MoSKito Concepts

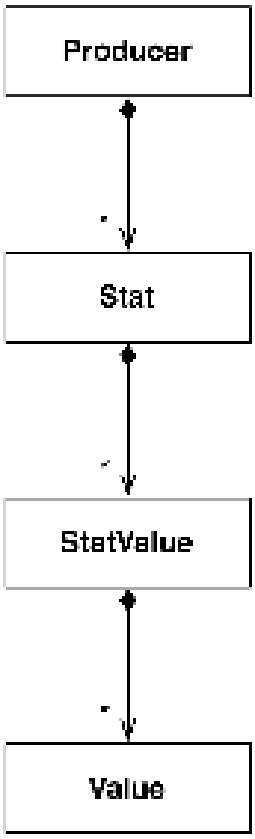
After reading this section, you will know...

- the structure of performance data, collected by MoSKito;
- essential tools inside MoSKito.

In this section:

Producers, Stats and Values

At the core of MoSKito are **Producers, Stats** and **Values.** The dependency is pretty straight-forward, each one owns (contains, aggregates) the following.



Producers

'produce' statistics, aka **stats**. Stats can be everything, and therefore producers can be everything.

Producer is a piece of code that gets connected to a source of statistics (this may be a class, method, service, servlet or **anything** that generates statistic data) and makes a count every time the source gets active.

Producers may count data, related to:

- **performance** (threads, memory, caches, storages, services, etc.) or
- **business** (registrations, payments, conversion, partner performance, online Users, etc.).

FOR EXAMPLE:

All of things below may be producers:

- a business *Service*,
- an *HttpFilter* or *HttpServlet*,
- an *Action* or *Controller* in a MVC WebApp,
- a Jersey *Resource*,
- Mail *Gateway*,
- the *interface* to an external payment (or whatever) provider.

Producers are the source of all information in MoSKito. Accumulators and thresholds are based on the values, reported by a producer.

Stats

represent statistics of a single use case, as well as the cumulated statistics for the whole producer.

FOR EXAMPLE:

If we monitor URLs with an *HttpFilter,* than the associated **Producer** would hold a **Stats** object for:

- each URL it has ever saw,
- the cumulated statistics of all URLs.

If we monitor a service, than each Service method would have a corresponding **Stats** object, and one **Stats** object would represent the service as a whole.

StatValues

Statistics consists of values that depend on the appropriate **Stats** object.

FOR EXAMPLE:

ServiceStats holds the number of requests, total, min, max, last and average request duration, number of concurrent requests and so on.
CacheStats holds the number of requests, hits and the hit ratio.
MemoryStats holds info on free, used and available memory.
CounterStats contain ... Count.

For more details, refer to Built-in Stats Implementation Reference.

Values

Finally, each **StatValue** is a multidimensional counter existing in context of some intervals. Each interval is a dimension in which own measurement takes place.

You can think of StatValue as a HashMap of **Values**, with Intervals being keys. Furthermore, each **Value** actually contains (at least) two Numbers, one for the last complete interval and one for the currently running interval.

Thresholds

Within your application, some producers might be more important for you than the others. To keep a watchful eye on them, we created **thresholds**.

Thresholds continuously monitor a single producer and give a signal when its performance changes.

A thresholds marks the performance boundaries for a producer. When a producer goes beyond (above or below) these boundaries, threshold changes status, thus letting you know this producer needs attention.

In simple words, thresholds are "border guards": they give signal when a producer violates the boundaries you've set for it.

Thresholds can also be compared with traffic lights. They let you know the degree of such "boundary violation" by using different colors. Yellow would mean minor violation, orange show greater problem and so on. The scale ends up on purple, which would stand for total fallout.

Thresholds let you face the possible problem before it affects your entire app.

FOR EXAMPLE

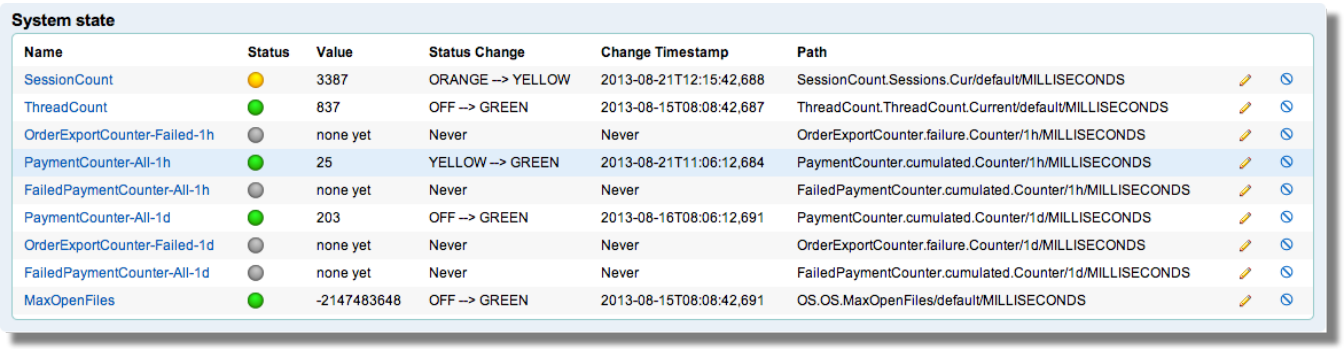
*Let us say we're watching **the session count** of our web application.*

So what we do is: we create a threshold on the basis of **SessionCount** producer and say:

"Ok, threshold, if the load is from 0 to **500** sessions, turn **green**.
If sessions are from **500** to **700**, turn **yellow**.
From **700** to **800**, turn **orange**.
Over **900**, turn **red**".

So, a quick glance on a threshold is enough to know if things go well or turn messy.

A threshold is set to monitor one parameter; a set of thresholds gives a complete picture of how your app performs at the moment.



System state

Name	Status	Value	Status Change	Change Timestamp	Path
SessionCount		3387	ORANGE --> YELLOW	2013-08-21T12:15:42,688	SessionCount.Sessions.Cur/default/MILLISECONDS
ThreadCount		837	OFF --> GREEN	2013-08-15T08:08:42,687	ThreadCount.ThreadCount.Current/default/MILLISECONDS
OrderExportCounter-Failed-1h		none yet	Never	Never	OrderExportCounter.failure.Counter/1h/MILLISECONDS
PaymentCounter-All-1h		25	YELLOW --> GREEN	2013-08-21T11:06:12,684	PaymentCounter.cumulated.Counter/1h/MILLISECONDS
FailedPaymentCounter-All-1h		none yet	Never	Never	FailedPaymentCounter.cumulated.Counter/1h/MILLISECONDS
PaymentCounter-All-1d		203	OFF --> GREEN	2013-08-16T08:06:12,691	PaymentCounter.cumulated.Counter/1d/MILLISECONDS
OrderExportCounter-Failed-1d		none yet	Never	Never	OrderExportCounter.failure.Counter/1d/MILLISECONDS
FailedPaymentCounter-All-1d		none yet	Never	Never	FailedPaymentCounter.cumulated.Counter/1d/MILLISECONDS
MaxOpenFiles		-2147483648	OFF --> GREEN	2013-08-15T08:08:42,691	OS.OS.MaxOpenFiles/default/MILLISECONDS

Logging and notification options are also available for thresholds.

To know more, refer to Thresholds section of MoSKito-Inspect User Guide

Accumulators

As we learned previously, each StatValue holds multiple values for multiple intervals, but only one value at a time (a value for an interval).

For example

If we work with the **SessionCount** *producer*, it will have only its **Sessions** *Stat,* with the **Current** *StatValue.* This *StatValue* will have a separate value for every monitoring interval, like **25** (sessions) for *1 minute* interval, and **35** (sessions) for *1 hour* interval. Anyway, it will have only and exactly *one value* for each configured interval.

Saving your app's memory

The main reason for storing only one value is **saving memory**, since it is where MoSKito-Essential is holding all the values. If we added 200 value-long history to each value (meaning: saving not only the last value, but 200 previous values, which would mean 200 minutes of history PLUS the current 1 minute interval), the memory footprint of MoSKito would explode.

> The MoSKito architecture differentiates between stats collected in the VM where they 'happen', and stats that are collected elsewhere. Now, the stats that are collected in the VM usually have multiple values, one for currently measured value, and one for each of the intervals. This amount can grow easily to huge numbers in a mature system. For example one of the installation runs with 1056 producers with 27067 statvalues and 216536 value holders. Each of those value holders holds one value for each interval and one *current* value. This makes 9 values in a default installation and almost 2 millions of values stored in the system.
>
> People often complain about not having the timeline for values in the MoSKito WebUI, the embedded UI. The above calculation makes clear that adding even 10 past values to the history would bloat the number easily to 20.000.000. And with 10 past values you can have a history of 50 minutes for the 5 minute interval, which is virtually nothing. To have at least week history would mean to have 2016 values for each 5m interval values - I think you get the point now.

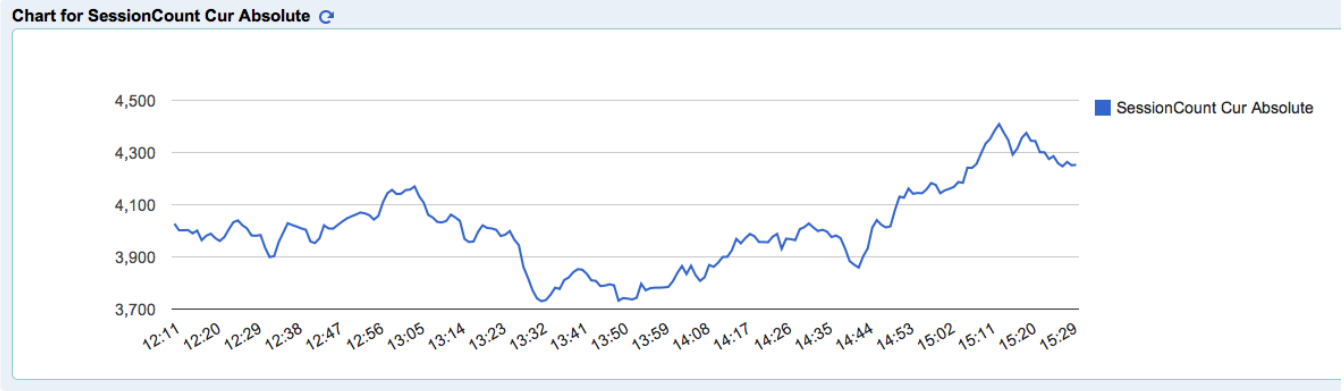
Instead of saving performance data directly in memory, you may connect MoSKito-Central and gather the data there, off-memory.

However, from time to time, there is data which one might want to analyse instantly and get its visual representation.The typical case is seeing how the *session count* is developing over time, with a visual representation of site usage. This is where **accumulators** come into play.

> An accumulator is a value collector (Tieable) for a given **interval**, **producer**, **stat** and **value**.

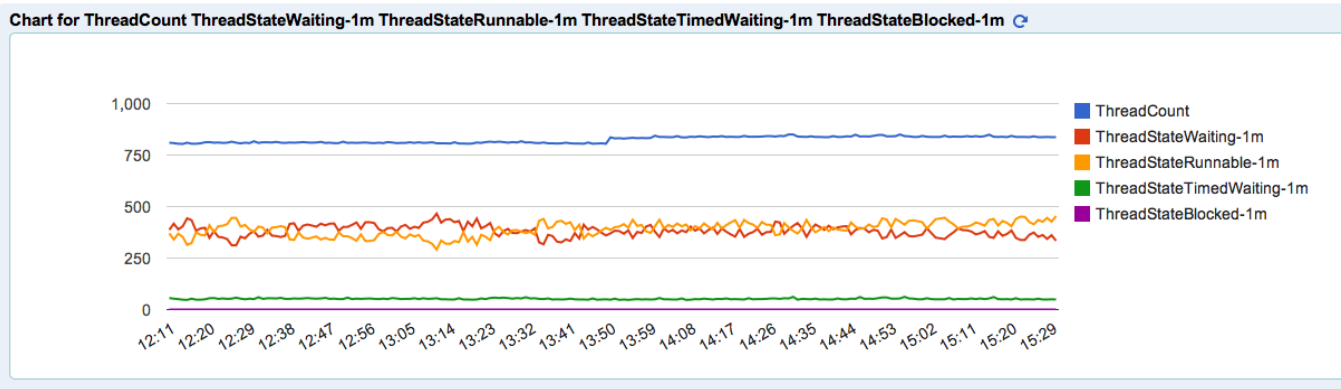
Basically, the accumulator attaches itself to the *Producer/Stat/StatValue/Interval* combination and accumulates (gathers and stores) the data, collected by this *Producer*. This data is then available for instant review, for example, in the form of a *line chart*.

Screenshot: SessionCount over the last two days



But **accumulators** can do more. First of all, they can be combined to give a greater picture of interconnection between different values in the Application:

Screenshots: Threads in different States



Accumulators also build the base for graphs in MoSKito UI (link) and MoSKito-Control.

To know more about adding and using accumulators, please refer to Accumulators section of MoSKito-WebUI manual.

Journeys

Journeys allow recording user actions in the form of actual calls/steps that occur inside a web application. Journeys show the methods /functions/services/resources a user triggers while using the app.

In simple words, a journey is a translation of any user action into a developer's or app admin's language.

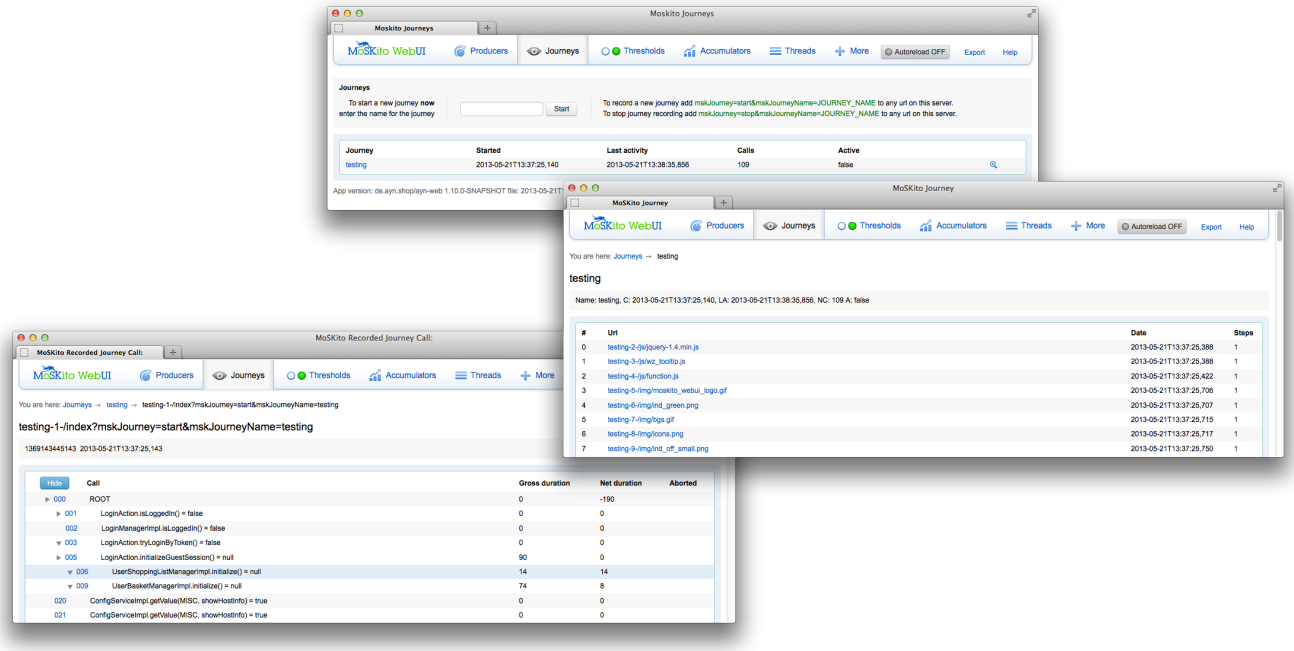
For example

When a user clicks a link to load a new page, inside the web app this click initiates a dozen of calls to system functions. The full step-by-step list of these operations and calls, with their timestamps, is what you get while recording a journey.

Journeys, being a part of MoSKito, are tied to producers. Thus, a line of recording is added to a journey when a system call passes through an existing MoSKito producer.

Journeys are helpful in many situations. The example below is the most typical.

Let us suppose a certain user action is taking too much time for the system to perform. Such a delay might mean that one of the calls to system methods is taking extra time.
As soon as we record this user action as a journey, we see all system steps/calls as they are initiated (with their timestamps), and easily identify those taking most of the time for running. This info is a roadmap for further improvement and optimisation.



Journeys also have the options for *RealTime* and *Life UserJourney* Analysis.

To know more, please refer to Journeys section of MoSKito-WebUI manual.

Go to Previous:

MoSKito-Essential Overview

Read Next:

Start Working, Step by Step