Extending Clessidra

Technical Document

Author: Reuben Abela

Date: 26th August 2013

Version 1.0

Table of Contents

[1 Introduction 1](#_Toc365296395)

[2 Built-in strategies 1](#_Toc365296396)

[3 Creating a new strategy 2](#_Toc365296397)

[3.1 Chaining etiquette 4](#_Toc365296398)

[3.2 Synchronization 5](#_Toc365296399)

[4 Customizing existing strategies 5](#_Toc365296400)

[4.1 Strategy grouping 5](#_Toc365296401)

[4.2 Messages 6](#_Toc365296402)

# Introduction

This document will outline how to extend Clessidra in order to create new strategies, or to some extent change the behaviour of existing strategies. Basic knowledge of the library will be assumed. This presentation can help you familiarise yourself with Clessidra.

[sscm:// mtsrv-codrep.malta.ccbill-hq.local:4900/Clessidra/Clessidra/RateLimiter/doc/presentation/Rate Limiter Presentation.pptx](sscm://%20malta-seapine.ccbill.com:4900/Clessidra/Clessidra/RateLimiter/doc/presentation/Rate%20Limiter%20Presentation.pptx)

[sscm:// malta-seapine.ccbill.com:4900/Clessidra/Clessidra/RateLimiter/doc/presentation/Rate Limiter Presentation.pptx](sscm://%20malta-seapine.ccbill.com:4900/Clessidra/Clessidra/RateLimiter/doc/presentation/Rate%20Limiter%20Presentation.pptx)

# Built-in strategies

Before deciding to extend Clessidra be sure that there is the need to do so. The library provides 3 built-in strategies.

* Invocation Rate Limiter – limits access based on how many times it was accessed in given amount of time.
* Concurrency Limiter – limits access based on concurrency with other running methods.
* Cost Based Limiter – limits access based on a pre-calculated cost on the request.

The implementation of these strategies can be found here.

com.ccbill.clessidra.strategy.BaseInvocationRateLimiterStrategy

com.ccbill.clessidra.strategy.BaseConcurrencyLimiterStrategy

com.ccbill.clessidra.strategy.BaseCostBasedLimiterStrategy

As highlighted in the presentation Clessidra supports differentiating the limit based on data related to the service adopting it. For this reason the logic behind the strategy is detached from the grouping of limits.

The Base Strategies are not usable in themselves, because they are abstract classes. The purpose of these Base Strategies is simply to hold the logic of their respective strategy. In the library, extending these 3 Base Strategies, there are 3 Default Strategies. These can be used for rate limiting in your spring beans configuration, but since they are not aware of any service data they will only have 1 common strategy group each.

com.ccbill.clessidra.strategy.defaults.DefaultInvocationRateLimiterStrategy

com.ccbill.clessidra.strategy.defaults.DefaultConcurrencyLimiterStrategy

com.ccbill.clessidra.strategy.defaults.DefaultCostBasedLimiterStrategy

# Creating a new strategy

The interface that defines a strategy can be found in:

com.ccbill.clessidra.interfaces.LimiterStrategy

For details about each method refer to the javadoc in the source code.

In the same package there are 2 other interfaces:

com.ccbill.clessidra.interfaces.CostBasedLimiterStrategy  
com.ccbill.clessidra.interfaces.RequiresPostInvocationCleanup

The CostBasedLimiterStrategy extends LimiterStrategy to provide a calculateCost() method.

RequiresPostInvocationCleanup is a marker interface, any strategy marked with this interface will have its rollback method called after invoking the @RateLimited method.

Many of the methods in the LimiterStrategy interface are implemented in

com.ccbill.clessidra.strategy.AbstractLimiterStrategy

To create a new strategy extend the AbstractLimiterStrategy and implement the following methods:

**public** LimiterStrategyConclusion hasLimitBeenExceededChain(String methodGroup, UUID invocationUUID, Object[] args);

**public** **void** rollback(String methodGroup, UUID invocationUUID, Object[] args);

**public** String getStrategyGroupKey(Object[] args);

Optionally you can also implement

**public** String getGenericExceededMessage(String methodGroup, UUID invocationUUID, Object[] args);

**public** String getDetailedExceededMessage(String methodGroup, UUID invocationUUID, Object[] args);

It is not necessary to follow the pattern of the inbuilt strategies and implement getStrategyGroupKey in a separate class.

## Chaining etiquette

A strategy can either be used as standalone or as a part of a chain. However it is its responsibility to call the next chained strategy. A well formed hasLimitBeenExceededChain() method would look like as shown below. This will ensure that each strategy is calling the next one and any decision taken, whether taken from the strategy itself or coming from a strategy down the chain, is passed to the previous strategy until it ends up in the rate limiter aspect.

**public** LimiterStrategyConclusion hasLimitBeenExceededChain(String methodGroup, UUID invocationUUID, Object[] args) {

**boolean** callNextInChain = **false**;

*read from history*

*boolean limitExceeded = decide if limit has been exceeded*

*if (!limitExceeded) {*

*callNextInChain = true*

*update history*

*}*

**if** (callNextInChain) {

**return** callNextChainedLimiterStrategy(methodGroup, invocationUUID, args);

} **else** {

**return** buildExceededConclusion(**this**, methodGroup, invocationUUID, args);

}

}

## Synchronization

Special attention has to be given to synchronization, although the library itself is completely flat, the adopting system might be running in a multi-threaded environment.

# Customizing existing strategies

## Strategy grouping

In the same way the 3 Default Strategies are deciding to put every request in the same group, you can group your requests as required.

**public** **class** DefaultInvocationRateLimiterStrategy **extends** BaseInvocationRateLimiterStrategy {

**public** DefaultInvocationRateLimiterStrategy(**int** numberOfInvocationsLimit, **int** perTimePeriodSeconds) {

**super**(numberOfInvocationsLimit, perTimePeriodSeconds);

}

**public** String getStrategyGroupKey(Object[] args) {

**return** "";

}

}

The getStrategyGroupKey method has Object[] as a parameter, this parameter will hold all the parameters being passed to the annotated method. This enables you to base your grouping on data being passed as a parameter. Alternatively you can autowire a relevant context from which to extract relevant data.

The following is an example:

@Autowired

**private** UserContextProvider userContextProvider;

**public** String getStrategyGroupKey(Object[] args) {

Long userOrgId = userContextProvider.getUserContextTO().getUserOrganizationId();

**if** (userOrgId != **null**) {

**return** userOrgId.toString();

}

**return** "";

}

## Messages

The messages returned in the LimiterStrategyConclusion are also customizable.

@Override

**public** String getDetailedExceededMessage(String methodGroup, UUID invocationUUID, Object[] args) {

**return** "Detailed message";

}

@Override

**public** String getGenericExceededMessage(String methodGroup, UUID invocationUUID, Object[] args) {

**return** "Generic message";

}