Docs

System Administration Category

System Administration

Anonymous · 09/09/2026

Table of Contents

System Administration Category

System Administration

Anonymous · 09/09/2026

System Administration

Last updated on 09/09/2026

Docker with Python and Caddy

Derafu Docker with Python and Caddy for Fabric

Anonymous · 09/09/2026 · #docker, #python

Derafu Docker with Python and Caddy for Fabric

A modern Docker setup for hosting Python websites with Caddy web server and SSH access for Fabric deployments — the Python counterpart of Docker with PHP and Caddy.

Features

- **Python 3.14**: Supported Python version with common extensions.
- **Caddy**: Modern web server with automatic HTTPS.
- **SSH Access**: For automated deployments with Fabric.
- **Automatic Site Discovery**: Just add your site folder and it works.
- **Development Domains**: Test with `.local` domains that map to production folders.
- **Automatic WWW Redirection**: For second-level domains (e.g., `example.com` → `www.example.com`).
- **Auto-HTTPS**: Certificates are automatically generated on-demand.
- **Environment Separation**: Development and production environments managed through Docker Compose override.
- **Optional PHP embed + `phpy`**: build PHP with `--enable-embed` and install `phpy` so Python code running in this container can host and call PHP libraries directly (e.g. Backbone Bridge Python). Disabled by default — see PHP Embed + phpy below.

Quick Start

Prerequisites

- Docker and Docker Compose installed on your system.
- SSH key for deployment access.

Setup

1. Clone this repository:

```
git clone https://github.com/derafu/docker-python3.14-caddy-server.git
cd docker-python3.14-caddy-server
```

2. Add your SSH public key to `config/ssh/authorized_keys` for admin, and default deployment, access:

```
cat ~/.ssh/id_rsa.pub > config/ssh/authorized_keys
```

3. Build and start the container:

```
docker-compose up -d
```

Testing Your First Site

1. Create the site directory structure and a Django project:

```
mkdir -p sites/www.example.com/
cd sites/www.example.com/
python3 -m venv venv
source venv/bin/activate
pip install django
django-admin startproject example .
```

2. Run it:

```
/scripts/start_sites.sh www.example.com
```

   If no site is specified, the script starts every available site under `/var/www/sites`.

3. Access it at `https://www.example.com` (production, requires DNS) or `https://www.example.com.local:8443` (development, requires a local `/etc/hosts` entry).

PHP Embed + `phpy` (Optional)

This image can optionally build PHP from source with `--enable-embed` and install `phpy`, so Python code can load and call a PHP library directly in the same process — the Python-hosts-PHP direction; the opposite direction, PHP-hosts-Python, is what Docker with PHP and Caddy provides.

This is **disabled by default**: no regular PHP package (apt, Homebrew, official Docker images) ships

with `--enable-embed`, so getting it requires compiling PHP from source, which adds several minutes to the image build. Enable it explicitly when you actually need it:

```
# In your .env file:
PHPY_ENABLED=true
PHPY_PHP_VERSION=8.5.3   # Optional, defaults to 8.5.3.

docker compose build
docker compose up -d
```

When enabled, the embed-enabled PHP build lives at `/opt/php` (`php`, `php-config`, `phpize` on `PATH`), and `phpy` is installed into the container's Python. Verify it works with:

```
docker compose exec webserver python3 -c "import phpy; print(phpy)"
```

Building with the default `PHPY_ENABLED=false` skips all of this and behaves exactly like a normal Python + Caddy image.

`phpy` needs a matching virtualenv

Since `phpy` is installed into the container's own Python, not into any particular virtualenv, create yours with `python3 -m venv --system-site-packages` so it can still see it. See Backbone Bridge Python for the Python-side package that builds on top of this.

Development vs Production Environment

Docker Compose's override mechanism separates the two:

- **Production** (`docker-compose.yml` alone): minimal configuration, required environment variables, essential ports (HTTP, HTTPS, SSH), no external volumes.
- **Development** (`docker-compose.override.yml`, merged automatically by `docker-compose up -d`): additional development ports and local volume mounts for live editing.

To run production-only:

```
docker-compose -f docker-compose.yml up -d
```

Access and Management

```
# SSH access.
ssh admin@localhost -p 2222

# Direct container shell.
```

```
docker exec -it derafu-sites-server-python-caddy bash

# Restart Caddy.
docker exec -it derafu-sites-server-python-caddy supervisorctl restart caddy

# Stop the container.
docker-compose down
```

Last updated on 09/09/2026

Docker with PHP and Caddy

Derafu Docker with PHP and Caddy

Anonymous · 09/09/2026 · #docker

Derafu Docker with PHP and Caddy

Last updated on 09/09/2026

Introduction

Docker with PHP and Caddy for Deployer

Anonymous · 09/09/2026 · #docker

Docker with PHP and Caddy for Deployer

A modern Docker setup for hosting PHP websites with Caddy web server and SSH access for Deployer deployments.

Current PHP version: 8.5.

The latest supported, and recommended, PHP version is 8.5. If you need to use another version, please use the docker-php7.4-caddy-server, docker-php8.3-caddy-server or docker-php8.4-caddy-server repositories. Remember change the PHP version in the examples below.

Features

- **Multiple PHP versions**: Supported PHP versions with common extensions: 7.4, 8.3, 8.4 and 8.5.
- **Caddy**: Modern web server with automatic HTTPS.
- **SSH Access**: For automated deployments with Deployer.
- **Automatic Site Discovery**: Just add your site folder and it works.
- **Development Domains**: Test with .local domains that map to production folders.
- **Automatic WWW Redirection**: For second-level domains (e.g., example.com → www.example.com).
- **Auto-HTTPS**: Certificates are automatically generated on-demand.
- **Environment Separation**: Development and production environments can be managed through Docker Compose override.

Quick Start

Prerequisites

- Docker and Docker Compose installed on your system.
- SSH key for deployment access.

Setup

1. Clone this repository:

   ```
   git clone https://github.com/derafu/docker-php8.5-caddy-server.git
   cd docker-php8.5-caddy-server
   ```

2. Add your SSH public key to `config/ssh/authorized_keys` for admin, and default deployment, access:

   ```
   cat ~/.ssh/id_rsa.pub > config/ssh/authorized_keys
   ```

3. Build and start the container:

   ```
   docker-compose up -d
   ```

   The `-d` parameter runs it in detached mode (background).

Verification

Check that the container is running:

```
docker-compose ps
```

View container logs:

```
docker-compose logs -f
```

The `-f` parameter allows you to follow logs in real-time.

Testing Your First Site

1. Create a test site directory structure:

   ```
   mkdir -p sites/www.example.com/public
   echo "<?php phpinfo();" > sites/www.example.com/public/index.php
   ```

2. Access the site at:

   - Production mode: https://www.example.com (requires DNS configuration).

    - Development mode: https://www.example.com.local:8443 (requires local hosts entry).

For local development, add to your `/etc/hosts` file:

```
127.0.0.1 www.example.com.local
```

Directory Structure

```
docker-php8.5-caddy-server/
├── config/                              # Configuration files.
│   ├── caddy/                           # Caddy configuration.
│   ├── php/                             # PHP configuration.
│   ├── ssh/                             # SSH configuration and authorized keys.
│   └── supervisor/                      # Supervisor configuration.
├── sites/                               # Web sites directory for local development.
│   └── www.example.com/                 # Example site.
│       └── public/                      # Public web files.
├── .env                                 # Docker Compose environment configuration.
├── Dockerfile                           # Container definition.
├── docker-compose.yml                   # Docker services configuration (production).
└── docker-compose.override-example.yml # Development-specific configuration.
```

Development vs Production Environment

This project uses Docker Compose's override functionality to separate different configurations.

Usage:

- **Development**: Rename `docker-compose.override-example.yml` to `docker-compose.override.yml` and then Docker Compose automatically merges both files:

  ```
  docker-compose up -d
  ```

- **Production**: Use only the base configuration (with `-f` or not creating the `docker-compose.override.yml` file):

  ```
  docker-compose -f docker-compose.yml up -d
  ```

Access and Management

SSH Access

Connect to the container via SSH:

```
ssh admin@localhost -p 2222
```

Direct Container Access

Access the container shell:

```
docker exec -it derafu-sites-server-php-caddy bash
```

Restarting Services

Restart Caddy web server:

```
docker exec -it derafu-sites-server-php-caddy supervisorctl restart caddy
```

Stopping the Container

```
docker-compose down
```

Rebuilding After Configuration Changes

Rebuild for development:

```
docker-compose build --no-cache
docker-compose up -d
```

Rebuild for production:

```
docker-compose -f docker-compose.yml build --no-cache
docker-compose -f docker-compose.yml up -d
```

Adding New Sites

1. Create the site directory structure:

```
mkdir -p sites/www.newsite.com/public
touch sites/www.newsite.com/public/index.php
```

2. No server restart required! Caddy automatically detects new sites.

3. For local development, add to your hosts file:

```
127.0.0.1 www.newsite.com.local
```

Environment Variables

Customize behavior through environment variables:

Variable	Description	Default
`SERVER_NAME`	Name for the docker container	derafu-sites-server
`CADDY_DEBUG`	Enable debug mode with `debug`	(empty)
`CADDY_EMAIL`	Email for Let's Encrypt	admin@example.com
`CADDY_HTTPS_ISSUER`	TLS issuer (internal, acme)	internal
`CADDY_HTTPS_ALLOW_ANY_HOST`	Allow any host for TLS	false
`CADDY_LOG_SIZE`	Log file max size	100mb
`CADDY_LOG_KEEP`	Number of log files to keep	5
`WWW_ROOT_PATH`	Web root path	/var/www/sites
`WWW_USER`	WWW and SSH user in the container	admin
`WWW_GROUP`	WWW group in the container	www-data
`HTTP_PORT`	HTTP port in host	8080
`HTTPS_PORT`	HTTPS port in host	8443
`SSH_PORT`	SSH port in host	2222

Domain Logic

The server handles domains in the following way:

1. **Development domains**: Any domain ending with `.local` (e.g., `www.example.com.local`)
    - Maps to the same directory as its production counterpart.

    - Uses internal self-signed certificates.

2. **Production domains**:
    - Redirects from non-www to www for second-level domains.
    - Automatically obtains and manages Let's Encrypt certificates (issuer `acme`).

Troubleshooting

SSL Certificate Issues

If you're having issues with SSL certificates in development:

- Ensure your browser trusts self-signed certificates.
- Try using HTTP instead of HTTPS for local development.

Permissions Issues

If you encounter permission issues:

```
docker exec -it derafu-sites-server-php-caddy chown -R admin:www-data /var/www/sites
```

Logs Location

Logs are available in the container and can be accessed with:

```
docker exec -it derafu-sites-server-php-caddy cat /var/log/caddy/access.log
```

Advanced Usage

Custom Caddy Configuration

For advanced configurations, modify the Caddyfile at `config/caddy/Caddyfile`.

Using with Deployer

This container is designed to work with Deployer for PHP deployments:

1. Set up your `deploy.php` configuration to connect to the SSH server on port 2222.
2. Use the `admin` user for authentication.
3. Set your deployment path to `/var/www/sites/www.yoursite.com`

Last updated on 09/09/2026

Docker Guide for Beginners

Docker Guide for Beginners

Anonymous · 09/09/2026

Docker Guide for Beginners

This guide provides an introduction to Docker for developers who are new to containerization. It covers basic concepts, essential commands, and best practices for working with Docker in the context of our PHP-Caddy server environment.

Introduction to Docker

Docker is a platform that enables developers to build, package, and run applications in containers. Containers are lightweight, portable, and self-sufficient environments that include everything an application needs to run.

Benefits of using Docker:

- **Consistency**: Applications run the same way across different environments.
- **Isolation**: Applications run in isolated environments without interfering with each other.
- **Portability**: Containers can run on any system that has Docker installed.
- **Efficiency**: Containers share the host system's kernel and use fewer resources than virtual machines.

Key Concepts

- **Container**: A runnable instance of an image that is isolated from the host and other containers. Think of it as a lightweight virtual machine.
- **Image**: A read-only template used to create containers. Images include the application code, runtime, libraries, and dependencies.
- **Dockerfile**: A text file that contains instructions to build a Docker image.
- **Docker Compose**: A tool for defining and running multi-container Docker applications using a YAML file.
- **Volume**: A persistent data storage mechanism that exists outside of containers.
- **Registry**: A repository for storing and distributing Docker images (e.g., Docker Hub).

Installation

macOS

1. Download Docker Desktop from https://www.docker.com/products/docker-desktop
2. Install the application
3. Start Docker Desktop

Windows

1. Download Docker Desktop from https://www.docker.com/products/docker-desktop
2. Install the application
3. Enable WSL 2 (Windows Subsystem for Linux) if prompted
4. Start Docker Desktop

Linux (Ubuntu)

```
# Update package index.
sudo apt-get update

# Install dependencies.
sudo apt-get install apt-transport-https ca-certificates curl gnupg lsb-release

# Add Docker's official GPG key.
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg

# Set up the stable repository.
echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null

# Install Docker Engine.
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io

# Install Docker Compose.
sudo curl -L
"https://github.com/docker/compose/releases/download/v2.18.1/docker-compose-$(uname -
s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Basic Docker Commands

Check Docker installation

```
docker --version
docker compose version
```

Working with Images

List all images

```
docker images
```

Pull an image from a registry

```
docker pull php:8.5-fpm
```

Build an image from a Dockerfile

```
docker build -t my-app:latest .
```

Remove an image

```
docker rmi image_name
```

Working with Containers

List running containers

```
docker ps
```

List all containers (including stopped)

```
docker ps -a
```

Create and start a container

```
docker run -d --name my-container image_name
```

Stop a container

```
docker stop container_name
```

Start a stopped container

```
docker start container_name
```

Remove a container

```
docker rm container_name
```

Execute a command in a running container

```
docker exec -it container_name bash
```

View container logs

```
docker logs container_name
```

Follow container logs in real-time

```
docker logs -f container_name
```

Working with Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to configure all the application's services, networks, and volumes.

Basic Docker Compose Commands

Start all services defined in docker-compose.yml

```
docker compose up -d
```

The `-d` flag runs containers in the background (detached mode).

Stop all services

```
docker compose down
```

View logs from all services

```
docker compose logs
```

View logs from a specific service

```
docker compose logs service_name
```

Follow logs in real-time

```
docker compose logs -f
```

Rebuild services

```
docker compose build --no-cache
```

Restart a specific service

```
docker compose restart service_name
```

Execute a command in a service container

```
docker compose exec service_name command
```

Example: Access bash in the webserver service

```
docker compose exec webserver bash
```

Docker Compose Override

This project uses Docker Compose's override functionality to separate different configurations:

Use both docker-compose.yml and docker-compose.override.yml

```
docker compose up -d
```

Use only docker-compose.yml

```
docker compose -f docker-compose.yml up -d
```

Docker Compose Environment Variables

Docker Compose can use environment variables from:

1. An `.env` file in the same directory.
2. Environment variables set in the shell.
3. Default values specified in the docker-compose.yml file.

Example .env File

```
SERVER_NAME=derafu-sites-server
HTTP_PORT=8080
HTTPS_PORT=8443
SSH_PORT=2222
```

Variable Substitution in docker-compose.yml

```
services:
  webserver:
    ports:
      - "${HTTP_PORT:-8080}:80"
```

The syntax `${VARIABLE:-default}` means “use the value of VARIABLE if set, otherwise use ‘default’”.

Common Workflows

Initial Setup

1. Clone the repository:

   ```
   git clone https://github.com/derafu/docker-php8.5-caddy-server.git
   cd docker-php8.5-caddy-server
   ```

2. Copy the example .env file:

   ```
   cp .env-dist .env
   ```

3. Add your SSH public key for deployment access:

   ```
   cat ~/.ssh/id_rsa.pub > config/ssh/authorized_keys
   ```

4. Build and start the containers:

   ```
   docker compose up -d
   ```

Adding a New Site

1. Create the site directory structure:

   ```
   mkdir -p sites/www.newsite.com/public
   echo "<?php phpinfo();" > sites/www.newsite.com/public/index.php
   ```

2. For local development, add to your hosts file:

   ```
   127.0.0.1 www.newsite.com.local
   ```

3. Access the site at https://www.newsite.com.local:8443

Updating After Configuration Changes

```
docker compose build --no-cache
docker compose up -d
```

Accessing the Container

```
docker compose exec webserver bash
```

Checking Logs

```
# View Caddy logs.
docker compose exec webserver cat /var/log/caddy/access.log

# Follow PHP-FPM logs.
docker compose exec webserver tail -f /var/log/php-fpm.log
```

Troubleshooting

Container Won't Start

1. Check for port conflicts:

   ```
   netstat -tuln | grep 8080
   ```

2. Check container logs:

   ```
   docker compose logs webserver
   ```

Permission Issues

If you encounter permission issues with mounted volumes:

```
docker compose exec webserver chown -R admin:admin /var/www/sites
```

Network Issues

If containers can't communicate:

1. Check if containers are on the same network:

   ```
   docker network ls
   docker network inspect <network_name>
   ```

2. Check if services are running:

   ```
   docker compose ps
   ```

Rebuilding from Scratch

If you need to start over completely:

```
# Stop and remove containers, networks, volumes, and images.
docker compose down -v --rmi all

# Rebuild and start
docker compose up -d --build
```

Best Practices

1. **Use .dockerignore files**: Exclude unnecessary files from the build context.
2. **Minimize image layers**: Combine RUN commands where possible.
3. **Use specific tags for base images**: Avoid using 'latest' which can change unexpectedly.
4. **Keep containers stateless**: Store persistent data in volumes.
5. **Use environment variables** for configuration that changes between environments.
6. **Regularly update base images** to get security patches.
7. **Use health checks** to ensure services are running correctly.
8. **Limit container privileges** for better security.

Further Resources

- Official Docker Documentation
- Docker Compose Documentation
- Docker Hub - Repository of Docker images.
- Docker Curriculum - Comprehensive Docker tutorial.
- Play with Docker - Browser-based Docker playground.

Last updated on 09/09/2026

Xdebug

Debugging with Xdebug

Anonymous · 09/09/2026

Debugging with Xdebug

This Docker container includes Xdebug to facilitate both interactive debugging and code coverage reporting. By default, Xdebug is installed but disabled to prevent performance impact in production environments.

Enabling Xdebug

To activate Xdebug for interactive debugging:

1. Access the container:

   ```
   docker exec -it php84-caddy bash
   ```

2. Edit the Xdebug configuration file:

   ```
   vim /usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini
   ```

3. Change the mode setting:

   ```
   xdebug.mode = develop,debug
   ```

4. Restart PHP-FPM:

   ```
   supervisorctl restart php-fpm
   ```

Code Coverage for Unit Tests

To run tests with code coverage without permanently modifying the configuration:

```
docker exec -it php84-caddy bash -c "cd /var/www/sites/your-domain &&
```

```
XDEBUG_MODE=coverage vendor/bin/phpunit --coverage-html ./coverage"
```

This will generate an HTML coverage report in the `coverage` directory of your project.

Visual Studio Code Configuration

1. Install the PHP Debug extension.
2. Add this configuration to your `launch.json`:

```
{
    "name": "Listen for Xdebug",
    "type": "php",
    "request": "launch",
    "port": 9003,
    "pathMappings": {
        "/var/www/sites/your-domain": "${workspaceFolder}"
    }
}
```

Performance Considerations

Xdebug can significantly impact performance:

- **Production environments**: Keep Xdebug disabled (`xdebug.mode = off`).
- **Development environments**: Only enable when necessary.
- **Performance impact**: PHP execution can be 2-3x slower with Xdebug enabled.
- **Memory usage**: Increased memory consumption when active.

Always return the configuration to `xdebug.mode = off` when you've finished debugging.

Available Xdebug Modes

Xdebug 3 offers different modes that can be configured based on your needs:

Mode	Description
`off`	Disables all functionality (default)
`develop`	Development features (enhanced error messages, var_dump improvements)
`coverage`	Code coverage analysis for PHPUnit
`debug`	Interactive debugging with IDE
`profile`	Performance profiling

Mode	Description
`trace`	Function call tracing

Troubleshooting

No Connection to IDE

1. Check if Xdebug is properly enabled:

   ```
   docker exec -it php84-caddy php -i | grep xdebug.mode
   ```

2. Ensure your IDE is listening for connections.

3. Verify the host and port settings:

   ```
   xdebug.client_host = host.docker.internal
   xdebug.client_port = 9003
   ```

4. Check Docker networking:

   ```
   docker exec -it php84-caddy ping host.docker.internal
   ```

Connection Timeouts

If the connection times out, add the following to your Xdebug configuration:

```
xdebug.start_with_request = yes
xdebug.discover_client_host = false
xdebug.log = /var/log/xdebug.log
xdebug.log_level = 7
```

Then check the log file for connection issues:

```
docker exec -it php84-caddy cat /var/log/xdebug.log
```

Last updated on 09/09/2026

Deployer Project

PHP Deployer for multiple sites

Anonymous · 09/09/2026 · #php

PHP Deployer for multiple sites

Derafu Deployer is a PHP deployment tool built on top of Deployer that simplifies managing deployments for multiple websites on a single server.

Features

- Deploy multiple sites from a single configuration.
- Deploy individual sites or all sites at once.
- Support for different deployment environments (development, production).
- Shared files and directories between releases.
- Writable directories configuration.
- Asset building support for sites with Node.js/npm.
- Run custom commands during the deployment.
- OPcache reset after deployments.
- Simple and flexible configuration.
- Support for different Git branches per site.

Requirements

- PHP 8.4 or higher.
- SSH access to your servers.
- Git repositories for your projects.

Installation

```
composer create-project derafu/deployer
```

Note: The tool is designed to be used standalone, not inside other project.

Configuration

The configuration can be stored in multiple files:

- `sites.php`: Legacy file and currently **deprecated**.
- `config/sites.yaml`: New file for the sites configuration when you need only one environment.
- `config/ABC.sites.yaml`: File for separated environments sites configuration.

The `ABC` is called the *source* of the configuration. For example, `dev.sites.yaml` for the development environment, `prod.sites.yaml` for the production environment, etc.

Sites Configuration

Create a YAML file and configure the sites you want to deploy using the following structure in each file:

- A key with the site/domain name and a value with the repository URL.
- A key with the site/domain name and a value with an array of detailed configuration options.

For example, create the file `config/sites.yaml` with the following content:

```
# Simple configuration with just the repository URL.
www.example.com: git@github.com:example/example.git

# Extended configuration with options.
www.complex-site.com:
    repository: git@github.com:example/complex-site.git
    branch: dev
    deploy_path: /var/www/custom/path/complex-site
    shared_files: ['.env', 'config/settings.php']
    shared_dirs: ['var/uploads', 'var/logs']
    writable_dirs: ['var', 'tmp', 'var/cache']
    writable_mode: chmod
    writable_use_sudo: false
    writable_recursive: true
    writable_chmod_mode: 0775
```

Add a site to the configuration via CLI

For simple configurations you can use:

```
./site-add.sh www.example.com git@github.com:example/example.git
```

This will always add the configuration to the `config/sites.yaml` file.

Using sources

You can use sources to create a configuration for multiple environments. For example, you can create a source for the development environment and a source for the production environment.

- `config/dev.sites.yaml`: Development environment configuration.
- `config/prod.sites.yaml`: Production environment configuration.

This file can have the same sites, but with different configuration options. Then you can use the `--source` option to specify the source of the configuration to select the appropriate environment for the site.

Available Configuration Options

Option	Description	Default
repository	Git repository URL	*Required*
branch	Git branch to deploy	main
deploy_path	Deployment path on server	/var/www/sites/[site-name]
shared_files	Files to share between releases	[]
shared_dirs	Directories to share between releases	[]
writable_dirs	Directories to make writable	[‘var’, ‘tmp’]
writable_mode	Mode for writable directories	chmod
writable_use_sudo	Whether to use sudo for writable directories	false
writable_recursive	Apply writable permissions recursively	true
writable_chmod_mode	Chmod mode for writable directories	0777

Deployer Server Configuration

The server configuration is defined in the file `deploy.php`. By default, a local environment (localhost) and a remote environment (if `DEPLOYER_HOST` is set) are configured:

```
// Default local environment.
host('localhost')
    ->setRemoteUser('admin')
    ->setPort(2222)
    ->setLabels(['stage' => 'local']);

// Remote environment (only if DEPLOYER_HOST is set).
if (getenv('DEPLOYER_HOST')) {
    $stage = getenv('DEPLOYER_STAGE') ?: 'prod';
    host(getenv('DEPLOYER_HOST'))
        ->setRemoteUser(getenv('DEPLOYER_USER') ?: 'admin')
```

```
        ->setPort(getenv('DEPLOYER_PORT') ?: 2222)
        ->setLabels(['stage' => $stage]);
    set('default_selector', 'stage=' . $stage);
}
```

You can modify these settings or add additional environments as needed.

We recommend to use the environment variables and not to hardcode the values in the file `deploy.php`.

Variable	Description	Default
`DEPLOYER_HOST`	Remote host	
`DEPLOYER_USER`	Remote user	`admin`
`DEPLOYER_PORT`	Remote port	`2222`
`DEPLOYER_STAGE`	Environment stage	`prod`

Actions (shell scripts)

Actions allow you to run custom commands during the deployment. For example, you can run a custom command to rebuild the cache after the deployment.

There are three types of actions:

- `initial`: Run initial actions before the deployment, just after the code is updated.
- `final`: Run final actions after the deployment, just before the symlink is created.
- `success`: Run success actions after the deployment, just before the success message.

The actions are defined in the `.deployer/actions` directory. Each action is a Shell script file:

- `.deployer/actions/initial.sh`: Initial actions script.
- `.deployer/actions/final.sh`: Final actions script.
- `.deployer/actions/success.sh`: Success actions script.

All the actions are optional, the deployment will continue if the action file is not present.

Usage

List Available Sites

To see the list of configured sites and usage information use `derafu:sites:list` command.

```
vendor/bin/dep derafu:sites:list
```

You can filter the list of sites by source using the `--source` option.

```
vendor/bin/dep derafu:sites:list --source=prod
```

Deploy a Single Site

Local Environment (localhost)

```
vendor/bin/dep derafu:deploy:single --site=www.example.com
```

Remote Environment

This usually is used for production environments, but can be used for any remote environment.

```
DEPLOYER_HOST=hosting.example.com vendor/bin/dep derafu:deploy:single --
site=www.example.com
```

You can also specify the SSH user and port with the environment variables `DEPLOYER_USER` and `DEPLOYER_PORT`.

```
DEPLOYER_HOST=hosting.example.com DEPLOYER_USER=deployuser DEPLOYER_PORT=22
vendor/bin/dep derafu:deploy:single --site=www.example.com
```

You can also specify the source of the configuration to select the appropriate environment for the site.

```
DEPLOYER_HOST=hosting.example.com DEPLOYER_USER=deployuser DEPLOYER_PORT=22
vendor/bin/dep derafu:deploy:single --source=prod --site=www.example.com
```

Deploy All Sites

Yo can deploy all sites of the configuration in deployer, or only the sites of a specific source. It's similar to the deploy a single site, but using `derafu:deploy:all`.

Local Environment (localhost)

```
vendor/bin/dep derafu:deploy:all
```

For a specific source:

```
vendor/bin/dep derafu:deploy:all --source=prod
```

Remote Environment

```
DEPLOYER_HOST=hosting.example.com vendor/bin/dep derafu:deploy:all
```

For a specific source:

```
DEPLOYER_HOST=hosting.example.com vendor/bin/dep derafu:deploy:all --source=prod
```

And you can also specify the SSH user and port with the environment variables `DEPLOYER_USER` and `DEPLOYER_PORT`.

Unlock a Deployment

If a deployment gets stuck or locked, you can unlock it `--unlock` option.

```
DEPLOYER_HOST=hosting.example.com vendor/bin/dep derafu:deploy:single --
site=www.example.com --unlock
```

Using aliases

You can create aliases for easy deployment. For example:

```
DEPLOYER_DIR="$HOME/dev/php/deployer"
alias site-deploy-stage="cd $DEPLOYER_DIR && DEPLOYER_HOST=example.com
DEPLOYER_USER=admin DEPLOYER_PORT=2223 vendor/bin/dep derafu:deploy:single --site"
alias site-deploy-prod="cd $DEPLOYER_DIR && DEPLOYER_HOST=example.com
DEPLOYER_USER=admin DEPLOYER_PORT=2224 vendor/bin/dep derafu:deploy:single --site"
```

Also, yo can specify the source of the configuration to select the appropriate environment for the site.

```
DEPLOYER_DIR="$HOME/dev/php/deployer"
alias site-deploy-stage="cd $DEPLOYER_DIR && DEPLOYER_HOST=example.com
DEPLOYER_USER=admin DEPLOYER_PORT=2223 vendor/bin/dep derafu:deploy:single --
source=dev --site"
alias site-deploy-prod="cd $DEPLOYER_DIR && DEPLOYER_HOST=example.com
DEPLOYER_USER=admin DEPLOYER_PORT=2224 vendor/bin/dep derafu:deploy:single --
source=prod --site"
```

The you can use the aliases to deploy the site to the appropriate environment.

```
site-deploy-stage www.example.com
site-deploy-prod www.example.com
```

Deployment Process

For each site, the deployment process performs the following steps:

1. **Info** (`deploy:info`): Shows information about the deployment.
2. **Setup** (`deploy:setup`): Sets up the deployment environment.
3. **Check Remote** (`deploy:check_remote`): Verifies SSH connection and deployment path.
4. **Lock** (`deploy:lock`): Locks the deployment.
5. **Release** (`deploy:release`): Creates a new release.
6. **Update Code** (`deploy:update_code`): Fetches code from the Git repository.
7. **Initial Actions** (`deploy:initial_actions`): Runs custom commands in the initial of the deployment.
8. **Environment** (`deploy:env`): Sets up the environment variables.
9. **Shared Files/Dirs** (`deploy:shared`): Links shared files and directories.
10. **Writable Dirs** (`deploy:writable`): Makes specified directories writable.
11. **Vendors** (`deploy:vendors`): Installs PHP dependencies with Composer.
12. **Assets** (`deploy:assets`): Builds frontend assets if package.json exists (`npm install && npm run build`).
13. **Final Actions** (`deploy:final_actions`): Runs custom commands in the final of the deployment.
14. **Symlink** (`deploy:symlink`): Creates a symlink to the new release.
15. **Unlock** (`deploy:unlock`): Removes the deployment lock.
16. **Cleanup** (`deploy:cleanup`): Removes old releases (keeps 5 by default).
17. **OPcache Reset** (`opcache:reset`): Resets the OPcache.
18. **Success Actions** (`deploy:success_actions`): Runs custom commands in the success of the deployment.
19. **Success Message** (`deploy:success`): Shows the success message.

Last updated on 09/09/2026

GitHub Project

Webhook handling and other sysadmin tasks

Anonymous · 09/09/2026 · #php

Webhook handling and other sysadmin tasks

A lightweight, no-dependency PHP library for handling GitHub webhooks, with a focus on security and extensibility.

Overview

This library provides a simple, secure way to handle GitHub webhooks, allowing you to easily react to GitHub events such as push notifications, pull requests, workflow runs, and more.

Features

- **Zero Dependencies**: Doesn't require any external packages.
- **Secure**: Built with security best practices, including HMAC signature validation.
- **Extensible**: Easily add custom handlers for any GitHub event.
- **Typed**: Fully typed for modern PHP 8.3 environments.
- **Event-driven**: Handle different GitHub webhook events with separate handlers.

Installation

Via Composer

```
composer require derafu/github
```

Manual Installation

Clone the repository:

```
git clone https://github.com/derafu/github.git
```

Usage

Basic Setup

1. Create a webhook endpoint:

```
<?php
// webhook.php

declare(strict_types=1);

namespace Derafu\GitHub\Webhook;

use Derafu\GitHub\Logger;
use Derafu\GitHub\Webhook\Handler;
use Derafu\GitHub\Webhook\Response;
use Exception;

// Load the autoloader.
require 'vendor/autoload.php';

// Load configuration.
$config = require 'config.php';

// Create the handler.
$logger = new Logger();
$handler = new Handler($config, $logger);

// Handle the webhook.
try {
    $notification = $handler->handle();
    $response = $notification->getResponse();
} catch (Exception $e) {
    $response = new Response([
        'code' => $e->getCode() ?: 1,
        'data' => [
            'message' => $e->getMessage(),
            'logs' => $logger->getLogs(),
        ],
    ]);
}

// Send the response.
http_response_code($response->getHttpCode());
header('Content-Type: application/json');
echo json_encode($response->toArray(), JSON_PRETTY_PRINT);
```

2. Create a configuration file:

```
<?php
// config.php

declare(strict_types=1);

use Derafu\GitHub\Webhook\EventHandler\WorkflowRunHandler;
use Derafu\GitHub\Webhook\Notification;

return [
    // Add handlers for different events.
    'handlers' => [
        'workflow_run' => fn (Notification $notification) =>
WorkflowRunHandler::deploy(
            $notification
        ),
    ],
];
```

GitHub Webhook Configuration

1. Go to your GitHub repository.
2. Navigate to Settings > Webhooks > Add webhook.
3. Set the Payload URL to your webhook endpoint (e.g., `https://example.com/webhook.php`).
4. Set Content type to `application/json`.
5. Set a Secret that matches your configuration.
6. Choose which events to receive.
7. Ensure the webhook is active.

Handler Examples

Workflow Run Handler

This handler deploys your application when a GitHub Action workflow completes successfully:

```
<?php

namespace Derafu\GitHub\Webhook\EventHandler;

use Derafu\GitHub\Webhook\Notification;

final class WorkflowRunHandler
{
    public static function deploy(
        Notification $notification,
        string $deployer,
        array $sites
```

```
    ): ?string {
        $payload = $notification->getPayload();

        $branch = $payload->workflow_run->head_branch;
        $workflow = $payload->workflow->name;
        $status = $payload->workflow_run->status;
        $conclusion = $payload->workflow_run->conclusion;

        // Check for matching sites and deploy if conditions are met.
        foreach ($sites as $site => $config) {
            // Deployment logic here.
        }

        return null;
    }
}
```

The included handler uses Docker and Deployer.

Custom Handler Example

You can create handlers for any GitHub event:

```
<?php
// In your config.php

'push' => function (Notification $notification) {
    $payload = $notification->getPayload();
    $repo = $payload->repository->name;
    $branch = str_replace('refs/heads/', '', $payload->ref);

    // Your custom logic here.

    $notification->setResponse("Processed push to $repo on branch $branch.");
},
```

Security Best Practices

- Always validate the webhook signature with your secret.
- Use environment variables for storing secrets.
- Escape any arguments used in shell commands with `escapeshellarg()`.
- Validate repository names against a whitelist.
- Implement proper error handling and logging.

Supported Events

The library supports all GitHub webhook events, including:

- `fork`
- `ping`
- `push`
- `pull_request`
- `release`
- `star`
- `status`
- `workflow_run`

For the events not explicitly declared, you can configure a closure for the `default` handler and take any action from there.

Last updated on 09/09/2026