Application Functions SDK Current and Future State

The App Functions Software Development Kit (SDK) is a library that is available for developers to extract and consume events from core data in the EdgeX Framework. Currently the only supported language is Golang, with the intention that community developed and supported SDKs will come in the future for other languages. It is currently available as a Golang module to remain operating system (OS) agnostic and to comply with the latest EdgeX guidelines on dependency management.

The intention of the SDK is to address the scalability concerns of the existing Client and Export Services as well as provide a flexible enough solution for exporting data outside of EdgeX without encumbering the EdgeX development community itself with trying to support all major cloud providers and export solutions. For the Edinburgh release cycle, the existing Client and Export Service remain supported and are still considered the primary way to export data out of EdgeX. However, it is encouraged for new development efforts adopting EdgeX that the App Functions SDK be leveraged moving forward with the intention that by the Fuji release, the SDK will be moved into release status and become the primary method of consuming data from EdgeX.

With the current export services, developers register their endpoints or MQTT clients with the provided registration services and as events are consumed from Core Data, the export service would then relay that information to the registered endpoints in a sequential fashion. Requiring the individual export service to rebroadcast data to all registered endpoints overtime creates a bottleneck and leaves applications with a potential delay in receiving events. Furthermore, the overhead and complexity of managing all registered endpoints becomes an added burden to EdgeX services. Finally, the Export services have also begun to address a bit more than is sustainable in regard to supporting all the major cloud provider endpoints. Providing a SDK and removing cloud specific exports is one way to remain agnostic to cloud providers and enables 3rd parties to support their use of any given cloud solution and eliminates the dependency on EdgeX to support the ever-changing cloud environment.

Thus, the Application Working Group decided that providing a SDK that connects directly to a message bus by which Core Data events are published eliminates performance issues as well as allow the developers extra control on what happens with that data as soon as it is available. Furthermore, it emphasizes configuration over registration for consuming the data. The application services can be customized to a client's needs and thereby also removing the need for client registration.

Before deciding on a SDK we evaluated Function-as-a-service (FaaS) frameworks and determined viability of leveraging what currently exists today. Ultimately, we decided that frameworks such as OpenFaaS, Nuclio, and others were a bit too heavy for running at the edge. Primary reasons include performance, CI/CD pipeline implications, deployment, and the amount of “machinery” or framework code that must be running in order to execute a full-fledged FaaS environment. After looking at the options available, the Applications Working Group decided the best move would be adopt the “spirit of FaaS” in an SDK with the intention of revisiting a full FaaS framework at a later time.

To the Applications Working group, capturing the “spirit of FaaS” meant providing the following pieces of functionality.

First, the SDK needed to provide the necessary plumbing to connect to and receive EdgeX events from Core Data. In other words, similar to FaaS in that the mechanism by which your set of functions ("pipeline") is triggered is abstracted away and determined solely by configuration as specified.

Secondly, provide a standardized and traceable way of passing data from one function to the next. This is realized by a "MessageEnvelope" that contains the EdgeX CorrelationID as well the payload itself which contains the []byte of information. The primary goal with this concept is to enable future expansion of metadata as necessary to be attached with the envelope as its moved about in the EdgeX system.

Third, encourage the idea of developing small, stateless, isolated units of work that allow applications developers to focus on the logic of their application, and easily support scalability of an application that leverages the app-functions-sdk in the future.

Lastly, allow applications to leverage configuration as provided by the registry to be able to adjust settings of an application such as message bus subscription/publish topics without having to recompile and redeploy applications

With these concepts in mind, the Application Functions SDK was developed and currently encompasses the following features:

One of the most common use cases for working with data that come from CoreData is to filter data down to what is relevant for a given application and to format it. To help facilitate this, four primary functions ported over from the existing services today are included in the SDK. The first is the FilterByDeviceID function which will remove events that do not match the specified IDs and will cease execution of the pipeline if no event matches. The second is the FilterByValueDescriptor which exhibits the same behavior as FilterByDeviceID except filtering on Value Descriptor instead of DeviceID.

The third and fourth provided functions in the SDK transform the data received to either XML or JSON by calling TransformToXML() or TransformToJSON().

Typically, after filtering and transforming the data as needed, exporting is the last step in a pipeline to ship the data where it needs to go. There are two primary functions included in the SDK to help facilitate this. The first is HTTPPost(string url) function that will POST the provided data to a specified endpoint, and the second is an MQTTPublish() function that will publish the provided data to an MQTT Broker as specified in the configuration.

There was a conscious decision by the team to NOT support authenticated HTTPS endpoints as well as MQTTs endpoints for the Edinburgh Release Cycle. These items will be addressed once EdgeX has full support for secret management as well as Vault integration.

The SDK leverages the abstract registry/consul integration from go-mod-registry **(Appendix A)** for supporting configuration management.

Structure logging is included as part of the SDK and is used by the SDK internally and is also available as part of the context during each pipeline execution for developers to leverage in

provided functions. The library used for logging is provided by go-mod-core-contracts **(Appendix B).**

System Management Agent Integration is included as well by supporting /metrics, /config, and /ping endpoints. Adding application services to the system management agent manifest of services would also allow the application services to be started, stopped and restarted.

There are two primary triggers that have been included in the SDK that initiate the start of the function pipeline. First is via a POST HTTP Endpoint "/trigger" with the EdgeX event data as the body. Second is the MessageBus subscription with connection details as specified in the configuration. See the documentation provided in the Readme.md for more information **(Appendix C)**

Finally, data may be sent back to the message bus or HTTP response by calling.complete() on the context. If the trigger is HTTP, then it will be an HTTP Response. If the trigger is MessageBus, then it will be published to the configured host and topic.

There are three example applications provided in the app-functions-sdk-go repository in the /examples directory that attempt to show basic structure of building an application with the app functions sdk. They also focus on how to leverage various built in provided functions as mentioned above as well as how to write your own in the case that the SDK does not provide what is needed.

1. Simple Filter XML -> Demonstrates Filter of data by device ID and transforming data to XML
2. Simple Filter XML Post -> Same example as #1, but result published to HTTP Endpoint
3. Simple Filter XML MQTT -> Same example as #1, but result published to MQTT Broker

The features in the initial implementation of the App Functions SDK should be sufficient to provide the foundation for basic filtering and export needs. There are some functions in the existing export services that are not yet available in application functions and are intended to be included in a later release. This includes the Encryption Transformer, the Compressor Transformer, and Valid Event Check **(See Appendix D)**. The primary purpose for leaving this out was to address core pieces of functionality that would set up the ease of adding additional functions in the future.

The above has addressed the current state of the Application Functions SDK. This last section aims to help tee up the discussion for items that have been slated for the Fuji release of EdgeX.

Fuji Release Topics

The following is the set of features that the meeting group settled upon as a reasonable set of target features for Fuji for October 2019 (generally assuming the same resource contributions as provided by the community today).

1) Be able to archive the current Export Services and Export Client micro services (meaning be able to provide all of the same functionality as provided by the export services today). The exception to this is that the new application services will not provide all the cloud endpoint distribution mechanisms (Azure IoT, Google IoT Core, AWS IoT, etc.). The basis of most of the cloud distribution endpoints utilize MQTT or MQTTS at the base. In supporting MQTT/S and HTTP/S, the platform allows for EdgeX to support 3rd parties to build the application services to support cloud endpoints, but without the project actually developing and maintaining this code long term.
2) Be able to send data to the rules engine. Do we get rid of the rules engine? Currently Java based and receives data via ZMQ -> Distro Services -> Rules Engine.
3) Provide tutorials on how to build a cloud-supporting endpoint (Azure, AWS, etc.) to help those with cloud needs –but without providing the code directly as part of the project itself.
4) Provide the first attempt (even if partially) to solve the command problem –that is how to provide the EdgeX device commands to the north side.
5) Provide Vault integration –allowing the secrets, certificates, tokens needed to connect to secure endpoints or to encrypt the north bound data to be retrieved from secure storage and not stored in the open or in Consul unprotected.
6) Expand triggers
    a. Timer based – allow execution of a given pipeline based on a timer interval
7) Expand current set of built in transforms:
    a. Filters
        i. value-ranges or data related, value out of range/non-compliant
        ii. select for one or more devices or value descriptor (today's only does one)
        iii. location based/geo-fenced
        iv. De-duplication values (eliminate repeated posts of 72 degrees for example)
        v. Eliminate noise from readings related to precision (considering 0.1112 versus 0.1115 when the reading is only accurate to 2 or 3 decimals)
        vi. Time based
    b. Format transforms:
        i. CSV, YAML, TOML, RAML
        ii. 3rd party formats (Haystack, etc.); although the concern here is that EdgeX have to maintain all of these; consider allowing integration of your own formatter without EdgeX owning the transform
    c. Transform
        i. Cloud ready (Azure, AWS, etc.)
        ii. Unit conversion (F to C, feet to meters, etc.)
    d. Encryption (payloads) – depends on vault integration
    e. Signing – depends on vault integration – near last step in pipeline
    f. Compression
        i. ZIP
        ii. TAR.GZ
        iii. Lossy vs lossless compression especially as it pertains to audio/video
    g. Media transformation for binary data (JPG to GIF, MPEG to WAV, etc.)
        i. Codec implications

h. Enrich (metadata additions, commands, ...)
i. Additional endpoints – maybe additional protocols
j. HTTPS and MQTTS (with token exchange via OAuth, JWT, ...)

Other considerations

Binary data (CBOR) in the new application services should not require a separate functions SDK. Further, these application services should be managed, orchestrated or built the same as application services that handle non-binary data. The application service functions may obviously be implemented to process binary versus non-binary data.

Multi-tenancy for application services is considered out of scope. There may be a need for data to be tagged for a particular tenant or in a certain way to ultimately serve a tenant's needs, but otherwise it was the opinion of the meeting group that multi-tenancy is handled above the level of EdgeX.

On the issue of scale, multiple instances of application service should be viewed as the way to address the volume of traffic to the application services (using a load balancer to spread traffic to copies of the same application services). We discussed and considered whether the application services or framework would potentially do the load balancing or routing to favor particular scale needs. Unanimously, this was considered a bad idea and one that would make the services more complex and potentially brittle. We also decided that the should be the message bus should be responsible for being able to scale as well.

While not part of application services, there was some discussion about finding a way to detect and report a sensor not sending data. This may or may not include some sort of function in the application services.

Appendices

Appendix A (go-mod-registry)

https://github.com/edgexfoundry/go-mod-registry

Registry client library for use by Go implementation of EdgeX micro services. This project contains the abstract Registry interface and an implementation for Consul. These interface functions initialize a connection to the Registry service, registering the service for discovery and health checks, push and pull configuration values to/from the Registry service and pull dependent service endpoint information and status.

Appendix B (go-mod-core-contracts)

https://github.com/edgexfoundry/go-mod-core-contracts

This module contains the contract models used to describe data as it is passed via Request/Response between various EdgeX Foundry services. It also contains service clients for each service within the edgex-go repository. The definition of the various models and clients can be found in their respective top-level directories.

The default encoding for the models is JSON, although in at least one case -- DeviceProfile -- YAML encoding is also supported since a device profile is defined as a YAML document.

Appendix C – Message Bus Trigger

A message bus trigger will execute the pipeline everytime data is received off of the configured topic.

Type and Topic configuration

Here's an example:

```
Type="messagebus"
SubscribeTopic="events"
PublishTopic=""
```

The `Type=` is set to "messagebus". EdgeX Core Data is publishing data to the `events` topic. So to receive data from core data, you can set your `SubscribeTopic=` either to `""` or `"events"`. You may also designate a `PublishTopic=` if you wish to publish data back to the message bus. `edgexcontext.complete([]byte outputData)` - Will send data back to back to the message bus with the topic specified in the `PublishTopic=` property

Message bus connection configuration

The other piece of configuration required are the connection settings:

```
[MessageBus]
Type = 'zero' #specifies of message bus (i.e zero for ZMQ)
    [MessageBus.PublishHost]
        Host = '*'
        Port = 5564
        Protocol = 'tcp'
    [MessageBus.SubscribeHost]
        Host = 'localhost'
        Port = 5563
        Protocol = 'tcp'
```

By default, `EdgeX Core Data` publishes data to the `events` topic on port 5563. The publish host is used if publishing data back to the message bus.

Important Note: Publish Host **MUST** be different for every topic you wish to publish to since the SDK will bind to the specific port. 5563 for example cannot be used to publish since `EdgeX Core Data` has bound to that port. Similarly, you cannot have two separate instances of the app functions SDK running publishing to the same port.

Appendix D – Existing export service functions

https://docs.edgexfoundry.org/Ch-Distribution.html

Compressor Transformer–A transformer component compresses the data string to be delivered to the clients, for any clients that have requested their data be compressed either by GZIP or ZIP methods.

Encryption Transformer–An encryption component encrypts the data to be sent to the client, using the client provided keys and vectors.

Valid Event Check–The first component in the pipe and filter, before the copier (described in the previous section) is a filter that can be optionally turned on or off by configuration. This filter is a general purpose data checking filter which assesses the device- or sensor-provided Event, with associated Readings, and ensures the data conforms to the ValueDescriptor associated with the Readings.

- For example, if the data from a sensor is described by its metadata profile as adhering to a “Temperature” value descriptor of floating number type, with the value between -100° F and 200° F, but the data seen in the Event and Readings is not a floating point number, for example if the data in the reading is a word such as “cold,” instead of a number, then the Event is rejected (no client receives the data) and no further processing is accomplished on the Event by the Export Distro service.