Contributing to opentelemetry-go

The Go special interest group (SIG) meets regularly. See the OpenTelemetry community repo for information on this and other language SIGs.

See the public meeting notes for a summary description of past meetings. To request edit access, join the meeting or get in touch on Slack.

Development

You can view and edit the source code by cloning this repository:

```
git clone https://github.com/open-telemetry/opentelemetry-go.git
```

Run `make test` to run the tests instead of `go test`.

There are some generated files checked into the repo. To make sure that the generated files are up-to-date, run `make` (or `make precommit` - the `precommit` target is the default).

The `precommit` target also fixes the formatting of the code and checks the status of the go module files.

If after running `make precommit` the output of `git status` contains `nothing to commit, working tree clean` then it means that everything is up-to-date and properly formatted.

Pull Requests

How to Send Pull Requests

Everyone is welcome to contribute code to `opentelemetry-go` via GitHub pull requests (PRs).

To create a new PR, fork the project in GitHub and clone the upstream repo:

```
$ go get -d go.opentelemetry.io/otel
```

(This may print some warning about "build constraints exclude all Go files", just ignore it.)

This will put the project in `${GOPATH}/src/go.opentelemetry.io/otel`. You can alternatively use `git` directly with:

```
$ git clone https://github.com/open-telemetry/opentelemetry-go
```

(Note that `git clone` is *not* using the `go.opentelemetry.io/otel` name - that name is a kind of a redirector to GitHub that `go get` can understand, but `git` does not.)

This would put the project in the `opentelemetry-go` directory in current working directory.

Enter the newly created directory and add your fork as a new remote:

```
$ git remote add <YOUR_FORK> git@github.com:<YOUR_GITHUB_USERNAME>/opentelemetry-go
```

Check out a new branch, make modifications, run linters and tests, update `CHANGELOG.md`, and push the branch to your fork:

```
$ git checkout -b <YOUR_BRANCH_NAME>
# edit files
# update changelog
$ make precommit
$ git add -p
$ git commit
$ git push <YOUR_FORK> <YOUR_BRANCH_NAME>
```

Open a pull request against the main `opentelemetry-go` repo. Be sure to add the pull request ID to the entry you added to `CHANGELOG.md`.

How to Receive Comments

- If the PR is not ready for review, please put `[WIP]` in the title, tag it as `work-in-progress`, or mark it as `draft`.
- Make sure CLA is signed and CI is clear.

How to Get PRs Merged

A PR is considered to be **ready to merge** when:

- It has received two approvals from Collaborators/Maintainers (at different companies). This is not enforced through technical means and a PR may be **ready to merge** with a single approval if the change and its approach have been discussed and consensus reached.
- Feedback has been addressed.
- Any substantive changes to your PR will require that you clear any prior Approval reviews, this includes changes resulting from other feedback. Unless the approver explicitly stated that their approval will persist across changes it should be assumed that the PR needs their review again. Other project members (e.g. approvers, maintainers) can help with this if there are any questions or if you forget to clear reviews.
- It has been open for review for at least one working day. This gives people reasonable time to review.
- Trivial changes (typo, cosmetic, doc, etc.) do not have to wait for one day and may be merged with a single Maintainer's approval.
- `CHANGELOG.md` has been updated to reflect what has been added, changed, removed, or fixed.
- Urgent fix can take exception as long as it has been actively communicated.

Any Maintainer can merge the PR once it is **ready to merge**.

Design Choices

As with other OpenTelemetry clients, opentelemetry-go follows the opentelemetry-specification.

It's especially valuable to read through the library guidelines.

Focus on Capabilities, Not Structure Compliance

OpenTelemetry is an evolving specification, one where the desires and use cases are clear, but the method to satisfy those uses cases are not.

As such, Contributions should provide functionality and behavior that conforms to the specification, but the interface and structure is flexible.

It is preferable to have contributions follow the idioms of the language rather than conform to specific API names or argument patterns in the spec.

For a deeper discussion, see: https://github.com/open-telemetry/opentelemetry-specification/issues/165

Style Guide

One of the primary goals of this project is that it is actually used by developers. With this goal in mind the project strives to build user-friendly and idiomatic Go code adhering to the Go community's best practices.

For a non-comprehensive but foundational overview of these best practices the Effective Go documentation is an excellent starting place.

As a convenience for developers building this project the `make precommit` will format, lint, validate, and in some cases fix the changes you plan to submit. This check will need to pass for your changes to be able to be merged.

In addition to idiomatic Go, the project has adopted certain standards for implementations of common patterns. These standards should be followed as a default, and if they are not followed documentation needs to be included as to the reasons why.

Configuration

When creating an instantiation function for a complex `struct` it is useful to allow variable number of options to be applied. However, the strong type system of Go restricts the function design options. There are a few ways to solve this problem, but we have landed on the following design.

`config` Configuration should be held in a `struct` named `config`, or prefixed with specific type name this Configuration applies to if there are multiple `config` in the package. This `struct` must contain configuration options.

```go
// config contains configuration options for a thing.
type config struct {
    // options ...
}
```

In general the `config` `struct` will not need to be used externally to the package and should be unexported. If, however, it is expected that the user will likely want to build custom options for the configuration, the `config` should be exported. Please, include in the documentation for the `config` how the user can extend the configuration.

It is important that `config` are not shared across package boundaries. Meaning a `config` from one package should not be directly used by another.

Optionally, it is common to include a `newConfig` function (with the same naming scheme). This function wraps any defaults setting and looping over all options to create a configured `config`.

```go
// newConfig returns an appropriately configured config.
func newConfig([]Option) config {
    // Set default values for config.
    config := config{/* [...] */}
    for _, option := range options {
        option.Apply(&config)
    }
    // Preform any validation here.
    return config
}
```

If validation of the `config` options is also preformed this can return an error as well that is expected to be handled by the instantiation function or propagated to the user.

Given the design goal of not having the user need to work with the `config`, the `newConfig` function should also be unexported.

`Option` To set the value of the options a `config` contains, a corresponding `Option` interface type should be used.

```go
type Option interface {
  Apply(*config)
}
```

The name of the interface should be prefixed in the same way the corresponding `config` is (if at all).

Options All user configurable options for a `config` must have a related unexported implementation of the `Option` interface and an exported configuration function that wraps this implementation.

The wrapping function name should be prefixed with `With*` (or in the special case of a boolean options `Without*`) and should have the following function signature.

```go
func With*(...) Option { ... }
```

`bool` Options

```go
type defaultFalseOption bool

func (o defaultFalseOption) Apply(c *config) {
    c.Bool = bool(o)
}

// WithOption sets a T* to have an option included.
func WithOption() Option {
    return defaultFalseOption(true)
}
```

```go
type defaultTrueOption bool

func (o defaultTrueOption) Apply(c *config) {
    c.Bool = bool(o)
}

// WithoutOption sets a T* to have Bool option excluded.
func WithoutOption() Option {
    return defaultTrueOption(false)
}
```

Declared Type Options

```go
type myTypeOption struct {
    MyType MyType
}

func (o myTypeOption) Apply(c *config) {
    c.MyType = o.MyType
}

// WithMyType sets T* to have include MyType.
func WithMyType(t MyType) Option {
    return myTypeOption{t}
}
```

Instantiation Using this configuration pattern to configure instantiation with a `New*` function.

```go
func NewT*(options ...Option) T* {...}
```

Any required parameters can be declared before the variadic `options`.

Dealing with Overlap Sometimes there are multiple complex `struct` that share common configuration and also have distinct configuration. To avoid repeated portions of `config`s, a common `config` can be used with the union of options being handled with the `Option` interface.

For example.

```go
// config holds options for all animals.
type config struct {
    Weight      float64
    Color       string
    MaxAltitude float64
}

// DogOption apply Dog specific options.
type DogOption interface {
    ApplyDog(*config)
}

// BirdOption apply Bird specific options.
type BirdOption interface {
    ApplyBird(*config)
}

// Option apply options for all animals.
type Option interface {
    BirdOption
    DogOption
}

type weightOption float64
func (o weightOption) ApplyDog(c *config)  { c.Weight = float64(o) }
func (o weightOption) ApplyBird(c *config) { c.Weight = float64(o) }
func WithWeight(w float64) Option          { return weightOption(w) }

type furColorOption string
func (o furColorOption) ApplyDog(c *config) { c.Color = string(o) }
func WithFurColor(c string) DogOption       { return furColorOption(c) }

type maxAltitudeOption float64
func (o maxAltitudeOption) ApplyBird(c *config) { c.MaxAltitude = float64(o) }
func WithMaxAltitude(a float64) BirdOption      { return maxAltitudeOption(a) }
func NewDog(name string, o ...DogOption) Dog    {...}
func NewBird(name string, o ...BirdOption) Bird {...}
```

Interface Type

To allow other developers to better comprehend the code, it is important to ensure it is sufficiently documented. One simple measure that contributes to this aim is self-documenting by naming method parameters. Therefore, where appropriate, methods of every exported interface type should have their parameters appropriately named.

Approvers and Maintainers

Approvers:

- Evan Torrie, Verizon Media
- Josh MacDonald, LightStep
- Sam Xie
- David Ashpole, Google
- Gustavo Silva Paiva, LightStep

Maintainers:

- Anthony Mirabella, AWS
- Tyler Yahn, Splunk

Become an Approver or a Maintainer

See the community membership document in OpenTelemetry community repo.