Metrics Documentation

Release 3.1.0

Coda Hale

Jun 06, 2021

CONTENTS

Metrics provides a powerful toolkit of ways to measure the behavior of critical components **in your production environment.**

With modules for common libraries like **Jetty**, **Logback**, **Log4j**, **Apache HttpClient**, **Ehcache**, **JDBI**, **Jersey** and reporting backends like **Ganglia** and **Graphite**, Metrics provides you with full-stack visibility.

CHAPTER

ONE

GETTING STARTED

***Getting Started* will guide you through the process of adding Metrics to an existing application. We'll go through the various measuring instruments that Metrics provides, how to use them, and when they'll come in handy.**

1.1 Setting Up Maven

You need the `metrics-core` library as a dependency:

```
<dependencies>
    <dependency>
        <groupId>io.dropwizard.metrics</groupId>
        <artifactId>metrics-core</artifactId>
        <version>${metrics.version}</version>
    </dependency>
</dependencies>
```

Note: Make sure you have a `metrics.version` property declared in your POM with the current version, which is 3.1.0.

Now it's time to add some metrics to your application!

1.2 Meters

A meter measures the rate of events over time (e.g., "requests per second"). In addition to the mean rate, meters also track 1-, 5-, and 15-minute moving averages.

```
private final Meter requests = metrics.meter("requests");

public void handleRequest(Request request, Response response) {
    requests.mark();
    // etc
}
```

This meter will measure the rate of requests in requests per second.

1.3 Console Reporter

A Console Reporter is exactly what it sounds like - report to the console. This reporter will print every second.

```
ConsoleReporter reporter = ConsoleReporter.forRegistry(metrics)
       .convertRatesTo(TimeUnit.SECONDS)
       .convertDurationsTo(TimeUnit.MILLISECONDS)
       .build();
   reporter.start(1, TimeUnit.SECONDS);
```

1.4 Complete getting started

So the complete Getting Started is

```
  package sample;
  import com.codahale.metrics.*;
  import java.util.concurrent.TimeUnit;

  public class GetStarted {
    static final MetricRegistry metrics = new MetricRegistry();
    public static void main(String args[]) {
      startReport();
      Meter requests = metrics.meter("requests");
      requests.mark();
      wait5Seconds();
    }

  static void startReport() {
      ConsoleReporter reporter = ConsoleReporter.forRegistry(metrics)
          .convertRatesTo(TimeUnit.SECONDS)
          .convertDurationsTo(TimeUnit.MILLISECONDS)
          .build();
      reporter.start(1, TimeUnit.SECONDS);
  }

  static void wait5Seconds() {
      try {
          Thread.sleep(5*1000);
      }
      catch(InterruptedException e) {}
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
↪apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>somegroup</groupId>
```

(continues on next page)

(continued from previous page)

```
  <artifactId>sample</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Example project for Metrics</name>

  <dependencies>
    <dependency>
      <groupId>io.dropwizard.metrics</groupId>
      <artifactId>metrics-core</artifactId>
      <version>${metrics.version}</version>
    </dependency>
  </dependencies>
</project>
```

Note: Make sure you have a `metrics.version` property declared in your POM with the current version, which is 3.1.0.

To run

```
mvn package exec:java -Dexec.mainClass=sample.First
```

1.5 The Registry

The centerpiece of Metrics is the `MetricRegistry` class, which is the container for all your application's metrics. Go ahead and create a new one:

```
final MetricRegistry metrics = new MetricRegistry();
```

You'll probably want to integrate this into your application's lifecycle (maybe using your dependency injection framework), but `static` field is fine.

1.6 Gauges

A gauge is an instantaneous measurement of a value. For example, we may want to measure the number of pending jobs in a queue:

```
public class QueueManager {
    private final Queue queue;

    public QueueManager(MetricRegistry metrics, String name) {
        this.queue = new Queue();
        metrics.register(MetricRegistry.name(QueueManager.class, name, "size"),
                         new Gauge<Integer>() {
                             @Override
                             public Integer getValue() {
                                 return queue.size();
                             }
                         });
```

(continues on next page)

(continued from previous page)

```
    }
}
```

When this gauge is measured, it will return the number of jobs in the queue.

Every metric in a registry has a unique name, which is just a dotted-name string like `"things.count"` or `"com.example.Thing.latency"`. `MetricRegistry` has a static helper method for constructing these names:

```
MetricRegistry.name(QueueManager.class, "jobs", "size")
```

This will return a string with something like `"com.example.QueueManager.jobs.size"`.

For most queue and queue-like structures, you won't want to simply return `queue.size()`. Most of `java.util` and `java.util.concurrent` have implementations of `#size()` which are **O(n)**, which means your gauge will be slow (potentially while holding a lock).

1.7 Counters

A counter is just a gauge for an `AtomicLong` instance. You can increment or decrement its value. For example, we may want a more efficient way of measuring the pending job in a queue:

```
private final Counter pendingJobs = metrics.counter(name(QueueManager.class, "pending-
↪jobs"));

public void addJob(Job job) {
    pendingJobs.inc();
    queue.offer(job);
}

public Job takeJob() {
    pendingJobs.dec();
    return queue.take();
}
```

Every time this counter is measured, it will return the number of jobs in the queue.

As you can see, the API for counters is slightly different: `#counter(String)` instead of `#register(String, Metric)`. While you can use `register` and create your own `Counter` instance, `#counter(String)` does all the work for you, and allows you to reuse metrics with the same name.

Also, we've statically imported `MetricRegistry`'s `name` method in this scope to reduce clutter.

1.8 Histograms

A histogram measures the statistical distribution of values in a stream of data. In addition to minimum, maximum, mean, etc., it also measures median, 75th, 90th, 95th, 98th, 99th, and 99.9th percentiles.

```
private final Histogram responseSizes = metrics.histogram(name(RequestHandler.class,
↪"response-sizes"));

public void handleRequest(Request request, Response response) {
```

(continues on next page)

(continued from previous page)

```
    // etc
    responseSizes.update(response.getContent().length);
}
```

This histogram will measure the size of responses in bytes.

1.9 Timers

A timer measures both the rate that a particular piece of code is called and the distribution of its duration.

```
private final Timer responses = metrics.timer(name(RequestHandler.class, "responses"));

public String handleRequest(Request request, Response response) {
    final Timer.Context context = responses.time();
    try {
        // etc;
        return "OK";
    } finally {
        context.stop();
    }
}
```

This timer will measure the amount of time it takes to process each request in nanoseconds and provide a rate of requests in requests per second.

1.10 Health Checks

Metrics also has the ability to centralize your service's health checks with the `metrics-healthchecks` module.

First, create a new `HealthCheckRegistry` instance:

```
final HealthCheckRegistry healthChecks = new HealthCheckRegistry();
```

Second, implement a `HealthCheck` subclass:

```
public class DatabaseHealthCheck extends HealthCheck {
    private final Database database;

    public DatabaseHealthCheck(Database database) {
        this.database = database;
    }

    @Override
    public HealthCheck.Result check() throws Exception {
        if (database.isConnected()) {
            return HealthCheck.Result.healthy();
        } else {
            return HealthCheck.Result.unhealthy("Cannot connect to " + database.
↪getUrl());
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Then register an instance of it with Metrics:

```
healthChecks.register("postgres", new DatabaseHealthCheck(database));
```

To run all of the registered health checks:

```
final Map<String, HealthCheck.Result> results = healthChecks.runHealthChecks();
for (Entry<String, HealthCheck.Result> entry : results.entrySet()) {
    if (entry.getValue().isHealthy()) {
        System.out.println(entry.getKey() + " is healthy");
    } else {
        System.err.println(entry.getKey() + " is UNHEALTHY: " + entry.getValue().
↪getMessage());
        final Throwable e = entry.getValue().getError();
        if (e != null) {
            e.printStackTrace();
        }
    }
}
```

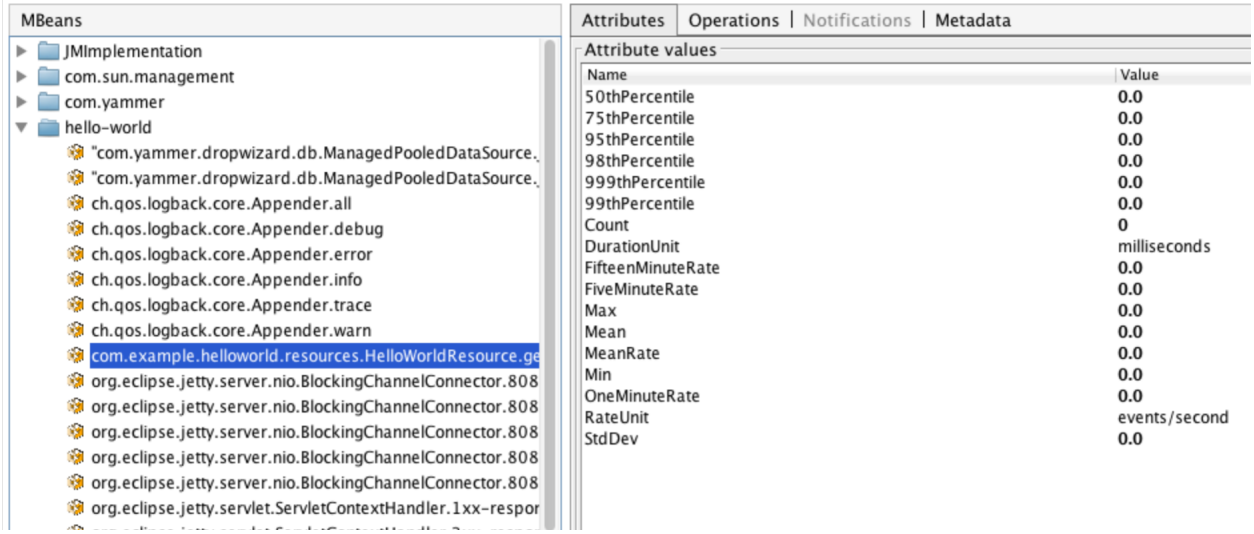
Metrics comes with a pre-built health check: `ThreadDeadlockHealthCheck`, which uses Java's built-in thread deadlock detection to determine if any threads are deadlocked.

1.11 Reporting Via JMX

To report metrics via JMX:

```
final JmxReporter reporter = JmxReporter.forRegistry(registry).build();
reporter.start();
```

Once the reporter is started, all of the metrics in the registry will become visible via **JConsole** or **VisualVM** (if you install the MBeans plugin):



Tip: If you double-click any of the metric properties, VisualVM will start graphing the data for that property. Sweet, eh?

1.12 Reporting Via HTTP

Metrics also ships with a servlet (`AdminServlet`) which will serve a JSON representation of all registered metrics. It will also run health checks, print out a thread dump, and provide a simple "ping" response for load-balancers. (It also has single servlets–`MetricsServlet`, `HealthCheckServlet`, `ThreadDumpServlet`, and `PingServlet`–which do these individual tasks.)

To use this servlet, include the `metrics-servlets` module as a dependency:

```
<dependency>
    <groupId>io.dropwizard.metrics</groupId>
    <artifactId>metrics-servlets</artifactId>
    <version>${metrics.version}</version>
</dependency>
```

Note: Make sure you have a `metrics.version` property declared in your POM with the current version, which is 3.1.0.

From there on, you can map the servlet to whatever path you see fit.

1.13 Other Reporting

In addition to JMX and HTTP, Metrics also has reporters for the following outputs:

- `STDOUT`, using *ConsoleReporter* from `metrics-core`
- `CSV` files, using *CsvReporter* from `metrics-core`
- SLF4J loggers, using *Slf4jReporter* from `metrics-core`
- Ganglia, using *GangliaReporter* from `metrics-ganglia`
- Graphite, using *GraphiteReporter* from `metrics-graphite`

CHAPTER

TWO

USER MANUAL

This goal of this document is to provide you with all the information required to effectively use the Metrics library in your application. If you're new to Metrics, you should read the Getting Started guide first.

2.1 Metrics Core

The central library for Metrics is `metrics-core`, which provides some basic functionality:

- Metric *registries*.
- The five metric types: *Gauges*, *Counters*, *Histograms*, *Meters*, and *Timers*.
- Reporting metrics values via *JMX*, the *console*, *CSV* files, and *SLF4J loggers*.

2.1.1 Metric Registries

The starting point for Metrics is the `MetricRegistry` class, which is a collection of all the metrics for your application (or a subset of your application).

Generally you only need one `MetricRegistry` instance per application, although you may choose to use more if you want to organize your metrics in particular reporting groups.

Global named registries can also be shared through the static `SharedMetricRegistries` class. This allows the same registry to be used in different sections of code without explicitly passing a `MetricRegistry` instance around.

Like all Metrics classes, `SharedMetricRegistries` is fully thread-safe.

2.1.2 Metric Names

Each metric is associated with a `MetricRegistry`, and has a unique *name* within that registry. This is a simple dotted name, like `com.example.Queue.size`. This flexibility allows you to encode a wide variety of context directly into a metric's name. If you have two instances of `com.example.Queue`, you can give them more specific: `com.example.Queue.requests.size` vs. `com.example.Queue.responses.size`, for example.

`MetricRegistry` has a set of static helper methods for easily creating names:

```
MetricRegistry.name(Queue.class, "requests", "size")
MetricRegistry.name(Queue.class, "responses", "size")
```

These methods will also elide any `null` values, allowing for easy optional scopes.

2.1.3 Gauges

A gauge is the simplest metric type. It just returns a *value*. If, for example, your application has a value which is maintained by a third-party library, you can easily expose it by registering a `Gauge` instance which returns that value:

```
registry.register(name(SessionStore.class, "cache-evictions"), new Gauge<Integer>() {
    @Override
    public Integer getValue() {
        return cache.getEvictionsCount();
    }
});
```

This will create a new gauge named `com.example.proj.auth.SessionStore.cache-evictions` which will return the number of evictions from the cache.

JMX Gauges

Given that many third-party libraries often expose metrics only via JMX, Metrics provides the `JmxAttributeGauge` class, which takes the object name of a JMX MBean and the name of an attribute and produces a gauge implementation which returns the value of that attribute:

```
registry.register(name(SessionStore.class, "cache-evictions"),
                  new JmxAttributeGauge("net.sf.ehcache:type=Cache,scope=sessions,
↪name=eviction-count", "Value"));
```

Ratio Gauges

A ratio gauge is a simple way to create a gauge which is the ratio between two numbers:

```
public class CacheHitRatio extends RatioGauge {
    private final Meter hits;
    private final Timer calls;

    public CacheHitRatio(Meter hits, Timer calls) {
        this.hits = hits;
        this.calls = calls;
    }

    @Override
    public Ratio getRatio() {
        return Ratio.of(hits.getOneMinuteRate(),
                        calls.getOneMinuteRate());
    }
}
```

This gauge returns the ratio of cache hits to misses using a meter and a timer.

Cached Gauges

A cached gauge allows for a more efficient reporting of values which are expensive to calculate:

```
registry.register(name(Cache.class, cache.getName(), "size"),
                  new CachedGauge<Long>(10, TimeUnit.MINUTES) {
                      @Override
                      protected Long loadValue() {
                          // assume this does something which takes a long time
                          return cache.getSize();
                      }
                  });
```

Derivative Gauges

A derivative gauge allows you to derive values from other gauges' values:

```
public class CacheSizeGauge extends DerivativeGauge<CacheStats, Long> {
    public CacheSizeGauge(Gauge<CacheStats> statsGauge) {
        super(statsGauge);
    }

    @Override
    protected Long transform(CacheStats stats) {
        return stats.getSize();
    }
}
```

2.1.4 Counters

A counter is a simple incrementing and decrementing 64-bit integer:

```
final Counter evictions = registry.counter(name(SessionStore.class, "cache-evictions"));
evictions.inc();
evictions.inc(3);
evictions.dec();
evictions.dec(2);
```

All `Counter` metrics start out at 0.

2.1.5 Histograms

A `Histogram` measures the distribution of values in a stream of data: e.g., the number of results returned by a search:

```
final Histogram resultCounts = registry.histogram(name(ProductDAO.class, "result-counts
↪");
resultCounts.update(results.size());
```

`Histogram` metrics allow you to measure not just easy things like the min, mean, max, and standard deviation of values, but also quantiles like the median or 95th percentile.

Traditionally, the way the median (or any other quantile) is calculated is to take the entire data set, sort it, and take the value in the middle (or 1% from the end, for the 99th percentile). This works for small data sets, or batch processing systems, but not for high-throughput, low-latency services.

The solution for this is to sample the data as it goes through. By maintaining a small, manageable reservoir which is statistically representative of the data stream as a whole, we can quickly and easily calculate quantiles which are valid approximations of the actual quantiles. This technique is called **reservoir sampling**.

Metrics provides a number of different `Reservoir` implementations, each of which is useful.

Uniform Reservoirs

A histogram with a uniform reservoir produces quantiles which are valid for the entirely of the histogram's lifetime. It will return a median value, for example, which is the median of all the values the histogram has ever been updated with. It does this by using an algorithm called Vitter's R), which randomly selects values for the reservoir with linearly-decreasing probability.

Use a uniform histogram when you're interested in long-term measurements. Don't use one where you'd want to know if the distribution of the underlying data stream has changed recently.

Exponentially Decaying Reservoirs

A histogram with an exponentially decaying reservoir produces quantiles which are representative of (roughly) the last five minutes of data. It does so by using a forward-decaying priority reservoir with an exponential weighting towards newer data. Unlike the uniform reservoir, an exponentially decaying reservoir represents **recent data**, allowing you to know very quickly if the distribution of the data has changed. *Timers* use histograms with exponentially decaying reservoirs by default.

Sliding Window Reservoirs

A histogram with a sliding window reservoir produces quantiles which are representative of the past `N` measurements.

Sliding Time Window Reservoirs

A histogram with a sliding time window reservoir produces quantiles which are strictly representative of the past `N` seconds (or other time period).

> **Warning:** While `SlidingTimeWindowReservoir` is easier to understand than `ExponentiallyDecayingReservoir`, it is not bounded in size, so using it to sample a high-frequency process can require a significant amount of memory. Because it records every measurement, it's also the slowest reservoir type.

2.1.6 Meters

A meter measures the *rate* at which a set of events occur:

```
final Meter getRequests = registry.meter(name(WebProxy.class, "get-requests", "requests
↪"));
getRequests.mark();
getRequests.mark(requests.size());
```

Meters measure the rate of the events in a few different ways. The *mean* rate is the average rate of events. It's generally useful for trivia, but as it represents the total rate for your application's entire lifetime (e.g., the total number of requests handled, divided by the number of seconds the process has been running), it doesn't offer a sense of recency. Luckily, meters also record three different *exponentially-weighted moving average* rates: the 1-, 5-, and 15-minute moving averages.

Hint: Just like the Unix load averages visible in `uptime` or `top`.

2.1.7 Timers

A timer is basically a *histogram* of the duration of a type of event and a *meter* of the rate of its occurrence.

```
final Timer timer = registry.timer(name(WebProxy.class, "get-requests"));

final Timer.Context context = timer.time();
try {
    // handle request
} finally {
    context.stop();
}
```

Note: Elapsed times for it events are measured internally in nanoseconds, using Java's high-precision `System.nanoTime()` method. Its precision and accuracy vary depending on operating system and hardware.

2.1.8 Metric Sets

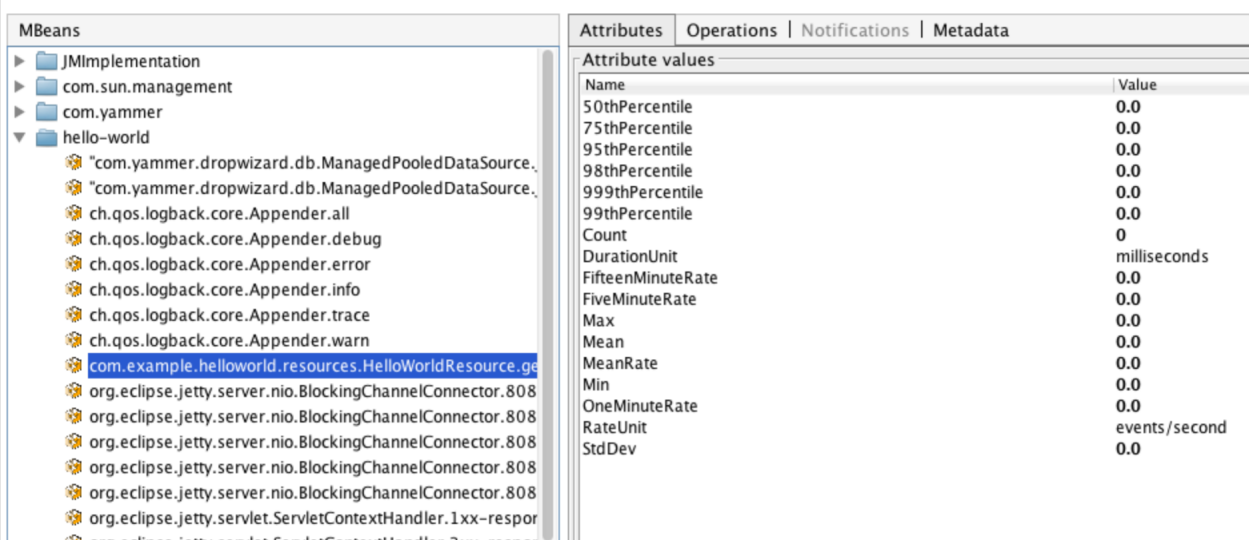
Metrics can also be grouped together into reusable metric sets using the `MetricSet` interface. This allows library authors to provide a single entry point for the instrumentation of a wide variety of functionality.

2.1.9 Reporters

Reporters are the way that your application exports all the measurements being made by its metrics. `metrics-core` comes with four ways of exporting your metrics: *JMX*, *console*, *SLF4J*, and *CSV*.

JMX

With `JmxReporter`, you can expose your metrics as JMX MBeans. To explore this you can use VisualVM (which ships with most JDKs as `jvisualvm`) with the VisualVM-MBeans plugins installed or JConsole (which ships with most JDKs as `jconsole`):



Tip: If you double-click any of the metric properties, VisualVM will start graphing the data for that property. Sweet, eh?

Warning: We don't recommend that you try to gather metrics from your production environment. JMX's RPC API is fragile and bonkers. For development purposes and browsing, though, it can be very useful.

To report metrics via JMX:

```
final JmxReporter reporter = JmxReporter.forRegistry(registry).build();
reporter.start();
```

Console

For simple benchmarks, Metrics comes with `ConsoleReporter`, which periodically reports all registered metrics to the console:

```
final ConsoleReporter reporter = ConsoleReporter.forRegistry(registry)
                                                .convertRatesTo(TimeUnit.SECONDS)
                                                .convertDurationsTo(TimeUnit.
↪MILLISECONDS)
                                                .build();
reporter.start(1, TimeUnit.MINUTES);
```

CSV

For more complex benchmarks, Metrics comes with `CsvReporter`, which periodically appends to a set of `.csv` files in a given directory:

```
final CsvReporter reporter = CsvReporter.forRegistry(registry)
                                        .formatFor(Locale.US)
                                        .convertRatesTo(TimeUnit.SECONDS)
                                        .convertDurationsTo(TimeUnit.MILLISECONDS)
                                        .build(new File("~/projects/data/"));
reporter.start(1, TimeUnit.SECONDS);
```

For each metric registered, a `.csv` file will be created, and every second its state will be written to it as a new row.

SLF4J

It's also possible to log metrics to an SLF4J logger:

```
final Slf4jReporter reporter = Slf4jReporter.forRegistry(registry)
                                            .outputTo(LoggerFactory.getLogger("com.
↪example.metrics"))
                                            .convertRatesTo(TimeUnit.SECONDS)
                                            .convertDurationsTo(TimeUnit.MILLISECONDS)
                                            .build();
reporter.start(1, TimeUnit.MINUTES);
```

Other Reporters

Metrics has other reporter implementations, too:

- *MetricsServlet* is a servlet which not only exposes your metrics as a JSON object, but it also runs your health checks, performs thread dumps, and exposes valuable JVM-level and OS-level information.
- *GangliaReporter* allows you to constantly stream metrics data to your Ganglia servers.
- *GraphiteReporter* allows you to constantly stream metrics data to your Graphite servers.

2.2 Health Checks

Metrics also provides you with a consistent, unified way of performing application health checks. A health check is basically a small self-test which your application performs to verify that a specific component or responsibility is performing correctly.

To create a health check, extend the `HealthCheck` class:

```
public class DatabaseHealthCheck extends HealthCheck {
    private final Database database;

    public DatabaseHealthCheck(Database database) {
        this.database = database;
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    protected Result check() throws Exception {
        if (database.ping()) {
            return Result.healthy();
        }
        return Result.unhealthy("Can't ping database");
    }
}
```

In this example, we've created a health check for a `Database` class on which our application depends. Our fictitious `Database` class has a `#ping()` method, which executes a safe test query (e.g., `SELECT 1`). `#ping()` returns `true` if the query returns the expected result, returns `false` if it returns something else, and throws an exception if things have gone seriously wrong.

Our `DatabaseHealthCheck`, then, takes a `Database` instance and in its `#check()` method, attempts to ping the database. If it can, it returns a **healthy** result. If it can't, it returns an **unhealthy** result.

Note: Exceptions thrown inside a health check's `#check()` method are automatically caught and turned into unhealthy results with the full stack trace.

To register a health check, either use a `HealthCheckRegistry` instance:

```
registry.register("database", new DatabaseHealthCheck(database));
```

You can also run the set of registered health checks:

```
for (Entry<String, Result> entry : registry.runHealthChecks().entrySet()) {
    if (entry.getValue().isHealthy()) {
        System.out.println(entry.getKey() + ": OK");
    } else {
        System.out.println(entry.getKey() + ": FAIL");
    }
}
```

2.3 Instrumenting Ehcache

The `metrics-ehcache` module provides `InstrumentedEhcache`, a decorator for Ehcache caches:

```
final Cache c = new Cache(new CacheConfiguration("test", 100));
MANAGER.addCache(c);
this.cache = InstrumentedEhcache.instrument(registry, c);
```

Instrumenting an `Ehcache` instance creates gauges for all of the Ehcache-provided statistics:

hits	The number of times a requested item was found in the cache.
in-memory-hits	Number of times a requested item was found in the memory store.
off-heap-hits	Number of times a requested item was found in the off-heap store.
on-disk-hits	Number of times a requested item was found in the disk store.
misses	Number of times a requested item was not found in the cache.
in-memory-misses	Number of times a requested item was not found in the memory store.
off-heap-misses	Number of times a requested item was not found in the off-heap store.
on-disk-misses	Number of times a requested item was not found in the disk store.
objects	Number of elements stored in the cache.
in-memory-objects	Number of objects in the memory store.
off-heap-objects	Number of objects in the off-heap store.
on-disk-objects	Number of objects in the disk store.
mean-get-time	The average get time. Because ehcache supports JDK1.4.2, each get time uses `System.currentTimeMillis()`, rather than nanoseconds. The accuracy is thus limited.
mean-search-time	The average execution time (in milliseconds) within the last sample period.
eviction-count	The number of cache evictions, since the cache was created, or statistics were cleared.
searches-per-second	The number of search executions that have completed in the last second.
accuracy	A human readable description of the accuracy setting. One of "None", "Best Effort" or "Guaranteed".

It also adds full timers for the cache's `get` and `put` methods.

The metrics are all scoped to the cache's class and name, so a `Cache` instance named `users` would have metric names like `net.sf.ehcache.Cache.users.get`, etc.

2.4 Reporting to Ganglia

The `metrics-ganglia` module provides `GangliaReporter`, which allows your application to constantly stream metric values to a Ganglia server:

```
final GMetric ganglia = new GMetric("ganglia.example.com", 8649, UDPAddressingMode.
↪MULTICAST, 1);
final GangliaReporter reporter = GangliaReporter.forRegistry(registry)
                                                .convertRatesTo(TimeUnit.SECONDS)
                                                .convertDurationsTo(TimeUnit.
↪MILLISECONDS)
                                                .build(ganglia);
reporter.start(1, TimeUnit.MINUTES);
```

2.5 Reporting to Graphite

The `metrics-graphite` module provides `GraphiteReporter`, which allows your application to constantly stream metric values to a Graphite server:

```
final Graphite graphite = new Graphite(new InetSocketAddress("graphite.example.com",␣
↪2003));
final GraphiteReporter reporter = GraphiteReporter.forRegistry(registry)
                                                  .prefixedWith("web1.example.com")
                                                  .convertRatesTo(TimeUnit.SECONDS)
```

(continues on next page)

(continued from previous page)

```
                                                    .convertDurationsTo(TimeUnit.
↪MILLISECONDS)
                                                    .filter(MetricFilter.ALL)
                                                    .build(graphite);
reporter.start(1, TimeUnit.MINUTES);
```

If you prefer to write metrics in batches using pickle, you can use the `PickledGraphite`:

```
final Graphite pickledGraphite = new PickledGraphite(new InetSocketAddress("graphite.
↪example.com", 2004));
final GraphiteReporter reporter = GraphiteReporter.forRegistry(registry)
                                                    .prefixedWith("web1.example.com")
                                                    .convertRatesTo(TimeUnit.SECONDS)
                                                    .convertDurationsTo(TimeUnit.
↪MILLISECONDS)
                                                    .filter(MetricFilter.ALL)
                                                    .build(pickledGraphite);
reporter.start(1, TimeUnit.MINUTES);
```

2.6 Instrumenting Apache HttpClient

The `metrics-httpclient` module provides `InstrumentedHttpClientConnManager` and `InstrumentedHttpClients`, two instrumented versions of Apache HttpClient 4.x classes.

`InstrumentedHttpClientConnManager` is a thread-safe `HttpClientConnectionManager` implementation which measures the number of open connections in the pool and the rate at which new connections are opened.

`InstrumentedHttpClients` follows the `HttpClients` builder pattern and adds per-HTTP method timers for HTTP requests.

2.6.1 Metric naming strategies

The default per-method metric naming and scoping strategy can be overridden by passing an implementation of `HttpClientMetricNameStrategy` to the `InstrumentedHttpClients.createDefault` method.

A number of pre-rolled strategies are available, e.g.:

```
HttpClient client = InstrumentedHttpClients.createDefault(registry,␣
↪HttpClientMetricNameStrategies.HOST_AND_METHOD);
```

2.7 Instrumenting JDBI

The `metrics-jdbi` module provides a `TimingCollector` implementation for JDBI, an SQL convenience library.

To use it, just add a `InstrumentedTimingCollector` instance to your `DBI`:

```
final DBI dbi = new DBI(dataSource);
dbi.setTimingCollector(new InstrumentedTimingCollector(registry));
```

`InstrumentedTimingCollector` keeps per-SQL-object timing data, as well as general raw SQL timing data. The metric names for each query are constructed by an `StatementNameStrategy` instance, of which there are many implementations. By default, `StatementNameStrategy` uses `SmartNameStrategy`, which attempts to effectively handle both queries from bound objects and raw SQL.

2.8 Instrumenting Jersey 1.x

The `metrics-jersey` module provides `InstrumentedResourceMethodDispatchAdapter`, which allows you to instrument methods on your Jersey 1.x resource classes:

An instance of `InstrumentedResourceMethodDispatchAdapter` must be registered with your Jersey application's `ResourceConfig` as a singleton provider for this to work.

```
public class ExampleApplication {
     private final DefaultResourceConfig config = new DefaultResourceConfig();

     public void init() {
         config.getSingletons().add(new␣
↪InstrumentedResourceMethodDispatchAdapter(registry));
         config.getClasses().add(ExampleResource.class);
     }
 }


 @Path("/example")
 @Produces(MediaType.TEXT_PLAIN)
 public class ExampleResource {
     @GET
     @Timed
     public String show() {
         return "yay";
     }
 }
```

The `show` method in the above example will have a timer attached to it, measuring the time spent in that method.

Use of the `@Metered` and `@ExceptionMetered` annotations is also supported.

2.9 Instrumenting Jersey 2.x

Jersey 2.x changed the API for how resource method monitoring works, so a new module `metrics-jersey2` provides `InstrumentedResourceMethodApplicationListener`, which allows you to instrument methods on your Jersey 2.x resource classes:

The `metrics-jersey2` module provides `InstrumentedResourceMethodApplicationListener`, which allows you to instrument methods on your Jersey 2.x resource classes:

An instance of `InstrumentedResourceMethodApplicationListener` must be registered with your Jersey application's `ResourceConfig` as a singleton provider for this to work.

```
public class ExampleApplication extends ResourceConfig {
    .
```

(continues on next page)

(continued from previous page)

```
    .
    .
    register(new InstrumentedResourceMethodApplicationListener (new MetricRegistry()));
    config = config.register(ExampleResource.class);
    .
    .
    .
}

@Path("/example")
@Produces(MediaType.TEXT_PLAIN)
public class ExampleResource {
    @GET
    @Timed
    public String show() {
        return "yay";
    }
}
```

The `show` method in the above example will have a timer attached to it, measuring the time spent in that method.

Use of the `@Metered` and `@ExceptionMetered` annotations is also supported.

2.10 Instrumenting Jetty

The `metrics-jetty8` (Jetty 8.0), `metrics-jetty9-legacy` (Jetty 9.0), and `metrics-jetty9` (Jetty 9.1 and higher) modules provides a set of instrumented equivalents of Jetty classes: `InstrumentedBlockingChannelConnector`, `InstrumentedHandler`, `InstrumentedQueuedThreadPool`, `InstrumentedSelectChannelConnector`, and `InstrumentedSocketConnector`.

The `Connector` implementations are simple, instrumented subclasses of the Jetty connector types which measure connection duration, the rate of accepted connections, connections, disconnections, and the total number of active connections.

`InstrumentedQueuedThreadPool` is a `QueuedThreadPool` subclass which measures the ratio of idle threads to working threads as well as the absolute number of threads (idle and otherwise).

`InstrumentedHandler` is a `Handler` decorator which measures a wide range of HTTP behavior: dispatch times, requests, resumes, suspends, expires, the number of active, suspected, and dispatched requests, as well as meters of responses with `1xx`, `2xx`, `3xx`, `4xx`, and `5xx` status codes. It even has gauges for the ratios of `4xx` and `5xx` response rates to overall response rates. Finally, it includes meters for requests by the HTTP method: `GET`, `POST`, etc.

2.11 Instrumenting Log4j

The `metrics-log4j` and `metrics-log4j2` modules provide `InstrumentedAppender`, a Log4j `Appender` implementation (for log4j 1.x and log4j 2.x correspondingly) which records the rate of logged events by their logging level.

You can add it to the root logger programmatically.

For log4j 1.x:

```
InstrumentedAppender appender = new InstrumentedAppender(registry);
appender.activateOptions();
LogManager.getRootLogger().addAppender(appender);
```

For log4j 2.x:

```
Filter filter = null;        // That's fine if we don't use filters; https://logging.
↪apache.org/log4j/2.x/manual/filters.html
PatternLayout layout = null; // The layout isn't used in InstrumentedAppender

InstrumentedAppender appender = new InstrumentedAppender(metrics, filter, layout, false);
appender.start();

LoggerContext context = (LoggerContext) LogManager.getContext(false);
Configuration config = context.getConfiguration();
config.getLoggerConfig(LogManager.ROOT_LOGGER_NAME).addAppender(appender, level, filter);
context.updateLoggers(config);
```

2.12 Instrumenting Logback

The `metrics-logback` module provides `InstrumentedAppender`, a Logback `Appender` implementation which records the rate of logged events by their logging level.

You add it to the root logger programmatically:

```
final LoggerContext factory = (LoggerContext) LoggerFactory.getILoggerFactory();
final Logger root = factory.getLogger(Logger.ROOT_LOGGER_NAME);

final InstrumentedAppender metrics = new InstrumentedAppender(registry);
metrics.setContext(root.getLoggerContext());
metrics.start();
root.addAppender(metrics);
```

2.13 JVM Instrumentation

The `metrics-jvm` module contains a number of reusable gauges and *metric sets* which allow you to easily instrument JVM internals.

Supported metrics include:

- Run count and elapsed times for all supported garbage collectors
- Memory usage for all memory pools, including off-heap memory
- Breakdown of thread states, including deadlocks
- File descriptor usage
- Buffer pool sizes and utilization (Java 7 only)

2.14 JSON Support

Metrics comes with `metrics-json`, which features two reusable modules for Jackson.

This allows for the serialization of all metric types and health checks to a standard, easily-parsable JSON format.

2.15 Metrics Servlets

The `metrics-servlets` module provides a handful of useful servlets:

2.15.1 HealthCheckServlet

`HealthCheckServlet` responds to `GET` requests by running all the [health checks](#health-checks) and returning `501 Not Implemented` if no health checks are registered, `200 OK` if all pass, or `500 Internal Service Error` if one or more fail. The results are returned as a human-readable `text/plain` entity.

`HealthCheckServlet` requires that the servlet context has a `HealthCheckRegistry` named `com.codahale.metrics.servlets.HealthCheckServlet.registry`. You can subclass `MetricsServletContextListener`, which will add a specific `HealthCheckRegistry` to the servlet context.

2.15.2 ThreadDumpServlet

`ThreadDumpServlet` responds to `GET` requests with a `text/plain` representation of all the live threads in the JVM, their states, their stack traces, and the state of any locks they may be waiting for.

2.15.3 MetricsServlet

`MetricsServlet` exposes the state of the metrics in a particular registry as a JSON object.

`MetricsServlet` requires that the servlet context has a `MetricRegistry` named `com.codahale.metrics.servlets.MetricsServlet.registry`. You can subclass `MetricsServletContextListener`, which will add a specific `MetricRegistry` to the servlet context.

`MetricsServlet` also takes an initialization parameter, `show-jvm-metrics`, which if `"false"` will disable the outputting of JVM-level information in the JSON object.

2.15.4 PingServlet

`PingServlet` responds to `GET` requests with a `text/plain/200 OK` response of `pong`. This is useful for determining liveness for load balancers, etc.

2.15.5 AdminServlet

`AdminServlet` aggregates `HealthCheckServlet`, `ThreadDumpServlet`, `MetricsServlet`, and `PingServlet` into a single, easy-to-use servlet which provides a set of URIs:

- `/`: an HTML admin menu with links to the following:
  - `/healthcheck`: `HealthCheckServlet`
  - `/metrics`: `MetricsServlet`
  - `/ping`: `PingServlet`
  - `/threads`: `ThreadDumpServlet`

You will need to add your `MetricRegistry` and `HealthCheckRegistry` instances to the servlet context as attributes named `com.codahale.metrics.servlets.MetricsServlet.registry` and `com.codahale.metrics.servlets.HealthCheckServlet.registry`, respectively. You can do this using the Servlet API by extending `MetricsServlet.ContextListener` for MetricRegistry:

```
public class MyMetricsServletContextListener extends MetricsServlet.ContextListener {

    public static final MetricRegistry METRIC_REGISTRY = new MetricRegistry();

    @Override
    protected MetricRegistry getMetricRegistry() {
        return METRIC_REGISTRY;
    }

}
```

And by extending `HealthCheckServlet.ContextListener` for HealthCheckRegistry:

```
public class MyHealthCheckServletContextListener extends HealthCheckServlet.
↪ContextListener {

    public static final HealthCheckRegistry HEALTH_CHECK_REGISTRY = new␣
↪HealthCheckRegistry();

    @Override
    protected HealthCheckRegistry getHealthCheckRegistry() {
        return HEALTH_CHECK_REGISTRY;
    }

}
```

Then you will need to register servlet context listeners either in you `web.xml` or annotating the class with `@WebListener` if you are in servlet 3.0 environment. In `web.xml`:

```
<listener>
        <listener-class>com.example.MyMetricsServletContextListener</listener-class>
</listener>
<listener>
        <listener-class>com.example.MyHealthCheckServletContextListener</listener-class>
</listener>
```

You will also need to register `AdminServlet` in `web.xml`:

```
<servlet>
        <servlet-name>metrics</servlet-name>
        <servlet-class>com.codahale.metrics.servlets.AdminServlet</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>metrics</servlet-name>
        <url-pattern>/metrics/*</url-pattern>
</servlet-mapping>
```

2.16 Instrumenting Web Applications

The `metrics-servlet` module provides a Servlet filter which has meters for status codes, a counter for the number of active requests, and a timer for request duration. By default the filter will use `com.codahale.metrics.servlet.InstrumentedFilter` as the base name of the metrics. You can use the filter in your `web.xml` like this:

```
<filter>
    <filter-name>instrumentedFilter</filter-name>
    <filter-class>com.codahale.metrics.servlet.InstrumentedFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>instrumentedFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

An optional filter init-param `name-prefix` can be specified to override the base name of the metrics associated with the filter mapping. This can be helpful if you need to instrument multiple url patterns and give each a unique name.

```
<filter>
    <filter-name>instrumentedFilter</filter-name>
    <filter-class>com.codahale.metrics.servlet.InstrumentedFilter</filter-class>
    <init-param>
        <param-name>name-prefix</param-name>
        <param-value>authentication</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>instrumentedFilter</filter-name>
    <url-pattern>/auth/*</url-pattern>
</filter-mapping>
```

You will need to add your `MetricRegistry` to the servlet context as an attribute named `com.codahale.metrics.servlet.InstrumentedFilter.registry`. You can do this using the Servlet API by extending `InstrumentedFilterContextListener`:

```
public class MyInstrumentedFilterContextListener extends␣
↪InstrumentedFilterContextListener {
    public static final MetricRegistry REGISTRY = new MetricRegistry();

    @Override
    protected MetricRegistry getMetricRegistry() {
        return REGISTRY;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

2.17 Third Party Libraries

If you're looking to integrate with something not provided by the main Metrics libraries, check out the many third-party libraries which extend Metrics:

- metrics-librato provides a reporter for Librato Metrics, a scalable metric collection, aggregation, monitoring, and alerting service.
- metrics-spring provides integration with Spring
- sematext-metrics-reporter provides a reporter for SPM.
- wicket-metrics provides easy integration for your Wicket application.
- metrics-guice provides integration with Guice.
- metrics-guice-servlet provides Guice Servlet integration with AdminServlet.
- metrics-scala provides an API optimized for Scala.
- metrics-clojure provides an API optimized for Clojure.
- metrics-cassandra provides a reporter for Apache Cassandra.
- MetricCatcher Turns JSON over UDP into Metrics so that non-jvm languages can know what's going on too.
- metrics-reporter-config DropWizard-eqsue YAML configuration of reporters.
- metrics-elasticsearch-reporter provides a reporter for elasticsearch
- metrics-statsd provides a Metrics 2.x and 3.x reporter for StatsD
- metrics-datadog provides a reporter to send data to Datadog
- metrics-new-relic provides a reporter which sends data to New Relic.
- metrics-influxdb provides a reporter which announces measurements to InfluxDB
- metrics-cdi provides integration with CDI environments
- metrics-aspectj provides integration with AspectJ
- camel-metrics provides component for your Apache Camel route
- metrics-munin-reporter provides a reporter for Munin
- jersey2-metrics provides integration with Jersey 2.
- jersey-metrics-filter provides integration with Jersey 1.
- hdrhistogram-metrics-reservoir provides a Reservoir backed by HdrHistogram.
- metrics-circonus provides a registry and reporter for sending metrics (including full histograms) to Circonus.

CHAPTER

THREE

ABOUT METRICS

3.1 Contributors

Many, many thanks to:

- Alan Woodward
- Alex Lambert
- Basil James Whitehouse III
- Benjamin Gehrels
- Bernardo Gomez Palacio
- Brian Ehmann
- Brian Roberts
- Bruce Mitchener
- C. Scott Andreas
- ceetav
- Charles Care
- Chris Birchall
- Chris Burroughs
- Christopher Swenson
- Ciamac Moallemi
- Cliff Moon
- Collin VanDyck
- Dag Liodden
- Dale Wijnand
- Dan Brown
- Dan Everton
- Dan Revel
- David Sutherland
- Diwaker Gupta
- Drew Stephens

- Edwin Shin
- Eric Daigneault
- Evan Jones
- François Beausoleil
- Gerolf Seitz
- Greg Bowyer
- Jackson Davis
- James Casey
- Jan-Helge Bergesen
- Jason A. Beranek
- Jason Slagle
- JD Maturen
- Jeff Hodges
- Jesper Blomquist
- Jesse Eichar
- John Ewart
- John Wang
- Justin Plock
- Kevin Clark
- Mahesh Tiyyagura
- Martin Traverso
- Matt Abrams
- Matt Ryall
- Matthew Gilliard
- Matthew O'Connor
- Mathijs Vogelzang
- Mårten Gustafson
- Michał Minicki
- Neil Prosser
- Nick Telford
- Niklas Konstenius
- Norbert Potocki
- Pablo Fernandez
- Paul Bloch
- Paul Brown
- Paul Doran

- Paul Sandwald
- Realbot
- Robby Walker
- Ryan Kennedy
- Ryan W Tenney
- Sam Perman
- Sean Laurent
- Shaneal Manek
- Steven Schlansker
- Stewart Allen
- Thomas Dudziak
- Tobias Lidskog
- Yang Ye
- Wolfgang Schell

3.2 Release Notes

3.2.1 v3.0.1: Jul 23 2013

- Fixed NPE in `MetricRegistry#name`.
- `ScheduledReporter` and `JmxReporter` now implement `Closeable`.
- Fixed cast exception for async requests in `metrics-jetty9`.
- Added support for `Access-Control-Allow-Origin` to `MetricsServlet`.
- Fixed numerical issue with `Meter` EWMA rates.
- Deprecated `AdminServletContextListener` in favor of `MetricsServlet.ContextListener` and `HealthCheckServlet.ContextListener`.
- Added additional constructors to `HealthCheckServlet` and `MetricsServlet`.

3.2.2 v3.0.0: June 10 2013

- Renamed `DefaultWebappMetricsFilter` to `InstrumentedFilter`.
- Renamed `MetricsContextListener` to `InstrumentedFilterContextListener` and made it fully abstract to avoid confusion.
- Renamed `MetricsServletContextListener` to `AdminServletContextListener` and made it fully abstract to avoid confusion.
- Upgraded to Servlet API 3.1.
- Upgraded to Jackson 2.2.2.
- Upgraded to Jetty 8.1.11.

3.2.3 v3.0.0-RC1: May 31 2013

- Added `SharedMetricRegistries`, a singleton for sharing named metric registries.
- Fixed XML configuration for `metrics-ehcache`.
- Fixed XML configuration for `metrics-jersey`.
- Fixed XML configuration for `metrics-log4j`.
- Fixed XML configuration for `metrics-logback`.
- Fixed a counting bug in `metrics-jetty9`'s InstrumentedHandler.
- Added `MetricsContextListener` to `metrics-servlet`.
- Added `MetricsServletContextListener` to `metrics-servlets`.
- Extracted the `Counting` interface.
- Reverted `SlidingWindowReservoir` to a synchronized implementation.
- Added the implementation version to the JAR manifests.
- Made dependencies for all modules conform to Maven Enforcer's convergence rules.
- Fixed `Slf4jReporter`'s logging of 99th percentiles.
- Added optional name prefixing to `GraphiteReporter`.
- Added metric-specific overrides of rate and duration units to `JmxReporter`.
- Documentation fixes.

3.2.4 v3.0.0-BETA3: May 13 2013

- Added `ScheduledReporter#report()` for manual reporting.
- Fixed overly-grabby catches in `HealthCheck` and `InstrumentedResourceMethodDispatchProvider`.
- Fixed phantom reads in `SlidingWindowReservoir`.
- Revamped `metrics-jetty9`, removing `InstrumentedConnector` and improving the API.
- Fixed OSGi imports for `sun.misc`.
- Added a strategy class for `HttpClient` metrics.
- Upgraded to Jetty 9.0.3.
- Upgraded to Jackson 2.2.1.
- Upgraded to Ehcache 2.6.6.
- Upgraded to Logback 1.0.13.
- Upgraded to HttpClient 4.2.5.
- Upgraded to gmetric4j 1.0.3, which allows for host spoofing.

3.2.5 v3.0.0-BETA2: Apr 22 2013

- Metrics is now under the `com.codahale.metrics` package, with the corresponding changes in Maven artifact groups. This should allow for an easier upgrade path without classpath conflicts.
- `MetricRegistry` no longer has a name.
- Added `metrics-jetty9` for Jetty 9.
- `JmxReporter` takes an optional domain property to disambiguate multiple reporters.
- Fixed Java 6 compatibility problem. (Also added Java 6 as a CI environment.)
- Added `MetricRegistryListener.Base`.
- Switched `Counter`, `Meter`, and `EWMA` to use JSR133's `LongAdder` instead of `AtomicLong`, improving contended concurrency.
- Added `MetricRegistry#buildMap()`, allowing for custom map implementations in `MetricRegistry`.
- Added `MetricRegistry#removeMatching(MetricFilter)`.
- Changed `metrics-json` to optionally depend on `metrics-healthcheck`.
- Upgraded to Jetty 8.1.10 for `metrics-jetty8`.

3.2.6 v3.0.0-BETA1: Apr 01 2013

- Total overhaul of most of the core Metrics classes:
  - Metric names are now just dotted paths like `com.example.Thing`, allowing for very flexible scopes, etc.
  - Meters and timers no longer have rate or duration units; those are properties of reporters.
  - Reporter architecture has been radically simplified, fixing many bugs.
  - Histograms and timers can take arbitrary reservoir implementations.
  - Added sliding window reservoir implementations.
  - Added `MetricSet` for sets of metrics.
- Changed package names to be OSGi-compatible and added OSGi bundling.
- Extracted JVM instrumentation to `metrics-jvm`.
- Extracted Jackson integration to `metrics-json`.
- Removed `metrics-guice`, `metrics-scala`, and `metrics-spring`.
- Renamed `metrics-servlet` to `metrics-servlets`.
- Renamed `metrics-web` to `metrics-servlet`.
- Renamed `metrics-jetty` to `metrics-jetty8`.
- Many more small changes!

3.2.7 v2.2.0: Nov 26 2012

- Removed all OSGi bundling. This will be back in 3.0.
- Added `InstrumentedSslSelectChannelConnector` and `InstrumentedSslSocketConnector`.
- Made all metric names JMX-safe.
- Upgraded to Ehcache 2.6.2.
- Upgraded to Apache HttpClient 4.2.2.
- Upgraded to Jersey 1.15.
- Upgraded to Log4j 1.2.17.
- Upgraded to Logback 1.0.7.
- Upgraded to Spring 3.1.3.
- Upgraded to Jetty 8.1.8.
- Upgraded to SLF4J 1.7.2.
- Replaced usage of `Unsafe` in `InstrumentedResourceMethodDispatchProvider` with type erasure trickery.

3.2.8 v2.1.5: Nov 19 2012

- Upgraded to Jackson 2.1.1.

3.2.9 v2.1.4: Nov 15 2012

- Added OSGi bundling manifests.

3.2.10 v2.1.3: Aug 06 2012

- Upgraded to Apache HttpClient 4.2.1.
- Changed `InstrumentedClientConnManager` to extend `PoolingClientConnectionManager` instead of the deprecated `ThreadSafeClientConnManager`.
- Fixed a bug in `ExponentiallyDecayingSample` with long periods of inactivity.
- Fixed problems with re-registering metrics in JMX.
- Added support for `DnsResolver` instances to `InstrumentedClientConnManager`.
- Added support for formatted health check error messages.

3.2.11 v2.1.2: Apr 11 2012

- Fixed double-registration in `metrics-guice`.

3.2.12 v2.1.1: Mar 13 2012

- Fixed instrumentation of all usages of `InstrumentedHttpClient`.

3.2.13 v2.1.0: Mar 12 2012

- Added support for Java 7's direct and mapped buffer pool stats in `VirtualMachineMetrics` and `metrics-servlet`.
- Added support for XML configuration in `metrics-ehcache`.
- `metrics-spring` now support `@Gauge`-annotated fields.
- Opened `GraphiteReporter` up for extension.
- Added `group` and `type` to `metrics-annotations`, `metrics-guice`, `metrics-jersey`, and `metrics-spring`.
- Fixed handling of non-int gauges in `GangliaReporter`.
- Fixed `NullPointerException` errors in `metrics-spring`.
- General improvements to `metrics-spring`, including allowing custom `Clock` instances.

3.2.14 v2.0.3: Feb 24 2012

- Change logging of `InstanceNotFoundException` exceptions thrown while unregistering a metric in `JmxReporter` to `TRACE`. It being `WARN` resulted in huge log dumps preventing process shutdowns when applications had ~1K+ metrics.
- Upgraded to Spring 3.1.1 for `metrics-spring`.
- Upgraded to JDBI 2.31.2.
- Upgraded to Jersey 1.12.
- Upgraded to Jetty 7.6.1.
- Fixed rate units for meters in `GangliaReporter`.

3.2.15 v2.0.2: Feb 09 2012

- `InstrumentationModule` in `metrics-guice` now uses the default `MetricsRegistry` and `HealthCheckRegistry`.

3.2.16 v2.0.1: Feb 08 2012

- Fixed a concurrency bug in `JmxReporter`.

3.2.17 v2.0.0: Feb 07 2012

- Upgraded to Jackson 1.9.4.
- Upgraded to Jetty 7.6.0.
- Added escaping for garbage collector and memory pool names in `GraphiteReporter`.
- Fixed the inability to start and stop multiple reporter instances.
- Switched to using a backported version of `ThreadLocalRandom` for `UniformSample` and `ExponentiallyDecayingSample` to reduce lock contention on random number generation.
- Removed `Ordered` from `TimedAnnotationBeanPostProcessor` in `metrics-spring`.
- Upgraded to JDBI 2.31.1.
- Upgraded to Ehcache 2.5.1.
- Added `#timerContext()` to Scala `Timer`.

3.2.18 v2.0.0-RC0: Jan 19 2012

- Added FindBugs checks to the build process.
- Fixed the catching of `Error` instances thrown during health checks.
- Added `enable` static methods to `CsvReporter` and changed `CsvReporter(File, MetricsRegistry)` to `CsvReporter(MetricsRegistry, File)`.
- Slimmed down `InstrumentedEhcache`.
- Hid the internals of `GangliaReporter`.
- Hid the internals of `metrics-guice`.
- Changed `metrics-httpclient` to consistently associate metrics with the `org.apache` class being extended.
- Hid the internals of `metrics-httpclient`.
- Rewrote `InstrumentedAppender` in `metrics-log4j`. It no longer forwards events to an appender. Instead, you can just attach it to your root logger to instrument logging.
- Rewrote `InstrumentedAppender` in `metrics-logback`. No major API changes.
- Fixed bugs with `@ExceptionMetered`-annotated resource methods in `metrics-jersey`.
- Fixed bugs generating `Snapshot` instances from concurrently modified collections.
- Fixed edge case in `MetricsServlet`'s thread dumps where one thread could be missed.
- Added `RatioGauge` and `PercentGauge`.
- Changed `InstrumentedQueuedThreadPool`'s `percent-idle` gauge to be a ratio.
- Decomposed `MetricsServlet` into a set of focused servlets: `HealthCheckServlet`, `MetricsServlet`, `PingServlet`, and `ThreadDumpServlet`. The top-level servlet which provides the HTML menu page is now `AdminServlet`.
- Added `metrics-spring`.

3.2.19 v2.0.0-BETA19: Jan 07 2012

- Added absolute memory usage to `MetricsServlet`.
- Extracted `@Timed` etc. to `metrics-annotations`.
- Added `metrics-jersey`, which provides a class allowing you to automatically instrument all `@Timed`, `@Metered`, and `@ExceptionMetered`-annotated resource methods.
- Moved all classes in `metrics-scala` from `com.yammer.metrics` to `com.yammer.metrics.scala`.
- Renamed `CounterMetric` to `Counter`.
- Renamed `GaugeMetric` to `Gauge`.
- Renamed `HistogramMetric` to `Histogram`.
- Renamed `MeterMetric` to `Meter`.
- Renamed `TimerMetric` to `Timer`.
- Added `ToggleGauge`, which returns `1` the first time it's called and `0` every time after that.
- Now licensed under Apache License 2.0.
- Converted `VirtualMachineMetrics` to a non-singleton class.
- Removed `Utils`.
- Removed deprecated constructors from `Meter` and `Timer`.
- Removed `LoggerMemoryLeakFix`.
- `DeathRattleExceptionHandler` now logs to SLF4J, not syserr.
- Added `MetricsRegistry#groupedMetrics()`.
- Removed `Metrics#allMetrics()`.
- Removed `Metrics#remove(MetricName)`.
- Removed `MetricsRegistry#threadPools()` and `#newMeterTickThreadPool()` and added `#newScheduledThreadPool`.
- Added `MetricsRegistry#shutdown()`.
- Renamed `ThreadPools#shutdownThreadPools()` to `#shutdown()`.
- Replaced `HealthCheck`'s abstract `name` method with a required constructor parameter.
- `HealthCheck#check()` is now `protected`.
- Moved `DeadlockHealthCheck` from `com.yammer.metrics.core` to `com.yammer.metrics.utils`.
- Added `HealthCheckRegistry#unregister(HealthCheck)`.
- Fixed typo in `VirtualMachineMetrics` and `MetricsServlet`: `commited` to `committed`.
- Changed `MetricsRegistry#createName` to `protected`.
- All metric types are created exclusively through `MetricsRegistry` now.
- `Metrics.newJmxGauge` and `MetricsRegistry.newJmxGauge` are deprecated.
- Fixed heap metrics in `VirtualMachineMetrics`.
- Added `Snapshot`, which calculates quantiles.
- Renamed `Percentiled` to `Sampling` and dropped `percentile` and `percentiles` in favor of producing `Snapshot` instances. This affects both `Histogram` and `Timer`.

- Renamed `Summarized` to `Summarizable`.
- Changed order of `CsvReporter`'s construction parameters.
- Renamed `VirtualMachineMetrics.GarbageCollector` to `VirtualMachineMetrics.GarbageCollectorStats`.
- Moved Guice/Servlet support from `metrics-servlet` to `metrics-guice`.
- Removed `metrics-aop`.
- Removed `newJmxGauge` from both `Metrics` and `MetricsRegistry`. Just use `JmxGauge`.
- Moved `JmxGauge` to `com.yammer.metrics.util`.
- Moved `MetricPredicate` to `com.yammer.metrics.core`.
- Moved `NameThreadFactory` into `ThreadPools` and made `ThreadPools` package-visible.
- Removed `Timer#values()`, `Histogram#values()`, and `Sample#values()`. Use `getSnapshot()` instead.
- Removed `Timer#dump(File)` and `Histogram#dump(File)`, and `Sample#dump(File)`. Use `Snapshot#dump(File)` instead.

3.2.20 v2.0.0-BETA18: Dec 16 2011

- Added `DeathRattleExceptionHandler`.
- Fixed NPE in `VirtualMachineMetrics`.
- Added decorators for connectors and thread pools in `metrics-jetty`.
- Added `TimerMetric#time()` and `TimerContext`.
- Added a shorter factory method for millisecond/second timers.
- Switched tests to JUnit.
- Improved logging in `GangliaReporter`.
- Improved random number generation for `UniformSample`.
- Added `metrics-httpclient` for instrumenting Apache HttpClient 4.1.
- Massively overhauled the reporting code.
- Added support for instrumented, non-`public` methods in `metrics-guice`.
- Added `@ExceptionMetered` to `metrics-guice`.
- Added group prefixes to `GangliaReporter`.
- Added `CvsReporter`, which outputs metric values to `.csv` files.
- Improved metric name sanitization in `GangliaReporter`.
- Added `Metrics.shutdown()` and improved metrics lifecycle behavior.
- Added `metrics-web`.
- Upgraded to ehcache 2.5.0.
- Many, many refactorings.
- `metrics-servlet` now responds with `501 Not Implememented` when no health checks have been registered.
- Many internal refactorings for testability.
- Added histogram counts to `metrics-servlet`.

- Fixed a race condition in `ExponentiallyDecayingSample`.
- Added timezone and locale support to `ConsoleReporter`.
- Added `metrics-aop` for Guiceless support of method annotations.
- Added `metrics-jdbi` which adds instrumentation to JDBI.
- Fixed NPE for metrics which belong to classes in the default package.
- Now deploying artifacts to Maven Central.

3.2.21 v2.0.0-BETA17: Oct 07 2011

- Added an option message to successful health check results.
- Fixed locale issues in `GraphiteReporter`.
- Added `GangliaReporter`.
- Added per-HTTP method timers to `InstrumentedHandler` in `metrics-jetty`.
- Fixed a thread pool leak for meters.
- Added `#dump(File)` to `HistogramMetric` and `TimerMetric`.
- Upgraded to Jackson 1.9.x.
- Upgraded to slf4j 1.6.2.
- Upgraded to logback 0.9.30.
- Upgraded to ehcache 2.4.5.
- Surfaced `Metrics.removeMetric()`.

3.2.22 v2.0.0-BETA16: Aug 23 2011

- Fixed a bug in GC monitoring.

3.2.23 v2.0.0-BETA15: Aug 15 2011

- Fixed dependency scopes for `metrics-jetty`.
- Added time and VM version to `vm` output of `MetricsServlet`.
- Dropped `com.sun.mangement`-based GC instrumentation in favor of a `java.lang.management`-based one. `getLastGcInfo` has a nasty native memory leak in it, plus it often returned incorrect data.
- Upgraded to Jackson 1.8.5.
- Upgraded to Jetty 7.4.5.
- Added sanitization for metric names in `GraphiteReporter`.
- Extracted out a `Clock` interface for timers for non-wall-clock timing.
- Extracted out most of the remaining statics into `MetricsRegistry` and `HealthCheckRegistry`.
- Added an init parameter to `MetricsServlet` for disabling the `jvm` section.
- Added a Guice module for `MetricsServlet`.
- Added dynamic metric names.

- Upgraded to ehcache 2.4.5.
- Upgraded to logback 0.9.29.
- Allowed for the removal of metrics.
- Added the ability to filter metrics exposed by a reporter to those which match a given predicate.

3.2.24 v2.0.0-BETA14: Jul 05 2011

- Moved to Maven for a build system and extracted the Scala façade to a `metrics-scala` module which is now the only cross-built module. All other modules dropped the Scala version suffix in their `artifactId`.
- Fixed non-heap metric name in `GraphiteReporter`.
- Fixed stability error in `GraphiteReporter` when dealing with unavailable servers.
- Fixed error with anonymous, instrumented classes.
- Fixed error in `MetricsServlet` when a gauge throws an exception.
- Fixed error with bogus GC run times.
- Link to the pretty JSON output from the `MetricsServlet` menu page.
- Fixed potential race condition in histograms' variance calculations.
- Fixed memory pool reporting for the G1 collector.

3.2.25 v2.0.0-BETA13: May 13 2011

- Fixed a bug in the initial startup phase of the `JmxReporter`.
- Added `metrics-ehcache`, for the instrumentation of `Ehcache` instances.
- Fixed a typo in `metrics-jetty`'s `InstrumentedHandler`.
- Added name prefixes to `GraphiteReporter`.
- Added JVM metrics reporting to `GraphiteReporter`.
- Actually fixed `MetricsServlet`'s links when the servlet has a non-root context path.
- Now cross-building for Scala 2.9.0.
- Added `pretty` query parameter for `MetricsServlet` to format the JSON object for human consumption.
- Added `no-cache` headers to the `MetricsServlet` responses.

3.2.26 v2.0.0-BETA12: May 09 2011

- Upgraded to Jackson 1.7.6.
- Added a new instrumented Log4J appender.
- Added a new instrumented Logback appender. Thanks to Bruce Mitchener (@waywardmonkeys) for the patch.
- Added a new reporter for the Graphite aggregation system. Thanks to Mahesh Tiyyagura (@tmahesh) for the patch.
- Added scoped metric names.
- Added Scala 2.9.0.RC{2,3,4} as build targets.

- Added meters to Jetty handler for the percent of responses which have `4xx` or `5xx` status codes.
- Changed the Servlet API to be a `provided` dependency. Thanks to Mårten Gustafson (@chids) for the patch.
- Separated project into modules:
  - `metrics-core`: A dependency-less project with all the core metrics.
  - `metrics-graphite`: A reporter for the [Graphite](http://graphite.wikidot.com) aggregation system.
  - `metrics-guice`: Guice AOP support.
  - `metrics-jetty`: An instrumented Jetty handler.
  - `metrics-log4j`: An instrumented Log4J appender.
  - `metrics-logback`: An instrumented Logback appender.
  - `metrics-servlet`: The Metrics servlet with context listener.

3.2.27 v2.0.0-BETA11: Apr 27 2011

- Added thread state and deadlock detection metrics.
- Fix `VirtualMachineMetrics`' initialization.
- Context path fixes for the servlet.
- Added the `@Gauge` annotation.
- Big reworking of the exponentially-weighted moving average code for meters. Thanks to JD Maturen (@sku) and John Ewart (@johnewart) for pointing this out.
- Upgraded to Guice 3.0.
- Upgraded to Jackson 1.7.5.
- Upgraded to Jetty 7.4.0.
- Big rewrite of the servlet's thread dump code.
- Fixed race condition in `ExponentiallyDecayingSample`. Thanks to Martin Traverso (@martint) for the patch.
- Lots of spelling fixes in Javadocs. Thanks to Bruce Mitchener (@waywardmonkeys) for the patch.
- Added Scala 2.9.0.RC1 as a build target. Thanks to Bruce Mitchener (@waywardmonkeys) for the patch.
- Patched a hilarious memory leak in `java.util.logging`.

3.2.28 v2.0.0-BETA10: Mar 25 2011

- Added Guice AOP annotations: `@Timed` and `@Metered`.
- Added `HealthCheck#name()`.
- Added `Metrics.newJmxGauge()`.
- Moved health checks into `HealthChecks`.
- Upgraded to Jackson 1.7.3 and Jetty 7.3.1.

3.2.29 v2.0.0-BETA9: Mar 14 2011

- Fixed `JmxReporter` lag.
- Added default arguments to timers and meters.
- Added default landing page to the servlet.
- Improved the performance of `ExponentiallyDecayingSample`.
- Fixed an integer overflow bug in `UniformSample`.
- Added linear scaling to `ExponentiallyDecayingSample`.

3.2.30 v2.0.0-BETA8: Mar 01 2011

- Added histograms.
- Added biased sampling for timers.
- Added dumping of timer/histogram samples via the servlet.
- Added dependency on `jackon-mapper`.
- Added classname filtering for the servlet.
- Added URI configuration for the servlet.

3.2.31 v2.0.0-BETA7: Jan 12 2011

- Added `JettyHandler`.
- Made the `Servlet` dependency optional.

3.2.32 v2.0.0-BETA6: Jan 12 2011

- Fix `JmxReporter` initialization.

3.2.33 v2.0.0-BETA5: Jan 11 2011

- Dropped `Counter#++` and `Counter#--`.
- Added `Timer#update`.
- Upgraded to Jackson 1.7.0.
- Made JMX reporting implicit.
- Added health checks.

3.2.34 v2.0.0-BETA3: Dec 23 2010

- Fixed thread names and some docs.

3.2.35 v2.0.0-BETA2: Dec 22 2010

- Fixed a memory leak in `MeterMetric`.

3.2.36 v2.0.0-BETA1: Dec 22 2010

- Total rewrite in Java.

3.2.37 v1.0.7: Sep 21 2010

- Added `median` to `Timer`.
- Added `p95` to `Timer` (95th percentile).
- Added `p98` to `Timer` (98th percentile).
- Added `p99` to `Timer` (99th percentile).

3.2.38 v1.0.6: Jul 15 2010

- Now compiled exclusively for 2.8.0 final.

3.2.39 v1.0.5: Jun 01 2010

- Documentation fix.
- Added `TimedToggle`, which may or may not be useful at all.
- Now cross-building for RC2 and RC3.

3.2.40 v1.0.4: Apr 27 2010

- Blank `Timer` instances (i.e., those which have recorded no timings yet) no longer explode when asked for metrics for that which does not yet exist.
- Nested classes, companion objects, and singletons don't have trailing `$` characters messing up JMX's good looks.

3.2.41 v1.0.3: Apr 16 2010

- Fixed some issues with the implicit.ly plumbing.
- Tweaked the sample size for `Timer`, giving it 99.9% confidence level with a %5 margin of error (for a normally distributed variable, which it almost certainly isn't.)
- `Sample#iterator` returns only the recorded data, not a bunch of zeros.
- Moved units of `Timer`, `Meter`, and `LoadMeter` to their own attributes, which allows for easy export of Metrics data via JMX to things like Ganglia or whatever.

3.2.42 v1.0.2: Mar 08 2010

- `Timer` now uses Welford's algorithm for calculating running variance, which means no more hilariously wrong standard deviations (e.g., `NaN`).
- `Timer` now supports `+=(Long)` for pre-recorded, nanosecond-precision timings.

3.2.43 v1.0.1: Mar 05 2010

- changed `Sample` to use an `AtomicReferenceArray`

3.2.44 v1.0.0: Feb 27 2010

- Initial release