Profiling PHP applications with PhpStorm and Xdebug

This tutorial describes profiling PHP applications PhpStorm and Xdebug. If you prefer using Zend Debugger, see Profiling PHP applications with PhpStorm and Zend Debugger.

Using PhpStorm, we can analyze performance of our PHP code by profiling it. Profiling allows us to gather program execution statistics such as the names of functions executed, the number of times a function has been executed, how long a function took to execute, which other functions have been called into, and so on. This information can give us a hint on where our code can be improved.

Let's see how this works.

Requirements

PhpStorm makes use of Xdebug to collect profiler information. This debugging engines must be installed and configured on our system. See Xdebug Installation Guide for more information.

1. Enabling the Xdebug profiler

Profiling adds some overhead to the running application and generates a huge amount of information on disk. Therefore, it's best to only enable the Xdebug profiler when needed and disable it afterwards.

To enable the Xdebug profiler globally, edit the active php.ini file and add the following directives:

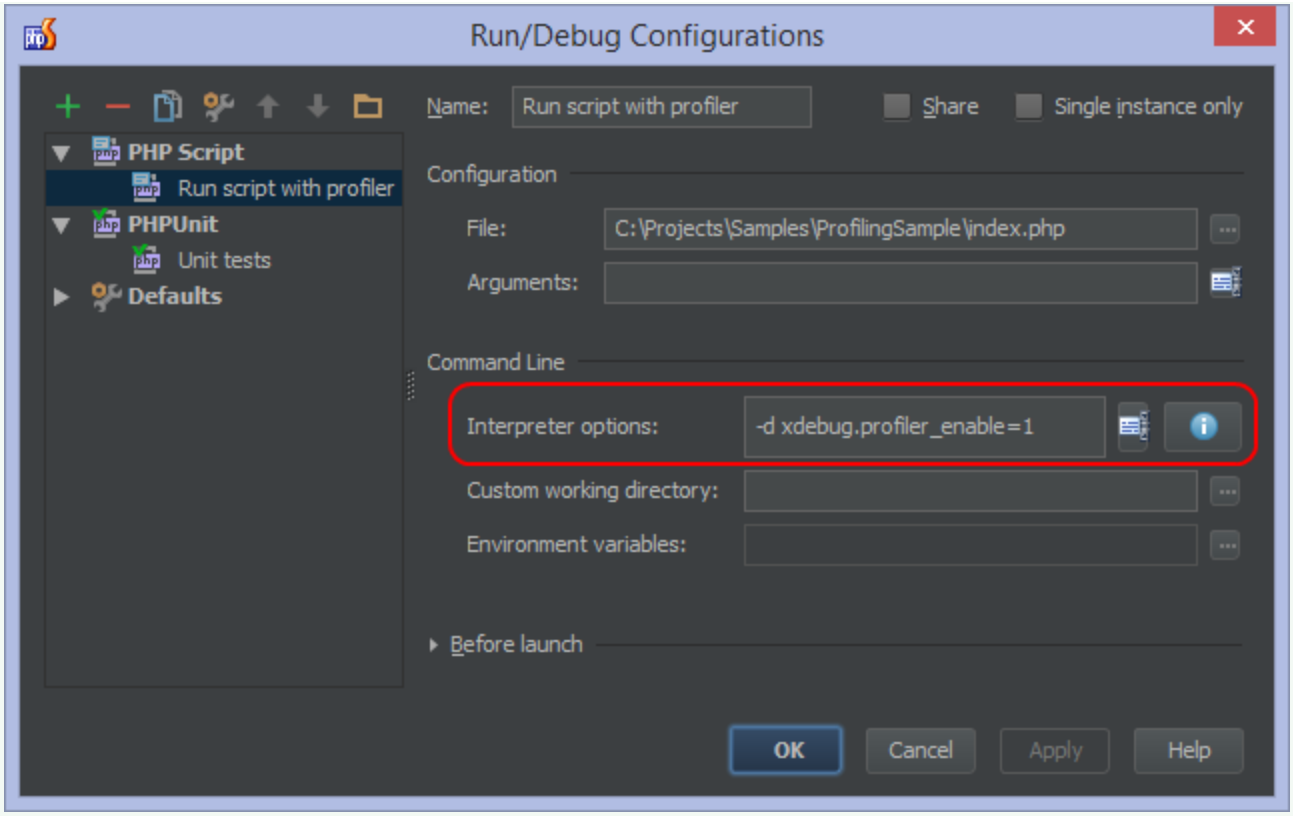
```
xdebug.profiler_enable=1
xdebug.profiler_output_dir=/path/to/store/snapshots
xdebug.profiler_enable_trigger=1
```

xdebug.profiler_enable enables or disables the profiler. xdebug.profiler_output_dir specifies where profiler snapshot data should be stored. If this setting is not specified, Xdebug will default to the /tmp folder.

A good approach to configuring the Xdebug profiler is to set xdebug.profiler_output_dir in php.ini and set xdebug.profiler_enable when needed. This can be done by specifying additional interpreter options in the PhpStorm Run Configuration:

-d xdebug.profiler_enable=1.

This will enable the profiler for the configuration but not for other configurations.



Xdebug features two ways of enabling the profiler, depending on the type of application we want to profile. Setting the xdebug.profiler_enable directive enables profiling for any application. For web applications, profiling can be enabled on demand by specifying a special GET/POST variable or a cookie. This can be done using the PhpStorm bookmarklets (or one of the Browser Debugging Extensions) and setting the xdebug.profiler_enable_trigger directive to 1 in php.ini.

2. Capturing a profiler snapshot

In order to be able to analyze information captured by the profiler, we first have to collect that information.

2.1. Capturing a profiler snapshot for web applications

To profile a web application, either enable the Xdebug profiler globally or use the PhpStorm bookmarklets (or one of the Browser Debugging Extensions) to start and stop the profiler on demand. Next, open the application in a browser to start collecting profiler data.

When analyzing a performance issue, using the bookmarklets or a Browser Debugging Extension is a very good approach: we can navigate through our application and only enable the profiler when using the feature in which a performance issue is occurring. This allows for capturing a targeted profiler snapshot.

2.2. Capturing a profiler snapshot for CLI applications and unit tests

To profile CLI applications and unit tests, either enable the Xdebug profiler globally or create a separate Run Configuration which enables the profiler. Next, run the application or unit tests to start collecting profiler data.

When analyzing a performance issue in unit tests, a good approach is to create a separate Run Configuration which only runs the unit tests in which a performance issue is suspected. This allows for capturing a targeted profiler snapshot.

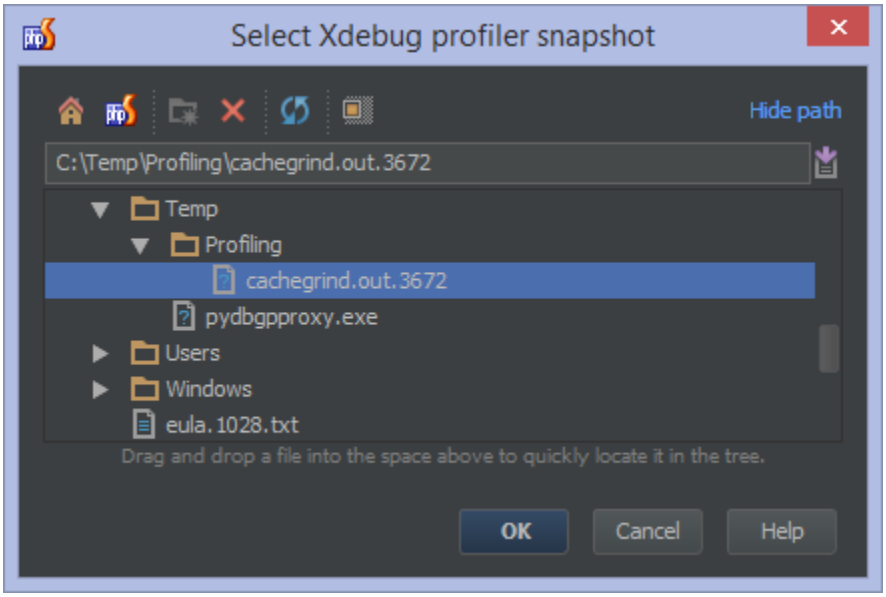
3. Analyzing a profiler snapshot

Let's examine a profiler snapshot.

3.1. Open the profiler snapshot

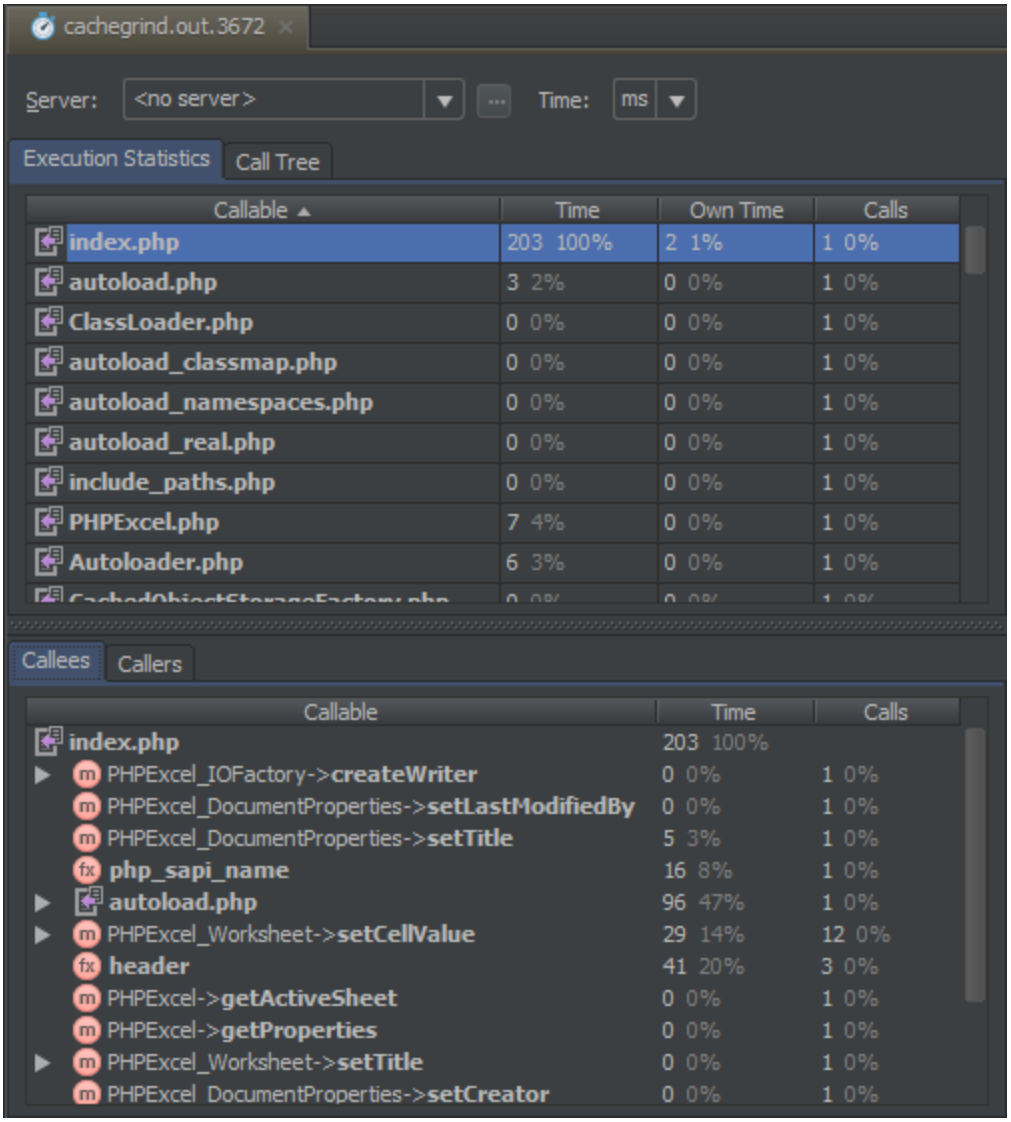
Using the Tools | Analyze XDebug profiler snapshot menu, we can open a profiler snapshot.

Snapshots are stored in the directory configured in php.ini (or /tmp if not configured explicitly). The name of the generated file always starts with "cachegrind.out." and ends with either the process ID of the PHP or webserver process or a crc32 hash of the directory containing the initially debugged script.



3.2. Execution Statistics tab

In the Execution Statistics view, we can examine the summary information about execution metrics of every called function. We can see all files, function calls, the number of times they have been called, and the time (absolute and relative) they took to execute.



The top grid shows us different metrics:

- Callable - the file that has been executed
- Time - the total execution time
- Own Time - the amount of time the function spent executing its own code (excluding calls to other functions)
- Calls - the number of calls

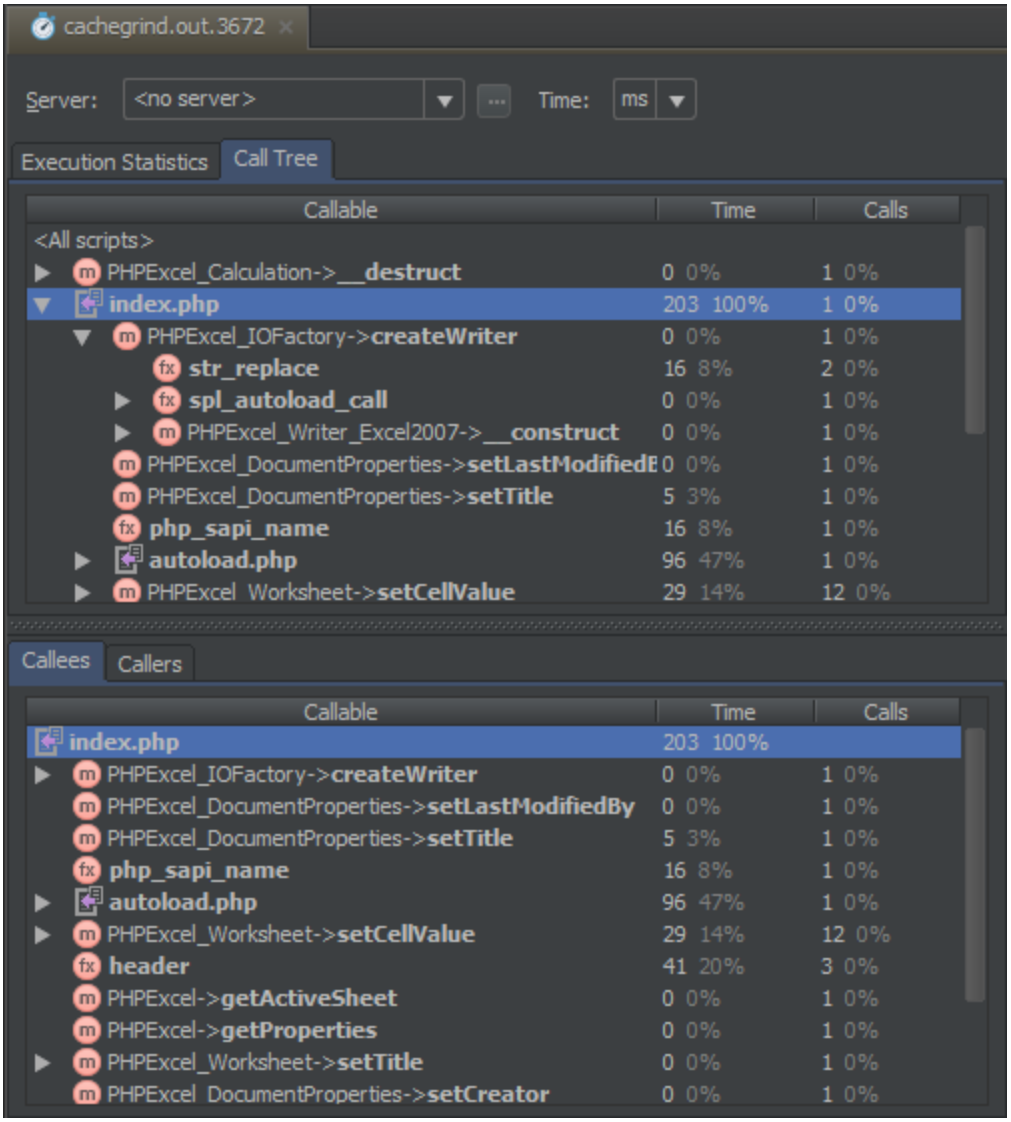
The bottom grid has two tabs, Callees (which functions the script calls into) and Callers (from where the script is called). We can see various metrics here as well:

- Callable - the function that has been executed
- Time - the total execution time
- Calls - the number of calls

Functions with high own times and/or large number of calls are definitely worth inspecting.

3.3. Call tree tab

The Call Tree view shows us the execution paths of our code. We can see more details about function execution times and so on.



The top grid shows us call trees (which function calls into which function) and different metrics:

- Callable - the file that has been executed
- Time - the total execution time
- Calls - the number of calls

The bottom grid has two tabs, Callees (which functions are being called) and Callers (from where the function is called). We can see various metrics here as well:

- Callable - the function that has been executed
- Time - the total execution time
- Calls - the number of calls