ChatGPT

Logz Banner

Build License: MIT Go Version Releases

An advanced logging and metrics management tool for Go with dynamic multi-channel notifications, seamless Prometheus integration, and a powerful yet flexible CLI.

Table of Contents

About the Project

Logz is a flexible and powerful solution for managing logs and metrics in modern systems. Built in **Go**, it can be used as a standalone logging service (with its CLI) or as a **module** in your Go applications. Logz provides out-of-the-box support for multiple notification channels such as **HTTP webhooks**, **ZeroMQ**, and **DBus**, and it easily hooks into popular tools like **Prometheus** for advanced monitoring. Its design prioritizes both developer experience and security, making logging a first-class citizen in your projects.

Why Logz?

- **Developer-Friendly:** Simple to set up and use. Whether via CLI or as a library, Logz streamlines your logging with clear APIs and sensible defaults.
- **Cloud & Edge Ready:** Lightweight Go binary with minimal overhead. Suitable for cloud deployments, microservices, and even edge devices, with easy containerization and small footprint.
- **Integrated Notifications:** Set up real-time alerts with dynamic notifiers. Trigger **webhooks** or send messages over **ZeroMQ/DBus** (and more) whenever important events occur, keeping you informed instantly.

- **Extensible & Customizable:** Designed for extensibility – you can add new notifier types or outputs as needed. For example, integrate with services like Slack or Discord via webhooks, or define custom channels (e.g., UDP sockets) by implementing a notifier interface.
- 🔒 **Secure & Robust:** Follows best practices (e.g., Go's `-trimpath` for build security). Validates metrics and log formats to prevent common errors. Offers distinct modes for standalone service vs. embedded use to avoid conflicts.

Features

Dynamic Multi-Channel Notifiers:
- Use multiple notification channels simultaneously for your logs. Configure notifiers via JSON/YAML config to send critical log events to various endpoints (HTTP, ZeroMQ, DBus, etc.) without changing application code.
- Easily extend notifications to other platforms. (Planned integrations include Slack, Discord, and email support – so you can get alerts in your chat or inbox with minimal setup.)

📊 **Monitoring and Metrics**:
- Exposes a Prometheus-compatible `/metrics` endpoint for all your log metrics, enabling you to gather and visualize logging statistics in real time.
- Supports dynamic metric management with optional persistence, so counters and gauges survive restarts if needed.

💻 **Powerful CLI**:
- Use Logz's CLI to log messages at various levels, tail logs, and manage the logging service. It comes with straightforward commands and flags to control output, format, and metadata on the fly.
- The CLI doubles as a background service: start a persistent logging daemon to aggregate logs from multiple sources or applications, and interact with it using commands (`start`, `stop`, `watch`, etc.).

Seamless Integration (Library Mode):
- Import Logz as a module to use it directly in your Go projects. Initialize a logger with one line and log from your code with rich context (structured metadata, custom prefixes, etc.).
- Works with web frameworks and middleware – for example, integrate with a Gin/GOBE server to automatically log HTTP requests, or plug Logz into your microservice architecture for consistent logging across services.

🔒 **Resilient and Secure**:
- Enforces Prometheus naming conventions and sane defaults to keep your logs and metrics clean.
- Robust error handling and optional debug modes ensure that even in failure cases (e.g., logging to a down notifier), Logz won't crash your app.
- Distinct run modes for CLI service vs. library usage prevent conflicts (the library usage won't spawn background services unless you want it to).

Installation

Requirements:

- **Go** version 1.21 or later
- *(Optional)* Prometheus for metrics scraping (if you plan to use Prometheus integration)

To install Logz, you can build from source:

```
# Clone the repository
git clone https://github.com/rafa-mori/logz.git

# Navigate to the project directory
cd logz

# Build the binary (requires Make)
make build

# (Optional) Install the binary to your system (e.g., /usr/local/bin)
make install

# (Optional) Add the installation directory to your PATH if not already in it
export PATH=$PATH:$(pwd)
```

Alternatively, add Logz as a module in your Go project by running:

```
go get github.com/rafa-mori/logz
```

(This allows you to use Logz's logging capabilities programmatically within your own Go code.)

Usage

Logz can be used in two primary ways: via its **CLI** for external log management, or as a **library** embedded in your Go application. In both cases, you benefit from the same core features like leveled logging, structured metadata, and notifier integrations.

CLI Usage

Once installed, the `logz` CLI lets you send logs and control the logging service. Here are some common commands:

```
# Log messages at different levels from the command line
logz info --msg "Starting the application."
logz error --msg "Database connection failed."

# Start the Logz service as a background (detached) process
logz start

# Stop the background Logz service
logz stop

# Tail and watch logs in real-time (from the service or aggregated sources)
logz watch
```

Logging with additional options: You can customize where logs go, their format, and attach metadata. Below are a few practical examples:

- **Log a Debug Message with Metadata** – for extra context in development or troubleshooting:

  ```
  logz debug \
    --msg "Debugging the payment module" \
    --output "stdout" \
    --metadata requestId=12345,user=admin
  ```

  Output:

  ```
  [2025-08-06T12:34:56Z]   DEBUG - Debugging the payment module
                          {"requestId":"12345","user":"admin"}
  ```

  In the above, the message is logged at DEBUG level with a timestamp and a bug emoji indicator. The metadata (requestId and user) is printed on a new indented line in a key/value JSON-like format, making it easy to scan.

- **Log an Info Message to a File** – for logging to a file on disk (useful in production):

  ```
  logz info \
    --msg "User signed up successfully." \
    --output "/var/log/myapp/actions.log" \
    --metadata userID=98765,plan=premium
  ```

  This will append the log to the file at `/var/log/myapp/actions.log`. You can specify any valid file path for the `--output`.

- **Log an Error Message in JSON Format** – for structured logging (great for log aggregators):

```
logz error \
  --msg "Failed to process payment" \
  --output "stdout" \
  --format "json" \
  --metadata errorCode=502,service="payments"
```

Output:

```
{
  "timestamp": "2025-08-06T12:35:30Z",
  "level": "ERROR",
  "message": "Failed to process payment",
  "metadata": {
    "errorCode": 502,
    "service": "payments"
  }
}
```

In JSON format, the log entry is fully structured with timestamp, level, message, and metadata fields, which can be easily parsed by machines or log management systems.

Note: The CLI will automatically start the Logz service in the background when you log messages if it's not already running. This service is what enables features like real-time log watching and metrics exposition. You can manage it with `logz start` and `logz stop` as shown.

Library Integration

Using Logz as a library allows you to embed advanced logging into your own Go applications. This is ideal for integrating with web frameworks, writing custom notifiers, or simply having more programmatic control over logging behavior.

To use Logz in code, import the `logz` module (or the sub-package `logz/logger` for direct logger usage) and obtain a logger instance. You can either use the global logger via a helper or create your own instance:

```
import (
    "github.com/rafa-mori/logz"
)

func main() {
    // Initialize global logger (singleton) with a custom prefix for your app/
service:
    logger :=
logz.GetLogger("MyApp")  // prefix will appear in logs to identify source
```

```go
    // Now use the logger to log messages at various levels:
    logger.Info("Application started", map[string]interface{}{
        "version": "1.4.0",
        "env":     "production",
    })
    // This will log an INFO level message with the given context metadata.

    // Example: using different levels
    logger.Debug("Debugging mode on", nil)
    logger.Error("An error occurred during processing", map[string]interface{}{
        "orderId": 112233,
        "error":   err.Error(),
    })
}
```

In the snippet above, `logz.GetLogger("MyApp")` returns a thread-safe global logger instance with the prefix "MyApp". The logger provides methods like `Info()`, `Debug()`, `Error()`, etc., each of which can take a message and an optional metadata map. The metadata is merged into the log entry, similar to how the CLI's `--metadata` works.

Integration with web frameworks (example): If you're using a framework like Gin (as in the GoBE project), you can integrate Logz in middleware to log requests:

```go
import (
    "github.com/gin-gonic/gin"
    "github.com/rafa-mori/logz"
)

func main() {
    r := gin.New()
    // Use Logz in a Gin middleware to log each request
    r.Use(func(c *gin.Context) {
        path := c.Request.URL.Path
        method := c.Request.Method
        logz.GetLogger("Web").Info("Incoming request", map[string]interface{}{
            "method": method,
            "path":   path,
        })
        c.Next()
    })

    // ... define routes and start server
}
```

The above middleware logs each HTTP request's method and path at INFO level using Logz. You can expand this idea to log response status, duration, etc. By using the Logz library, all these logs will also respect the global configuration – for example, they can trigger notifiers or be watched via `logz watch` if the Logz service is running.

Custom Channels: Logz's architecture allows implementing custom log writers or notifiers. For instance, you could direct certain logs to a UDP socket or another service by writing a small piece of code that implements the notifier interface and adding it to the configuration. *(Refer to the documentation or source for creating custom notifiers.)* This means you're not limited to the built-in channels – Logz can be extended to suit very specific logging needs in specialized environments.

Configuration

When run for the first time, Logz will generate a default configuration file (in JSON format) at:

`~/.kubex/logz/config.json` (on Linux/macOS)

This config file centralizes Logz's settings, including server ports, log level, and notifier setups. You can edit this file manually (or supply your own YAML/JSON) to configure Logz.

Example Configuration (JSON):

```json
{
  "port": "2112",
  "bindAddress": "0.0.0.0",
  "logLevel": "info",
  "notifiers": {
    "alertsWebhook": {
      "type": "http",
      "webhookURL": "https://example.com/webhook-endpoint",
      "authToken": "YOUR_SECRET_TOKEN"
    },
    "localBus": {
      "type": "dbus",
      "channel": "/com/example/logging"
    }
  }
}
```

In this example:
- The Logz service will bind to `0.0.0.0:2112` (this is where the Prometheus `/metrics` and any API endpoints are exposed).
- The default log level is set to **info** (so debug and trace logs would be suppressed globally until changed).
- Two notifiers are configured: one is an HTTP webhook (perhaps for sending alerts to an external service or chat), and another uses **DBus** (for local system notifications via the system bus).

You can define multiple notifiers under the `"notifiers"` section. Each notifier entry requires a `type` (e.g., `"http"`, `"zeromq"`, `"dbus"` or others) and corresponding fields for that type (for instance, HTTP notifiers use a `webhookURL` and optional `authToken`, ZeroMQ might use an address, etc.). Consult the documentation for all supported fields per notifier type.

Logz will load this configuration on startup (or when the service starts). If you update the config file, you should restart the Logz service (`logz stop` and `logz start`) to apply changes.

Prometheus Integration

One of Logz's strengths is its native integration with Prometheus for metrics. When the Logz service is running, it exposes an HTTP endpoint for Prometheus to scrape metrics:

```
http://localhost:2112/metrics
```

This endpoint provides various metrics about the logging system itself – for example, counts of logs by level, metrics from your application if you use Logz's metric features, and health stats of notifiers.

Example Prometheus Configuration:

To have Prometheus collect Logz metrics, add a job to your Prometheus `prometheus.yml`:

```
scrape_configs:
  - job_name: 'logz'
    static_configs:
      - targets: ['localhost:2112']
```

With the above, Prometheus will periodically hit the Logz service on port 2112 and record the metrics. You can then visualize these metrics on Grafana or analyze them to get insights like "error rate over time" or "number of notifications sent".

This makes it easy to include Logz as part of your observability stack – your application logs are not just written out, but also measured and monitored just like any other important aspect of your system.

Roadmap

Upcoming Features & Ideas:

- **Slack/Discord Notifiers**: First-class support for Slack and Discord channels (beyond generic webhooks). This will allow sending formatted alert messages to chat channels for critical logs or incidents, with minimal config.

- **Email Notifications**: Integration to send email alerts for certain log events (e.g., send an email when a fatal error occurs in production).
- **Integrated Dashboard**: A built-in web dashboard for viewing logs and metrics in real-time, simplifying debugging without external tools.
- **Advanced Configuration Validation**: Improving the config file handling with schema validation and helpful error messages, so misconfigurations are caught on startup.
- **Plugins System**: A plugin mechanism to allow community-contributed notifiers and formatters to be added to Logz without modifying core, enabling a broader ecosystem of integrations.

Have a suggestion or a feature request? Feel free to open an issue – Logz is a community-driven project and welcomes contributions!

Contributing

Contributions are welcome! If you have ideas for improvements or have found a bug, please open an issue or submit a pull request. When contributing code, please check out the Contributing Guide for best practices and project standards.

Whether it's a new notifier integration, a documentation improvement, or a bugfix, all contributions are appreciated. Join the discussions in the issue tracker to collaborate with other developers and help make Logz better.

Contact

💻 **Project Maintainer** – **Rafael Mori** (@rafa-mori)
If you have any questions, need help getting started, or just want to share how you're using Logz, feel free to reach out.

📫 **Email**: faelmori@gmail.com

Connect: You can follow the project on GitHub for updates. If you find this project useful or interesting, a star on the repository is appreciated!

Thank you for checking out Logz. Happy logging!
