The “Base” EdgeX Microservice

Author: Jim White

Last Updated: 4/9/18

Contents

This document provides design guidance for functionality to be included with each EdgeX microservice as it relates to the service’s name, availability, use of configuration – independent of OS, deployment environment, or use of registry service. This document also outlines how/what functions should be bundled into common shared language-specific libraries that are shared by all EdgeX microservices.

Base Service Requirements

All EdgeX Foundry microservices should be able to accomplish the following tasks:

- Register with the configuration/registration (referred to simply as “the registry” for the rest of this document) provider (today Consul) & get its configuration (per formula below)
    - Service reads its local configuration from the local file system (using ./res as the location of the local configuration by default but the default location can be overridden with a location designated by either command line parameter or environment variable).
    - If configured to use the registry, the service attempts connection to the registry and registers itself as an EdgeX microservice (the address of the registry is provided by the local configuration file). If unable to connect to the registry after the number of retries, it should exit and log the error.
- Respond to availability requests (it is up and working properly to take requests from other services or from outside EdgeX where applicable – more details below).

- Respond to shutdown requests
    - Taking care to clean up of any resources in an orderly fashion
    - Unregister itself from the registry
- Get the address to another EdgeX microservice by service name through the registry (when enabled), or via local configuration when the registry is not being used.

In programming environments that support it, these features should be provided by a base microservice class**,** library or package so that they can be shared by all microservices. For example, these features would be provided by a shared library package in Go or a base abstract "EdgeXService" class in Java. For the purposes of this document, this common set of features will be referred to as the "Base Interface" with implementation to be determined based on programming paradigm, but intended to be shared across multiple microservices.

In the future, the Base Interface might also include:

- Additional code to support securing the microservice
- Additional code to support the management of the microservice (see current System Management Requirements document located here for more requirements and details)
- Additional code to support collection of metric information, resource utilization, and performance of the service

Configuration Rules

- Bootstrap options provided by command line (note that each of these options has a default value if not provided):
    - -registry – use the registry set to either true or false (default false)
    - –confdir – path to local configuration file (default ./res)
    - –profile – configuration profile to use when loading the configuration (defaults to the default profile – which is the configuration file with no addition profile information attached – see below). This option would be used for microservices only when loading local configuration.
- Bootstrap options provided by environment variable (same options as above that would override the command line options). Note: in future implementations where there is a need to support multiple services running in the same context (such as the same container), then there must be some strategy for naming environment variables for each service so that services could have different settings.
- If registry bootstrap option is "true", configuration settings are provided by the registry
- If registry bootstrap option is "false", configuration settings are provided by local file system (would serve as the default configuration when configuration is not provided by the registry)

Microservice Naming

Today, the microservice obtains its name from central or local configuration. There are issues when a service needs to be renamed or multiple instances of the service need to exist.

Commented [Tony Espy1]: Just a note that the current core-* services use "-consul=y" to enable consul. I think the new option should just be "-registry", with no trailing value needed. If we really think the ability to specify "no-registry" is important, then we need to make sure we check the value case-insensitively, and if we want "true/false", we probably should accept "t/f". Likewise with "yes/no" we should also accept "y/n".

Commented [Tony Espy2]: If the options require "=", we should include in this doc.

Commented [Tony Espy3]: I'd rather see us just define them here, and if we get it wrong, we iterate. The CONSUL env var will almost always be applied to all services, and same with PROFILE. I think it's only CONF-DIR which is one that might be different for multiple services. I propose we just name the vars:

EDGEX_REGISTRY
EDGEX_CONF_DIR
EDGEX_PROFILE

- The name (for some microservices such as device services) is stored in the Metadata database (device services – and their name – are represented as objects) and associated to other non-service objects in the database (for example, devices are linked to device services by service name) in EdgeX.
- The name may also be used to name the service deployment container on a virtual deployment network.
- Going forward, the "name" of the service is not its unique identity but rather an indication of the API set it satisfies.
    a. The service name will identify the type or nature of the service but is not guaranteed to uniquely identify the actual instance. Essentially, the name is unique to an API offering (but not an instance of a process). There may someday be multiple, load-balanced core-data microservice instances. The name for each would still be "edgex-core-data". The name "edgex-core-data" could not be applied to other services that did not offer the Core Data API set (so the name is unique to the type of API) but there could be multiple "edgex-core-data" in a fog deployment.
    b. The proposed standard of the service API set name (aka service name) is edgex-<layer>-<name>. Examples: edgex-core-data, edgex-device-modbus, etc. TSC must govern the names of services (agreeing to the names of any new service API sets). Services (such as device services) may have their own specific extensions to the service API, but would still conform to some generic API set for that type of service.
    c. All services that come up, must try to register their service name with the registry (when enabled). In the event that the registry is not available, the service should enter a retry loop for a number of connection retries (number of retries and time between retries to be determined and provided in the local configuration – this should generally be consistent across services). If unable to connect to the registry after the number of retries, it should exit and log the error. More on both registry and no-registry environments is provided below.
    d. Today, there is but one of each service per name. There could be, in the future, multiple instances; for example core-data #1, core-data #2. EdgeX is not setup to run with multiple services with the same name. If the registry is in use, services will use it to query by service name, addresses (hostname/ip address and port) of other services it needs. In the case where the registry is not used, the local configuration should provide the hostname (or IP address) and port of the requested service.
    e. A service will not change its name. The service name is really an indication of the API set it offers. Since it represents an API set (versus unique identifier) the name will always remain constant.
    f. The service name for each service type is known to all of EdgeX. Therefore, the service name will no longer be provided by configuration (local or by the registry). Service names are defined as constants within each service.
- In the future (post California release), the registry could allow multiple instances of any particular service – and understand that many instances exist all at different addresses. In the

future, clients may also use the service name to call on some service to provide details of the API set – that is, allowing introspection of the API set by service name.

- Again in the future, clients of any particular API set (or named service) would not care that there are multiple instances. Clients of the future would go through a load-balanced registry to get an endpoint address to talk to (as they will do today, and not know or care that they are being directed to a specific designated instance) as determined by the load balancer. For example, when an edgex-core-data service needs to speak to edgex-core-metadata, it would request the host and port of edgex-core-metadata from the load-balanced registry service. It would then be handed the information (host and port) necessary to create an address for the service, but that might be one of many addresses for the edgex-core-metadata service (unbeknownst to the edgex-core-data service).

Service To Service Communications

When Service A needs to talk to Service B (or really the service offering B's APIs) and…

a. The registry is enabled then Service A should request the address of Service B from the registry by name (allows for load balancing or other redirection in the future)

b. The registry is not used (as in development), then Service A will use the address of Service B provided by the local configuration settings for Service B

Note, that the address is created by the client service from the host (or IP address) and port provided by the registry or local configuration. Implementation note for clarity: today, the client service must create the REST endpoint URLs from the host (or IP address) and port provided by the registry or local configuration along with any required additional API path or version information (v1 today).

This functionality will be provided by the "Base Interface" (as defined above) and could look something like the following pseudo code:

```
addressString getServiceAddress(servicename, registryaddress){
    if (registryaddress!= null)
        return configServiceGetServiceLocation(servicename)
    else
        return localConfigGetServiceLocation(servicename)
}
```

As an added convenience, the specific client service libraries (i.e. core-data-client or core-metadata-client) should build on top of the "Base Interface" functionality and offer a means to call on the microservice API of choice without first worrying about locating the address to make the call and return the results (or error condition) as appropriate. Note: services are always allowed to make their own calls outside of the client libraries.

```
result callAPI(servicename, registryaddress, serviceAPI, serviceAPIparameters) {
    address=addressString(servicename, registryaddress);
    return remoteCall(address, serviceAPI, serviceAPIparameters)
}
```

Device Service Naming and Metadata Registration

Device services abide by the same rules, in general, as the other EdgeX services but because of the more direct association between specific devices and specific managing instances of a device service, there are some deviations.

a. Device services of a particular protocol abide by the same naming standard as defined above inclusive of the protocol as part of the device name to readily identify its southward interface connectivity (ex: edgex-device-modbus).

b. The proposed standard of the device service API set name (aka service name) is edgex-device-<protocol support type>. Examples: edgex-device-modbus, edgex-device-bluetooth, etc. Again, the TSC must govern the names of services (agreeing to the names of any new service API sets).

c. Device services are also represented by objects in metadata.

d. When a device service is started, an object representing the service must also be created in metadata (metadata registration). Today, the address of the device service is supplied as a parameter in this metadata registration. When devices get registered in metadata, they must be managed by a specific instance of a device service. In addition, the address of the device is supplied as a parameter in metadata registration of the device. This allows other EdgeX services that need to speak to a specific device to know what specific device service instance to communicate with. In other words, because a specific instance of a device service is associated to specific devices, any client requesting to speak to a device through a device service cannot go through a registry and be routed to any device service, it must be given the explicit device service address for the specific device.

e. For this reason (among others) communications with device services and associated devices should happen almost exclusively through the edgex-core-command service – thereby requiring that only the command service really know the details of communications with device services.

Command-Service to Device-Service communications

When edgex-core-command needs to talk to a device (like a specific Modbus motor) through its owning device service (like edgex-device-modbus) then it requests the specific device service address (for that device) stored in metadata from the edgex-core-metadata service.

It gets access to the metadata service per the service-to-service communications mechanism above.

Even though communications with a device service is dependent on address information from metadata versus the registry, a device service must still register with the registry so that things like availability can be determined.

Commented [Tony Espy4]: I still think it'd be cleaner if command pulled the right device-service name from metadata, and then pulled the address from the registry...

Deployment and Service Names

EdgeX application architecture should not be tightly coupled to the deployment architecture.

The ability of service discovery (via Consul today) to resolve a service name to the instance of a service (and thus be able to get its address) should not be impacted by whether running in Docker (or other container) or otherwise. The resolved address may be an address bound to a Docker (or other deployment technology) network, but it should not matter to either the calling client service or the service itself.

Therefore, when a service registers itself with the registry, it should register itself with an address that is reachable by the other services in the EdgeX deployment.

Client services that request another service address from the registry should receive an address which is good for the deployment circumstances and without the need for further resolution.

Practically speaking, this means that a Docker container name should not have any bearing on the registration or use of an EdgeX microservice (so long as all of the EdgeX containers are participating in the same Docker network when Docker is used). The service name is and should be consistent across all environments and should serve as the key in the registry to an address understandable to all within the EdgeX deployment.

When EdgeX microservices are deployed outside of a container environment like Docker, the registry or local configuration (when the registry is not used) would still return an address for a service name that is relevant to that non-containerized environment.

When using the local configuration, multiple configuration files (called profiles) may exist to help provide addresses used for specific types of deployment. A configuration-docker.toml (a docker profile) configuration file would be used when running in a Docker environment. A configuration-snap.toml (a snap profile) would be used when running as a snap. The profile to be used is specified to the microservice through the –profile command or environment variable (see bootstratp properties above). See Configuration Files and Profile section below for more details.

Docker Container Names

Docker container/hostname names should be, where possible, named the same as the service name, but should not be required, and EdgeX should still work if they are not. Using the same name just helps in human understanding of the system. However, if the Docker containers were all A, B, C, … versus the service names (edgex-core-data, etc.) then EdgeX should still work. The registry (or local configuration file when a registry is not used) would report the appropriate address of any service based on the service name provided.

The local configuration file may need to use host names or IP addresses that are relevant to a Docker network address scheme (i.e. incorporating the Docker network IP addresses for example) when running services in Docker containers and not using the registry.

Service Availability

Service availability must be determined internally by each service.

Availability is defined as a service that is fully initialized, and capable of responding to API requests.   For example, edgex-core-data cannot respond to requests for historical data or to add new event/readings without the associated persistence store up and running (in cases where persistence is turned on). So edgex-core-data would not be “available” until it has established a connection with the database.

Currently, the ping API on each service indicates that the infrastructure to receive an HTTP request is available, but it does not always and accurately determine if a microservice is ready for handling all of its duties.

The microservices will be refactored so that the services are successfully initialized before the ping API returns true – in other words, indicating when the service is initialized and ready to handle requests as determined by each service. It can be determined independent or dependent of other services (edgex-export-distro may require edgex-export-client be up) and infrastructure (like the database or message infrastructure availability). Immature services (typically those still under development) may simply define availability as being able to accept HTTP requests. As services become more industrialized and ready for production, they should respond to the ping API only when really ready for use by the entire EdgeX platform.

When an EdgeX microservice starts, it should not be considered "available" until the following ordered events occur:

- Service configuration is loaded (locally or from the registry as described above)
- Required external services are initialized and/or verified to be available (that which makes the service truly "available" and ready to take on requests from the other services)
- Start responding affirmatively (pong is the response today) to ping API requests. It should return an appropriate HTTP error response when responding negatively.
- If the registry is enabled, register itself

As an added convenience, any service-to-service communications function (as defined above in this document) could incorporate "service availability" into the "Base Interface". *This would allow any service to get the availability of any other service through the shared library.*

```
callAPIIfAvailable(servicename, registryaddress, serviceAPI, serviceAPIparameters) {
        address=addressString(servicename, registryaddress);
        pingAddress=getPingAddress(address)
        if serviceAvailable(pingAddress)
                return remoteCall(address, serviceAPI, serviceAPIparameters)
        else
                return HTTP 500("service not available")
}
```

Commented [Tony Espy5]: I still think services directly querying the ping address of services doesn't make any sense. In the future, the registry should gain the ability to notify all services of a service outage…

Also requiring services to monitor service availability via the registry allows us flexibility to enhance the health check mechanism without worry about breaking services.

Service Operating State

The operating state (enabled versus disabled) of an EdgeX service should be determined by a call to the ping API.

Metadata should no longer get and keep its own operating state information for services. Metadata may still keep and manage the operating state of sensors for device services as needed.

Service Availability and EdgeX Startup Scripts/Technology

Startup scripts can use the "availability" API (either directly calling on a service's availability API or indirectly by going through the registry when it is used) as the means to determine availability and prohibit starting the next service in the startup of EdgeX until the preceding service(s) is "available.

Case in point, Docker Compose does not have an ability to check availability (it can only make sure each dependent service is started before starting another container). However, someone can create a wait-for-it script to use with Docker Compose to start each service (one at a time) and check the new availability API between each service start. See https://docs.docker.com/compose/startup-order/ for details on wait-for-it scripts.

Other orchestration/deployment tools (e.g. systemd service units) can make use of this same mechanism to handle startup dependencies.

Commented [Tony Espy6]: Actually systemd has it's own mechanism for ordering service startup, however this can't be used till our services themselves grow the capability to probe their dependencies before registering themselves to the registry.

Configuration Files and Profiles

Each microservice should have a single, authoritative file for configuration information.

When the bootstrap option (via command line or environment variable) indicating the use of the registry is specified (the command line property today is –registery which is set to true or false), the microservice should read configuration from the registry, which in turn has been initialized by the edgex-config-seed service (today it initializes Consul). Only one configuration set should be loaded into the registry by the initialization service (edgex-config-seed).

When a bootstrap option to use the registry is not provided (the default or if set to false explicitly) to a service, then two alternate bootstrap options (supplied via command line or environment variable) can be provided to the service. The first property (-profile today when using command line variable) specifies which configuration file by profile name should be used to provide configuration to the microservice. The second property (-confdir today when using command line variables) specifies the location of configuration files in the file system of the running service to use.

The configuration file format is TOML and the default configuration file name (whether via config seed or local file system) is to be named configuration.toml. The configuration keys for services should be standardized and consistent for common settings across all EdgeX microservices – regardless of where the configuration emanates (local file system or config seed). Examples of common properties include the service's port address or a service's logging configuration.

Each EdgeX microservice is allowed to have configuration properties which are specific to that service. An example may be the ZeroMQ topic name for the message infrastructure carrying messages from edgex-core-data to the export services.

These rules apply to "version 2" of the EdgeX configuration. For backward compatibility, some configuration (such as the configuration of Java microservices) may not follow these rules. In Consul, old and possibly non-conforming configuration would be found in /config. All conforming configuration would be found in /config/v2.

Profiles

Configuration profiles are additional suffixes applied to configuration file names to suggest their use. One ending in "-docker" for example, would suggest the configuration is to be used in conditions where the microservice is running in a Docker deployment. While the profile names are arbitrary, EdgeX uses the profiles largely to suggest deployment circumstances. Use of the default configuration (configuration.toml) is an indication that the service is deployed outside of some container environment (normally considered a "development" environment running right on the existing OS).

As indicated above, which local configuration file is loaded by a service can be altered by using one of both optional startup options. The first option (-profile when using command line) specifies a suffix used to construct the configuration filename. If no profile is specified, the default filename (configuration.toml) is used. The second option (-confdir today when using the command line) specifies the location of configuration file in the file system. The default configuration directory today is "./res".

For example, if the following command line parameters are given: -registry=false –profile=docker –confdir=/res, it would mean that the service would load the configuration from the file /res/configuration-docker.toml.

Commented [Tony Espy7]: See previous comment about this option.

When running with the registry, users can create multiple configuration files designated with additional profile suffixes ("-profile-name"). For example, additional configuration files may be provided for deployment circumstances (configuration-docker.toml) or different language variants (configuration-pythong.toml) – although language variants are discouraged and it is recommended that the configuration for all languages adhere to the configuration standardization. The config-seed service shall be modified so that an additional profile bootstrap property is provided to config-seed service (today edgex-config-seed). At runtime, the profile property will dictate which of the profile configurations to load and use to initialize the registry.

The –profile option will be used in the core-config-seed (as it is used with the other services) to specify which configuration gets loaded into the registry.

This approach helps to simplify what a user would see in the registry. Only configuration data to be used by the services (as dictated by the profile) is loaded and seen through Consul (versus seeing all the profiles and all the configuration options today). Profiles that are not being used will not be loaded into the configuration/registration service. So, for example, if the bootstrap property supplied to the config seed service was “docker”, only service configuration related to Docker deployments would be seen in Consul and used by the EdgeX services.

Because the config-seed could be used to load any alternate configuration by profile designation, 3rd parties could provide their own configuration files in the config-seed service. This would allow for 3rd parties to deploy EdgeX to alternate and unconsidered environments.