
SwiftMetrics

A Metrics API package for Swift.

Almost all production server software needs to emit metrics information for observability. Because it's unlikely that all parties can agree on one specific metrics backend implementation, this API is designed to establish a standard that can be implemented by various metrics libraries which then post the metrics data to backends like Prometheus, Graphite, publish over statsd, write to disk, etc.

This is the beginning of a community-driven open-source project actively seeking contributions, be it code, documentation, or ideas. Apart from contributing to SwiftMetrics itself, we need metrics compatible libraries which send the metrics over to backend such as the ones mentioned above. What SwiftMetrics provides today is covered in the API docs, but it will continue to evolve with community input.

Getting started

If you have a server-side Swift application, or maybe a cross-platform (e.g. Linux, macOS) application or library, and you would like to emit metrics, targeting this metrics API package is a great idea. Below you'll find all you need to know to get started.

Adding the dependency

To add a dependency on the metrics API package, you need to declare it in your `Package.swift`:

```
1 // swift-metrics 1.x and 2.x are almost API compatible, so most clients
   should use
2 .package(url: "https://github.com/apple/swift-metrics.git", "1.0.0" ..<
   "3.0.0"),
```

and to your application/library target, add “Metrics” to your dependencies:

```
1 .target(
2     name: "BestExampleApp",
3     dependencies: [
4         // ...
5         .product(name: "Metrics", package: "swift-metrics"),
6     ]
7 ),
```

Emitting metrics information

```
1 // 1) let's import the metrics API package
2 import Metrics
3
4 // 2) we need to create a concrete metric object, the label works
   similarly to a `DispatchQueue` label
5 let counter = Counter(label: "com.example.BestExampleApp.
   numberOfRequests")
6
7 // 3) we're now ready to use it
8 counter.increment()
```

Selecting a metrics backend implementation (applications only)

Note: If you are building a library, you don't need to concern yourself with this section. It is the end users of your library (the applications) who will decide which metrics backend to use. Libraries should never change the metrics implementation as that is something owned by the application.

SwiftMetrics only provides the metrics system API. As an application owner, you need to select a metrics backend (such as the ones mentioned above) to make the metrics information useful.

Selecting a backend is done by adding a dependency on the desired backend client implementation and invoking the `MetricsSystem.bootstrap` function at the beginning of the program:

```
1 MetricsSystem.bootstrap(SelectedMetricsImplementation())
```

This instructs the `MetricsSystem` to install `SelectedMetricsImplementation` (actual name will differ) as the metrics backend to use.

As the API has just launched, not many implementations exist yet. If you are interested in implementing one see the “Implementing a metrics backend” section below explaining how to do so. List of existing SwiftMetrics API compatible libraries:

- SwiftPrometheus, support for Prometheus
- StatsD Client, support for StatsD
- OpenTelemetry Swift, support for OpenTelemetry which also implements other metrics and tracing backends
- Your library? Get in touch!

Swift Metrics Extras

You may also be interested in some “extra” modules which are collected in the Swift Metrics Extras repository.

Detailed design

Architecture

We believe that for the Swift on Server ecosystem, it's crucial to have a metrics API that can be adopted by anybody so a multitude of libraries from different parties can all provide metrics information. More concretely this means that we believe all the metrics events from all libraries should end up in the same place, be one of the backends mentioned above or wherever else the application owner may choose.

In the real world, there are so many opinions over how exactly a metrics system should behave, how metrics should be aggregated and calculated, and where/how to persist them. We think it's not feasible to wait for one metrics package to support everything that a specific deployment needs while still being simple enough to use and remain performant. That's why we decided to split the problem into two:

1. a metrics API
2. a metrics backend implementation

This package only provides the metrics API itself, and therefore, SwiftMetrics is a “metrics API package.” SwiftMetrics can be configured (using `MetricsSystem.bootstrap`) to choose any compatible metrics backend implementation. This way, packages can adopt the API, and the application can choose any compatible metrics backend implementation without requiring any changes from any of the libraries.

This API was designed with the contributors to the Swift on Server community and approved by the SSWG (Swift Server Work Group) to the “sandbox level” of the SSWG's incubation process.

[pitch](#) | [discussion](#) | [feedback](#)

Metric types

The API supports four metric types:

Counter: A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart. For example, you can use a counter to represent the number of requests served, tasks completed, or errors.

```
1 counter.increment(by: 100)
```

Gauge: A Gauge is a metric that represents a single numerical value that can arbitrarily go up and down. Gauges are typically used for measured values like temperatures or current memory usage,

but also “counts” that can go up and down, like the number of active threads. Gauges are modeled as a `Recorder` with a sample size of 1 that does not perform any aggregation.

```
1 gauge.record(100)
```

Meter: A Meter is similar to `Gauge` - a metric that represents a single numerical value that can arbitrarily go up and down. Meters are typically used for measured values like temperatures or current memory usage, but also “counts” that can go up and down, like the number of active threads. Unlike `Gauge`, `Meter` also supports atomic increments and decrements.

```
1 meter.record(100)
```

Recorder: A recorder collects observations within a time window (usually things like response sizes) and *can* provide aggregated information about the data sample, for example count, sum, min, max and various quantiles.

```
1 recorder.record(100)
```

Timer: A timer collects observations within a time window (usually things like request duration) and provides aggregated information about the data sample, for example min, max and various quantiles. It is similar to a `Recorder` but specialized for values that represent durations.

```
1 timer.recordMilliseconds(100)
```

Implementing a metrics backend (e.g. Prometheus client library)

Note: Unless you need to implement a custom metrics backend, everything in this section is likely not relevant, so please feel free to skip.

As seen above, each constructor for `Counter`, `Gauge`, `Meter`, `Recorder` and `Timer` provides a metric object. This uncertainty obscures the selected metrics backend calling these constructors by design. *Each application* can select and configure its desired backend. The application sets up the metrics backend it wishes to use. Configuring the metrics backend is straightforward:

```
1 let metricsImplementation = MyFavoriteMetricsImplementation()  
2 MetricsSystem.bootstrap(metricsImplementation)
```

This instructs the `MetricsSystem` to install `MyFavoriteMetricsImplementation` as the metrics backend (`MetricsFactory`) to use. This should only be done once at the beginning of the program.

Given the above, an implementation of a metric backend needs to conform to `protocol MetricsFactory`:

```
1 public protocol MetricsFactory {
2     func makeCounter(label: String, dimensions: [(String, String)]) ->
        CounterHandler
3     func makeMeter(label: String, dimensions: [(String, String)]) ->
        MeterHandler
4     func makeRecorder(label: String, dimensions: [(String, String)],
        aggregate: Bool) -> RecorderHandler
5     func makeTimer(label: String, dimensions: [(String, String)]) ->
        TimerHandler
6
7     func destroyCounter(_ handler: CounterHandler)
8     func destroyMeter(_ handler: MeterHandler)
9     func destroyRecorder(_ handler: RecorderHandler)
10    func destroyTimer(_ handler: TimerHandler)
11 }
```

The `MetricsFactory` is responsible for instantiating the concrete metrics classes that capture the metrics and perform aggregation and calculation of various quantiles as needed.

Counter

```
1 public protocol CounterHandler: AnyObject {
2     func increment(by: Int64)
3     func reset()
4 }
```

Meter

```
1 public protocol MeterHandler: AnyObject {
2     func set(_ value: Int64)
3     func set(_ value: Double)
4     func increment(by: Double)
5     func decrement(by: Double)
6 }
```

Recorder

```
1 public protocol RecorderHandler: AnyObject {
2     func record(_ value: Int64)
3     func record(_ value: Double)
4 }
```

Timer

```
1 public protocol TimerHandler: AnyObject {
2     func recordNanoseconds(_ duration: Int64)
3 }
```

Dealing with Overflows Implementation of metric objects that deal with integers, like `Counter` and `Timer` should be careful with overflow. The expected behavior is to cap at `.max`, and never crash the program due to overflow. For example:

```
1 class ExampleCounter: CounterHandler {
2     var value: Int64 = 0
3     func increment(by amount: Int64) {
4         let result = self.value.addingReportingOverflow(amount)
5         if result.overflow {
6             self.value = Int64.max
7         } else {
8             self.value = result.partialValue
9         }
10    }
11 }
```

Full example Here is a full, but contrived, example of an in-memory implementation:

```
1 class SimpleMetricsLibrary: MetricsFactory {
2     init() {}
3
4     func makeCounter(label: String, dimensions: [(String, String)]) ->
5         CounterHandler {
6         return ExampleCounter(label, dimensions)
7     }
8
9     func makeMeter(label: String, dimensions: [(String, String)]) ->
10        MeterHandler {
11        return ExampleMeter(label, dimensions)
12    }
13
14    func makeRecorder(label: String, dimensions: [(String, String)],
15        aggregate: Bool) -> RecorderHandler {
16        return ExampleRecorder(label, dimensions, aggregate)
17    }
18
19    func makeTimer(label: String, dimensions: [(String, String)]) ->
20        TimerHandler {
21        return ExampleTimer(label, dimensions)
22    }
23
24    // implementation is stateless, so nothing to do on destroy calls
25    func destroyCounter(_ handler: CounterHandler) {}
26    func destroyMeter(_ handler: TimerHandler) {}
27    func destroyRecorder(_ handler: RecorderHandler) {}
28    func destroyTimer(_ handler: TimerHandler) {}
29
30    private class ExampleCounter: CounterHandler {
31        init(_: String, _: [(String, String)]) {}
32    }
33}
```

```

28
29     let lock = NSLock()
30     var value: Int64 = 0
31     func increment(by amount: Int64) {
32         self.lock.withLock {
33             self.value += amount
34         }
35     }
36
37     func reset() {
38         self.lock.withLock {
39             self.value = 0
40         }
41     }
42 }
43
44 private class ExampleMeter: MeterHandler {
45     init(_: String, _: [(String, String)]) {}
46
47     let lock = NSLock()
48     var _value: Double = 0
49
50     func set(_ value: Int64) {
51         self.set(Double(value))
52     }
53
54     func set(_ value: Double) {
55         self.lock.withLock { _value = value }
56     }
57
58     func increment(by value: Double) {
59         self.lock.withLock { self._value += value }
60     }
61
62     func decrement(by value: Double) {
63         self.lock.withLock { self._value -= value }
64     }
65 }
66
67 private class ExampleRecorder: RecorderHandler {
68     init(_: String, _: [(String, String)], _: Bool) {}
69
70     private let lock = NSLock()
71     var values = [(Int64, Double)]()
72     func record(_ value: Int64) {
73         self.record(Double(value))
74     }
75
76     func record(_ value: Double) {
77         // TODO: sliding window
78         lock.withLock {

```

```

79         values.append((Date().nanoSince1970, value))
80         self._count += 1
81         self._sum += value
82         self._min = Swift.min(self._min, value)
83         self._max = Swift.max(self._max, value)
84     }
85 }
86
87 var _sum: Double = 0
88 var sum: Double {
89     return self.lock.withLock { _sum }
90 }
91
92 private var _count: Int = 0
93 var count: Int {
94     return self.lock.withLock { _count }
95 }
96
97 private var _min: Double = 0
98 var min: Double {
99     return self.lock.withLock { _min }
100 }
101
102 private var _max: Double = 0
103 var max: Double {
104     return self.lock.withLock { _max }
105 }
106 }
107
108 private class ExampleTimer: TimerHandler {
109     init(_: String, _: [(String, String)]) {}
110
111     let lock = NSLock()
112     var _value: Int64 = 0
113
114     func recordNanoseconds(_ duration: Int64) {
115         self.lock.withLock { _value = duration }
116     }
117 }
118 }
```

Security

Please see SECURITY.md for details on the security process.

Getting involved

Do not hesitate to get in touch as well, over on <https://forums.swift.org/c/server>