the number of maximum retries for restarting the container before giving up:

```
restart:
  name: on-failure
  retries: 3
```

Or as a single string (similar to Docker's command line option):

```
restart: "on-failure:3"
```

2.9 Links

Maestro also supports defining links to link same-host containers together via Docker's Links feature. Read more about Docker Links to learn more. Note that the format of the environment variables is not the same as the ones Maestro inserts into the container's environment, so software running inside the containers needs to deal with that on its own.

Defining links is done through the instance-level `links` section, with each link defined as a child in the format `name: alias`:

```
services:
  myservice:
    image: ...
    instances:
      myservice-1:
        # ...
        links:
          mongodb01: db
```

2.10 Working with image registries

When Maestro needs to start a new container, it will do whatever it can to make sure the image this container needs is available; the image full name is specified at the service level.

Maestro will first check if the target Docker daemon reports the image to be available. If the image is not available, or if the `-r` flag was passed on the command-line (to force refresh the images), Maestro will attempt to pull the image.

To do so, it will first analyze the name of the image and try to identify a registry name (for example, in `my-private-registry/my-image:tag`, the address of the registry is `my-private-registry`) and look for a corresponding entry in the `registries` section of the environment description file to look for authentication credentials. Note that Maestro will also look at each registry's address FQDN for a match as a fallback.

You can also put your credentials into `${HOME}/.dockercfg` in the appropriate format expected by Docker and `docker-py`. Maestro, via the `docker-py` library, will also be looking at the contents of this file for credentials to registries you are already logged in against.

If credentials are found, Maestro will login to the registry before attempting to pull the image.

Additionally, you can configure a retry policy or image pull errors or a per-registry basis. You can specify a maximum number of retries, and a list of returned HTTP status codes to retry on. For example, the following configuration will make two attempts to pull images from the `quay.io` registry if a 500 is returned.

```
registries:
  quay.io:
    registry: https://quay.io/v1/
    email: user@example.com
    username: user
    password: super-secret
    retry:
      attempts: 2
      when:
        - 500
```

2.11 Passing extra environment variables

You can pass in or override arbitrary environment variables by providing a dictionary of environment variables key/value pairs. This can be done both at the service level and the container level; the latter taking precedence:

```
services:
  myservice:
    image: ...
    env:
      FOO: bar
    instance-1:
      ship: host
      env:
        FOO: overrides bar
        FOO_2: bar2
```

Additionally, Maestro will automatically expand all levels of YAML lists in environment variable values. The following are equivalent:

```
env:
  FOO: This is a test
  BAR: [ This, [ is, a ], test ]
```

This becomes useful when used in conjunction with YAML references to build more complex environment variable values:

```
_globals:
  DEFAULT_JVM_OPTS: &jvmopts [ '-Xms500m', '-Xmx2g', '-showversion', '-server' ]

...

env:
  JVM_OPTS: [ *jvmopts, '-XX:+UseConcMarkSweep' ]
```

2.11.1 Examples of Docker images with Maestro orchestration

For examples of Docker images that are suitable for use with Maestro, you can look at the following repositories:

- http://github.com/signalfuse/docker-cassandra A Cassandra image. Nodes within the same cluster are automatically used as Gossip seed peers.
- http://github.com/signalfuse/docker-elasticsearch ElasicSearch with ZooKeeper-based discovery instead of the multicast-based discovery, to work in cloud environments.

- http://github.com/signalfuse/docker-zookeeper A ZooKeeper image, automatically creating a cluster with the other instances in the same environment.

2.12 Maestro guest functions

MaestroNG passes into your container's environment a lot of information that you might need for your application to run: port numbers that you have defined in your environment configuration file, addresses and ports of your dependencies, etc. All these environment variables are defined and documented in *How Maestro orchestrates and service auto-configuration*.

In order to make groking this environment information easier, MaestroNG provides a Python module of "guest functions" that allow you to write simple container init scripts in Python that act as the glue between this information and your application's configuration before starting the application itself.

A common use case, for example, is to get your ZooKeeper connection string from this environment – the rest you usually find in ZooKeeper itself via service discovery.

2.12.1 Basic usage

To make use of the Maestro guest utils functions, you'll need to have the Maestro package installed inside your container. You can easily achieve this by adding the following to your Dockerfile:

```
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update \
  && apt-get -y install python python-pip \
  && apt-get clean
RUN pip install maestro-ng
```

Then, from your Python script, simply do:

```
from maestro.guestutils import *
```

And you're ready to go! Feel free to change the *import* * to the list of specific functions you actually need in your script.

2.12.2 Reference

Here's a summary of the functions available at your disposal that will make your life much easier. All the examples given here are based on the [Zookeeper+Kafka example environment](examples/zookeeper+kafka.yaml), and assumed to be executed from within the *kafka-2* container.

get_environment_name()

Returns the name of the environment as defined in the description file. Could be useful to namespace information inside ZooKeeper for example.

```
>> get_environment_name()
'zk-kafka'
```

get_service_name()

Returns the name of the service the container is a member of.

```
>> get_service_name()
'kafka'
```

get_container_name()

Returns the name of the container instance.

```
>> get_container_name()
'kafka-2'
```

get_container_host_address()

Returns the IP address or hostname of the host of the container. Useful if your application needs to advertise itself to some service discovery system with its publicly reachable address. This would be *192.168.10.2* in our example.

```
>> get_container_host_address()
'192.168.10.2'
```

get_container_internal_address()

Returns the IP address assigned to the container itself by Docker (its private IP address). This is normally the IP address Docker assigned to the *eth0* interface inside the container and is usually in the *172.18.42.0/24* subnet.

```
>> get_container_internal_address()
# Might be different depending on the number of running containers on
# that host.
'172.18.42.1'
```

get_port(name, default=None)

Returns the exposed internal port number of a given named port for the current container. This is the port number your application uses _inside_ the container. This is useful to automatically configure the port your application should use directly from what you have specified in your environment file.

If no default is provided and the port name does not exist, the function will throw a *MaestroEnvironmentError* exception.

```
>> get_port('broker')
9092

>> get_port('unknown', 42)
42

>> get_port('unknown')
# MaestroEnvironmentError gets raised
```

get_node_list(service, ports=[], minimum=1)

This function is one of the most useful of the set. It takes in a service name and an optional list of port names and returns the list of IP addresses/hostname of the containers of that service. For each port specified, in order, it will append *:<port number>* to each host with the external port number.

The *minimum* parameter allows for specifying a minimum number of hosts to return, under which the function will throw a MaestroEnvironmentError exception. This helps enforce the presence of at least N hosts of that service you depend on in the environment.

Back to our example, you can return the list of ZooKeeper endpoints with their client ports by calling:

```
>> get_node_list('zookeeper', ports=['client'])
['192.168.10.2:2181', '192.168.10.2:2182', '192.168.10.2:2183']

>> get_node_list('zookeeper', ports=['client', 'peer'])
['192.168.10.2:2181:2888', '192.168.10.2:2182:2889', '192.168.10.2:2183:2890']
```

Note that Maestro provides information about all your declared dependencies in your environment, but also the information about all the instances of your service itself, so you can easily get a node list of your peers:

```
>> get_node_list(get_service_name(), ports=['broker'])
['192.168.10.2:9092', '192.168.10.2:9093', '192.168.10.2:9094']
```

get_specific_host(service, container)

Returns the hostname or IP address of a specific container from a given service.

```
>> get_specific_host('zookeeper', 'zk-node-2')
'192.168.10.2'
```

get_specific_port(service, container, port, default=None)

Returns the external port number of a specific named port of a given container. This is the externally reachable, routed port number for that particular target.

```
>> get_specific_port('zookeeper', 'zk-node-2', 'client')
2182
```

get_specific_exposed_port(service, container, port, default=None)

Returns the exposed internal port number of a specific named port of a given container. This is rarely needed (but is used internally by *get_port()*).

```
>> get_specific_exposed_port('zookeeper', 'zk-node-2', 'client')
2181
```