Application Work Group Face-to-Face meeting notes

1/24/19

Application Services – general design guidelines

- The application services will be built on a model that each application service is a single executable that includes a "framework" that includes code to get messages off of and onto a message bus. Which functions will be called on will be determined not by the SDK or framework but by a developers own application service function selection/orchestration.
- Go SDK (deemed "app functions SDK") will be placed in a single and separate repo.
    - Other language SDK's may be created in the future or be provided by 3rd parties. These would be in their own repo.
    - We should try to use modules as part of the SDK implementation (research to be accomplished)
- The SDK should ultimately offer a fixed (or semi-fixed) set of functions.
    - Under the mantra of crawl, walk, run, the SDK in Edinburgh should focus on just a couple of functions.
    - We need to avoid a scale issue of trying to address all endpoints in particular. Cloud providers and different protocol support could cause the SDK and application services to grow in an unsupported manner.
    - 3rd parties can add their own functions but the project should avoid incorporating these in the project's source and SDK unless this is viewed as a common and compelling need.
    - Functions are each defined in their own package
    - The SDK should allow tracing of execution into and out of the application service but not into and out of each function – via the correlation ID
- Developers will create new application services using the app function SDK.
    - The application services should be created in their own repo.
    - Like the DS SDK, these application services will reference the app function SDK.
    - The application service will have a function (like a main) that picks/calls the SDK provided functions or custom functions in the application service.
    - The application service project can include custom functions not found in the SDK.
    - The application service should create the executable that runs in the EdgeX deployment as one of potentially many north bound services.
    - The input/output topics for the application services must be provided via config.

Edinburgh Release Tasks and Assignments

- Create the message bus abstraction (interface). [DELL to work]
    - This work will be done in a Go module
- Create a Message envelope that includes the Correlation ID [DELL to work]
- Provide an implementation of the message bus abstraction/interface and message envelope in Core Data and Export Distro [DELL to work]
    - Use ZeroMQ in the implementation for Edinburgh
- Implement the Application Service's Functions SDK [INTEL to work]
    - Jim to get repo ready in holding

- The SDK will be made in a repo called app-functions-sdk-go (allowing future SDKs for other languages)
- The SDK will include Consul/config, logging and SMA integration
- The SDK will incorporate just two functions to start: Filter (device and value descriptor filters) and Formatter for JSON & XML (be a feature replication of the current Export Distro implementation)
- The SDK will implement two endpoint types: MQTT and HTTP
  - Piping data to the rules engine will be in a later release
  - A stretch goal is to offer a demo/tutorial on how to get data to a cloud endpoint (like Azure IoT or AWS IoT)
- Implement the SDK as a module (some research to be done)
- Provide at least a single example Application Service using the SDK
  - This may be delivered as part of the SDK like the example device service is provided with the Device Service SDK

1/25/19

Future Road-mapping

The following is a set of features and work items for application services for future releases. These features and work items will not be provided as part of the Edinburgh release.

- Create a more complete message envelope around the Event/Readings to pass into and through the application services and their associated functions. A message structure is required to send the Event/Readings, correlation id, any necessary metadata and the resulting byte array or string into and out of the functions that perform work in the application services. In other words, the message envelop allows a common structure to be used in calling on each function with the same inputs and outputs in an application service. Particular functions would need to use an Event/Readings and produce the byte array or string (example: JSON or XML formatters), while other functions use an Event/Readings and also produce modified Event/Readings (example: filters) while finally some functions use a byte array/string and produce a byte array/string (compression or encryption).
- Facilitate the export of command information and metadata to systems external to EdgeX (cloud, enterprise, etc.). This allows the 3rd parties to make actuation calls to EdgeX (without having to negotiate the command service blind of any knowledge of the connected devices, services, etc.).
- Allow for and explore alternate message bus under the framework (allowing for improved QoS, distributed bus, etc.)
- Provide the following functions:
  - Filters
    - value-ranges or data related, value out of range/non-compliant
    - remove precision issues
    - select for one or more devices or value descriptor (today's only does one)
    - location based/geo-fenced
    - De-duplication values (eliminate repeated posts of 72 degrees for example)

        - Eliminate noise from readings related to precision (considering 0.1112 versus 0.1115 when the reading is only accurate to 2 or 3 decimals)
        - Time based
    - Format transforms
        - CSV, YAML, TOML, RAML
        - 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
    - Transform
        - Cloud ready (Azure, AWS, etc.)
        - Unit conversion (F to C, feet to meters, etc.)
    - Encryption (payloads)
    - Signing
    - Compression
    - Media transformation for binary data (JPG to GIF, MPEG to WAV, etc.)
    - Enrich (metadata additions, commands, …)
    - Bring your own (custom functions – like Lambda functions)
    - Additional endpoints
        - HTTPS and MQTTS (with token exchange via OAuth, JWT, …)
        - Tutorials and demos for cloud providers but without always providing the cloud connector/endpoint
- Eventually, we want the application services built from the SDK to choose the functions called by configuration. The configuration should also determine the order that functions are called. This has been referred to as function selection and orchestration by config.
    - This may require the creation of the message envelope so that all function signatures are the same (all being passed the same message envelope).

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 for application services (and arguably for EdgeX as a whole). 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). It was 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.

- 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.

Fuji Release

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).

- 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.
- Be able to send data to the rules engine
- Provide a tutorial or two 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.
- 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.
- Provide Vault integration – allowing the secrets, certificates, tokens needed to connected 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.