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

Quick dos/donts

- Don't use os.Exit (or logging.Fatal) anywhere except the main function - this stops any 'defer' statements from running and also prints no message when it exits, which can cause confusion when trying to debug why a program suddenly stopped. If something has failed in such a way that it can't recover, panic instead. If you can't do anything with the error, just return the error from the function you are in (and handle this case further up).
- If you are returning an error from another package (or a standard library function), wrap it with errors.Wrap from github.com/pkg/errors which will add context to the error to make it easier to understand where it comes from. If you are creating a new error, use errors.New (or errors.Errorf instead of fmt.Errorf) from the same library which provide the same utility.

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"))`.
- Any new packages that need to log things should use the common logger specified in common/pkg/logging/logging.go. This can be used by importing in your package (in package.go, or another file)

  go log example

  ```
  package newpackage

  import (
          "github.com/ForgeCloud/saas/go/common/pkg/logging"
  )

  var (
          log = logging.Recorder
  )
  ```

- Programs should only exit in a 'main' function, and should log which error caused the program to fail at the 'Panic' level. This will make it obvious when looking at logs where and why the program exited.

  go error handling example

  ```
  func main() {
          // ... do some setup ...
          err := runProgramForReal()
          if err != nil {
                  log.WithError(err).Panic("Error while running program")
                  // or
                  // panic(err)
          }
  }
  ```

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

Mocking

Mocking is done using https://github.com/vektra/mockery which generates mocks using https://github.com/stretchr/testify.

What not to do

- Do not write a 'custom' mock - eg, write a struct which implements the interface you want to mock. If somebody needs to change the interface further down the line, they will then need to modify your mock as well. When using an autogenerated mock, they can just run 'go generate' to recreate it.
- Do not use a mock for an interface in the same package. Mocks should provide a description of the interface for something in the package - the package itself should not need to test the mock.

Generating mocks

When adding a new interface that will require a mock there are a few simple steps to follow

1. Follow the instructions to install mockery
2. Add a line either above the interface you've added or in a package.go file alongside the interface with a 'go generate' comment
3. Run 'go generate ./...' in the relevant folder - this will run 'mockery' and generate a mock for your interface under the 'mocks' folder

Your interface should look like this:

go mocking example

```
//go:generate mockery --all

// ThingDoer does a thing
type ThingDoer interface {
        DoThing() (error)
}
```

If you need to generate a mock for an interface but you also need to use that mock in the same package, this will cause import cycles. To get around this, add '--inpackage' to the list of arguments to 'mockery' in the generate comment.

Using mocks

If you want to use one of these mocks in one of your tests, there is a small utility function in go/common/pkg/testutil/mockhelper.go which can be used to do some common setup and mock assertion. An example of how to use this:

go mock example

```
func TestSomething(t *testing.T) {
        // create mock controller
        ctrl := testutil.NewController(t)
        // defer 'finish' - this will check that your mock assertions are satisfied
        defer ctrl.Finish()

        // Create the mock object
        mockThingdoer := &mocks.Thingdoer{}

        // register the mock object with the controller - this will prevent mock assertions from instantly
panicking, and will check for assertions at the end of the test
        ctrl.Register(&mockThingdoer.Mock)

        // Set mock assertions
        mockThingDoer.On("DoThing").Return(nil)

        // call your function, do basic checks
        err := useThingdoer(mockThingdoer)
        assert.NoError(t, err)

        // If 'DoThing' was not called, mockery will fail the test when the function ends
}
```

The testify documentation will have more information about what methods these mocks will have and how to use them.

Creating GCP API clients

When creating an API client (for example, a logging client that is used to read logs from the GCP API) and you only ever want to read data, pass a 'WithScope' option to the constructor to limit what the client can do:

```
crmService, err := cloudresourcemanager.NewService(ctx, option.WithScopes(cloudresourcemanager.
CloudPlatformReadOnlyScope))
if err != nil {
        return errors.Wrap(err, "creating service")
}

...
```

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
- Test assertions with nice output https://github.com/stretchr/testify
- Try to use the cloud.google.com/go/... client libraries rather than google.golang.org/... if one exists
- github.com/spf13/cobra and github.com/spf13/viper for command line utilities
- https://pkg.go.dev/golang.org/x/sync/errgroup for a more intuitive way to do a group of work in parallel
- When setting up logging, use the common logger in go/common/pkg/logging/logging.go (which under the hood uses https://github.com/sirupsen/logrus)
- github.com/gin-gonic/gin for HTTP servers
- github.com/hashicorp/go-multierror for handling a case where multiple errors might be accumulated in a loop