# Tolerance Documentation

***Release 0.1.0***

**sroze**

**Oct 31, 2017**

# Contents

Contents:

CHAPTER 1

# Introduction

Tolerance is a PHP library that provides fault tolerance and microservices related tools in order to be able to solve some of the problems introduced by microservices.

## Why ?

Software fails. Software communicates with other softwares. Software can be distributed or a set of services, which makes it even more subject to faults and monitoring/tracing problems.

Tolerance helps to run fault-tolerant operations, throttle (ie rate limiting) your outgoing or incoming messages, track messages across services and protocols and more.

## Getting started

The recommended way is to use Composer to install the *tolerance/tolerance* package.

```
$ composer require tolerance/tolerance
```

If you are using Symfony, then checkout the Symfony Bundle. Else, you should have a look to the different components.

- Operation runners
- Tracer
- Metrics
- Throttling

## Contributing

Everything is open-source and therefore use the GitHub repository to open an issue or a pull-request.

CHAPTER 2

# Operation runners

This component aims to run atomic tasks (called *operations*) by using different operation runners. They can retry in case of a temporary fault, buffer the operations, fallback them with a default result, rate limit the throughput of operations and more.

## Operations

An operation is an atomic piece of processing. This is for instance an API call to an third-party service, or a process that requires to talk to the database. We can use them for any process that is dependent on a non-trusted resource, starting with the network connection.

### From a callback

The first kind of operation is an operation defined by a PHP callable. This operation can be created with the `Callback` class, like this:

```
use Tolerance\Operation\Callback;

$operation = new Callback(function() use ($client) {
    return $client->get('/foo');
});
```

This class accepts any supported PHP callable, so you can also use object methods. For instance:

```
use Tolerance\Operation\Callback;

$operation = new Callback([$this, 'run']);
```

### Promise Operation

This class accepts any supported PHP callable, which must returns a *Promise*.

For instance:

```
use Tolerance\Operation\PromiseOperation;

$operation = new PromiseOperation(function () use ($nextHandler, $request) {
    return $nextHandler($request);
});
```

The `PromiseOperation` is runned by the `RetryPromiseOperationRunner`.

## *Raw* runners

There's a set of *raw* operation runners that know how to run the default operations:

- The *callback runner* that is able to run callback operations.
- The *chain runner* that is able to chain operation runners that supports different operation types.

### Callback runner

This is the runner that runs the Callback operations.

```
use Tolerance\Operation\Runner\CallbackOperationRunner;

$runner = new CallbackOperationRunner();
$result = $runner->run($operation);
```

### Chain runner

Constructed by other runners, usually the *raw* ones, it uses the first one that supports to run the operation.

```
use Tolerance\Operation\Runner\ChainOperationRunner;
use Tolerance\Operation\Runner\CallbackOperationRunner;

$runner = new ChainOperationRunner([
    new CallbackOperationRunner(),
]);

$result = $runner->run($operation);
```

Also, the `addOperationRunner` method allows you to add another runner on the fly.

## *Behavioural* runners

These operation runners decorates an existing one to add extra *behaviour*:

- The *retry runner* will retry the operation until it is successful or considered as failing too much.
- The *buffered runner* will buffer operations until you decide the run them.
- The *retry promise runner* will provide a new *Promise* to replace a rejected or incorrectly fulfilled one.

**Note:** The Throttling component also come with a Rate Limited Operation Runner

## Retry runner

This runner will retry to run the operation until it is successful or the wait strategy decide to fail.

```php
use Tolerance\Operation\Runner\CallbackOperationRunner;
use Tolerance\Operation\Runner\RetryOperationRunner;
use Tolerance\Waiter\SleepWaiter;
use Tolerance\Waiter\ExponentialBackOff;
use Tolerance\Waiter\CountLimited;

// Creates the strategy used to wait between failing calls
$waitStrategy = new CountLimited(
    new ExponentialBackOff(
        new SleepWaiter(),
        1
    ),
    10
);

// Creates the runner
$runner = new RetryOperationRunner(
    new CallbackOperationRunner(),
    $waitStrategy
);

$result = $runner->run($operation);
```

By default, the retry runner will catch all the exception. If you want to be able to catch only unexpected exceptions or only some, you can inject a `ThrowableCatcherVoter` implementation as the third argument of the `RetryOperationRunner`. For instance, you can catch every exception but Guzzle's `ClientException` ones.

```php
use Tolerance\Operation\ExceptionCatcher\ThrowableCatcherVoter;

$throwableCatcherVoter = new class() implements ThrowableCatcherVoter {
    public function shouldCatchThrowable(\Throwable $t)
    {
        return !$t instanceof ClientException;
    }
};

$runner = new RetryOperationRunner(
    new CallbackOperationRunner(),
    $waitStrategy,
    $throwableCatcherVoter
);
```

## Buffered runner

This runner will buffer all the operations to post-pone their execution.

```php
use Tolerance\Operation\Buffer\InMemoryOperationBuffer;
use Tolerance\Operation\Runner\BufferedOperationRunner;

$buffer = new InMemoryOperationBuffer();
$bufferedRunner = new BufferedOperationRunner($runner, $buffer);

// These 2 operations will be buffered
$bufferedRunner->run($firstOperation);
$bufferedRunner->run($secondOperation);
```

Once you've decided that you want to run all the operations, you need to call the `runBufferedOperations` method.

```php
$results = $bufferedRunner->runBufferedOperations();
```

The `$results` variable will be an array containing the result of each ran operation.

---

**Tip:** The Symfony Bridge automatically run all the buffered operations when the kernel terminates. Checkout the Symfony Bridge documentation

---

## Retry Promise runner

This runner will provide a new *Promise* until it is successful or the wait strategy decide to fail. It supports only the `PromiseOperation`.

```php
use Tolerance\Operation\Runner\RetryPromiseOperationRunner;
use Tolerance\Waiter\SleepWaiter;
use Tolerance\Waiter\ExponentialBackOff;
use Tolerance\Waiter\CountLimited;

// Creates the strategy used to wait between failing calls
$waitStrategy = new CountLimited(
    new ExponentialBackOff(
        new SleepWaiter(),
        1
    ),
    10
);

// Creates the runner
$runner = new RetryPromiseOperationRunner(
    $waitStrategy
);

$promise = $runner->run($operation);
```

By default, the promise retry runner will considered a *Fulfilled* Promise as successful, and will retry any *Rejected* Promise. If you want to be able to define you own catching strategy, you can inject a `ThrowableCatcherVoter` implementation as the second argument for the *Fulfilled* stragegy, and as the third argument for the *Rejected* strategy.

```php
use Tolerance\Operation\Exception\PromiseException;
use Tolerance\Operation\ExceptionCatcher\ThrowableCatcherVoter;

$throwableCatcherVoter = new class() implements ThrowableCatcherVoter {
```

```
    public function shouldCatchThrowable(\Throwable $t)
    {
        return !$throwable instanceof PromiseException
            || $throwable->isRejected()
            || !$throwable->getValue() instanceof Response
            || $throwable->getValue()->getStatusCode() >= 500
        ;
    }
};

$runner = new RetryPromiseOperationRunner(
    $waitStrategy,
    $throwableCatcherVoter
);
```

## Create your own

Provided operation runners might be sufficient in many cases, but you can easily create your own runners by implementing the OperationRunner interface.

CHAPTER 3

# Throttling

The principle of throttling a set of operation is to restrict the maximum number of these operations to run in a given time frame.

For instance, we want to be able to run a maximum of 10 requests per seconds per client. That means that the operations tagged as "coming from the client X" have to be throttled with a rate of 10 requests per seconds. It is important to note that the time frame can also be unknown and you can use your own *ticks* to achieve a rate limitation for concurrent processes for instance.

## Rate

Even if you may not need to extend these main objects of the Throttling, here are the description of the `Rate` and `RateMeasure` objects that are used by the rate limit implementations.

### Rate

The `Rate` interface simply defines a `getTicks()` method that should returns a number. The first implementation is the `TimeRate` that defines a number of operation in a given time range.

```
use Tolerance\Throttling\Rate\TimeRate;

$rate = new TimeRate(60, TimeRate::PER_SECOND)
$rate = new TimeRate(1, TimeRate::PER_MINUTE)
```

The second implementation is the `CounterRate` that simply defines a counter. This is mainly used to store a counter such as in the internals of the Leaky Bucket implementation or when you'll want to setup a rate limitation for parallel running processes for instance.

### Rate measure

The `RateMeasure` is mainly used in the internals to store a given `Rate` at a given time. The only implementation at the moment is the `ImmutableRateMeasure`.

### Storage

What you have to care about is the storage of these rate measures because they need to be stored in order to ensure the coherency or this rate limits, especially when running with concurrent requests.

#### In memory storage

The easiest way to start is to store the rate measures in memory. The major drawback is that in order to ensure your rate limitation you need to have your application running in a single long-running script.

```
use Tolerance\Throttling\RateMeasureStorage\InMemoryStorage;

$storage = new InMemoryStorage();
```

## Waiters

In any loop, you'll probably want to wait between calls somehow, to prevent DDoSing your other services or 3rd party APIs. Tolerance come with 2 default *raw* waiters:

- *SleepWaiter* that simply wait using PHP's *usleep* function
- *NullWaiter* that do not wait and it mainly used for tests

Once you are able to wait an amount of time, you may want to surcharge the waiters to apply different wait strategies such as an exponential back-off.

- The *linear* waiter simply waits a predefined amount of time.
- The *exponential back-off* waiter uses the well-known Exponential backoff algorithm to multiplicatively increase the amount of time of wait time.
- The *count limited* waiter simply adds a limit in the number of times it can be called.
- The *rate limit* waiter will wait the required amount of time to satisfy a rate limit.

**Note:** The Throttling component also come with a Rate Limited Operation Runner

### SleepWaiter

This implementation will use PHP's `sleep` function to actually pause your process for a given amount of time.

```
use Tolerance\Waiter\Waiter\SleepWaiter;

$waiter = new SleepWaiter();

// That will sleep for 500 milliseconds
$waiter->wait(0.5);
```

### NullWaiter

The `NullWaiter` won't actually wait anything. This is usually used for the testing, you should be careful using it in production.

```
use Tolerance\Waiter\Waiter\NullWaiter;

$waiter = new NullWaiter();
```

### Linear

How to simply always wait a predefined amount of time? There's the linear waiter. The following example show how it can be used to have a waiter that will always wait 0.1 seconds.

```
use Tolerance\Waiter\Waiter\SleepWaiter;
use Tolerance\Waiter\Waiter\Linear;

$waiter = new Linear(new SleepWaiter(), 0.1);
```

### Exponential back-off

> In a variety of computer networks, binary exponential backoff or truncated binary exponential backoff refers to an algorithm used to space out repeated retransmissions of the same block of data, often as part of network congestion avoidance.
>
> —Wikipedia

The `ExponentialBackOff` waiter decorates one of the raw waiters to add this additional exponential wait time.

```
use Tolerance\Waiter\Waiter\ExponentialBackOff;
use Tolerance\Waiter\Waiter\SleepWaiter;

// We use an initial collision number of 0
$waiter = new SleepWaiter();
$waitStrategy = new ExponentialBackOff($waiter, 0);

$waitStrategy->wait(); // 0.5s
$waitStrategy->wait(); // 1.5s
$waitStrategy->wait(); // 3.5s

// ...
```

### Count limited

This decoration strategy defines a maximum amount of waits. Once this limit is reached, it will throw the `CountLimitReached` exception.

```
// Wait for a maximum amount of 10 times
$waitingStrategy = new CountLimited($waitingStrategy, 10);
```

### Rate Limit

Using the Rate Limit Waiter, you will just have to call the `wait()` method of the waiter at the end of all your iterations in a loop for instance, to ensure that each the iteration rate will match the rate limit you've defined.

```
use Tolerance\Throttling\Rate\TimeRate;
use Tolerance\Throttling\RateLimit\LeakyBucket;
use Tolerance\Throttling\RateMeasureStorage\InMemoryStorage;
use Tolerance\Throttling\Waiter\RateLimitWaiter;
use Tolerance\Waiter\SleepWaiter;

$rate = new TimeRate(10, TimeRate::PER_SECOND);
$rateLimit = new LeakyBucket(new InMemoryStorage(), $rate);
$waiter = new RateLimitWaiter($rateLimit, new SleepWaiter());

for ($i = 0; $i < 100; $i++) {
    echo microtime(true)."\n";

    $waiter->wait('id');
}
```

The *optional* argument of the `wait` method is the identifier of the operation you want to isolate. That means that you can use the same waiter/rate limit for different type of operations if you want.

### Time Out

This decoration strategy defines a time out to your operation execution. Once this time out is exceeded, it will throw the `TimedOutExceeded` exception.

```
// Time out in 20 seconds
$waitingStrategy = new TimeOut($waitingStrategy, 20);
```

## Strategies

There are many existing algorithms for throttling, you need to choose the one that fits the best your needs. At the moment, only the following algorithm can be found in Tolerance:

- *Leaky bucket*, a *rolling time frame* rate limit

Each implementation implements the `RateLimit` interface that contains the following methods:

- `hasReachedLimit(string $identifier): bool` Returns true if the given identifier reached the limit
- `getTicksBeforeUnderLimit(string $identifier): float` Returns the number of ticks that represents the moment when the rate will be under the limit.
- `tick(string $identifier)` Register a tick on the bucket, meaning that an operation was executed

### Leaky bucket

The leaky bucket algorithm ensure that the number of operations won't exceed a rate on a given **rolling time frame**.

```php
use Tolerance\Throttling\Rate\TimeRate;
use Tolerance\Throttling\RateLimit\LeakyBucket;
use Tolerance\Throttling\RateMeasureStorage\InMemoryStorage;

$rateLimit = new LeakyBucket(
    new InMemoryStorage(),
    new TimeRate(10, TimeRate::PER_SECOND)
);
```

You can have a look to the LeakyBucket unit tests to have a better idea of how you can use it directly.

## Integrations

Once you've chosen your rate limit strategy you can either use it directly or integrates it with some of the existing components of Tolerance.

- *Operation Runner* is an `Operation Runner` that will run your operations based on the rate limit.

### Operation Runner

The Rate Limited Operation Runner is the integration of rate limiting with operation runners. That way you can ensure that all the operations you want to run will actually run at the given time rate.

```php
$rateLimit = /* The implementation you wants */;

$operationRunner = new RateLimitedOperationRunner(
    new SimpleOperationRunner(),
    $rateLimit,
    new SleepWaiter()
);

$operationRunner->run($operation);
```

By default, the identifier given to the rate limit is an empty string. The *optional* fourth parameter is an object implementing the `ThrottlingIdentifierStrategy` interface that will returns the identifier of the operation.

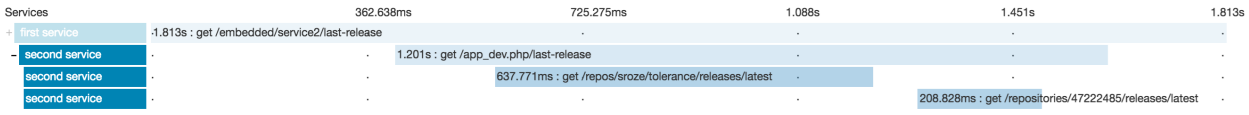
```php
class ThrottlingIdentifierStrategy implements ThrottlingIdentifierStrategy
{
    /**
     * {@inheritdoc}
     */
    public function getOperationIdentifier(Operation $operation)
    {
        if ($operation instanceof MyClientOperation) {
            return sprintf(
                'client-%s',
                $operation->getClient()->getIdentifier()
            );
        }

        return 'unknown-client';
    }
}
```

# CHAPTER 4

# Tracer

The Tracer component's goal is to be able to easily setup a tracing of your application messages across different services.

At the moment, the supported backend is Zipkin. Once configured, you'll be able to preview a given Trace and analyze the time spent by each service.




**Note:** The Symfony Bundle also integrates this component to ease the setup of traces in a Symfony application.

CHAPTER 5

# Metrics

This component contain a set of tools to collect and publish different metrics about the application.

## Collectors

Tolerance have built-in collectors that you can use directly.

- class.CollectionMetricCollector contain a collector of other collectors and will collect the metrics from all of them.
- class.RabbitMqCollector will grab metrics from the RabbitMq management API.

In order to create your own collector, you will just have to implement the interface.MetricCollector interface.

## Publishers

Tolerance have built-in publishers that you can use directly.

- class.CollectionMetricPublisher contain a collector of other publisher and will publish the metrics to all of them.
- class.HostedGraphitePublisher will publish the metrics to HostedGraphite.
- class.BeberleiMetricsAdapterPublisher will publish the metrics using a "collector" from the Beberlei's metrics library.

In order to create your own collector, you will just have to implement the interface.MetricPublisher interface.

## Operation Runners

If you are using Tolerance's operation runners you can decorate them with some additional operation runner that will publish some metrics. It's an easy way to collect metrics from your application with almost no effort.

## Success/Failure

This operation runner will increment a `:.failure` and a `:.success` metric at every run. You can therefore count the number of ran operation as well as their status.

```
use Tolerance\Operation\Runner\Metrics\SuccessFailurePublisherOperationRunner;

$runner = new SuccessFailurePublisherOperationRunner(
    $decoratedRunner,
    $metricPublisher,
    'metric_namespace'
);
```

**Note:** You can also uses Symfony's bridge to create and use this runner without any PHP code.

**Note:** The Symfony Bundle integration also uses this component to provide metrics.

# CHAPTER 6

# Symfony Bundle

The Tolerance library comes with a Symfony Bundle that automatically integrates most of the features automatically with a Symfony application.

## Getting started

Simply add the `ToleranceBundle` in your Symfony's AppKernel.

```
$bundles[] = new Tolerance\Bridge\Symfony\Bundle\ToleranceBundle\ToleranceBundle();
```

You can also checkout the example Symfony service and the test application.

## Operation runner

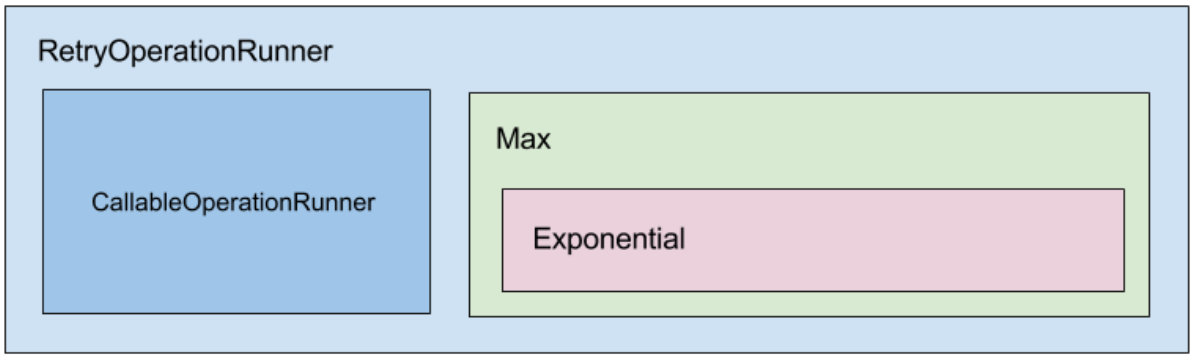
### Factory

When using simple operation runners, you can create them using the YML configuration of the bundle. Each operation runner have a name (`default` in the following example). The created operation runner will be available via the service named `tolerance.operation_runner.default`.

```
tolerance:
    operation_runners:
        default:
            retry:
                runner:
                    callback: ~

                waiter:
                    count_limited:
                        count: 10
                        waiter:
```

```
                                exponential_back_off:
                                    exponent: 1
                                    waiter:
                                        sleep: ~
```

In that example, that will create a operation runner that is the retry operation runner decorating a callable operation runner. The following image represents the imbrication of the different runners.




**Note:** This YML factory do not support recursive operation runner. That means that you can't use a chain runner inside another chain runner. If you need to create more complex operation runners, you should create your own service with a simple factory like the one that was in the tests before this YML factory.

**Tip:** If you just need to add a decorator on a created operation runner, simply uses Symfony DIC decorates features.

## Buffered termination

If you are using a buffered operation runner, it will automatically run all the buffered operations after the response it sent to the client (`kernel.terminate` event).

You can disable this feature with the following configuration:

```
tolerance:
    operation_runner_listener: false
```

This will automatically work with operation runners created using the factory. If you've created your own service, you will need to tag it with `tolerance.operation_runner` in order to declare it.

```
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.
↪com/schema/dic/services/services-1.0.xsd">

    <services>
        <service id="app.my_buffered_operation_runner" class=
↪"Tolerance\Operation\Runner\BufferedOperationRunner">
            <!-- Arguments... -->
```

```
            <tag name="tolerance.operation_runner" />
        </service>
    </services>
</container>
```

# Operation Wrappers

The purpose of this Symfony integration is to help you using operations and operation runners in an easy way. By using the AOP features provided by the JMSAopBundle you can wrap a Symfony service in an operation runner by simply using a tag or a YAML configuration.

**Important:** You need to first install JMSAopBundle in order to be able to use this AOP integration.

By default this feature is not activated so you need to activate it manually:

```
tolerance:
    aop:
        enabled: true
```

## Using a tag

Let's say now that you've a service for this `YourService` object that contains methods that are a bit risky and needs to be wrapped into an operation runner:

```
namespace App;

class YourService
{
    public function getSomething()
    {
        // This method needs to be in an operation runner because it's
        // doing something risky such as an API call.
    }
}
```

Once you've that, you can use the `tolerance.operation_wrapper` tag to wrap the different calls to some of your service's methods inside an operation runner.

```
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.
↪com/schema/dic/services/services-1.0.xsd">

    <services>
        <service id="app.your_service" class="App\YourService">
            <tag name="tolerance.operation_wrapper"
                 methods="getSomething"
                 runner="tolerance.operation_runner.default" />
        </service>
```

```
    </services>
</container>
```

The tag have 2 configuration options:

- `methods`: a comma separated names of the methods you want to *proxy*
- `runner`: the service name of the operation runner to use

And that's all, your calls to the method `getSomething` of your service will be wrapper inside a callback operation and run with the operation runner `operation_runner.service_name`.

### Using YAML

You can wrap some methods of a given class into a given operation runner. The following example shows how simple it can be to simply get metrics from some API calls for instance.

All the calls to the methods `requestSomething` and `findSomethingElse` to a service with the class `HttpThirdPartyClient` will be proxied through the operation runner `tolerance.operation_runners.3rdparty`. This metrics operation runner created in YAML will record the success and failure of the operations to a metric publisher.

```
tolerance:
aop:
    enabled: true

    wrappers:
        - class: "Acme\Your\HttpThirdPartyClient"
          methods:
              - requestSomething
              - findSomethingElse
          runner: tolerance.operation_runners.3rdparty

operation_runners:
    default:
        callback: ~

    3rdparty:
        success_failure_metrics:
            publisher: tolerance.metrics.publisher.statsd
            namespace: 3rdparty.outgoing.requests
```

## Tracer

This integration is created to require you the less effort possible to use Tolerance's Tracer component. Enable it with the following configuration.

```
tolerance:
    tracer:
        service_name: MyApplicationService

        zipkin:
            http:
                base_url: http://address.of.your.zipkin.example.com:9411
```

By default, it'll activate the following integrations:

- Request listener that reads the span informations from a request's header
- Monolog processor that adds the span information to the context of each log
- Registered Guzzle middleware that create a span when sending a request if you are using CsaGuzzleBundle

# Metrics

The Symfony bundle comes with an integration for the Metrics component that allows you to easily collect and publish metrics.

## Collectors

You can create metric collectors using YAML. The available types, at the moment, are the following:

- `:rabbitmq`: will get some metrics about a RabbitMq queue, from the management API.

The following YAML is a reference of the possible configuration.

```
tolerance:
    metrics:
        collectors:
            my_queue:
                type: rabbitmq
                namespace: metric.prefix
                options:
                    host: %rabbitmq_host%
                    port: %rabbitmq_management_port%
                    username: %rabbitmq_user%
                    password: %rabbitmq_password%
                    vhost: %rabbitmq_vhost%
                    queue: %rabbitmq_queue_name%
```

## Publishers

You can create publishers using YAML. The available types, at the moment, are the following:

- `:hosted_graphite`: publish some metrics to the HostedGraphite service.
- `:beberlei`: publish some metrics using a "collector" from beberlei/metrics.

The following YAML is a reference of the possible configuration.

```
tolerance:
    metrics:
        publishers:
            hosted_graphite:
                type: hosted_graphite
                options:
                    server: %hosted_graphite_server%
                    port: %hosted_graphite_port%
                    api_key: %hosted_graphite_api_key%

            statsd:
```

```
                type: beberlei
                options:
                    service: beberlei_metrics.collector.statsd
                    auto_flush: true
```

### Your own consumer and publishers

If you want to register your own consumers and publishers to the default collection services, you have to tag your services with the `:tolerance.metrics.collector` and `:tolerance.metrics.publisher` tags.

### Command

A command to collect and publish the metrics is built-in in the Bundle. As an example, you can run this command periodically to be able to graph metrics from your application.

```
app/console tolerance:metrics:collect-and-publish
```

If required, you can configure the collector and publisher used by this command:

```
tolerance:
    metrics:
        command:
            collector: tolerance.metrics.collector.collection
            publisher: tolerance.metrics.publisher.collection
```

### Request

If configured, the bundle will register listeners to send two metrics (a timing and an increment) at the end of each request.

In other words, you just have to put this YAML configuration in order to publish metrics about the duration and the number of requests to your Symfony application:

```
tolerance:
    metrics:
        request:
            namespace: my_api.http.request
            publisher: tolerance.metrics.publisher.statsd
```

## Guzzle

The Symfony bundle comes with an integration for Guzzle that allows automatic retry of failed requests.

### Configuration

First, you need to install the CsaGuzzleBundle.

Then you must enabled the bridge using the *guzzle* key:

```
tolerance:
    guzzle: true
```

Finally, just add the *retries* option in the configuration of your client:

```
csa_guzzle:
    clients:
        my_client:
            config:
                retries: 2
```

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search