PHPBench Documentation

Release

Daniel Leech

Jul 27, 2017

Contents

1 Book **3**
- 1.1 Introduction . . . 3
- 1.2 Installing . . . 4
- 1.3 Quick Start . . . 5
- 1.4 Writing Benchmarks . . . 9
- 1.5 Benchmark Runner . . . 16
- 1.6 Reports . . . 20
- 1.7 Storage and Querying . . . 25
- 1.8 Environment . . . 29
- 1.9 Report Generators . . . 31
- 1.10 Report Renderers . . . 33
- 1.11 Configuration . . . 34
- 1.12 FAQ . . . 37

2 Extensions **39**
- 2.1 XDebug . . . 39
- 2.2 DBAL . . . 42
- 2.3 Custom Extensions . . . 43

3 Indices and tables **47**

PHPBench is a benchmarking framework for PHP. Find it on Github.

CHAPTER 1

Book

Introduction

PHPBench is a benchmark runner for PHP. It enables you to write standard benchmarks for your application and classes and helps you to make smart decisions based on *comparative* results.

Features at a glance:

- **Revolutions and Iterations**: Spin and repeat.
- **Process Isolation**: Benchmarks are run in separate processes with no significant overhead from the runner.
- **Reporting**: Powerful and extensible reports.
- **Deferred Reporting**: Dump benchmarking results to an XML file and report on them later.
- **Memory Usage**: Keep an eye on the amount of memory used by benchmarking subjects.

Why PHPBench?

Performance can be monitored and measured in a number of ways: profiling (via. XDebug or Blackfire), injecting timing classes (e.g. Symfony Stopwatch, Hoa Bench) or with server tools such as NewRelic.

PHPBench differs from these tools in that it allows you to benchmark explicit scenarios independently of the application context, and to run these scenarios multiple times in order to obtain a degree of *confidence* about the stability of the results.

As a tool it is analogous to the test framework PHPUnit, but instead of *tests* we run *benchmarks* and generate reports.

Are There Other Benchmarking Frameworks?

You can try Athletic .

Installing

PHPBench can be installed either as dependency for your project or as a global package.

Install as a PHAR package

Installing as a PHAR is convenient, you will need to download the phar and the public key, this can be done with CURL as follows:

```
$ curl -o phpbench.phar https://phpbench.github.io/phpbench/phpbench.phar
$ curl -o phpbench.phar.pubkey https://phpbench.github.io/phpbench/phpbench.phar.
↪pubkey
```

You will probably then want make it executable and put it in your systems global path, on Linux systems:

```
$ chmod 0755 phpbench.phar
$ sudo mv phpbench.phar /usr/local/bin/phpbench
$ sudo mv phpbench.phar.pubkey /usr/local/bin/phpbench.pubkey
```

You can update the version at any time by using the `self-update` command:

```
$ phpbench self-update
```

Warning: PHPBench is unstable, installing as a PHAR will means that you are always updating to the latest version, the latest version may include BC breaks. Therefore it is recommended to include the package as a project dependency for continuous-integration.

Composer Install

To install PHPBench as a dependency of your project:

```
$ composer require phpbench/phpbench @dev --dev
```

You may then run PHPBench from your project's directory as follows:

```
$ ./vendor/bin/phpbench
```

Composer Global Install

You may install PHPBench globally using composer:

```
$ composer global require phpbench/phpbench @dev
```

Note: You will need to add Composer's global `bin` directory to your systems `PATH` variable (on linux). See the above link.

You may now run PHPBench simply as:

```
$ phpbench
```

Quick Start

This tutorial will walk you through creating a typical, simple, project that uses PHPBench as a dependency. You may also install PHPBench globally, see the *Installing* chapter for more information.

You may skip various sections according to your needs and use this as a general reference.

Create your project

Create a directory for the tutorial:

```
$ mkdir phpbench-tutorial
```

And create the following Composer file within it:

```
{
    "name": "acme/phpbench-test",
    "require-dev": {
        "phpbench/phpbench": "^1.0@dev"
    },
    "autoload": {
        "psr-4": {
            "Acme\\": "lib"
        }
    }
}
```

Now perform a Composer install:

```
$ composer install
```

Note: You may also install PHPBench globally, see the *Installing* chapter for more information.

PHPBench should now be installed. Now create two directories, `benchmarks` and `lib` which we will need further on:

```
$ mkdir benchmarks
$ mkdir lib
```

PHPBench configuration

In order for PHPBench to be able to autoload files from your library, you should specify the path to your bootstrap file (i.e. `vendor/autoload.php`). This can be done in the PHPBench *configuration*.

Create the file `phpbench.json` in the projects root directory:

```
{
    "bootstrap": "vendor/autoload.php"
}
```

Note: PHPBench does not **require** a bootstrap (or a configuration file for that matter). You may omit it if you do not need autoloading, or you want to include files manually.

Creating and running a benchmark

You will need some code to benchmark, create a simple class in `lib` which consumes *time itself*:

```php
<?php

namespace Acme;

class TimeConsumer
{
    public function consume()
    {
        usleep(100);
    }
}
```

In order to benchmark your code you will need to execute that code within a method of a benchmarking class. Benchmarking classes MUST have the `Bench` suffix and each benchmarking method must be prefixed with `bench`.

Create the following class in the `benchmarks` directory:

```php
<?php

use Acme\TimeConsumer;

class TimeConsumerBench
{
    public function benchConsume()
    {
       $consumer = new TimeConsumer();
       $consumer->consume();
    }
}
```

Now you can execute the benchmark as follows:

```
$ ./vendor/bin/phpbench run benchmarks/TimeConsumerBench.php --report=default
```

And you should see some output similar to the following:

```
PhpBench 0.8.0-dev. Running benchmarks.

\TimeConsumerBench

    benchConsume                    I0 P0          [μ Mo]/r: 173.00μs    [μSD μRSD]/r: 0.
↪00μs 0.00%

1 subjects, 1 iterations, 1 revs, 0 rejects
T: 173μs μSD/r 0.00μs μRSD/r: 0.00%
min [mean mode] max: 173.00 [173.00 1732.00] 173.00 (μs/r)

+------------------+--------------+-------+--------+------+------+-----+----------+-
↪----------+---------+-------+
```

```
| benchmark          | subject        | group | params | revs | iter | rej | mem       |␣
↪time       | z-score | diff   |
+------------------+---------------+-------+--------+------+------+-----+-----------+-
↪-----------+---------+-------+
| TimeConsumerBench | benchConsume   |       | []     | 1    | 0    | 0   | 265,936b |␣
↪173.0000μs | 0.00σ   | 0.00% |
+------------------+---------------+-------+--------+------+------+-----+-----------+-
↪-----------+---------+-------+
```

You may have guessed that the code was only executed once (as indicated by the `revs` column). To achieve a better measurement we should increase the number of times that the code is consecutively executed.

```php
<?php

// ...

class TimeConsumerBench
{
    /**
     * @Revs(1000)
     */
    public function benchConsume()
    {
        // ...
    }
}
```

Run the benchmark again and you should notice that the report states that 1000 revolutions were performed. *Revolutions* in PHPBench represent the number of times that the code is executed consecutively within a single measurement.

Currently we only execute the benchmark subject a single time, to verify the result you should increase the number of *iterations* using the `@Iterations` annotation (either as a replacement or in addition to `@Revs`:

```php
<?php

// ...

class TimeConsumerBench
{
    /**
     * @Revs(1000)
     * @Iterations(5)
     */
    public function benchConsume()
    {
        // ...
    }
}
```

Now when you run the report you should see that it contains 5 rows. One measurement for each iteration, and each iteration executed the code 1000 times.

Note: You can override the number of iterations and revolutions on the CLI using the `--iterations` and `--revs` options.

At this point it would be better for you to use the `aggregate` report rather than `default`:

```
$ php vendor/bin/phpbench run benchmarks/TimeConsumerBench.php --report=aggregate
```

Increase Stability

You will see the columns *stdev* and *rstdev*. *stdev* is the standard deviation of the set of iterations and *rstdev* is relative standard deviation.

Stability can be inferred from *rstdev*, with 0% being the best and anything about 2% should be treated as suspicious.

To increase stability you can use the `--retry-threshold` to automatically *repeat the iterations* until the *diff* (the percentage difference from the lowest measurement) fits within a given threshold:

```
$ php vendor/bin/phpbench run benchmarks/TimeConsumerBench.php --report=aggregate --
↪retry-threshold=5
```

Warning: Lower values for `retry-threshold`, depending on the stability of your system, generally lead to increased total benchmarking time.

Customize Reports

PHPBench also allows you to customize reports on the command line, try the following:

```
$ ./vendor/bin/phpbench run benchmarks/TimeConsumerBench.php --report='{"extends":
↪"aggregate", "cols": ["subject", "mode"]}'
```

Above we configure a new report which extends the `default` report that we have already used, but we use only the `subject` and `mode` columns. A full list of all the options for the default reports can be found in the *Report Generators* chapter.

Configuration

Now to finish off, lets add the path and new report to the configuration file:

```
{
    ...
    "path": "benchmarks",
    "reports": {
        "consumation_of_time": {
            "extends": "default",
            "title": "The Consumation of Time",
            "description": "Benchmark how long it takes to consume time",
            "cols": [ "subject", "mode" ]
        }
    }
}
```

Warning: JSON files are very strict - be sure not to have commas after the final elements in arrays or objects!

Above you tell PHPBench where the benchmarks are located and you define a new report, `consumation_of_time` with a title, description and sort order.

We can now run the new report:

```
$ php vendor/bin/phpbench run --report=consumation_of_time
```

Note: Note that we did not specify the path to the benchmark file, by default all benchmarks under the given or configured path will be executed.

This quick start demonstrated some of the features of PHPBench, but there is more to discover everything can be found in this manual. Happy benchmarking.

Writing Benchmarks

Benchmark classes have the following characteristics:

- The class and filename must be the same.
- Class methods that start with `bench` will be executed by the benchrunner and timed.

PHPBench does not require that the benchmark class be aware of PHPBench library - it does not need to extend a parent class or implement an interface.

The following is a simple benchmark class:

```php
<?php
// HashBench.php

class HashBench
{
    public function benchMd5()
    {
        hash('md5', 'Hello World!');
    }

    public function benchSha1()
    {
        hash('sha1', 'Hello World!');
    }
}
```

And it can be executed as follows:

```
$ phpbench run examples/HashBench.php --progress=dots
PhpBench 0.8.0-dev. Running benchmarks.

...

3 subjects, 30 iterations, 30000 revs, 0 rejects
T: 30543μs μSD/r 0.05μs μRSD/r: 4.83%
min mean max: 0.78 1.02 1.47 (μs/r)
```

Note: The above command does not generate a report, add `--report=default` to view something useful.

PHPBench reads docblock annotations in the benchmark class. Annotations can be placed in the class docblock, or on individual methods docblocks.

Note: Instead of prefixing a method with `bench` you can use the `@Subject` annotation or specify a *custom pattern*.

Improving Precision: Revolutions

When testing units of code where microsecond accuracy is important, it is necessary to increase the number of *revolutions* performed by the benchmark runner. The term "revolutions" (invented here) refers to the number of times the benchmark is executed consecutively within a single time measurement.

We can arrive at a more accurate measurement by determining the mean time from multiple revolutions (i.e. *time / revolutions*) than we could with a single revolution. In other words, more revolutions means more precision.

Revolutions can be specified using the `@Revs` annotation:

```
<?php

/**
 * @Revs(1000)
 */
class HashBench
{
    // ...
}
```

You may also specify an array:

```
<?php

/**
 * @Revs({1, 8, 64, 4096})
 */
class HashBench
{
    // ...
}
```

Revolutions can also be overridden from the *command line*.

Verifying and Improving Stability: Iterations

Iterations represent the number of times we will perform the benchmark (including all the revolutions). Contrary to revolutions, a time reading will be taken for *each iteration*.

By looking at the separate time measurement of each iteration we can determine how *stable* the readings are. The less the measurements differ from each other, the more stable the benchmark is, and the more you can trust the results.

Note: In a *perfect* environment the readings would all be *exactly* the same - but such an environment is unlikely to exist

Iterations can be specified using the `@Iterations` annotation:

```
<?php

/**
```

```
 * @Iterations(5)
 */
class HashBench
{
    // ...
}
```

As with *revolutions*, you may also specify an array.

Iterations can also be overridden from the *command line*.

You can instruct PHPBench to continuously run the iterations until the deviation of each iteration fits within a given margin of error by using the `--retry-threshold`. See *Retry Threshold* for more information.

Subject (runtime) State: Before and After

Any number of methods can be executed both before and after each benchmark **subject** using the `@BeforeMethods` and `@AfterMethods` annotations. Before methods are useful for bootstrapping your environment, for example:

```
<?php

/**
 * @BeforeMethods({"init"})
 */
class HashBench
{
    private $hasher;

    public function init()
    {
        $this->hasher = new Hasher();
    }

    public function benchMd5()
    {
        $this->hasher->md5('Hello World!');
    }
}
```

Multiple before and after methods can be specified.

Note: If before and after methods are used when the `@ParamProviders` annotations are used, then they will also be passed the parameters.

Benchmark (external) State: Before and After

Sometimes you will want to perform actions which establish an *external* state. For example, creating or populating a database, creating files, etc.

This can be achieved by creating **static** methods within your benchmark class and adding the `@BeforeClassMethods` and `@AfterClassMethods`:

These methods will be executed by the runner once per benchmark class.

```php
<?php

/**
 * @BeforeClassMethods({"initDatabase"})
 */
class DatabaseBench
{
    public static function initDatabase()
    {
        // init database here.
    }

    // ...
}
```

Note: These methods are static and are executed in a process that is separate from that from which your iterations will be executed. Therefore **state will not be carried over to your iterations!**.

Parameterized Benchmarks

Parameter sets can be provided to benchmark subjects. For example:

```php
<?php

class HashBench
{
    public function provideStrings()
    {
        return array(
            [
                'string' => 'Hello World!'
            ],
            [
                'string' => 'Goodbye Cruel World!'
            ]
        );
    }

    /**
     * @ParamProviders({"provideStrings"})
     */
    public function benchMd5($params)
    {
        hash('md5', $params['string']);
    }
}
```

The `benchMd5` subject will now be benchmarked with each parameter set.

Multiple parameter providers can be used, in which case the data sets will be combined into a cartesian product - all possible combinations of the parameters will be generated, for example:

```php
<?php

class HashBench
```

```
{
    public function provideStrings()
    {
        return array(
            array(
                'string' => 'Hello World!',
            ),
            array(
                'string' => 'Goodbye Cruel World!',
            ),
        );
    }

    public function provideNumbers()
    {
        return array(
            array(
                'algorithm' => 'md5',
            ),
            array(
                'algorithm' => 'sha1',
            ),
        );
    }

    /**
     * @ParamProviders({"provideStrings", "provideNumbers"})
     */
    public function benchHash($params)
    {
        hash($params['algorithm'], $params['string']);
    }
}
```

Will result in the following parameter benchmark scenarios:

```
<?php

// #0
array('string' => 'Hello World!', 'algorithm' => 'md5');

// #1
array('string' => 'Goodbye Cruel World!', 'algorithm' => 'md5');

// #2
array('string' => 'Hello World!', 'algorithm' => 'sha1');

// #3
array('string' => 'Goodbye Cruel World!', 'algorithm' => 'sha1');
```

Groups

You can assign benchmark subjects to groups using the `@Groups` annotation.

```
<?php
```

```
/**
 * @Groups({"hash"})
 */
class HashBench
{
    // ...
}
```

The group can then be targeted using the command line interface.

Skipping Subjects

You can skip subjects by using the `@Skip` annotation:

```
<?php

class HashBench extends Foobar
{
    /**
     * @Skip()
     */
    public function testFoobar()
    {
    }
}
```

Extending Existing Array Values

When working with annotations which accept an array value, you may wish to extend the values of the same annotation from ancestor classes. This can be accomplished using the `extend` option.

```
<?php

abstract class AbstractHash
{
    /**
     * @Groups({"md5"})
     */
    abstract public function benchMd5();
}

/**
 * @Groups({"my_hash_implementation"}, extend=true)
 */
class HashBench extends AbstractHash
{
    public function benchMd5()
    {
        // ...
    }
}
```

The `benchHash` subject will now be in both the `md5` and `my_hash_implementation` groups.

This option is available on all array valued (plural) annotations.

Recovery Period: Sleeping

Sometimes it may be necessary to pause between iterations in order to let the system recover. Use the `@Sleep` annotation, specifying the number of **microseconds** required:

```
<?php

class HashBench
{
    /**
     * @Iterations(10)
     * @Sleep(1000000)
     */
    public function benchMd5()
    {
        md5('Hello World');
    }
}
```

The above example will pause (sleep) for 1 second *after* each iteration.

Note: This can be overridden using the `--sleep` option from the CLI.

Microseconds to Minutes: Time Units

If you have benchmarks which take seconds or even minutes to execute then the default time unit, microseconds, is going to be far more visual precision than you need and will only serve to make the results more difficult to interpret.

You can specify *output* time units using the `@OutputTimeUnit` annotation (*precision* is optional):

```
<?php

class HashBench
{
    /**
     * @Iterations(10)
       @OutputTimeUnit("seconds", precision=3)
     */
    public function benchSleep()
    {
        sleep(2);
    }
}
```

The following time units are available:

- `microseconds`
- `milliseconds`
- `seconds`
- `minutes`
- `hours`
- `days`

Mode: Throughput Representation

The output mode determines how the measurements are presented, either *time* or *throughput*. *time* mode is the default and shows the average execution time of a single *revolution*. *throughput* shows how many *operations* are executed within a single time unit:

```
<?php

class HashBench
{
    /**
     * @OutputTimeUnit("seconds")
     * @OutputMode("throughput")
     */
    public function benchMd5()
    {
        hash('md5', 'Hello World!');
    }
}
```

PHPBench will then render all measurements for *benchMd5* similar to *363,874.536ops/s*.

Warming Up: Getting ready for the show

In some cases, it might be a good idea to execute a revolution or two before performing the revolutions time measurement.

For example, when benchmarking something that uses an class autoloader, the first revolution will always be slower because the autoloader will not to be called again.

Use the `@Warmup` annotation to execute any number of revolutions before actually measuring the revolutions time.

```
<?php

// ...
class ReportBench
{
    // ...

    /**
     * @Warmup(2)
     * @Revs(10)
     */
    public function benchGenerateReport()
    {
        $this->generator->generateMyComplexReport();
    }
}
```

As with *revolutions*, you may also specify an array.

Benchmark Runner

The benchmark runner is a command line application which executes the benchmarks and generates reports from the results.

Running Benchmarks

To run all benchmarks in a specific directory:

```
$ phpbench run /path/to
```

To run a single benchmark class, specify a specific file:

```
$ phpbench run /path/to/HashBench.php
```

To run a single method of a single benchmark class, use the `--filter` option:

```
$ phpbench run /path/to/HashBench.php --filter=benchMd5
```

Groups can be specified using the `--group` option:

```
$ phpbench run /path/to/HashBench.php --group=hash
```

Note: Both `--subject` and `--group` options may be specified multiple times.

Filtering

The `--filter` option accepts a regex without the delimiters and matches against a string such as `HashBench::benchMd5`, so all of the following are valid:

```
$ phpbench run /path/to --filter=benchFoo
$ phpbench run /path/to --filter=HashBench::benchFoo
$ phpbench run /path/to --filter=Hash.*
```

Overriding Iterations and Revolutions

The benchmark runner can override the number of *revolutions* and *iterations* which will be executed:

```
$ phpbench run /path/to/HashBench.php --iterations=10 --revs=1000
```

You may specify these options multiple times.

Overriding the Bootstrap

You can override or set the *Bootstrap* using the `--bootstrap` option:

```
$ phpbench run /path/to/HashBench.php --bootstrap=vendor/autoload.php
```

Generating Reports

By default PHPBench will run the benchmarks and tell you that the benchmarks have been executed successfully. In order to see some useful information you can specify that a report be generated.

By default there are two reports `default` and `aggregate`, and they can be specified directly using the `--report` option:

```
$ phpbench run /path/to/HashBench.php --report=default
```

See the *Reports* chapter for more information on how you can configure reports.

Note: If you want to suppress all other output and only show the output from the reports you can use the `--progress=none` option. This is especially useful when piping a report to another program.

Retry Threshold

PHPBench is able to dramatically improve the stability of your benchmarks by retrying the iteration set until all the deviations in time between iterations fit within a given margin of error.

You can set this as follows:

```
$ phpbench run /path/to/HashBench.php --retry-threshold=5
```

The retry threshold is the margin of error as a percentage which is allowed between deviations. Generally the lower this value, the higher the stability, but the longer it will take for a set of iterations to be resolved.

By default the retry threshold is disabled.

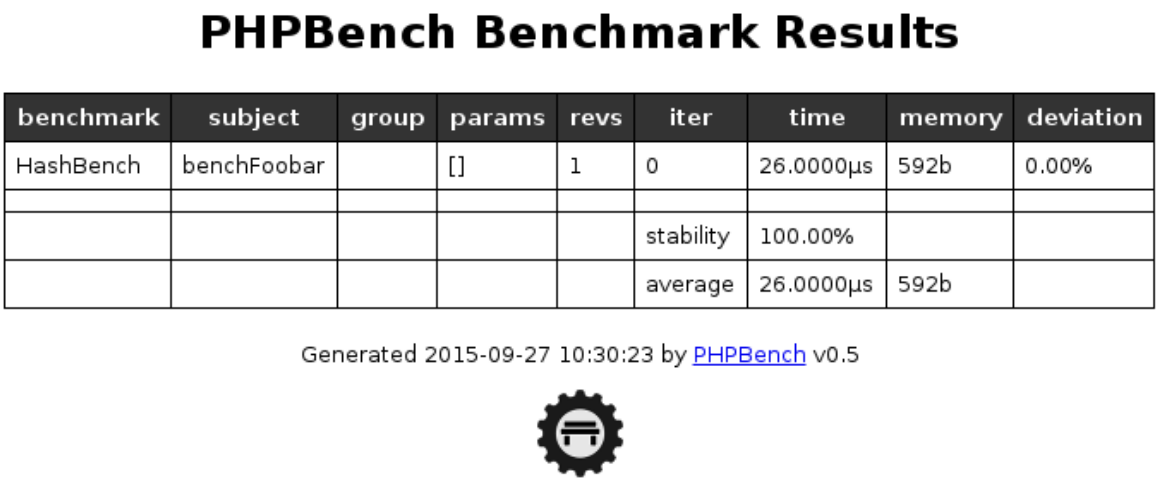
You may also set the retry threshold in the *configuration*.

Changing the Output Medium

By default PHPBench will output the reports to the console using the `console` output. The output can be changed using the `--output` option. For example, to render a HTML document:

```
$ phpbench run /path/to/HashBench.php --report=default --output=html
```

Example of HTML output:



PHPBench Benchmark Results

benchmark	subject	group	params	revs	iter	time	memory	deviation
HashBench	benchFoobar		[]	1	0	26.0000µs	592b	0.00%
					stability	100.00%		
					average	26.0000µs	592b	

Generated 2015-09-27 10:30:23 by PHPBench v0.5

See the *Report Renderers* chapter for more information.

Deferring Report Generation

You can store benchmark results which can then later be used later to generate reports.

There are two ways to do this: Firstly you may dump the results to an XML file, secondly you can use a storage driver to persist them.

To dump the benchmark results to an XML file use the `--dump-file` option:

```
$ phpbench run /path/to/HashBench.php --dump-file=report.xml
```

You can then generate reports using the `report` command:

```
$ phpbench report --file=report.xml --report=default
```

Alternatively (or in a addition) you may use the storage driver as follows:

```
$ phpbench run /path/to/HashBench.php --store
```

Then generate reports using a *query*:

```
$ phpbench report --query='benchmark: "MyBench"' --report=aggregate
```

This method is highly mighty. See the *storage* chapter for more information.

Comparing Results

You can compare the results of two or more sets of results using the *compare* report.

First you should generate a suite result document for each separate implementation and specify a *context*:

```
$ # .. configure for implementation A
$ phpbench run --context="Impl. A" --dump-file=impl-a.xml
$ # .. configure for implementation B
$ phpbench run --context="Impl. B" --dump-file=impl-b.xml
$ # .. configure for implementation C
$ phpbench run --context="Impl. C" --dump-file=impl-c.xml
```

Now you can use the *report* command and specify the *compare* report to compare the results for each implementation side-by-side:

```
$ phpbench report --file=impl-a.xml --file=impl-b.xml --file=impl-c.xml --
↪report=compare
+---------+-----------------+----------+---------+--------+-----------+-----------+--
↪---------+
| context | benchmark       | subject  | group   | params | t:Impl. A | t:Impl. C |␣
↪t:Impl. B |
+---------+-----------------+----------+---------+--------+-----------+-----------+--
↪---------+
| Impl. A | HashingBenchmark | benchMd5 | hashing | []     | 2.4448μs  | 4.3039μs  |␣
↪1.5003μs  |
+---------+-----------------+----------+---------+--------+-----------+-----------+--
↪---------+
```

Progress Reporters

By default PHPBench issues a single `.` for each benchmark subject executed. This is the `dots` progress reporter. Different progress reporters can be specified using the `--progress` option:

Fig. 1.1: blinken progress logger.

```
$ phpbench run /path/to/HashBench.php --progress=classdots
```

The built-in progress loggers are:

- `verbose`: The default logger, format: *[R<retry nb.>] I<iter nb.> P<parameter set nb.> <mean|mode per rev.> <standard deviation per rev.> <relative standard deviation per rev.>*).
- `travis`: Similar to verbose, but with no fancy console manipulation. Perfect for travis.
- `dots`: Shows one dot per subject (like PHPUnit).
- `classdots`: Shows the benchmark class, and then a dot for each subject.
- `blinken`: Highly visual progress logger.
- `histogram`: Shows a histogram with 8 vertical levels and 16 bins for each iteration set.

Note: PHPBench is aware of the `CONTINUOUS_INTEGRATION` environment variable set by travis. If this variable is set then the default logger will automatically be changed to *travis* and the *dots* progress logger will not do any fancy stuff.

All of the progress reports contain the following footer:

```
3 subjects, 30 iterations, 30000 revs, 0 rejects
min [mean mode] max: 0.84 [1.13 1.12] 1.66 (μs/r)
T: 33987μs μSD/r 0.16μs μRSD/r: 14.92%
```

It provides a summary of the minimum, mean, mode, and maximum subject times, given microseconds per revolution. T is the aggregate total time, μSD/r is the mean standard deviation, and μRSD/r is the mean relative standard deviation.

Warning: These summary statistics can be misleading. You should always verify the individual subject statistics before drawing any conclusions.

Configuration File

A custom configuration file can be specified with the `--config` option. See the *Configuration* chapter for more information on configuration.

Reports

PHPBench includes a primitive reporting framework. It allows for *report generators* which generate *reports* from one or more benchmarking suite results.

Reports can be generated for each `run` that your perform, or using historical data by using the `report` command.

The reports are then *renderered* using a *report renderer* to various outputs (e.g. console, HTML, markdown, CSV).

This chapter will deal with generating reports and assume that the `console` renderer is used.

Generating Reports

To report after a benchmarking run:

```
$ phpbench run --report=aggregate
```

Multiple reports can be specified:

```
$ phpbench run --report=aggregate --report=env
```

The report command operates in a similar way but requires you to provide some data, either from XML dumps or from a *storage* query:

```
$ phpbench report --query='benchmark: "MyBench"' --report=aggregate
```

For more information on storage and the query language see *Storage and Querying*.

Configuring Reports

All reports can be configured either in the *report configuration* or directly on the command line using a simplified JSON encoded string instead of the report name:

```
$ phpbench run --report='generator: "table", cols: [ "suite", "subject", "mean" ],␣
↪break: ["benchmark"]'
```

In each case it is required to specify the `generator` key which corresponds to the registered name of the *report generator*.

You may also **extend** an existing report configuration:

```
$ phpbench run --report='extend: "aggregate", break: ["benchmark", "revs"]'
```

This will merge the given keys onto the configuration for the aggregate report.

Table Generator

For details about the table generator see the *table* reference, this section will simply offer practical examples.

Note: Here we give the report configuration as an argument on the command line, it is important to note that reports can also be defined in the *configuration*.

Selecting columns

You can select exactly which columns you need using the `cols` option. If you make a mistake an exception will be thrown showing all the valid possibilities, see the *columns* reference.

The following examples will make use of this option for brevity.

Breaking into multiple tables

Use the `break` option to split tables based on the unique values of the given keys:

```
$ phpbench run --report='generator: "table", break: ["revs"], cols: ["subject", "mean
↪"]'

    revs: 1
    +-------------+---------+
    | subject     | mean    |
    +-------------+---------+
    | benchMd5    | 3.300μs |
    | ...         | ...     |
    +-------------+---------+

    revs: 10
    +-------------+---------+
    | subject     | mean    |
    +-------------+---------+
    | benchMd5    | 0.700μs |
    | ...         | ...     |
    +-------------+---------+

    revs: 100
    +-------------+---------+
    | subject     | mean    |
    +-------------+---------+
    | benchMd5    | 0.447μs |
    | ...         | ...     |
    +-------------+---------+
```

Multiple columns may be specified:

```
$ phpbench run --report='generator: "table", break: ["benchmark", "revs"], cols: [
↪"subject", "mean"]'

benchmark: HashingBenchmark, revs: 1
+-------------+---------+
| subject     | mean    |
+-------------+---------+
| benchMd5    | 3.400μs |
| benchSha1   | 4.700μs |
| benchSha256 | 4.700μs |
+-------------+---------+

benchmark: HashingBenchmark, revs: 10
+-------------+---------+
| subject     | mean    |
+-------------+---------+
| benchMd5    | 0.720μs |
| benchSha1   | 0.970μs |
| benchSha256 | 1.320μs |
+-------------+---------+
```

Comparing Values

To compare values by factor horizontally, use the `compare` option, for example to compare mean times against revs:

```
$ phpbench run --report='generator: "table", compare: "revs", cols: ["subject", "mean
↪"]'

    +-------------+-------------+--------------+---------------+
    | subject     | revs:1:mean | revs:10:mean | revs:100:mean |
    +-------------+-------------+--------------+---------------+
    | benchMd5    | 3.800μs     | 0.890μs      | 0.535μs       |
    | benchSha1   | 5.600μs     | 0.930μs      | 0.651μs       |
    | benchSha256 | 5.500μs     | 1.490μs      | 1.114μs       |
    +-------------+-------------+--------------+---------------+
```

By default the mean is used as the comparison value, you may also select different value columns using `compare_fields`, e.g. to show both `mean` and `mode`:

```
$ phpbench run --report='generator: "table", compare: "revs", cols: ["subject", "mean
↪"], compare_fields: ["mean", "mode"]'
```

Note: The compare function "squashes" the non-statistical columns which have the same values - sometimes this may result in there being more than one "statstic" for the `compare` column. In such cases extra columns are added suffixed with an index, for example: `revs:10:mean#1`.

Difference Between Rows

You can show the percentage of difference from the lowest column value in the table () by specifying the `diff` column. By default this will use the `mean`, you can specify a different value using the `diff_col` option, e.g. `diff_col: "mode"`.

```
$ phpbench run --report='generator: "table", cols: ["subject", "revs", "mean", "diff"]
↪'
+---------------+------+---------+--------+
| subject       | revs | mean    | diff   |
+---------------+------+---------+--------+
| benchVariance | 100  | 12.05μs | 0.00%  |
| benchStDev    | 100  | 12.53μs | +4.03% |
+---------------+------+---------+--------+
```

Sorting

Sorting can be achieved on multiple columns in either ascending (`asc`) or descending (`desc`) order.

```
$ phpbench run --report='generator: "table", cols: ["subject", "revs", "mean", "diff
↪"], sort: {subject: "asc", mean: "desc"}'

    +-------------+------+---------+---------+
    | subject     | revs | mean    | diff    |
    +-------------+------+---------+---------+
    | benchMd5    | 1    | 3.600μs | +89.32% |
    | benchMd5    | 10   | 0.680μs | +43.44% |
    | benchMd5    | 100  | 0.420μs | +8.43%  |
    | benchSha1   | 1    | 5.000μs | +92.31% |
    | benchSha1   | 10   | 0.900μs | +57.27% |
    | benchSha1   | 100  | 0.494μs | +22.15% |
    | benchSha256 | 1    | 4.600μs | +91.64% |
```

```
    | benchSha256 | 10    | 1.320μs | +70.86% |
    | benchSha256 | 100   | 0.847μs | +54.59% |
    +-------------+-------+---------+---------+
```

Default Reports

Configured reports can be executed simply by name as follows:

```
$ phpbench run --report=aggregate
```

The following are reports defined by PHPBench, other reports can be defined in your *configuration*.

aggregate

Shows aggregate details of each set of iterations:

```
+-----------------+-------------+---------+--------+------+-----+----------+---------
↪+---------+---------+---------+---------+--------+
| benchmark       | subject     | groups  | params | revs | its | mem_peak | best    ␣
↪| mean    | mode    | worst   | stdev   | rstdev |
+-----------------+-------------+---------+--------+------+-----+----------+---------
↪+---------+---------+---------+---------+--------+
| HashingBenchmark | benchMd5    | hashing | []     | 1000 | 10  | 272,616b | 2.470μs␣
↪| 2.636μs | 2.621μs | 2.805μs | 0.093μs | 3.55%  |
| HashingBenchmark | benchSha1   | hashing | []     | 1000 | 10  | 272,616b | 2.640μs␣
↪| 2.837μs | 2.903μs | 2.937μs | 0.097μs | 3.43%  |
| HashingBenchmark | benchSha256 | hashing | []     | 1000 | 10  | 272,616b | 2.735μs␣
↪| 3.021μs | 2.988μs | 3.247μs | 0.159μs | 5.26%  |
+-----------------+-------------+---------+--------+------+-----+----------+---------
↪+---------+---------+---------+---------+--------+
```

It is uses the `table` generator, see *table* for more information.

default

The default report presents the result of *each iteration*:

```
+-----------------+-------------+---------+--------+------+------+-----+----------+--
↪--------+---------+--------+
| benchmark       | subject     | groups  | params | revs | iter | rej | mem_peak |␣
↪time     | z-score | diff   |
+-----------------+-------------+---------+--------+------+------+-----+----------+--
↪--------+---------+--------+
| HashingBenchmark | benchMd5    | hashing | []     | 1000 | 0    | 0   | 268,160b |␣
↪0.8040μs | -1σ     | -3.48% |
| HashingBenchmark | benchMd5    | hashing | []     | 1000 | 1    | 0   | 268,160b |␣
↪0.8620μs | +1.00σ  | +3.48% |
| HashingBenchmark | benchSha256 | hashing | []     | 1000 | 0    | 0   | 268,160b |␣
↪1.2880μs | +1.00σ  | +1.98% |
| HashingBenchmark | benchSha256 | hashing | []     | 1000 | 1    | 0   | 268,160b |␣
↪1.2380μs | -1σ     | -1.98% |
| HashingBenchmark | benchSha1   | hashing | []     | 1000 | 0    | 0   | 268,160b |␣
↪0.9030μs | -1σ     | -4.7%  |
```

```
| HashingBenchmark | benchSha1   | hashing | []      | 1000 | 1    | 0   | 268,160b |␣
↪0.9920μs | +1.00σ  | +4.70% |
+------------------+-------------+---------+---------+------+------+-----+----------+--
↪--------+---------+--------+
```

It is uses the `table` generator, see *table* for more information.

env

This report shows information about the environment that the benchmarks were executed in.

```
+--------------+---------+------------------------------------------+
| provider     | key     | value                                    |
+--------------+---------+------------------------------------------+
| uname        | os      | Linux                                    |
| uname        | host    | dtlt410                                  |
| uname        | release | 4.2.0-1-amd64                            |
| uname        | version | #1 SMP Debian 4.2.6-1 (2015-11-10)       |
| uname        | machine | x86_64                                   |
| php          | version | 5.6.15-1                                 |
| unix-sysload | l1      | 0.52                                     |
| unix-sysload | l5      | 0.64                                     |
| unix-sysload | l15     | 0.57                                     |
| vcs          | system  | git                                      |
| vcs          | branch  | env_info                                 |
| vcs          | version | edde9dc7542cfa8e3ef4da459f0aaa5dfb095109 |
+--------------+---------+------------------------------------------+
```

Generator: *table*.

Columns:

- **provider**: Name of the environment provider (see `PhpBench\\Environment\\Provider` in the code for more information).
- **key**: Information key.
- **value**: Information value.

See the *Environment* chapter for more information.

Note: The information available will differ depending on platform. For example, `unit-sysload` is unsurprisingly only available on UNIX platforms, where as the VCS field will appear only when a *supported* VCS system is being used.

Storage and Querying

PHPBench allows benchmarking results to be persisted using a configured storage driver. You can inspect the results with either the `show` or `report` commands.

Configuring a Storage Driver

PHPBench will use XML storage by default, which is fine for most purposes. If you want advanced functionality (e.g. the ability to query benchmarks) then you can install the *Doctrine DBAL extension*.

The XML storage driver will place benchmarks in a folder called `_storage` by default, this can be changed in the configuration as follows:

```
{
    'xml_storage_path' => '_storage'
}
```

Storing Results

In order to store benchmarking runs you simply need to give `--store` option when running your benchmarks:

```
$ phpbench run --store
```

Viewing the History

Once you have stored some benchmark runs you can use the history command to see what you have got:

```
$ phpbench log
run 875c827946204db23eadd4b10e76b7189e10dde2
Date:    2016-03-19T09:46:52+01:00
Branch:  git_log
Context: <none>
Scale:   1 subjects, 60 iterations, 120 revolutions
Summary: (best [mean] worst) = 433.467 [988.067] 504.600 (μs)
         T: 59,284.000μs μRSD/r: 9.911%

run 9d38a760e6ebec0a466c80f148264a7a4bb7a203
Date:    2016-03-19T09:46:39+01:00
Branch:  git_log
Context: <none>
Scale:   1 subjects, 30 iterations, 30 revolutions
Summary: (best [mean] worst) = 461.800 [935.720] 503.300 (μs)
         T: 28,071.600μs μRSD/r: 4.582%

...
```

Report Generation

You can report on a single given run ID using the `show` command:

```
$ phpbench show 9d38a760e6ebec0a466c80f148264a7a4bb7a203
```

You may also specify a different report with the `--report` option. In order to compare two or more reports, you should use the `report` command as detailed in the following section.

Meta UUIDs

It is possible to specify "meta" UUIDs, such as `latest`:

```
$ phpbench show latest
```

And also you may use the `-<n>` suffix to view the "nth" entry in the history from the latest:

```
$ phpbench show latest-1
```

Would show the second latest entry. Meta UUIDs can be used anywhere where you would normally specify a UUID, including queries.

Querying

Important: The XML storage driver does not support querying, if you require this functionality install the *Doctrine DBAL extension*.

PHPBench uses a query language very similar to that of MongoDB. A simple example:

```
$ phpbench report --report=aggregate --query='subject: "benchMd5", run: 239"'
```

Would show the results in an aggregate report for the benchmarking subject `benchMd5` from run `239`.

A more complex example:

```
$ phpbench report --report=aggregate --query='$and: [ { subject: "benchMd5" }, {␣
↪date: { $gt: "2016-02-09" } } ]'
```

This would generate a suite collection containing all the `benchMd5` subjects created after `2016-02-09`.

Logical Operators

Logical operators must have as a value an array of constraints.

$and

Return only the records which meet both of the given constraints:

```
$and: [ { field1: "value1" }, { field2: "value2" } ]
```

$or

Return only the records which meet at least one of the given constraints:

```
$or: [ { field1: "value1" }, { field2: "value2" } ]
```

Logical Comparisons

$eq

Note that that equality is assumed if the value for a field is a scalar:

```
subject: "benchMd5"
```

The verbose equality comparison would be:

```
subject: { $eq: "benchMd5" }
```

$neq

Non-equality comparison:

```
run: { $neq: 12 }
```

$gt, $gte

Greater than and greater than or equal to comparisons:

```
date: { $gt: "2016-02-10" }
```

$lt, $lte

Greater than and greater than or equal to comparisons:

```
date: { $lt: "2016-02-10" }
```

$in

Matches when the field value matches any one of the given values:

```
run: { $in: [ 10, 11, 12 ] }
```

$regex

Provides regular expression capabilities for pattern matching strings in queries:

```
benchmark: { $regex: "FooBarBench" }

benchmark: { $regex: "Foo.*Bench" }
```

Fields

The following fields are currently available for querying:

- **benchmark**: The benchmark class name.
- **subject**: The subject name (e.g. `benchMd5`)
- **revs**: The number of revolutions.
- **date**: The date.
- **run**: The run ID (as inferred from the `phpbench history` command).
- **group**: The group name.
- **param**: Query a parameter value, parameter name in square brackets.

Parameters may be queried with the *param* field - the parameter name should be enclosed in square brackets as follows:

```
param[nb_elements]: 10

param[points]: { $gt: 50 }
```

Archiving

Archiving provides a way to export and reimport data from and to the configured storage. This allows you to:

- Backup your results (for example to a GIT repository).
- Migrate to other storage drivers.

By default PHPBench is configured to use an `XML` archiver, which will dump results to a directory in the current working directory, `_archive`.

To archive:

```
$ phpbench archive
```

To restore:

```
$ phpbench archive --restore
```

Both operations are idempotent - they will skip any existing records.

You may configure a different archiver in the configuration:

```
{
    "archiver": "xml"
}
```

Environment

PHPBench will try and record as much information about the current environment as it can. This is facilitated by "environment provider" classes which implement the `PhpBench\\Environment\\ProviderInterface` and are registered with the `environment_provider` tag in the DI container.

This information is recorded in the XML document:

```
<env>
  <uname os="Linux" host="dtlt410" release="4.2.0-1-amd64" version="#1 SMP Debian 4.2.
↪6-1 (2015-11-10)" machine="x86_64"/>
  <php version="5.6.15-1"/>
  <unix-sysload l1="1.04" l5="0.63" l15="0.55"/>
  <vcs system="git" branch="env_info" version=
↪"edde9dc7542cfa8e3ef4da459f0aaa5dfb095109"/>
</env>
```

This information can be readily viewed with the *env* report and can also be displayed when using the *table report generator*.

GIT

Class: `PhpBench\\Environment\\Provider\\Git`. **Available**: When PHPBench is run in the *root* directory of a GIT repository.

The GIT provider will provide VCS information, including the branch and vesion (i.e. the `commitsh`).

PHP

Class: `PhpBench\\Environment\\Provider\\Php`. **Available**: Always

Provides the PHP version.

Uname

Class: `PhpBench\\Environment\\Provider\\Uname`. **Available**: Always

Provides information about the operating system obtained through the php_uname command.

Unix Sysload

Class: `PhpBench\\Environment\\Provider\\UnixSysload`. **Available**: On non-windows systems.

Provides the CPU load for the following time periods: 1 minute, 5 minutes and 15 minutes.

Baseline

Class: `PhpBench\Environment\Provider\Baseline` **Available**: Always

Provides baseline measurements, by default it will provide mean times for executing the following micro-benchmarks (1000 revolutions):

- `nothing`: An empty method.
- `md5`: Calculation of an MD5 hash.
- `file_rw`: File read and write.

These measurements can help determine the relative speed of the system under test compared to other systems.

Report Generators

PHPBench generates reports using report generators. These are classes which implement the `PhpBench\Report\GeneratorInterface` and produce a report XML document which will later be rendered by using a *renderer* (the `console` renderer by default).

This chapter will describe the default report generators.

table

The table generator is the main report generator - it is the generator that allows you to analyze your benchmarking results.

Class: `PhpBench\Report\Generator\TableGenerator`.

Options:

- **title**: *(string)* Title of the report.
- **description**: *(string)* Description of the report.
- **cols**: *(array)* List of columns to display, see below.
- **break**: *(array)* List of columns; break into multiple tables based on specified columns.
- **compare**: *(string)* Isolate and compare values (default `mean` time) based for the given column.
- **compare_fields**: *(array)* List of fields to compare based on the column specified with **compare**.
- **diff_col**: *(string)* If the `diff` column is given in `cols`, use this column as the value on which to determine the `diff` (default `mean`).
- **sort**: *(assoc_array)* Sort specification, can specify multiple columns; e.g. `{ mean: "asc", benchmark: "desc" }`.
- **pretty_params**: *(boolean)* Pretty print the `params` field.
- **iterations**: *(boolean)* Include the results of every individual iteration (default `false`).
- **labels**: *(array)* Override the default column names, either as a numerical array or as a *colName* => *label* hash.

Columns

Here we divide the columns into three sets, *conditions* are those columns which determine the execution context, *variant statistics* are aggregate statistics relating to a set of iterations and *iteration statistcs* relate to single iterations (as provided when `iterations` option is set to `true`).

Conditions:

- **suite**: Identifier of the suite.
- **date**: Date the suite was generated,
- **stime**: Time the suite was generated
- **benchmark**: Short name of the benchmark class (i.e. no namespace).
- **benchmark_full**: Fully expanded name of benchmark class.
- **subject**: Name of the subject method.
- **groups**: Comma separated list of groups.

- **params**: Parameters (represented as JSON).
- **revs**: Number of revolutions.
- **its**: Number of iterations.

Variant Statistics:

- **mem_peak**: (mean) Peak memory used by each iteration as retrieved by memory_get_peak_usage.
- **mem_final**: (mean) Memory allocated to PHP at the end of the benchmark (*memory_get_usage*).
- **mem_real**: (mean) Memory allocated by the system for PHP at the end of the benchmark (*memory_get_usage(true)*).
- **min**: Minimum time of all iterations in variant.
- **max**: Maximum time of all iterations in variant.
- **worst**: Synonym for `max`.
- **best**: Synonym for `min`.
- **sum**: Total time taken by all iterations in variant,
- **stdev**: Standard deviation
- **mean**: Mean time taken by all iterations in variant.
- **mode**: Mode of all iterations in variant.
- **variance**: The variance of the variant.
- **rstdev**: The relative standard deviation.

Iteration Statistics:

- **mem_peak**: Peak memory used by each iteration as retrieved by memory_get_peak_usage.
- **mem_final**: Memory allocated to PHP at the end of the benchmark (*memory_get_usage*).
- **mem_real**: Memory allocated by the system for PHP at the end of the benchmark (*memory_get_usage(true)*).
- **iter**: Index of iteration.
- **rej**: Number of rejections the iteration went through (see *Retry Threshold*.
- **time_net**: Time in (microseconds) it took for the iteration to complete.
- **time_rev**: Time per revolution (`time_net / nb revs`).
- **z-vaue**: The number of standard deviations away from the mean of the iteration set (the variant).

In addition any number of environment columns are added in the form of `<provider_name>_<key>`, so for example the column for the VCS branch would be `vcs_branch`.

composite

This report generates multiple reports.

Class: `PhpBench\Report\Generator\CompositeGenerator`.

Options:

- **reports**: *(array)*: List of report names.

env

This is a simple generator which generates a report listing all of the environmental factors for each suite.

Class: `PhpBench\Report\Generator\EnvGenerator`.

Options:

- **title**: *(string)* Title of the report.
- **description**: *(string)* Description of the report.

Report Renderers

Reports are rendered to an output medium using classes implementing the `PhpBench\Report\RendererInterface`.

The configuration for a renderer is known here as an *output*. The user may define new outputs either in the *configuration* file or on the CLI. The renderer may also supply default outputs.

console

Renders directly to the console.

Class: `PhpBench\Report\Renderer\ConsoleRenderer`.

Options:

- **table_style**: *(string)* Table style to use, one of: `default`, `compact`, `borderless` or `symfony-style-guide`.

Default outputs:

- `console`: Renderers the report directly to the console. This is the **default** output method.

xslt

The XSLT renderer the path to an XSLT template which will be used to transform the report XML document into an output *file*.

Class: `PhpBench\Report\Renderer\XsltRenderer`.

Options:

- **title**: *(string)*: Title to use for the document (where applicable).
- **template**: *(string)*: Path to the XSL template.
- **file**: *(string)*: Path to the output file (existing files will be overwritten). You can use the `%report_name%` token, it will be replaced with the name of the report.

Default outputs:

- `html`: Render the report as a single HTML page.
- `markdown`: Render the report as a GitHub Flavored Markdown document.

delimited

The delimited renderer outputs the report as a delimited value list (for example a tab separated list of values). Such data can be easily imported into applications such as GNUPlot.

Class: `PhpBench\Report\Renderer\DelimitedRenderer`.

Options:

- **delimiter**: *(string)*: Path to the output file (existing files will be overwritten).
- **header**: *(boolean)*: If a header should be included in the output.

Default outputs:

- `delimiter`: The delimiter to use.

debug

Output the raw XML of the report document. Useful for debugging.

Options:

none

Default outputs:

- `debug`: Outputs the report document's XML.

Configuration

Unless overridden with the `--config` option, PHPBench will try to load its configuration from the current working directory. It will check for the existence each of the files `phpbench.json` and `phpbench.json.dist` in that order and use one if it exists.

```
{
    "bootstrap": "vendor/autoload.php",
    "path": "path/to/benchmarks",
    "outputs": {
         "my_output": {
             "extends": "html",
             "file": "my_report.html",
             "title": "Hello World"
         }
    },
    "reports": {
        "my_report": {
            "extends": "aggregate",
            "exclude": ["benchmark"]
        }
    }
}
```

Note: Typically you should use `phpbench.json.dist` in your projects. This allows the end-user of your library to override your configuration by creating `phpbench.json`.

Bootstrap

You can include a single file, the bootstrap file, before the benchmarks are executed. Typically this will be the class autoloader (e.g. `vendor/autoload.php`).

It is specified with the `bootstrap` key:

```
{
    "bootstrap": "vendor/autoload.php",
}
```

Note: You can override (or initially set) the bootstrap using the `--bootstrap` CLI option.

Path

Specify the default path to the benchmarks:

```
{
    "path": "tests/benchmarks"
}
```

Progress Logger

Specify which progress logger to use:

```
{
    "progress": "dots"
}
```

Retry Threshold

Set the *Retry Threshold*:

```
{
    "retry_threshold": 5
}
```

Reports

List of report definitions:

```
{
    "reports": {
        "my_report": {
            "extends": "aggregate",
            "exclude": ["benchmark"]
        }
    }
}
```

The key is the name of the report that you are defining, and the object properties are the options for the report. Each report must specify either the `generator` or `extends` key, specifying the *generator* or report to extend respectively.

See the *Report Generators* chapter for more information on report configuration.

Outputs

Custom output definitions:

```
"outputs": {
     "my_output": {
         "extends": "html",
         "file": "my_report.html",
         "title": "Hello World"
     }
}
```

Note that:

- The key of each definition is the output name.
- As with reports, each definition *MUST* include either the `renderer` or `extends` key.
- All other options are passed to the renderer as options.

See the *Report Renderers* chapter for more information.

Time Unit and Mode

Specify the *default time unit*. Note that this will be overridden by individual benchmark/subjects and when the `time-unit` option is passed to the CLI.

```
{
    "time_unit": "milliseconds"
}
```

Similarly the *Mode: Throughput Representation* can be set using the *output_mode* key:

```
{
    "output_mode": "throughput"
}
```

PHP Binary and INI settings

You can change the PHP binary and INI settings used to execute the benchmarks:

```
{
    "php_binary": "hhvm",
    "php_config": {
        "memory_limit": "10M"
    }
}
```

Prefixing the Benchmarking Process

You can prefix the benchmarking command line using the `php_wrapper` option:

```
{
    "php_wrapper": "blackfire run"
}
```

Note: This can also be set using the `--php-wrapper` CLI option.

Customizing the subject matching pattern

By default PHPBench will identify subject methods when they have a `bench` prefix. It is possible to change the regex pattern used to identify subjects as follows:

```
{
    "subject_pattern": "^spin_"
}
```

The above will allow you to have benchmark class such as:

```
<?php

class Foobar
{
    public function spin_kde()
    {
        // ...
    }

    public function spin_lcd()
    {
        // ...
    }
}
```

Note: You can also explicitly declare that methods are benchmark subjects by using the `@Subject` annotation.

FAQ

Why does PHPBench slow on Windows?

Process spawning on Windows is more expensive than on Linux, PHPBench spawns many processes. Actual benchmarking time however is not affected.

Why does PHPBench look terrible on Windows?

PHPBench makes use of ansi escape sequences in most of its progress loggers. The default Windows console does not support these sequences, so the output can look very bad.

You can mitigate this by using the *travis* logger, which does not issue any of these escape sequences.

You may also consider using *Cgywin*, *emuCon* or *ansiCon* programs to enhance your console. You may also switch to Linux.

Why do *setUp* and *tearDown* methods not automatically get called?

PHPBench supports the annotations `BeforeMethods` and `AfterMethods` which can be placed at the class level and/or the method level. These methods are plural. If we were to automatically add `setUp` to the chain then the annotation would read read one thing, but the benchmark would do another (i.e. execute the method indicated by the annotation and the "magic" setUp method).

If you want to support `setUp` and `tearDown` you can create a simple base class such as:

```
/**
 * @BeforeMethods({"setUp"})
 * @AfterMethods({"tearDown"})
 */
abstract class BenchmarkCase
{
    public function setUp()
    {
    }

    public function tearDown()
    {
    }
}
```

CHAPTER 2

Extensions

XDebug

The XDebug extension allows you to easily profile your code using function traces or by generating cachegrind profiles.

The commands are very similar to the standard `run` command with the difference that only single iterations are performed.

Note: XDebug needs to be installed, however it does NOT need to be activated by default. PHPBench will automatically try and load and configure the extension even if it is disabled.

Tip: You can use the XDebug executors with the standard `run` command by specifying either `xdebug_profile` or `xdebug_trace` with the `--executor` option.

Installation

The XDebug extension is bundled with PHPBench, it just needs to be activated:

```
{
   "extensions": {
       "PhpBench\\Extensions\\XDebug\\XDebugExtension"
   }
}
```

Alternatively you can activate it directly from the CLI using the `extension` option:

```
$ phpbench xdebug:profile examples/HashBench.php --extension=
↪"PhpBench\\Extensions\\XDebug\\XDebugExtension"
```

Function Tracing

Function tracing provides a simple way to profile your benchmark subjects:

```
$ ./bin/phpbench xdebug:trace benchmarks/Micro/Math/StatisticsBench.php

+-----+-------+----------+-----------+-----------+-----------------------------------
↪---+-------------------------------------------------+
| #   | Level | Mem      | Time      | Time inc. | Function
↪    | File                                            |
+-----+-------+----------+-----------+-----------+-----------------------------------
↪---+-------------------------------------------------+
| 777 | 4     | 922,976b | 0.015284s | 192µs     | PhpBench\Math\Statistics::stdev()
↪    | ./benchmarks/Micro/Math/StatisticsBench.php:40 |
| 778 | 5     | 923,024b | 0.015294s | 168µs     |
↪PhpBench\Math\Statistics::variance() | ./lib/Math/Statistics.php:29
↪   |
| 779 | 6     | 923,024b | 0.015304s | 34µs      |   PhpBench\Math\Statistics::mean()
↪    | ./lib/Math/Statistics.php:44                    |
| 780 | 7     | 923,072b | 0.015313s | 9µs       |    array_sum()
↪    | ./lib/Math/Statistics.php:73                    |
| 781 | 7     | 923,120b | 0.015326s | 8µs       |    count()
↪    | ./lib/Math/Statistics.php:79                    |
+-----+-------+----------+-----------+-----------+-----------------------------------
↪---+-------------------------------------------------+
```

You can enable showing arguments using the `--show-args` option:

```
$ ./bin/phpbench xdebug:trace benchmarks/Micro/Math/StatisticsBench.php --show-args
+-----+-------+----------+-----------+-----------+-----------------------------------
↪-------------------------------------------+-----------------------------------
↪-----------+
| #   | Level | Mem      | Time      | Time inc. | Function
↪                                           | File
↪           |
+-----+-------+----------+-----------+-----------+-----------------------------------
↪-------------------------------------------+-----------------------------------
↪-----------+
| 777 | 4     | 922,912b | 0.021977s | 244µs     |
↪PhpBench\Math\Statistics::variance(                                                | .
↪/benchmarks/Micro/Math/StatisticsBench.php:33 |
|     |       |          |           |           |  array (0 => 10, 1 => 100, 2 => 42,
↪ 3 => 84, 4 => 11, 5 => 12, 6 => 9, 7 => 6)    |
↪           |
|     |       |          |           |           |  ???
↪                                          |
↪           |
|     |       |          |           |           | )
↪                                          |
↪           |
| 778 | 5     | 922,960b | 0.021992s | 49µs      |  PhpBench\Math\Statistics::mean(
↪                                          | ./lib/Math/Statistics.php:44
↪           |
|     |       |          |           |           |   array (0 => 10, 1 => 100, 2 =>
↪42, 3 => 84, 4 => 11, 5 => 12, 6 => 9, 7 => 6)  |
↪            |
|     |       |          |           |           |  )
↪                                          |
↪           |
```

```
| 779 | 6      | 923,008b | 0.022005s | 13μs       |   array_sum(
↪                                                  | ./lib/Math/Statistics.php:73
↪             |
|     |        |          |           |            |    array (0 => 10, 1 => 100, 2 =>
↪42, 3 => 84, 4 => 11, 5 => 12, 6 => 9, 7 => 6) |
↪             |
|     |        |          |           |            |   )
↪                                                  |
↪             |
| 780 | 6      | 923,056b | 0.022024s | 11μs       |   count(
↪                                                  | ./lib/Math/Statistics.php:79
↪             |
|     |        |          |           |            |    array (0 => 10, 1 => 100, 2 =>
↪42, 3 => 84, 4 => 11, 5 => 12, 6 => 9, 7 => 6) |
↪             |
|     |        |          |           |            |   )
↪                                                  |
↪             |
+-----+--------+----------+-----------+------------+-------------------------------------
↪--------------------------------------------------+-------------------------------------
↪-------------+
```

Profiling (cachegrind)

The profile command is very similar to the run command:

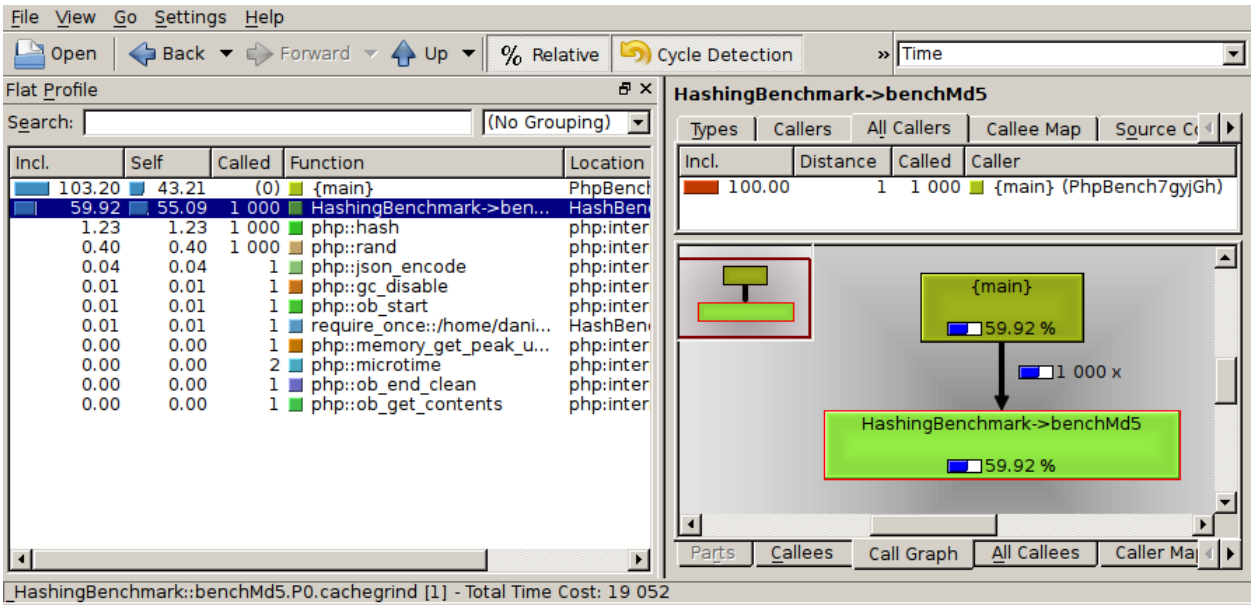
```
$ phpbench xdebug:profile examples/HashBench.php --progress=none

3 profile(s) generated:

    profile/_HashingBenchmark::benchMd5.P0.cachegrind
    profile/_HashingBenchmark::benchSha1.P0.cachegrind
    profile/_HashingBenchmark::benchSha256.P0.cachegrind
```

A single profile is generated for each subject in the benchmark and placed in the directory `profile` by default.

The generated profiles can be viewed with a cachegrind viewer such as *kcachgrind* (linux) or webgrind (web based).



It is possible to automatically launch a GUI for each of the profiles using the `--gui` option. By default PHPBench will attempt to locate the `kcachegrind` executable. If you do not have `kcachegrind` you can specify a different executable using the `--gui-bin` option.

DBAL

The DBAL extension provides a storage driver for storing results to any database supported by doctrine dbal. By default it will use a file-based sqlite database which will be created in your current working directory and named `.phpbench.sqlite`.

Currently it will only store basic metrics for each iteration, *time*, *memory* (peak), *z-value*, *deviation* and the summary statistics.

> **Warning:** The DBAL extension does not provide an advanced storage capability and may be dropped from the core before the 1.0 release. In this case it will be available as an officially unsupported extension in a separate repository.

Installation

The DBAL depends on the `doctrine/dbal` package. If you are using PHPBench as a dependency of your project you will need to ensure that you have this package installed, install it with composer:

```
$ composer require --dev "doctrine/dbal"
```

You will then need to enable the extension (which is bundled with PHPBench):

```
{
    "storage": "dbal",
    "extensions": {
        "PhpBench\\Extensions\\Dbal\\DbalExtension"
    }
}
```

In addition to adding the extension we also set `sqlite` as our storage driver.

Configuration

You may configure the dbal with the `storage.dbal.connection` key in your `phpbench.json` file. For example, to change the path of the sqlite database:

```
{
    ...
    "storage.dbal.connection": {
        "driver": "pdo_sqlite",
        "path": "/path/to/database"
    }
}
```

Or to use another database driver, e.g. `mysql`:

```
{
    ...
    "storage.dbal.connection": {
        "driver": "pdo_mysql",
        "dbname": "myproject_benchmarks",
        "user": "root",
    }
}
```

If you are not using the `pdo_sqlite` driver, you will need to initialize the database using the `dbal:migrate` command detailed below.

Warning: Multiple projects cannot currently share the same sqlite database.

Migration

The `dbal:migrate` command will update the database schema to the latest version, if the database is "new" then it will create the schema.

Important: Whilst it is normally safe to run this command, there is no guarantee that your data will be migrated properly. If you care about your data, then it is advisable to *archive* your data before migrating the database.

Running the command with no options will tell you how many operations need to be executed on the database. To actually migrate you need to supply the `--force` option:

```
$ phpbench dbal:migrate --force
```

You may also manually inspect the statements that will be executed using the `--dump-sql` option:

```
$ phpbench dbal:migrate --dump-sql
```

Custom Extensions

Note: Check out the example extension here.

Writing custom extensions is quite easy, it is necessary to create a container extension and then register that extension in your configuration:

```
<?php

namespace Acme\PhpBench\Extension\Example;

use PhpBench\DependencyInjection\ExtensionInterface;
use PhpBench\DependencyInjection\Container;
use Acme\PhpBench\Extension\Example\Command\FooCommand;
use Acme\PhpBench\Extension\Example\Progress\FooLogger;

class ExampleExtension implements ExtensionInterface
{
    public function getDefaultConfig()
    {
        // default configuration for this extension
        return [
            'acme.example.message' => 'Hello World',
            'acme.progress.character' => 'x'
        ];
    }

    public function load(Container $container)
    {
        // register a command
        $container->register('acme.example.foo', function (Container $container) {
            return new FooCommand(
                $container->getParameter('acme.example.message')
            );
        }, ['console.command' => []]);

        // register a progress logger
        $container->register('acme.example.progress_logger', function (Container
↪$container) {
            return new FooLogger($container->get('benchmark.time_unit'));
        }, ['progress_logger' => ['name' => 'foo']]);

    }

    // called after load() can be used to add tagged services to existing services.
    public function build(Container $container)
    {
    }
}
```

Note: The third argument of the `register` method, this is a list of **tags**. Tags tell PHPUnit what these services are and how to use them. Checkout the CoreExtension to investigate all of the available tags.

and activate the extension in your `phpbench.json` file:

```
{
    "extensions": [
```

```
        "Acme\\PhpBench\\Extension\\Example\\ExampleExtension"
    ]
}
```

PHPBench as a project dependency

If you are using PHPBench as a `require-dev` dependency of your project, and the extension is in your projects autoloader, then you are done, congratulations!

PHPBench as a PHAR

If you are using the PHAR version of PHPBench then you will need to tell PHPBench where it can find an autoloader for your extension (or extensions):

```
{
    "extension_autoloader": "/home/daniel/www/phpbench/phpbench-example-extension/
↪vendor/autoload.php"
}
```

If you have multiple extensions you may consider creating an "extension project" e.g.

```
$ mkdir phpbench-extensions
$ cd phpbench-extensions
$ composer require vendor/my-phpbench-extension-1
$ composer require vendor/my-phpbench-extension-2
```

and then using the `autoload.php` of this project.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search