System Management high-level API (re-)design

Jim White

Version 2

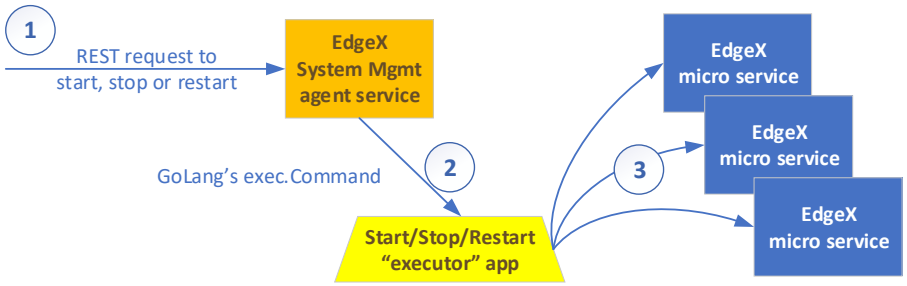
Updated: 1/8/19

This document follows a lot from what is in https://github.com/edgexfoundry/edgex-go/pull/772/files. Where it deviates, we are trying not to provide for all the executor options (non-scalable) but allow these other executors to exist via the call to an independently created external executor in a standard/interface way – this per our 11/13/18 sys management WG call.

At a system management work group meeting on 11/13/18, it was decided that our management facility needed to be able to start/stop/restart EdgeX micro services through a call to either

- Docker Compose
- or some outside, undefined executable app that adheres to a defined API

It was further decided in a 1/8/19 meeting of the same work group, that the Docker Compose need could be handled by just having an executable app that makes the Docker Compose calls – thus preserving a common pattern for all start/stop/restart functionality.



The executable applications that the system management agent (SMA) micro service calls on are referred to as the "executors" of the start, stop, and restart functionality. How the executor performs its duties is left for the implementor and is generally dictated by the available operating system, platform environment (existence and use of Docker for example) and associated programming language resources.

EdgeX will supply at least two executors for demonstration purposes.

- An executor that uses Docker Compose calls to facilitate start, stop and restart functionality (the reference implementation)
- An executor that uses a Linux shell script that start services and a find (using Linux process status: ps -ax) and kill process by PID means to stop the services. This executor is obviously tied to Linux environments.

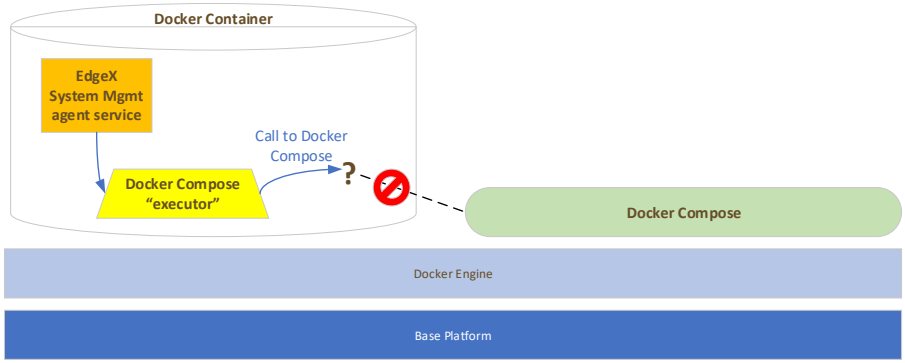
The executor design allows use of other orchestrating software (example: Kubernetes or Swarm), scripts, OS specific technology (snaps or sysd), etc. to be used without having to build anything

into the SMA – allowing for a more scalable solution in EdgeX but still allowing use of all sorts of implementation technology outside of EdgeX.

The SMA will be informed of what executor to use for start/stop/restart functionality through a configuration option – appPath. The appPath will specify the location (which may be platform dependent) and executable to be called to start, stop or restart. The SMA will make a call to execute the executor's functionality via GoLang's exec.Command. See below for more details on the execution call.

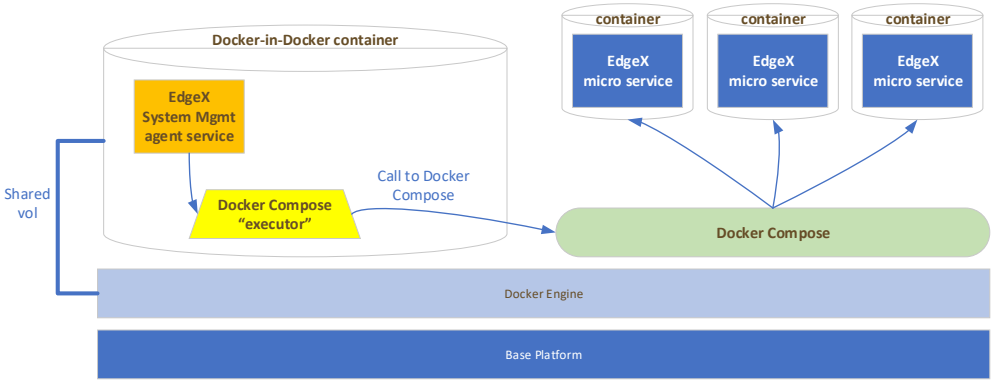
Docker Compose Executor

When using Docker Compose to execute the start, stop and restart of the other EdgeX micro services, the executor will make command line calls to Docker Compose. However, this is not as straightforward as one would think. Complexity comes from the fact that the SMA (and associated executor) is itself containerized in this type of environment and so a call from within a normal container to Docker Compose would fail as Docker Compose is not installed inside of the container.

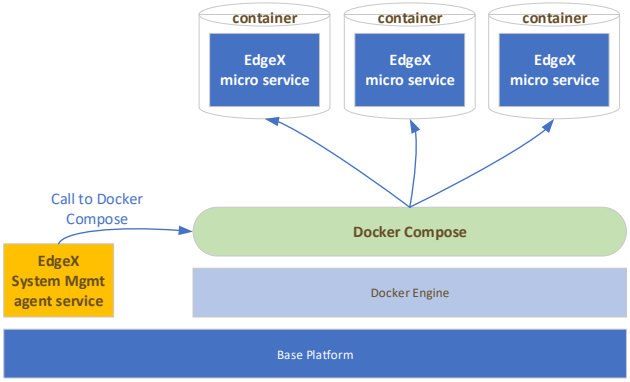


Even if Docker Compose were part of the SMA's container, a call to Docker Compose to start (or stop or restart) the other services would be internal to the SMA's container. This would not be helpful since it would try to start the EdgeX services inside of the SMA's container and not on the Docker Engine where all the containers exist.

The answer to solve this issue is that the SMA must run inside of a special container – a Docker-in-Docker container and that container must share a volume with the Docker Engine. This acts in a way as to expose the Docker Compose calls out to the Docker Engine running on the base platform. Thereby allowing the SMA (and its executor) to effect calls to start, stop and restart the original EdgeX services running on the same Docker Engine as the SMA.



Note that in the Delhi release, this obstacle required that the SMA be deployed outside of Docker so that it's calls to Docker Compose were made directly – a non-desired and probably non-sensical deployment scenario.



Docker Compose Executor Internals

Again, the makeup of the executors is at the implementer's discretion. However, it is likely that inside of the Docker Compose Executor, there would be code similar to code today in internal/system/agent/executor/docker.go.

```
StartService(services string, params []strings){
   //Docker Compose call would ignore params
   call to start via docker-compose
StopService(services string, params []strings){
   //Docker Compose call would ignore params
   call to stop via docker-compose
ReStartService(services string, params []strings){
   //Docker Compose call would ignore params
   call to restart via docker-compose
```

Location of the Docker Compose Executor code

The Docker Compose Executor (or any executor maintained by the EdgeX project) will reside in a separate Go repository. In this case edgex-docker-compose-executor. As the executor does not need any EdgeX models, libraries, etc. it can stand alone, be managed independently, and be

built separately. The only packages that it should need is some type of logging and the Go Lang os/exec package.

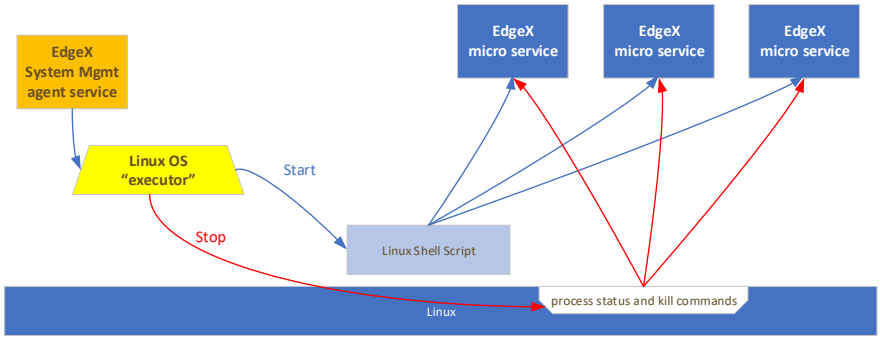
Linux OS Executor (stretch goal for Edinburgh)

For the Edinburgh release, the Docker Compose Executor will be used as the reference implementation executor for use with the SMA. A second Executor will be provided for use in non-container solutions to provide start, stop and restart functionality and to provide a second example to those looking to build their own executors. This Executor will take advantage of Linux OS capability and Shell scripts to provide the start, stop and restart functionality.

Linux OS Executor Internals

This Linux OS Executor will use calls to a shell script similar to the edgex-launch script in the edgex-go repository (see https://github.com/edgexfoundry/edgex-go/blob/master/bin/edgex-launch.sh) to start the services. It will use Linux process status calls to "kill" services in order to stop them (similar in code submitted by Ian Johnson below).

```
func (oe *ExecuteOs) Stop(service string, params []string) error {
        cmd := exec.Command("ps", "-ax")
        out, err := cmd.CombinedOutput()
        if err != nil {
                return err
        }
        pid, err := parseProcessListing(string(out), service)
        if err != nil {
                return err
        }
        proc, err := os.FindProcess(pid)
        if err != nil {
                return err
        }
        return proc.Kill()
}
```



Location of the Linux OS Executor code

The Linux OS Executor (or any executor maintained by the EdgeX project) will reside in a separate Go repository. In this case edgex-executor-linux-os. As the executor does not need any EdgeX models, libraries, etc. it can stand alone, be managed independently, and be built

separately. The only packages that it should need is some type of logging and the Go Lang os/exec package.

EdgeX SMA API Trace

A request (by REST call) of the SMA to start, stop or restart service(s) begins at the router.

```
In internal/system/agent/router.go
        b.HandleFunc("/operation", operationHandler).Methods(http.MethodPost)
```

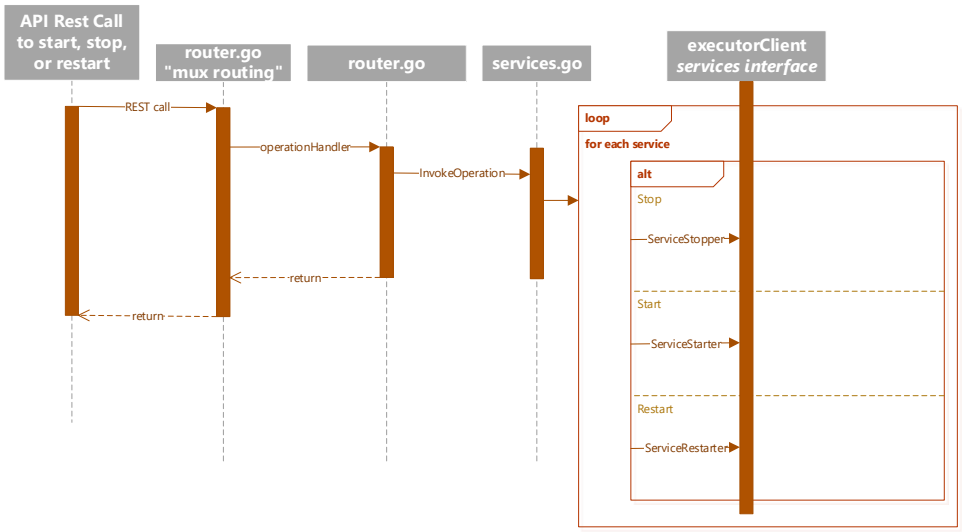
The router passes the request to the operationHandler function where action, services and parameters are determined.

```
In router.go
        operationHandler( )
                // invoke the operation and check the result
                err = InvokeOperation(o.Action, o.Services, o.Parameters)
```

The invocation function (InvokeOperation) then makes the call to the specific start, stop or restart interface on the executor client to invoke the specific operation (with the applicable service name and parameters list) for each of the services. The executor client represents the "executor" in the SMA and is the component that knows the actual location of the real executor.

```
In internal/system/agent/services.go
   func InvokeOperation(action string, services []string, params []string) error {
      for each service
         if Stop
            executorClient.(interfaces.ServiceStopper)
         if Start
            executorClient.(interfaces.ServiceStarter)
         if Restart
            executorClient.(interfaces.ServiceRestarter)
```

The functions (ServiceStopper, ServiceStarter and ServiceRestarter) are all defined by internal/system/agent/interfaces/services.go. This interface allows for other future implementations (outside of the "executor" model).



services.go Interface Implementation

The executor client instance (that implements the internal/system/agent/interfaces/services.go function definitions) is established on bootstrapping of the SMA. Specifically, it is defined in init.go.

```
In init.go
func newExecutorClient(operationsType string) (interfaces.ExecutorClient, error) {

        // use the configuration to provide the executor with the location (appPath)
        // to the actual executor program.
    return &executor.ExecuteApp{}, nil
}
```

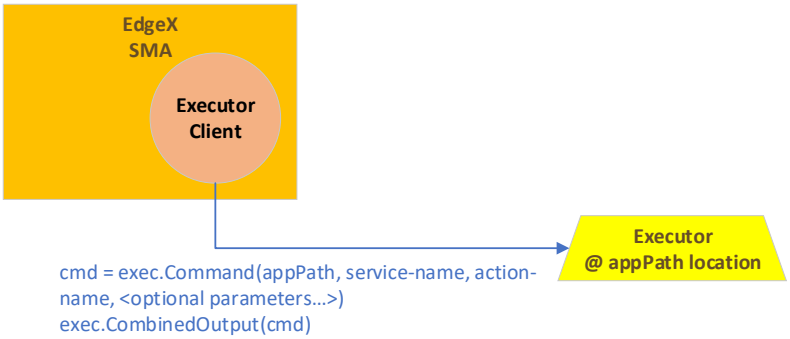
The executor client provides the ServiceStopper, ServiceStarter and ServiceRestart functions that each make calls to Go Lang's os/exec.Command(name string, arg… string) and then subsequently exec.CombinedOutput to run the command and get the returned standard output and standard error returns.

```
cmd := exec.Command(appPath, serviceName, actionName, parameters…>)
stdoutStderr, err := cmd.CombinedOutput()
```

Go Lang Exec calls

When the executor client makes the Go exec.command call, name string is the appPath to the Executor application. The remaining args are the service name (EdgeX micro service name), action name (start, stop, or restart) and then any, optionally, additional parameters the Executor may need or want.

executable-name (which is the appPath) service-name action-name optional-parameters



Parameter strings provided by configuration and passed from the SMA through to the executor can be used to provide platform or environmental indicators to the Executor. For example, in some environments, there is the ability to enable/disable restart of services on restart or reboot of the underlying platform. The parameters list would be separated by a space if there was more than one.

Per the Go Lang documentation, the call to exec.CombinedOutput will return the standard output and standard error back to the SMA. Therefore, the Executor programs should write a single 0 or 1 string to the standard output to indicate the results of the Executor's execution of the request. A return of string 0 indicates that the execution completed its task and exited "normally" or without issue. A return of string 1 indicates that the execution did not complete "normally" and the caller should check the standard error for more information. The Executor should always return some information string indicting why the non-normal return when 1 is returned on the standard out. The Executor should not return anything on the standard error when 0 is returned on the standard out. See https://golang.org/pkg/os/exec/ for more details.

Note for future implementations: Go Lang plugin architecture may someday be used for the Executor. Today, plugins are a Linux only feature in Go Lang and therefore not being used for this implementation.