List of CI/CD Changes for Adopting Go Modules

We want to move away from glide and vendoring for dependency management. Utilizing Go modules will allow us to do that but will require changes to the CI/CD pipeline. This document will attempt to capture a list of those changes at a high level.

One thing to note – Where code changes are referenced, assume "edgex-go" as the example repository. These changes will need to be made to any repo that is to integrate with modules, but just be aware that in my mind as I'm writing this, I'm thinking of edgex-go and making it our first use case.

1.) Code changes
    a. Developer needs to create a PR wherein a "go.mod" file containing all of the relevant dependencies, similar to the content in the glide.yaml, is added to the repo.
2.) CI Build Agent
    a. Agent should have Go 1.11.2 minimum installed
    b. Agent should perform a git clone of the repository to a location *outside of the GOPATH.*
        i. If for some reason working in the GOPATH is preferrable then the following environment variable must be set -- `GO111MODULE=on`
    c. Eliminate `make prepare` from Jenkins verify job
        i. Also in makefile
    d. Agent executes `make test` as done today
        i. The combination of the new `go` command plus the go.mod file will cause an import of all relevant dependencies at this time.
        ii. https://github.com/golang/go/wiki/Modules
            1. Standard commands like `go build` or `go test` will automatically add new dependencies as needed to satisfy imports (updating `go.mod` and downloading the new dependencies).
    e. Agent executes `make build && make docker` as done today
3.) Dockerfile changes
    a. WORKDIR currently points to location in the GOPATH. Change this or else set the GO111MODULE environment variable.
    b. CONCERN: With the removal of the vendor directory the line `COPY . .` will no longer copy dependencies from the build agent into the docker image. Therefore when a `go build` is performed inside the image, it will cause the module to download its dependencies. This will be done for every docker image in the repo, adding to build times
        i. Review the following PR for possible mitigation
        ii. https://github.com/edgexfoundry/ci-management/pull/261
4.) Tags

a. We need to change the style of our tags for any repos we want to publish as modules. Currently, our versioning scheme does not include a "v" before the version number. Example:
    i. https://github.com/edgexfoundry/edgex-go/releases/tag/0.7.1
b. When a module refers to the above release, the version is lost the first time a *go build* is run. Go will replace the version specified in the go.mod file with a combination timestamp + SHA. The following example reflects a dependency on the version above:
    i. github.com/edgexfoundry/edgex-go v0.0.0-20181210184951-f7a0b44a0188
c. In order to fix this, we need to tag our releases using "v0.7.1" instead of simply "0.7.1". We can then refer to them in a go.mod like so without the explicit version getting lost:
    i. github.com/edgexfoundry/edgex-go v0.7.1
        1. Yes, the leading "v" is required. See the "From Repository to Modules" section of this article.
            a. https://research.swtch.com/vgo-module

5.) Versioning of modules
    a. See additional section below

Use of Versioning With Modules Within Edgex-go and Related Repositories

Go modules utilize a versioning approach called "semantic import versioning"[1]. This is actually a combination of two sub-approaches to versioning:

1.) Import uniqueness
    a. This requires that the code defined within a given package will always be backward compatible. That is, any code that depends on a given package should always be guaranteed of a successful build with that package as well as consistent functionality.
2.) Semantic versioning
    a. The mechanism of versioning identifies changes of a codebase through time via the use of major version, minor version and patch (e.g. v2.1.3)

The recommended practice to indicate module versioning within the import path only targets those modules that are past version 1. Version 1 modules do not require additional information. Once a module revs to version 2 and beyond, it is recommended to add version information to the import path of a package. For example, if my application imports "github.com/edgexfoundry/edgex-go/pkg/contracts" then I am implicitly working with V1 of that package. According to the import uniqueness requirement above, I am guaranteed to

[1] For more information, see the original Go versioning proposal -- https://blog.golang.org/versioning-proposal

never receive a breaking change in that package[2]. Once a new major version of the edgex-go/pkg module is published, then the publisher will provide a new import path – "github.com/edgexfoundry/edgex-go/pkg/V2/contracts".

In practice this means that the following two directories in the same repo are tagged differently:
github.com/edgexfoundry/edgex-go/pkg/contracts – tagged v1.0.0
github.com/edgexfoundry/edgex-go/pkg/V2/contracts – tagged v2.0.0

After the v2.0.0 release, the v1.0.0 tag must remain for those applications that still import the older versioned module. In our current DevOps process, we tag the top-level repo itself and the release tag applies to everything. How do we manage tags within sub-directories that are used not for product release purposes, but for dependency management?

During the recent Application Services face-to-face meeting, we discussed the possibility of exposing Message Queue and Service Registry abstractions to multiple services. It is undesirable to go on copying a pattern from one repo to another. One option is to simply put these previously encapsulated packages into the externally available /pkg folder, but an obvious alternative is to expose these capabilities as modules[3].

An example illustration of a possible direction for our code organization and subsequent DevOps practices might look like this:

```
/edgex-go                                          tag: v1.2.0
    /modules
        /messaging                                 tag: v1.0.0
        /registry                                  tag: v1.0.1
        /contracts                                 tag: v1.0.0
            /clients (currently /pkg/clients)
            /models (currently /pkg/models)
```

With versioning starting at version 2, the paths might look like this:

```
/edgex-go                                          tag: v2.5.3
    /modules
        /messaging/V2                              tag: v2.0.0
        /registry/V2                               tag: v2.1.0
        /contracts/V2                              tag: v2.0.0
            /clients (currently /pkg/clients)
            /models (currently /pkg/models)
```

[2] Obviously we have not been adhering to this so far in our project. Subsequent to the Edinburgh release, it will be a requirement.
[3] This is starting to look more like the practice of shared libraries common to many other languages.

Potential application dependencies might look something like this. Note that just because a module is found in a sub-folder of a parent's repository, that does not mean it is included with the parent module. Modules are indicated by the presence of a go.mod file. They have nothing to do with the repo structure:

- Device-sdk-go → edgex-go/modules/contracts ← edgex-go
- Device-sdk-go → edgex-go/modules/registry ← edgex-go
- Application-services → edgex-go/modules/messaging ← edgex-go
- Application-services → edgex-go/modules/registry ← edgex-go

One other thing to note is that while modules do require a main.go in their root just like any other Go program, this file does not have to contain a main() function. Its minimum requirement is to declare the root package of the module. Obviously a package (library) of re-usable types doesn't need to be executed on its own.

To summarize:

1.) We should identify what challenges, if any, may come to light as the result of tagging subdirectories within a larger repo. The tag applied to the parent repo (edgex-go) equates to a product release. Tags applied to sub-directories (like edgex-go/modules) are merely version dependencies.
2.) The parent repo itself does still have to be a module since modules are the manner in which Go will manage dependencies going forward. This is how we get rid of glide.
3.) We should discuss the manner in which our code should be organized to facilitate publication of EdgeX related modules. Is the above proposal sound?

Further recommended background reading
https://research.swtch.com/vgo-module