Backend style guide

Grafana's backend has been developed for a long time with a mix of code styles. This guide explains how we want to write Go code in the future.

Unless stated otherwise, use the guidelines listed in the following articles:

- Effective Go
- Code Review Comments
- Go: Best Practices for Production Environments

Linting and formatting

To ensure consistency across the Go codebase, we require all code to pass a number of linter checks.

We use the standard following linters:

- gofmt
- golint
- go vet

In addition to the standard linters, we also use:

- revive with a custom config
- GolangCI-Lint
- gosec

To run all linters, use the `lint-go` Makefile target:

```
make lint-go
```

Testing

We value clean and readable code, that is loosely coupled and covered by unit tests. This makes it easier to collaborate and maintain the code.

Tests must use the standard library, `testing`. For assertions, prefer using testify.

The majority of our tests uses GoConvey but that's something we want to avoid going forward.

In the `sqlstore` package we do database operations in tests and while some might say that's not suited for unit tests. We think they are fast enough and provide a lot of value.

Assertions

Use respectively `assert.*` functions to make assertions that should *not* halt the test ("soft checks") and `require.*` functions to make assertions that *should* halt the test ("hard checks"). Typically you want to use the latter type of check to assert that errors have or have not happened, since continuing the test after such an assertion fails is chaotic (the system under test will be in an undefined state) and you'll often have segfaults in practice.

Sub-tests

Use `t.Run` to group sub-test cases, since it allows common setup and teardown code, plus lets you run each test case in isolation when debugging. Don't use `t.Run` to e.g. group assertions.

Cleanup

Use `t.Cleanup` to clean up resources in tests. It's a less fragile choice than `defer`, since it's independent of which function you call it in. It will always execute after the test is over in reverse call order (last `t.Cleanup` first, same as `defer`).

Mock

Optionally, we use `mock.Mock` package to generate mocks. This is useful when you expect different behaviours of the same function.

Tips

- Use `Once()` or `Times(n)` to make this mock only works `n` times.
- Use `mockedClass.AssertExpectations(t)` to guarantee that the mock is called the times asked.
  - If any mock set is not called or its expects more calls, the test fails.
- You can pass `mock.Anything` as argument if you don't care about the argument passed.
- Use `mockedClass.AssertNotCalled(t, "FunctionName")` to assert that this test is not called.

Example

This is an example to easily create a mock of an interface.

Given this interface:

```go
func MyInterface interface {
    Get(ctx context.Context, id string) (Object, error)
}
```

Mock implementation should be like this:

```go
import

func MockImplementation struct {
    mock.Mock
}

func (m *MockImplementation) Get(ctx context.Context, id string) error {
    args := m.Called(ctx, id) // Pass all arguments in order here
    return args.Get(0).(Object), args.Error(1)
}
```

And use it as the following way:

```go

objectToReturn := Object{Message: "abc"}
errToReturn := errors.New("my error")

myMock := &MockImplementation{}
defer myMock.AssertExpectations(t)

myMock.On("Get", mock.Anything, "id1").Return(objectToReturn, errToReturn).Once()
myMock.On("Get", mock.Anything, "id2").Return(Object{}, nil).Once()

anyService := NewService(myMock)
resp, err := anyService.Call("id1")

assert.Equal(t, resp.Message, objectToReturn.Message)
assert.Error(t, err, errToReturn)

resp, err = anyService.Call("id2")
assert.Nil(t, err)
```

Mockery

When an interface to test is too big, it's annoying to mock each function manually. To avoid this, you can use `mockery` library to generate the mocks.

The command is like the following (there are more options documented if you need to use another one):

```
mockery --name InterfaceName --structname MockImplementationName --inpackage --
filename my_implementation_mock.go
```

- `--name` : Interface to mock
- `--structname` : Mock implementation name
- `--inpackage` : To use the same package name as the interface
- `--filename` : Your mock generated file name

If any interface signature changes, executing the command again updates the mock.

Additionally, you can put `go:generate` command on the top of the file as a comment. It's useful because some IDEs like Goland and Visual Studio Code allows executing scripts from the IDE.

```
package <package>

import (
    ...
)

//go:generate mockery --name InterfaceName --structname MockImplementationName --
inpackage --filename my_implementation_mock.go
```

Globals

As a general rule of thumb, avoid using global variables, since they make the code difficult to maintain and reason about, and to write tests for. The Grafana codebase currently does use a lot of global variables, especially when it comes to configuration, but that is a problem we're trying to solve.

Pointers

In general, use value types and only reach for pointers when there's a real need. The reason being that pointers increase the risk of bugs, since a pointer can be nil and dereferencing a nil pointer leads to a panic (AKA segfault). Valid reasons to use a pointer include (but not necessarily limited to):

- You might need to pass a modifiable argument to a function
- Copying an object might incur a performance hit (benchmark to check your assumptions, copying is often faster than allocating heap memory)
- You might *need* `nil` to tell if a variable isn't set, although usually it's better to use the type's zero value to tell instead

Database

In database related code, we follow certain patterns.

Foreign keys

While they can be useful, we don't generally use foreign key constraints in Grafana, for historical and technical reasons. See this comment by Torkel for context.

Unique columns

If a column, or column combination, should be unique, add a corresponding uniqueness constraint through a migration.

JSON

The simplejson package is used a lot throughout the backend codebase, but it's legacy, so if at all possible avoid using it in new code. Use json-iterator instead, which is a more performant drop-in alternative to the standard encoding/json package. While encoding/json is a fine choice, profiling shows that json-iterator may be 3-4 times more efficient for encoding. We haven't profiled its parsing performance yet, but according to json-iterator's own benchmarks, it appears even more superior in this department.