Go

These guidelines for writing Go code are WIP

Overview

This document covers common coding styles and guidelines for all ForgeRock products.

Copyright notices

Within the FRaaS codebases we are not currently adding licence headers to individual source files. This practice diverges from the standard practice of doing so across other projects.

The ForgeRock Go coding style

Documentation

- Each Go project should include one or more README.md files detailing
  - An overview of the project and its purpose
  - How to build, test, deploy and configure the executable
- All public constants, structs, fields, interfaces and functions must have Go doc
- All packages must have Go doc - Packages with more than one source file should consider providing package documentation using a doc.go file

Source code layout

In addition to good documentation, having a consistent approach to organising code across directories and within a given source file makes it easier for engineers to move between projects and get up to speed quickly.

- Each Go project should use a standard layout
- Each Go source file should, as far as possible, be readable top-to-bottom with public / high level functions at the top of source files and private / helper functions lower down:
  - vars/consts
  - interfaces
  - structs
  - methods
  - constructors
  - public functions
  - private functions

Linting rules

Where possible, agreed standards relating to source files should be enforced by linting during continuous integration. The linting rules currently in use by the FRaaS team are:

golangci-lint_config_base.yml

```
linters:
  # inverted configuration with `enable-all` and `disable` is not scalable during updates of golangci-lint
  disable-all: true
  enable:
    - deadcode
    - errcheck
    - gofmt
    - goimports
    - gosimple
    - govet
    - ineffassign
    - structcheck
    - typecheck
    - unused
    - varcheck

run:
  tests: true
```

Logging

- Avoid logging directly at the site where an error is produced or returned. Instead let the entry point to processing do the logging. If the returned error message is insufficient (often they are already sufficient) use `errors.Wrap` to add context to the returned error.
- Use the `github.com/pkg/errors` package instead of `errors`, and user `errors.WithStack(err)` wherever an error is produced or returned from an external package. `errors.WithStack` produces a stack trace pointing to the line of code which produced the error, which also prevents us from having to add our own custom error message so that we can correlate the error message to the line of code which produced it.This can also be done wherever there's an `errors.New`, I.E `errors.WithStack(errors.New("my error"))`.

Idiomatic Go

In addition to the points raised above, we should endeavour to write idiomatic Go. Guidance for what these idioms are and how to follow them can be found in:

- Effective Go
- https://github.com/golang/go/wiki/CodeReviewComments

Commonly used libraries

- (Pending review and approval from others) Validator validates struct data to ensure input matches what's required. github.com/go-playground/validator