Throttling-библиотека своими руками

# Задачи

Обычно одной из главных задач в разработке приложений является максимально возможная производительность. Однако, есть случаи, когда целесообразно искусственно ограничить производительность.

На практике мне приходилось сталкиваться с несколько подобными задачами, опишу три наиболее характерных.

* В чате необходимо ограничить кол-во посылаемых пользователем в единицу времени. сообщений. Предположим, не более одного сообщения в секунду и при этом не более 100 сообщений в час.
* Внешний сервис возвращает историю обслуживания по VIN автомобиля. Кол-во запросов ограничено, скажем, 1000 запросов в час от одного клиента независимо от запрашиваемого VINа. Таким образом, 1000 запросов по одному и тому же VINу исчерпывают часовой лимит и следующий запрос по другому вину будет отклонен и придется ждать час.до следующего запроса.
* В системе учета тары по RFID меткам возле ворот могут скопится несколько меток, которые персонал фабрики забыл и не собирается оперативно убирать. Тем не менее, эти метки считываются наравне с полезными и засоряют эфир. Таким образом, хотелось бы ограничить кол-во считываний таких меток, предположим, 20000 в час.

Во всех случаях ограничивается не производительность системы как таковая, а производительность системы в расчете на пользователя / автомобиль / метку. Таким образом, ресурсы системы распределяются более равномерно для всех пользователей / автомобилей / меток.

# Существующие решения

В случае возникновения проблемы первым делом стоит поискать готовые решения. Для .Net обнаружилось только одно популярное решение [WebApiThrottle](https://github.com/stefanprodan/WebApiThrottle). На первый взгляд, достаточно популярная и законченная, однако, как показала практика, недостаточно гибкая библиотека.

То есть все стандартные задачи типа black / white списков, ограничения кол-ва запросов в единицу времени по клиентскому IP, контроллеру она решает, но как только нужно ограничивать по параметру, который станет известен, скажем, в середине обработки запроса (тот же производитель и дилер VINа невозможно определить на входе в метод контролллера), то становится очевидно, что WebApiThrottle нам не помощник.

А значит, нужно разрабатывать свое решение, которое бы решило не только все свои задачи, но и было бы достаточно универсальным для использования другими разработчиками.

# Базовые требования к библиотеке

Адаптируемость библиотеки в разных условиях определяется гибкостью и расширяемостью архитектуры. Для этого нужно иметь возможность

* переопределить любую часть системы на любом уровне абстракции
* использовать существующий код по максимуму

С другой стороны, хотелось бы максимальной простоты использования и наличия реализаций под наиболее популярные и тривиальные проблемы.

Поэтому хотелось бы

* иметь возможность конфигурации как кодом в runtime, так и через конфигурационный файл.
* простейшие логические правила “и”/”или”.
* реализацию как в виде Handler и Filter для Asp.Net, так и Middleware для Owin.
* хотя бы в простейшем виде логировать работу

# Расширенные требования к библиотеке

Как известно, аппетит приходит во время еды.

Кто сказал, что измерять можно только кол-во запросов?

Ведь метрик у запроса гораздо больше. Например, размер запроса/ответа, количество абсолютного или процессорного времени, потраченного на обработку, кол-во обращений всех сетевых обращений или кол-во обращений к базе данных и т.д.

Отсюда вытекает требование двухфазности обработки, т.е. чтобы замеры можно было производить и до выполнения запроса, и после (при этом никто не запрещает вручную правила в середине выполнения, как в случае с VIN). Например, необходимо ограничивать размер ответа, который можно измерить лишь после его, ответа, формирования и уже в зависимости от индивидуальных предпочтений либо заблокировать выдачу ответа, либо ответить (ведь ресурсы уже потрачены и ответ сформирован, а на этом этапе можно лишь записать статистику и вернуть ответ клиенту).

Кто сказал, что решение о запросе должно быть бинарным: пропустить или заблокировать? Возможно, имеет смысл отвечать вещественным числом от 0(запрос заблокирован) до 1(запрос зазрешен), при этом промежуточные значения будут означать приоритет обработки для следующего уровня либо для самого клиента. Таким образом, запрос с результатом 0.9 будет более приоритетным, чем запрос с результатом 0.3.

Кто сказал, что хранение состояния должно быть локальным? Ведь это работает только на одном узле либо с некоторыми оговорками на кластере с закреплёнными за узлами сессиями. Почему бы не использовать общее сетевое хранилище, скажем, Redis? Более того, на Redis нужно будет осуществлять манипуляцию статистикой с помощью “придворного” скрипта Lua. А раз мы используем удалённое хранилище, конечно же, необходимо поддерживать async/await интерфейс.

# Примеры использования

Чтобы не утомлять читателя рассмотрением внутренней архитектуры библиотеки (вероятно, она станет темой отдельной статьи), перейдём к рассмотрению примеров использования.

Итак, нужно ограничить доступ к сайту 2 вызовами в 10 секунд на каждый отдельный клиентский IP.

Рассмотрим Asp.Net Web Api сайт. Сконфигурируем throttling кодом (полный исходный код примера можно найти [ЗДЕСЬ]).

Public static void Register(HttpConfiguration config)

{

var storage = new LocalMemoryStorage();

var keyer = new IPKeyer();

var processor = new RequestCountPerPeriodProcessorPhased { Count = 2, Period = TimeSpan.FromSeconds(10) };

var rule = new StorageKeyerProcessorRule<IaspArgs, PassBlockVerdict> { Storage = storage, Keyer = keyer, Processor = processor } as Irule;

var throttlingHandler = new ThrottlingHandler(rule);

config.MessageHandlers.Add(throttlingHandler);

config.Routes.MapHttpRoute(

name: “DefaultApi”,

routeTemplate: “api/{controller}/{id}”,

defaults: new { id = RouteParameter.Optional }

);

}

Первым делом, создаётся хранилище для статистики (в данном случае, локальное).

Следующим шагом создаём keyer – функтор, определяющий ячейку, относительно которой вычисляется статистика.

Затем, создаём процессор, определяющий способ принятия решения. В данном случае мы воспользуемся самым популярным, вычисляющим количество запросов в единицу времени.

После создаём правило (класс, реализующий базовый интерфейс системы – IRule). Используем наиболее популярное правило, объединяющее хранилище, хэш-функтор и процессор.

Далее, создаём ThrottlingHandler (потомок System.Net.Http.DelegatingHandler) и добавляем его в Web Api конвеер.

Готово, теперь каждому IP разрешено делать не более 2 вызовов каждые 10 секунд.

Конфигурация кодом – это, конечно, полезно, но гибче это делать с помощью конфигурационного файла.

Рассмотрим для разнообразия решение той же самой задачи для Owin Self Hosted приложения с использованием .Net конфигурации.

<configSections>

<section name="ipThrottling" type="Dxw.Throttling.Owin.Configuration.PassBlockConfigurationSectionHandler, Dxw.Throttling.Owin"/>

</configSections>

<ipThrottling>

<storages>

<storage type="Dxw.Throttling.Core.Storages.LocalMemoryStorage, Dxw.Throttling.Core" name="local" />

</storages>

<rules>

<rule type="Dxw.Throttling.Owin.Rules.OwinStorageKeyerProcessorRule, Dxw.Throttling.Owin">

<storage name="local" />

<keyer type="Dxw.Throttling.Owin.Keyers.IPKeyer, Dxw.Throttling.Owin" />

<processor type="Dxw.Throttling.Core.Processors.RequestCountPerPeriodProcessorPhased, Dxw.Throttling.Core"

count="2" period="00:00:10" />

</rule>

</rules>

</ipThrottling>

Собственно, семантически конфигурация эквивалентна первым четырём строкам кода Asp.Net примера.

Затем используем конфигурацию в коде.

var throttlingConfig = ConfigurationManager.GetSection("ipThrottling")

as Throttling.Core.Configuration.ThrottlingConfiguration<IOwinArgs, PassBlockVerdict>;

var rule = throttlingConfig.Rule;

appBuilder.Use(typeof(ThrottlingPassBlockMiddleware), rule);

Собственно, добиваемся того же результата: теперь наше Owin Self Hosted приложение позволяет не более двух запросов в течении десяти секунд.

Конечно, не всегда уместно и удобно пользоваться DelegateHandler’ом или Middleware для дросселирования.

Во-первых, это может быть неоптимальным решением с точки зрения производительности, если дросселировать нужно только запросы к части ресурсов.

Во-вторых, иногда хочется более выразительного и читабельного кода.

Для этих целей библиотека содержит ThrottleAttribute – неследник ActionFilterAttribute. Рассмотрим пример использования. Пусть нам необходимо ограничить частоту вызова к методам Post и Put некоего контроллера, а остальным методам дросселировать не требуется. Создадим конфигурацию.

# Выводы

Библиотека используется в трёх независимых коммерческих проектах. Во всех случаях понадобилась реализация специализированных хэш-функторов, поскольку ключи оказались характеры лишь для своих предметных областей. Однако остальные части системы (хранилища, процессоры, конфигурация) остались неизменны. Таким образом, несмотря на наличие типичной для дросселирования логики, главным преимуществом библиотеки на практике является приспосабливаемость к самым нетривиальным и каверзным задачам.

# Архитектура

Throttler - это по сути расширенный случай Firewall’а с той лишь разницей, что последний не руководствуется предыдущим состоянием, т.е. пропускает либо блокирует запрос без оглядки на предыдущую историю.

Поэтому логично, что базовой сущностью будет правило (Rule). Соответственно, основным интерфейсом будет