Building a Logger Wrapper in Go with Support for Multiple Logging Libraries

Ansu Jain

When building a Go application, logging is an essential part of the development process. The Go standard library provides a basic logging package, but it doesn't offer many advanced features. Fortunately, there are many third-party logging libraries available that provide more features and flexibility.

However, using different logging libraries throughout your codebase can make it hard to maintain and scale. That's where a logger wrapper comes in handy. In this article, we'll discuss how to create a logger wrapper that can work with different logging libraries.

Understanding Logger Wrapper

A logger wrapper is a layer between your application code and the underlying logging library. It provides a simplified API that abstracts away the complexity of different logging libraries, making it easier to switch between them or add new ones.

To create a logger wrapper, you'll need to define an interface that describes the logging methods your application needs. For instance, a basic logger interface could include methods like Info, Warn, and Error. Then, you can create an implementation of this interface that uses a specific logging library, such as logrus or zap.

Once you have defined the interface and implementations, you can create a logger wrapper that exposes the interface methods and allows you to switch between the implementations. This way, you can use the same logging API throughout your codebase and change the underlying logging library without having to change the application code.

Github Source Code : https://github.com/ansu/multilogger

Defining the Logger Interface

Let's start by defining a simple logger interface that includes the methods we need:

```
type Logger interface {
 Debug(msg string, fields map[string]interface{})
 Info(msg string, fields map[string]interface{})
 Warn(msg string, fields map[string]interface{})
 Error(msg string, fields map[string]interface{})
 Fatal(msg string, fields map[string]interface{})
}
```

In this interface, we have five methods: Debug,Info, Warn, Error and Fatal. These methods take a format string and a take a fields map.

Implementing the Logger Interface with Logrus

The next step is to implement the Logger interface using Logrus. We'll create a LogrusLogger struct that contains a Logrus logger instance. Here's an example:

```
type LogrusLogger struct {
 logger *logrus.Logger
 ctx    context.Context
}

func NewLogrusLogger(loggerType string, ctx context.Context)
*LogrusLogger {
 logger := logrus.New()
 logger.Out = os.Stdout

 return &LogrusLogger{logger: logger, ctx: ctx}
}

func (l *LogrusLogger) Debug(msg string, fields map[string]interface{})
{

 l.logger.WithFields(fields).Debug(msg)
}

func (l *LogrusLogger) Info(msg string, fields map[string]interface{}) {
 l.addContextCommonFields(fields)

 l.logger.WithFields(fields).Info(msg)
}

func (l *LogrusLogger) Warn(msg string, fields map[string]interface{}) {
 l.logger.WithFields(fields).Warn(msg)
}

func (l *LogrusLogger) Error(msg string, fields map[string]interface{})
{
 l.logger.WithFields(fields).Error(msg)
}
```

```
func (l *LogrusLogger) Fatal(msg string, fields map[string]interface{})
{
 l.logger.WithFields(fields).Fatal(msg)
}

func (l *LogrusLogger) addContextCommonFields(fields
map[string]interface{}) {
 if l.ctx != nil {
  for k, v := range l.ctx.Value("commonFields").(map[string]interface{})
{
   if _, ok := fields[k]; !ok {
    fields[k] = v
   }
  }
 }
}
```

In this example, we create a LogrusLogger struct that wraps a Logrus logger instance. The NewLogrusLogger() function creates a new Logrus logger instance, and we implement each of the logging methods using the Logrus logger instance.

Implementing the Logger Interface with Zap

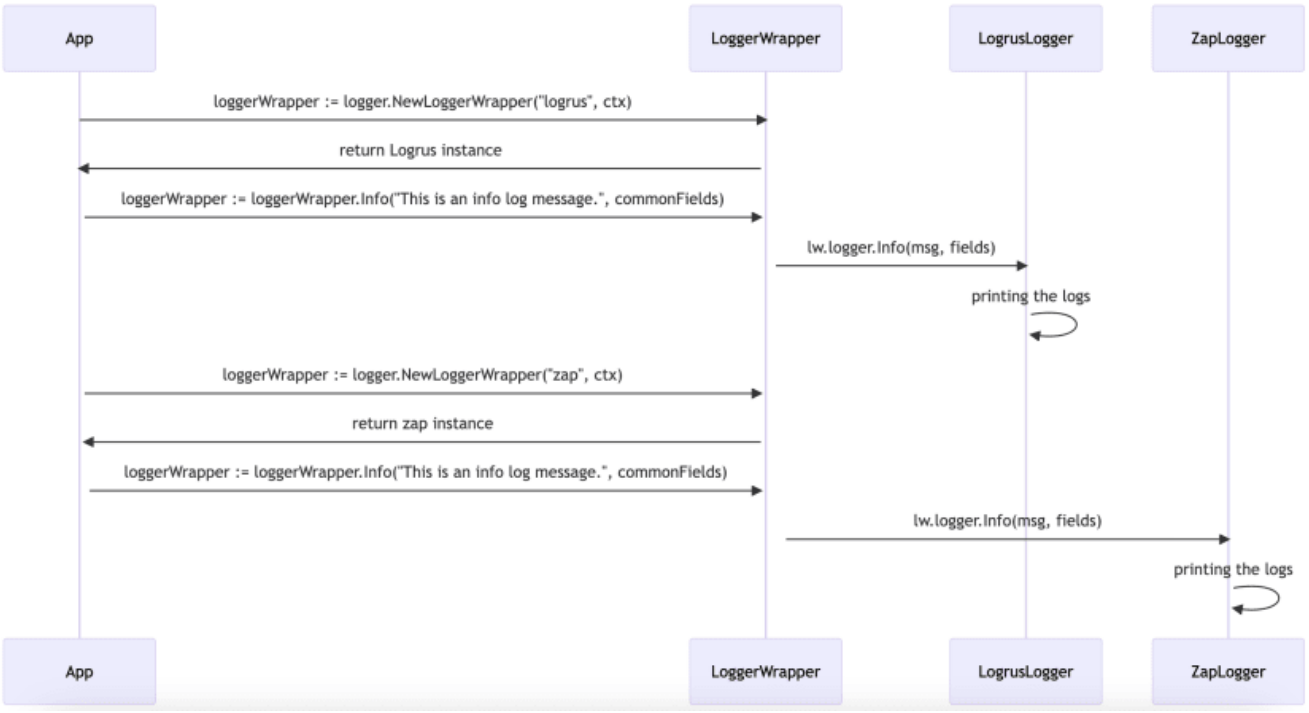
```
type ZapLogger struct {
 logger *zap.Logger
 ctx    context.Context
}

func NewZapLogger(loggerType string, ctx context.Context) *ZapLogger {
 logger, _ := zap.NewProduction()

 return &ZapLogger{logger: logger, ctx: ctx}
}

func (l *ZapLogger) Debug(msg string, fields map[string]interface{}) {
 l.addContextCommonFields(fields)

 l.logger.Debug("", zap.Any("args", fields))
```

```
}

func (l *ZapLogger) Info(msg string, fields map[string]interface{}) {
 l.addContextCommonFields(fields)

 l.logger.Info("", zap.Any("args", fields))
}

func (l *ZapLogger) Warn(msg string, fields map[string]interface{}) {
 l.addContextCommonFields(fields)

 l.logger.Warn("", zap.Any("args", fields))
}

func (l *ZapLogger) Error(msg string, fields map[string]interface{}) {
 l.addContextCommonFields(fields)

 l.logger.Error("", zap.Any("args", fields))
}

func (l *ZapLogger) Fatal(msg string, fields map[string]interface{}) {
 l.addContextCommonFields(fields)

 l.logger.Fatal("", zap.Any("args", fields))
}

func (l *ZapLogger) addContextCommonFields(fields
map[string]interface{}) {
 if l.ctx != nil {
  for k, v := range l.ctx.Value("commonFields").(map[string]interface{})
{
   if _, ok := fields[k]; !ok {
    fields[k] = v
   }
  }
 }
}
```

LoggerWrapper which will be consumed by Application Layer:

```
type LoggerWrapper struct {
 logger Logger
}

func NewLoggerWrapper(loggerType string, ctx context.Context)
*LoggerWrapper {
 var logger Logger

 switch loggerType {
 case "logrus":
  logger = NewLogrusLogger(loggerType, ctx)
 case "zap":
  logger = NewZapLogger(loggerType, ctx)
 default:
  logger = NewLogrusLogger(loggerType, ctx)
 }

 return &LoggerWrapper{logger: logger}
}

func (lw *LoggerWrapper) Debug(msg string, fields
map[string]interface{}) {

 lw.logger.Debug(msg, fields)
}

func (lw *LoggerWrapper) Info(msg string, fields map[string]interface{})
{
 lw.logger.Info(msg, fields)
}

func (lw *LoggerWrapper) Warn(msg string, fields map[string]interface{})
{
 lw.logger.Warn(msg, fields)
}

func (lw *LoggerWrapper) Error(msg string, fields
map[string]interface{}) {
```

```
 lw.logger.Error(msg, fields)
}

func (lw *LoggerWrapper) Fatal(msg string, fields
map[string]interface{}) {
 lw.logger.Fatal(msg, fields)
}
```



During the first step, the application is initialized with either logrus or zap package. Once the initialization is done, all the logs used in the application will go through the respective package. For instance, if the application is initialized with logrus, all logs will be processed through the logrus package. Similarly, if the application is initialized with zap, all logs will go through the zap package.

You might be curious as to why I have utilized contextual references and what advantages we can derive from them.

To log common attributes that should be included in each log, you can use a context object that contains the common attributes.Common attributes that are often logged in each log entry include:

Request tracing information, such as request ID, ALB ID (if it exists), and span ID for microservice communication. This information can be used to correlate logs across multiple services and to trace the flow of a request through the system.

We can use a middleware to automatically add the common fields to each log message without having to pass the context object explicitly to each logging method.Middleware function set the common fields in

context which logger library can consume directly.

In conclusion, we have built a multi-logger package that supports both Logrus and Zap loggers. You can download source code from here : MultiLogger github repo

We have demonstrated how to use the logger wrapper to log messages with fields, and how to use a context object to add common fields to each log. By using a logger wrapper, we can easily switch between different logging frameworks without changing our application code.

Overall, the multi-logger package provides a simple and flexible way to handle logging in Go applications.

**I hope you found this article helpful . If you enjoyed reading this, please like the article and consider following me for more programming tutorials.

**