go-autorest

GO reference | Azure Pipelines never built | go report A+

Package go-autorest provides an HTTP request client for use with Autorest-generated API client packages.

An authentication client tested with Azure Active Directory (AAD) is also provided in this repo in the package `github.com/Azure/go-autorest/autorest/adal`. Despite its name, this package is maintained only as part of the Azure Go SDK and is not related to other "ADAL" libraries in github.com/AzureAD.

Overview

Package go-autorest implements an HTTP request pipeline suitable for use across multiple goroutines and provides the shared routines used by packages generated by Autorest.

The package breaks sending and responding to HTTP requests into three phases: Preparing, Sending, and Responding. A typical pattern is:

```
req, err := Prepare(&http.Request{},
  token.WithAuthorization())

resp, err := Send(req,
  WithLogging(logger),
  DoErrorIfStatusCode(http.StatusInternalServerError),
  DoCloseIfError(),
  DoRetryForAttempts(5, time.Second))

err = Respond(resp,
      ByDiscardingBody(),
  ByClosing())
```

Each phase relies on decorators to modify and / or manage processing. Decorators may first modify and then pass the data along, pass the data first and then modify the result, or wrap themselves around passing the data (such as a logger might do). Decorators run in the order provided. For example, the following:

```
req, err := Prepare(&http.Request{},
  WithBaseURL("https://microsoft.com/"),
  WithPath("a"),
  WithPath("b"),
  WithPath("c"))
```

will set the URL to:

```
https://microsoft.com/a/b/c
```

Preparers and Responders may be shared and re-used (assuming the underlying decorators support sharing and re-use). Performant use is obtained by creating one or more Preparers and Responders shared among multiple go-routines, and a single Sender shared among multiple sending go-routines, all bound together by means of input / output channels.

Decorators hold their passed state within a closure (such as the path components in the example above). Be careful to share Preparers and Responders only in a context where such held state applies. For example, it may not make sense to share a Preparer that applies a query string from a fixed set of values. Similarly, sharing a Responder that reads the response body into a passed struct (e.g., `ByUnmarshallingJson`) is likely incorrect.

Errors raised by autorest objects and methods will conform to the `autorest.Error` interface.

See the included examples for more detail. For details on the suggested use of this package by generated clients, see the Client described below.

Helpers

Handling Swagger Dates

The Swagger specification (https://swagger.io) that drives AutoRest (https://github.com/Azure/autorest/) precisely defines two date forms: date and date-time. The github.com/Azure/go-autorest/autorest/date package provides time.Time derivations to ensure correct parsing and formatting.

Handling Empty Values

In JSON, missing values have different semantics than empty values. This is especially true for services using the HTTP PATCH verb. The JSON submitted with a PATCH request generally contains only those values to modify. Missing values are to be left unchanged. Developers, then, require a means to both specify an empty value and to leave the value out of the submitted JSON.

The Go JSON package (`encoding/json`) supports the `omitempty` tag. When specified, it omits empty values from the rendered JSON. Since Go defines default values for all base types (such as "" for string and 0 for int) and provides no means to mark a value as actually empty, the JSON package treats default values as meaning empty, omitting them from the rendered JSON. This means that, using the Go base types encoded through the default JSON package, it is not possible to create JSON to clear a value at the server.

The workaround within the Go community is to use pointers to base types in lieu of base types within structures that map to JSON. For example, instead of a value of type `string` , the workaround uses `*string` . While this enables distinguishing empty values from those to be unchanged, creating pointers to a base type (notably constant, in-line values) requires additional variables. This, for example,

```
s := struct {
  S *string
}{ S: &"foo" }
```

fails, while, this

```
v := "foo"
s := struct {
  S *string
}{ S: &v }
```

succeeds.

To ease using pointers, the subpackage `to` contains helpers that convert to and from pointers for Go base types which have Swagger analogs. It also provides a helper that converts between `map[string]string` and `map[string]*string`, enabling the JSON to specify that the value associated with a key should be cleared. With the helpers, the previous example becomes

```
  s := struct {
    S *string
  }{ S: to.StringPtr("foo") }
```

Install

```
go get github.com/Azure/go-autorest/autorest
go get github.com/Azure/go-autorest/autorest/azure
go get github.com/Azure/go-autorest/autorest/date
go get github.com/Azure/go-autorest/autorest/to
```

Using with Go Modules

In v12.0.1, this repository introduced the following modules.

- autorest/adal
- autorest/azure/auth
- autorest/azure/cli
- autorest/date
- autorest/mocks
- autorest/to
- autorest/validation
- autorest
- logger
- tracing

Tagging cumulative SDK releases as a whole (e.g. `v12.3.0`) is still enabled to support consumers of this repo that have not yet migrated to modules.

License

See LICENSE file.
