# Comodojo daemon Documentation

***Release 1.0.0***

**Marco Giovinazzi**

**Nov 07, 2018**

# 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 took just a couple of lines:

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

(continues on next page)

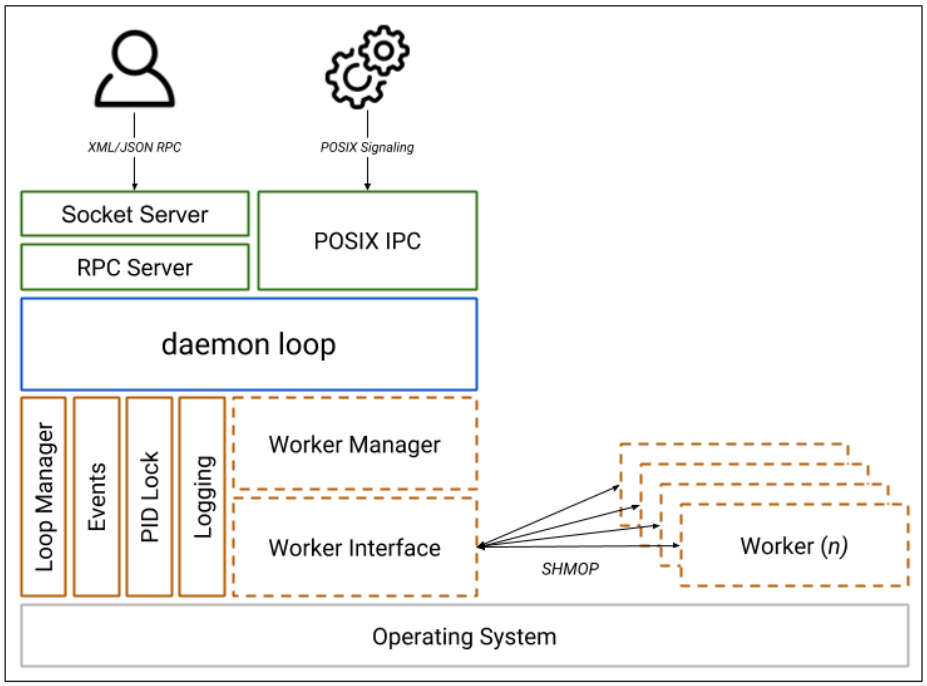


Fig. 1: comodojo/daemon v1.X architecture

(continued from previous page)

```
use \Comodojo\RpcServer\RpcMethod;

class Daemon extends AbstractDaemon {

    public function setup() {
        $echo = RpcMethod::create("my.echo", function($params, $daemon) {
            $message = $params->get('message');
            return $message;
        }, $daemon)
            ->setDescription("I'm here to reply your data")
            ->addParameter('string','message')
            ->setReturnType('string');

        $this->getSocket()
            ->getRpcServer()
            ->methods()
            ->add($echo);
    }

}
```

## 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 with 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 namespace My\Echo\Daemon;

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

class Daemon extends AbstractDaemon {

    public function setup() {

        // define the echo method
        $echo = RpcMethod::create("my.echo", function($params, $daemon) {
            $message = $params->get('message');
            return $message;
        }, $daemon)
            ->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)

```
            ->methods()
            ->add($echo);
    }

}
```

The *my.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.

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

require "vendor/autoload.php";

use \My\Echo\Daemon;

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

$daemon = new Daemon($configuration);

$daemon->init();
```

**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.

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

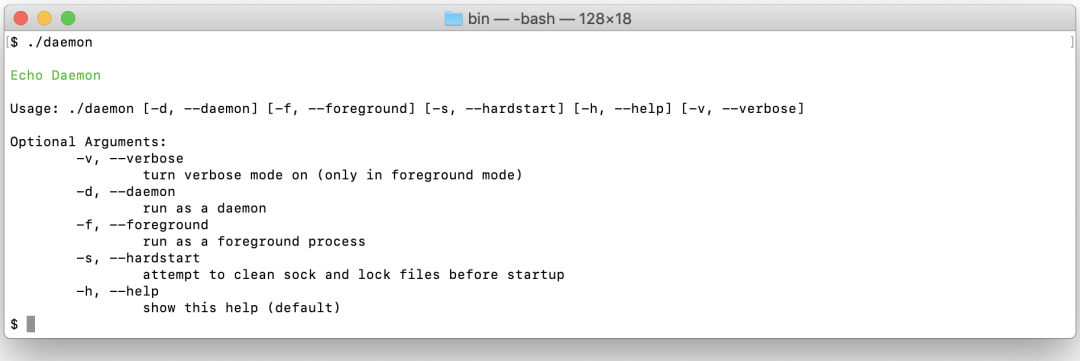


Fig. 1: comodojo/daemon default console

CHAPTER 4

# Daemon configuration

## 4.1 General configuration

TBW

## 4.2 Advanced configuration

TBW

CHAPTER 5

# Using Workers

TBW