Executor Gets Service Metrics (Redesign - 2.0) – Notes

Overview

This redesign is unified around organizing the underlying implementation wherein the service management abstraction has a unifying interface for gathering different types of metrics, with three implementations that will be maintained in the Go code base:

1. A *custom executor* method which calls out to some binary (defined by the configuration.toml file) with some specific arguments and gets the result over stdout from that process.
2. A *direct-service* executor method which is actually the existing implementation we have put behind an interface where the CPU method of the interface just calls out over REST to the specified process endpoint like today.
3. A *docker executor* method which uses the docker CLI (Command Line Interface) to call "docker stats" for the specified service and metrics.

This is supplemented by a configuration variable (i.e. *MetricsMechanism*). The Snap executor would use this variable to go to the services and fetch their associated metrics. Likewise, in a similar pattern, the Docker executor will go to the services and fetch their metrics (using the built-in "docker stats" command for this purpose).

By way of this redesign, we don't throw out any (existing, tested) code, other than brush it up so everything remains behind proper interfaces.

Interfaces

The interface (functions, inputs, return params) are as follows:

```
type MetricsFetcher interface {
	Metrics(ctx context.Context, service string) ([]byte, error)
}
```

➔ The central interface (above) is defined in: edgexfoundry/edgex-go/internal/system/agent/interfaces/*Operations.go*

➔ But to begin at the beginning, the incoming request (for metrics) arrives at the following handler function in the router:

```
func metricsFetchHandler(w http.ResponseWriter, r *http.Request) { … }
```

➔ The handler function invokes:

```
send, err := InvokeMetrics(services, ctx)
```

➔ Which leads to the following service function:

```
func InvokeMetrics(services []string, ctx context.Context) (MetricsRespMap, error) { … }
```

Where the (JSON) response is handled through a map with type-less values (to accommodate various types such as integers, strings, Booleans, etc.), hence the interface{} below:

```
type MetricsRespMap struct {
        Metrics map[string]interface{}
}
```

➔ The function (above) invokes the central interface through method receivers such as the following:

```
es := ExecuteService{}
out, err := es.Metrics(ctx, service)

ea := ExecuteApp{}
out, err := ea.Metrics(ctx, service)
```

➔ The interface is invoked via the interface through method receivers as mentioned above, and that depends on which implementation is appropriate, based on the value specified in the *configuration* TOML file as follows:

```
# The MetricsMechanism can be one of the following three options, proceeding which is
a non-operational default:
MetricsMechanism = 'custom'
# MetricsMechanism = 'executor'
# MetricsMechanism = 'direct-service'
```

➔ Finally, here are the relevant signatures of the interface implementations:

```
func (ec *ExecuteService) Metrics(ctx context.Context, service string) ([]byte, error) { … }

func (ea *ExecuteApp) Metrics(ctx context.Context, service string) ([]byte, error) {
```

Ideally there would be no difference between the current implementation (i.e. native golang reported metrics) and the snap implementation of this. The only complication that would arise from running in a snap versus running natively is that the snap would be confined by the interfaces we declare and so some methods of reading metrics would be denied such as using *ptrace*. The main interfaces necessary to perform this type of metrics gathering we do today are already used by the snap so we don't need to worry.

Benefits

This redesign, thus, has a unifying interface for gathering different types of metrics, with three implementations that will be maintained in the Go code base. The benefits of abstracting out at the Go layer are as follows:

1. It's easier to maintain Go code
2. It's portable by default
3. It reduces complexity for the user
4. It allows us to organize the internal complexity in a known developer friendly format using Go interfaces
5. This list of internal interface implementations could be entirely left at these 3 for the rest of time and all further implementations folks want to use (k8s, etc.) can be implemented on top of 1, so there's no code creep for the project.

The End