Comodojo daemon Documentation

Release 1.0.0

Marco Giovinazzi

Apr 06, 2019

Contents

This library provides tools to create, control and interact with complex, multi-process PHP daemons.

Table of Contents:

CHAPTER 1

General concepts

This library provides basic tools to create solid PHP daemons that can:

- spawn and control multiple workers,
- communicate via unix/inet sockets using structured RPC calls,
- receive and handle POSIX signals using a signal-to-event bridge, and
- maintain small memory footprint.

The following picture shows the high level architecture of the comodojo/daemon package.

1.1 The big picture

According to wikipedia:

> [...] a daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user.

Starting from the ground up, the structure of this library reflects the above definition: the `\Comodojo\Daemon\Process` abstract class provides all the basic methods to create a standard *nix process that can handle OS signals and set its own niceness.

The `\Comodojo\Daemon\Daemon` abstract class extends the previous one with all the fancy daemon features. When extended and instantiated, this class, basically:

- forks itself and close the parent process (to became an orphaned process)
- detaches from STDOUT, STDERR, STDIN and became a session leader
- creates and inject event listeners to react to common *nix signals (SIGTERM, SIGINT, SIGCHLD)
- creates a communication socket
- start the internal daemon loop

Creating a simple echo daemon this way requires just a couple of lines:

```
1 <?php namespace DaemonExamples;
2
3 use \Comodojo\Daemon\Daemon as AbstractDaemon;
```

(continues on next page)

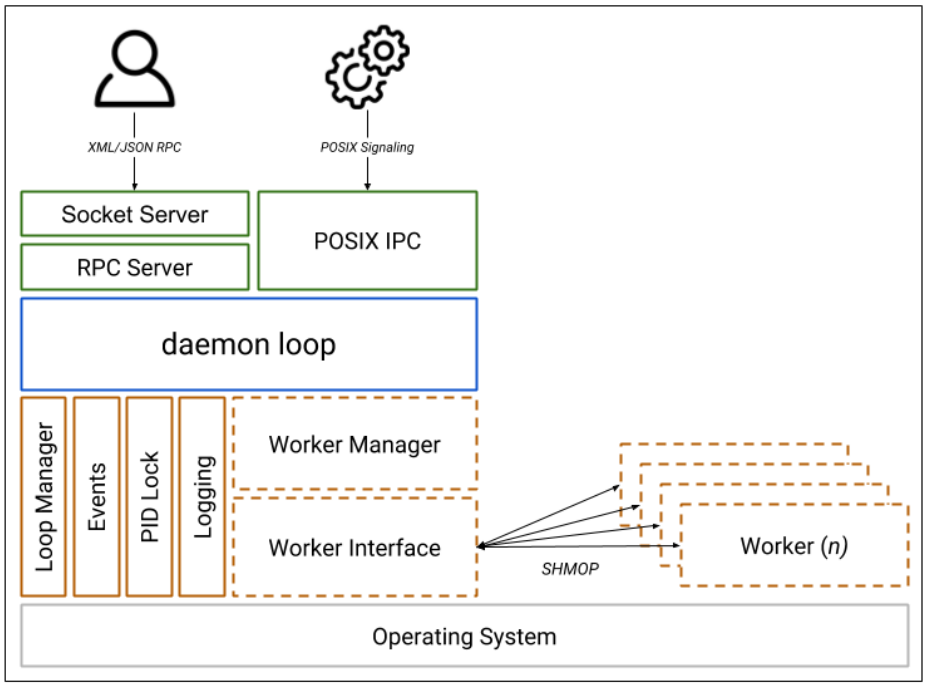

Fig. 1: comodojo/daemon v1.X architecture

(continued from previous page)

```php
use \Comodojo\RpcServer\RpcMethod;

class EchoDaemon extends AbstractDaemon {

    public function setup() {

        // define the echo method using lambda function
        $echo = RpcMethod::create("examples.echo", function($params, $daemon) {
            $message = $params->get('message');
            return $message;
        }, $this)
            ->setDescription("I'm here to reply your data")
            ->addParameter('string','message')
            ->setReturnType('string');

        // inject the method to the daemon internal RPC server
        $this->getSocket()
            ->getRpcServer()
            ->methods()
            ->add($echo);

    }

}
```

Note: This code is available in the daemon-examples github repository.

1.2 Daemon loop

The daemon itself is designed to handle communication via socket or at the OS level. That's why the main loop in comodojo/daemon is implemented ad the socket level, i.e. the daemon loop endlessly waiting for incoming connections. Once received, the socket calls the internal RPC server to execute the command (if any). This behaviour can not be changed.

Note: See comodojo/rpcserver github repo for more information about RPC server.

1.3 Socket communication

TBW

1.4 POSIX signals and signal-to-event bridge

Once received, a POSIX signal is automatically converted into a `\Comodojo\Daemon\Events\PosixEvent` event that will fire hooked listeners. In this way the framework can be customized to react to specific events according to user needs.

Predefined listeners are in place to handle most common system events; the `\Comodojo\Daemon\Listeners\StopDaemon`, for example, is designed to react on SIGTERM and to close the daemon gracefully.

1.5 Workers and Worker management

Workers are the standard way to create extended logic inside a project based on comodojo/daemon.

A worker is a child process, forked from the daemon, that implements another kind of loop; the daemon itself constantly monitors the status of the worker and keeps an always open bidirectional communication channel using shared memory segments (SHMOP).

In other words, a worker can actually do a "specialized work" independently from the parent process, without exposing another socket, relying on the daemon for external communications.

CHAPTER 2

Installation

First install composer, then:

```
composer require comodojo/daemon
```

2.1 Requirements

To work properly, comodojo/daemon requires PHP >=5.6.0.

Following PHP extension are also required:

- ext-posix: PHP interface to *nix Process Control Extensions
- ext-pcntl: process Control support in PHP
- ext-shmop: read, write, create and delete Unix shared memory segments
- ext-sockets: low-level interface to the socket communication functions

CHAPTER 3

Using the library

Creating a daemon using this library requires at least two steps:

1. create your own daemon class, defining methods to be exposed via RPC socket,
2. create the daemon exec file, that will init the above mentioned class providing basic configuration.

Workers can be also injected to the daemon in the second step.

3.1 Defining the daemon

Your new daemon should extend the `\Comodojo\Daemon\Daemon` abstract class, implementing the abstract `setup` method.

The main purpose of this method is to define all the commands that the daemon will accept from the input socket.

Let's take as an example the dummy *echo* daemon mentioned in *General concepts* section:

```php
<?php namespace DaemonExamples;

use \Comodojo\Daemon\Daemon as AbstractDaemon;
use \Comodojo\RpcServer\RpcMethod;

class EchoDaemon extends AbstractDaemon {

    public function setup() {

        // define the echo method using lambda function
        $echo = RpcMethod::create("examples.echo", function($params, $daemon) {
            $message = $params->get('message');
            return $message;
        }, $this)
            ->setDescription("I'm here to reply your data")
            ->addParameter('string','message')
            ->setReturnType('string');

        // inject the method to the daemon internal RPC server
        $this->getSocket()
            ->getRpcServer()
```

(continues on next page)

(continued from previous page)

```
22              ->methods()
23              ->add($echo);
24
25      }
26
27  }
```

Note: This code is available in the daemon-examples github repository.

The *examples.echo* RPC method expects a string parameter *message* that will be replied by the server.

Now that we have our first daemon, let's figure out how to start it.

3.2 Creating the exec script

The exec script typically provides only the basic configuration to the daemon class.

Following an example exec script that init the daemon using an inet/tpc socket on port 10042.

```
1  #!/usr/bin/env php
2  <?php
3
4  $base_path = realpath(dirname(__FILE__)."/../");
5  require "$base_path/vendor/autoload.php";
6
7  use \DaemonExamples\EchoDaemon;
8
9  $configuration = [
10     'description' => 'Echo Daemon',
11     'sockethandler' => 'tcp://127.0.0.1:10042'
12 ];
13
14 // Create a new instance of EchoDaemon
15 $daemon = new EchoDaemon($configuration);
16
17 // Start the daemon!
18 $daemon->init();
```

Note: This code is available in the daemon-examples github repository.

Note: for a complete list of configuration parameters, refer to the *Daemon configuration* section.

Once saved and made executable, the daemon is ready start.

3.3 Running the daemon

If called with no arguments, the exec script will present the default daemon console:

The *-d* (run as a daemon) and the *-f* (run in foreground) arguments are the most important to understand. If *-d* is selected, the script will act as a daemon (forking itself, detaching from IO, ...), while the *-f* keeps the script in foreground and the standard shell IO.

So, it's trivial to understand that the main purpose of the *-f* argument is to enable the debug at run-time.

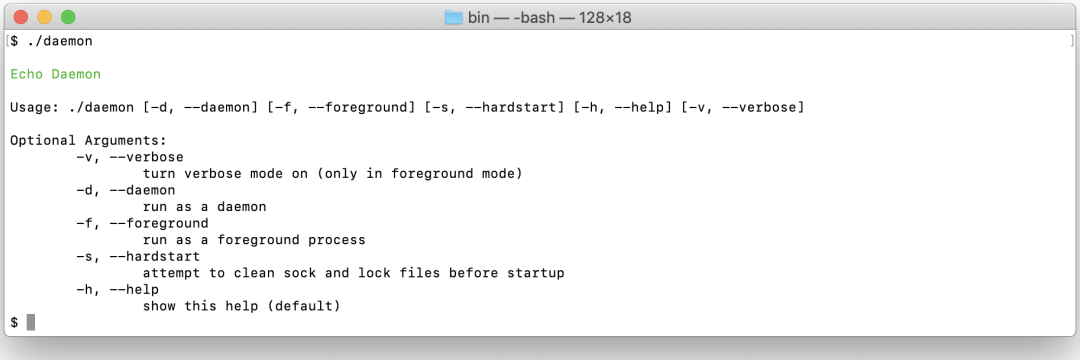

Fig. 1: comodojo/daemon default console

Two typical combination of arguments are the following:

- run the daemon, (eventually) cleaning the socket and the locker: *./daemon -d -s*
- run the daemon in foreground, enabling debug: *./daemon -f -v*

3.4 Interacting with the daemon

TBW

CHAPTER 4

Daemon configuration

A daemon created using this package can be configured using an array of parameters provided as the first input argument to the `\Comodojo\Daemon\Daemon` abstract class. As an example:

```
#!/usr/bin/env php
<?php

use \DaemonExamples\EchoDaemon;

$configuration = [
    'description' => 'Echo Daemon',
    'sockethandler' => 'tcp://127.0.0.1:10042'
];

// Create a new instance of EchoDaemon
$daemon = new EchoDaemon($configuration);
```

Note: This code is available in the **'daemon-examples github repository'_**.

4.1 Configuration parameters

Following a list of accepted configuration parameters.

4.1.1 sockethandler

Address and type of the socket handler (see the PHP socket documentation).

Example: 'sockethandler' => 'tcp://127.0.0.1:60001'

Default: 'sockethandler' => 'unix://daemon.sock'

4.1.2 pidfile

Location (relative to the base path) of the daemon's pid file.

Default: 'pidfile' => 'daemon.pid'

Note: Prepend a slash to the file loaction to make it absolute (e.g. /tmp/daemon.pid).

4.1.3 socketbuffer

Size of the socket buffer (see the PHP socket documentation).

Default: 'socketbuffer' => 1024

4.1.4 sockettimeout

Timeout for the select() system call (see the PHP socket documentation).

Default: 'sockettimeout' => 2

4.1.5 socketmaxconnections

Maximum number of connection accepted by the socket.

Default: 'socketmaxconnections' => 10

4.1.6 niceness

Define the nice value of the daemon process (see the nice unix command on wikipedia).

Default: 'niceness' => 0

4.1.7 arguments

Definition of command line arguments, in the climate format (see climate documentation).

Default: 'arguments' => '\Comodojo\Daemon\Console\DaemonArguments'

4.1.8 description

Description banner in the daemon command line.

Default: 'description' => 'Comodojo Daemon'

CHAPTER 5

Using Workers

In comodojo/daemon workers are, essentially, child processes that run in parallel maintaining a communication channel with the master daemon. Each worker has its own loop that can be configured from the daemon.

5.1 Creating a worker

The simplest way to create a worker, is to extend the `\Comodojo\Daemon\Worker\AbstractWorker` abstract class implementing the `loop()` method.

There are two other optional methods, `spinup()` and `spindown` that can be used to control the worker startup and execute action before shutting down.

As an example, let's consider the following *CopyWorker*: it's job is to check if a specific *test.txt" file exists in the *tmp* directory and, if it's there, duplicate the file.

```
<?php namespace DaemonExamples;

use \Comodojo\Daemon\Worker\AbstractWorker;

class CopyWorker extends AbstractWorker {

    protected $path;

    // Source file
    protected $file = 'test.txt';

    // Destination file
    protected $copy = 'copy_test.txt';

    public function spinup() {

        $this->logger->info("CopyWorker ".$this->getName()." spinning up...");
        $this->path = realpath(dirname(__FILE__)."/../../tmp/");

    }

    public function loop() {

```

(continues on next page)

(continued from previous page)

```
        $filename = $this->path."/".$this->file;

        if ( file_exists($filename) ) {
            copy($filename, $this->path."/".$this->copy);
        }

    }

    public function spindown() {

        $this->logger->info("CopyWorker ".$this->getName()." spinning down.");
        unlink($this->path."/".$this->copy);

    }

}
```

Note: This code is available in the daemon-examples github repository.

5.2 Adding a worker to the daemon

In order to run, a worker should be installed in the daemon before calling the `init()` method. The internal workers stack `Comodojo\Daemon\Worker\Manager` can be accessed using the `$daemon::getWorkers()` getter.

The `install()` method can be used to push a worker into the stack, specifying the looptime:

```
#!/usr/bin/env php
<?php

$base_path = realpath(dirname(__FILE__)."/../");
require "$base_path/vendor/autoload.php";

use \DaemonExamples\CopyDaemon;
use \DaemonExamples\CopyWorker;

$configuration = [
    'description' => 'Copy Daemon',
    'sockethandler' => 'tcp://127.0.0.1:10042'
];

$daemon = new CopyDaemon($configuration);

// Create a CopyWorker with name: handyman
$handyman = new CopyWorker("handyman");

// Install the worker into the stack configuring a 10 secs looptime and enabling␣
↪the forever watchdog
$daemon->getWorkers()->install($handyman, 10, true);

$daemon->init();
```

Note: This code is available in the daemon-examples github repository.

5.3 The forever switch

The `install()` method allows also to enable the *forever* mode for the worker. When the third argument is set to *true*, the internal watchdog of the daemon will restart the worker in case of crash, with no need to restart the whole daemon. On the contrary, in case of *false* a controlled shutdown of the whole daemon will be triggered if one worker goes down.

5.4 Communicating with the worker

When a worker is created, the daemon will open a bidirectional communication channel using standard Unix shared memory segments. This channel will be kept opened for the entire life of the process.

Using this channel:

1. the daemon is able to pool the worker to konw its state (running, paused, ...) and trigger actions if the daemon crashes (worker watchdog);
2. the user can send commands to the worker using the daemon RPC socket.

While the first point is totally automated, the second one requires a user interaction.

5.4.1 Using default commands

By default, the RPC socket expose a couple of method to manage workers:

1. `worker.list()` - get the list of the currently installed workers
2. **`worker.status(worker_name)` - get the status of the worker**
    - 0 => SPINUP
    - 1 => LOOPING
    - 2 => PAUSED
    - 3 => SPINDOWN
3. `worker.pause(worker_name*)` - pause the worker
4. `worker.resume(worker_name*)` - resume the worker

These commands are automatically sent to the communication channel (using shmop), trapped by the worker loop and then propagated as `\Comodojo\Daemon\Events\WorkerEvent`. A listener on the worker side is responsible for executing the related action.

For example, this RPC request can be used to request the status of all workers:

```
$request = \Comodojo\RpcClient\RpcRequest::create("worker.status", []);
```

And the following one to pause the *handyman* worker:

```
$request = RpcRequest::create("worker.pause", ["handyman"]);
```

5.4.2 Defining custom actions

Custom actions can be defined in the worker to trap user defined commands, using the same mechanism described in the previous section.

As an example, let's customize the CopyDaemon/CopyWorker to change the output filename if a *handyman.changename* request is received.

To create this custom action, first step is to create a custom listener to handle the WorkerEvent:

```php
<?php namespace DaemonExamples;

use \League\Event\AbstractListener;
use \League\Event\EventInterface;

class ChangeNameListener extends AbstractListener {

    public function handle(EventInterface $event) {

        // get the current worker instance
        $worker = $event->getWorker()->getInstance();

        // invoke the changeName method
        $worker->changeName();

        return true;

    }

}
```

This listener should be hooked to a custom event at the worker level. The modified version of the CopyWorker is:

```php
<?php namespace DaemonExamples;

use \Comodojo\Daemon\Worker\AbstractWorker;

class CopyWorker extends AbstractWorker {

    protected $path;

    protected $file = 'test.txt';

    protected $copy = 'copy_test.txt';

    public function spinup() {

        $this->logger->info("CopyWorker ".$this->getName()." spinning up...");
        $this->path = realpath(dirname(__FILE__)."/../../tmp/");

        // Hook on daemon.worker.changename event to change the output file name
        $this->getEvents()
            ->subscribe('daemon.worker.changename',
↪'\DaemonExamples\ChangeNameListener');

    }

    public function loop() {

        $filename = $this->path."/".$this->file;

        if ( file_exists($filename) ) {
            $this->logger->info("Copying file ".$this->file." to ".$this->copy);
            copy($filename, $this->path."/".$this->copy);
        }

    }

    public function spindown() {

        $this->logger->info("CopyWorker ".$this->getName()." spinning down.");
        unlink($this->path."/".$this->copy);
```

(continues on next page)

(continued from previous page)

```
39
40      }
41
42      // this method will be invoked by the listener for daemon.worker.changename
    ↪event
43      public function changeName() {
44          $this->logger->info("Changing filename...");
45          $this->copy = 'copy_test_2.txt';
46
47      }
48
49  }
```

Note: This code is available in the daemon-examples github repository.

The last step is to create a custom RPC Method in the daemon that can handle the *handyman.changename* request translating it to a message *changename* propagated in the communication channel (output side):

```
1   <?php namespace DaemonExamples;
2
3   use \Comodojo\Daemon\Daemon as AbstractDaemon;
4   use \Comodojo\RpcServer\RpcMethod;
5
6   class CopyDaemon extends AbstractDaemon {
7
8       public function setup() {
9
10          // define the changename method using lambda function
11          $change = RpcMethod::create("handyman.changename", function($params,
    ↪$daemon) {
12              return $daemon->getWorkers()
13              ->get("handyman")
14              ->getOutputChannel()
15              ->send('changename') > 0;
16          }, $this)
17              ->setDescription("Change the output file name")
18              ->setReturnType('string');
19
20          // inject the method to the daemon internal RPC server
21          $this->getSocket()
22              ->getRpcServer()
23              ->methods()
24              ->add($change);
25
26      }
27
28  }
```

Note: This code is available in the daemon-examples github repository.