

Docs

Website Skeleton

Skeleton of a Derafu Website

Anonymous · 08/10/2026 · #php

Table of Contents

Website Skeleton	3
<i>Project Structure</i>	14
<i>Assets</i>	17
<i>GitHub Actions</i>	21

Docs » Base for Applications Category » Website Skeleton

Website Skeleton

Skeleton of a Derafu Website

Anonymous · 08/10/2026 · #php

Skeleton of a Derafu Website

github [derafu/website-skeleton](#)

This guide explains how to create a new website using the [Derafu Website Skeleton](#), the [Docker](#) and [PHP Deployer](#) projects for local development and deployment to the production server.

WIP: Work in Progress

The Road to Success is Always Under Construction.

What it is

`derafu/website-skeleton` is the template of a site: everything is in its repository (templates, configuration, `public/`, Vite, translations and tests). It requires [derafu/foundation](#), the package with the dependencies and the base configuration of every web application. To start a library and not a site, use [derafu/library-skeleton](#). The next pages describe the [structure](#), the [assets](#) and the [GitHub Actions](#) of a site, and the quality tools are in [Code Quality](#).

Quick Start

Create a new website based on this base:

```
composer create-project derafu/website-skeleton www.example.com --stability=dev
```

Review what you need to do, install, or know

Configure ZSH on local macOS machine

If you're on macOS and use ZSH as your shell, you can configure ZSH to accept comments in commands with:

```
echo "setopt interactivecomments" >> ~/.zshrc
```

```
source ~/.zshrc
```

SSH Key on local machine

It's necessary to have an SSH key on your local machine. This key will be used to configure your Docker container.

With the following command, you'll get an existing key or create a new one in `$HOME/.ssh/id_rsa` and `$HOME/.ssh/id_rsa.pub`. If a new one is created, you'll need to enter your email to identify the key owner. If you prefer, you can enter your username and machine name (in case you have multiple SSH keys in different environments).

```
SSH_KEY="$HOME/.ssh/id_rsa"
if [ -f "$SSH_KEY.pub" ]; then
    cat "$SSH_KEY.pub"
else
    echo -n "Enter your email: "; read COMMENT
    ssh-keygen -t rsa -b 4096 -N "" -C "$COMMENT" -f "$SSH_KEY"
    cat "$SSH_KEY.pub"
fi
```

With this command, the public key will be displayed on screen. If you need to see it again in the future, run:

```
cat $HOME/.ssh/id_rsa.pub
```

Important: The key in `$HOME/.ssh/id_rsa` is **private** and should never be shared.

Add SSH key to GitHub

1. Go to [GitHub](#).
2. In **Title**, enter the same email or comment you chose when creating the key.
3. In **Key**, paste the public key extracted with `cat $HOME/.ssh/id_rsa.pub`.
4. Click on **Add SSH key**.

Docker container with PHP and Caddy

Prepare [Docker](#) in `$DOCKER_DIR` on your local machine:

```
DEV_DIR=$HOME/dev
DOCKER_DIR=$DEV_DIR/docker-sites-php
mkdir -p $DEV_DIR
git clone https://github.com/derafu/docker-php-caddy-server.git $DOCKER_DIR
cat $HOME/.ssh/id_rsa.pub > $DOCKER_DIR/config/ssh/authorized_keys
cd $DOCKER_DIR
```

Copy and edit the `.env` file and configure the environment variables as needed.

```
# It's recommended to at least configure the DEPLOYER_HOST variable.  
cp .env-dist .env
```

Warning

Don't proceed until you've reviewed and configured the `.env` file.

Build the Docker container:

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

With this configuration, the folder where sites to be developed will be installed will be in `$DOCKER_DIR/sites`. This folder will be shared between your local machine and the Docker container.

Connect to Docker container

Configure the dev SSH alias on your local machine:

```
echo "  
Host dev  
  HostName localhost  
  User admin  
  Port 2222  
  IdentityFile $HOME/.ssh/id_rsa  
  StrictHostKeyChecking no  
  UserKnownHostsFile /dev/null  
" >> $HOME/.ssh/config
```

Then you can enter the Docker container with:

```
ssh dev
```

Note: The alias name can be whatever you want; in this case, `dev` was used.

SSH key in Docker container

To work with private repositories and deploy to the production server, it's necessary that the SSH key from your local machine be added to the Docker container.

On your local machine, run:

```
scp $HOME/.ssh/id_rsa* dev:.ssh/
```

Basic Git configuration

Perform the following configurations inside the Docker container.

First, configure your GitHub email and name:

Warning

Before pasting this command in the Docker container, edit it to use your GitHub email and name.

```
git config --global user.email "you@example.com" # Your github email.
git config --global user.name "Your Name"       # Your name.
```

Configure case sensitivity and avoid mixing changes:

```
git config --global core.ignorecase false      # Case sensitive.
git config --global pull.rebase false          # Rebase instead of merge.
git config --global merge.ff false             # Fast forward.
```

Configure text editor:

```
git config --global core.editor nano           # Default editor nano or any other.
```

Configure commit signing in Git

First, create the SSH key on your local machine:

```
SSH_KEY="$HOME/.ssh/id_ed25519"
if [ -f "$SSH_KEY.pub" ]; then
    cat "$SSH_KEY.pub"
else
    echo -n "Enter your email: "; read COMMENT
    ssh-keygen -t ed25519 -N "" -C "$COMMENT" -f "$SSH_KEY"
    cat "$SSH_KEY.pub"
fi
```

Then add the SSH key to the Docker container:

```
scp $HOME/.ssh/id_ed25519* dev:.ssh/
```

Finally, inside the Docker container, configure Git to sign commits:

```
git config --global commit.gpgSign true           # Sign commits.
git config --global user.signingkey ~/.ssh/id_ed25519 # Your ssh key.
git config --global gpg.format ssh                 # Use ssh key.
git config --global tag.gpgSign true               # Sign tags.
```

Develop a website

Create a new website

Enter the container and run:

```
site-create www.example.com
```

Note: Creating the website requires as a subsequent step that you configure its repository on GitHub and add the site to the `$DEPLOYER_DIR/sites.php` file using the `site-add` command.

Clone a website

Enter the container and run:

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

Note: When cloning an existing website, the site is automatically added to the `$DEPLOYER_DIR/sites.php` file.

Visit the website in the browser

On your machine, configure `/etc/hosts` by adding the local development domain:

```
echo "127.0.0.1          www.example.com.local" | sudo tee -a /etc/hosts
```

Then you can access the website via the URL <https://www.example.com.local:8443>

The language of the website

The texts of the templates are in English and are translated with the domain website: the Spanish translations are in `resources/translations/website+intl-icu.es.php`, and `App\Translation\WebsiteTranslationResourceProvider` gives them to the translator. The language of the site is the one of the environment variable `APP_LOCALE` (`es`, for example), or English if it is not set. A text that you add to a template is written in English with the filter `|trans`, and its translation goes in the catalogue; the test `tests/src/Translation/WebsiteMessagesTest.php` fails if a text is missing.

The contact page

The contact page is the one of [derafu/contact-form](#): [derafu/foundation](#) requires it and imports its routes and its services, so the site has no controller of its own for it.

The tests of the website

The site comes with tests that run it as it is: `tests/src/WebsiteTest.php` starts the kernel with the real configuration of `config/`, makes a request to the home page and to `/contact`, and checks the response in English and in Spanish. `tests/src/Translation/WebsiteMessagesTest.php` audits that every text of the templates has its translation.

The kernel of the tests runs in debug mode on purpose. Outside of debug mode the container is cached in `var/cache/<env>/` and a later run reuses it, so a change in `config/services.yaml` would not be seen. If you test the kernel in your site, do the same, or delete `var/cache/` before running.

Deploy a website to production

All these instructions are executed in the Docker container.

Add website to configuration file

If you created the site from scratch instead of cloning it, make sure the website is added to the `$DEPLOYER_DIR/sites.php` file. You can validate this by running:

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

Note: If the site requires special configuration, you'll need to manually edit the `$DEPLOYER_DIR/sites.php` file.

Style tests, code quality, and unit tests

Run style tests, code quality, and unit tests in the Docker container with:

```
site www.example.com
site-check
```

Push changes to GitHub

If everything is correct, push the changes to GitHub:

```
site www.example.com
site-send "Website update."
```

Note: If the GitHub repository has a [configured webhook](#), the website will be deployed automatically when pushing changes and passing the style tests, code quality, and unit tests in the GitHub Actions workflow.

Deployment

Note

It's not necessary to deploy to the production server if the GitHub repository has a [configured webhook](#).

If there are no errors, you can deploy to the production server with:

```
#DEPLOYER_HOST=hosting.example.com # Only if not configured in .env
site-deploy www.example.com
```

If an error occurs when deploying and you try to make a new deploy, it's very likely that the deploy is locked. If this happens, you can unlock and deploy again with:

```
#DEPLOYER_HOST=hosting.example.com # Only if not configured in .env
site-deploy-locked www.example.com
```

Update components

Update Docker

On your local machine, run:

```
DEV_DIR=$HOME/dev
DOCKER_DIR=$DEV_DIR/docker-sites-php
cd $DOCKER_DIR
git pull
docker-compose build --no-cache
docker-compose up -d
```

Warning

You must add the SSH keys to the Docker container again and configure Git inside the container.

Update PHP Deployer

Enter the container and run:

```
cd $DEPLOYER_DIR
git pull
composer update
```

Update website

Enter the container and run:

```
site www.example.com
site-update
```

Using development tools

Composer

Install dependencies:

```
composer install
```

Add development dependency:

```
composer require --dev dependency-name
```

Add production dependency:

```
composer require dependency-name
```

Remove dependency:

```
composer remove dependency-name
```

Update dependencies:

```
composer update
```

Note: Normally only `composer install` is used to install dependencies.

NPM

Install dependencies:

```
npm install
```

Add development dependency:

```
npm install --save-dev dependency-name
```

Add production dependency:

```
npm install --save dependency-name
```

Remove dependency:

```
npm uninstall dependency-name
```

Update dependencies:

```
npm update
```

Note: Normally only `npm install` is used to install dependencies.

Git

View change status:

```
git status
```

Add changes:

```
git add .
```

Make commit:

```
git commit -m "Commit message"
```

Push changes to GitHub:

```
git push
```

Update local repository:

```
git pull
```

Undo changes:

```
git checkout -- .
```

Note: Instead of using dot `.`, you can specify the files you want to add, commit, or revert.

PHP CS Fixer

Check code style:

```
composer phpcs
```

Fix code style:

```
composer phpcs-fix
```

PHP Unit

Run unit tests:

```
composer tests
```

Using terminal in Docker container

Enter a website directory:

```
cd $SITES_DIR/www.example.com
```

Exit directory:

```
cd ..
```

List files:

```
ls -la
```

View file content:

```
cat $DEPLOYER_DIR/sites.php
```

Edit a file:

```
nano $DEPLOYER_DIR/sites.php
```

Save and exit in nano:

```
Ctrl + X
```

Last updated on 08/10/2026

Docs » Base for Applications Category » Website Skeleton » Project Structure

Project Structure

Project Structure

Anonymous · 08/10/2026

Project Structure

derafu/website-skeleton provides a standardized directory structure for your sites (see [Foundation](#) for how it relates to the other packages). This document explains the purpose of each directory and key files.

Root Directories

`.github/`

Contains GitHub-specific configurations:

- `workflows/ci.yml`: Continuous Integration workflow that runs tests and code quality checks.
- `workflows/cd.yml`: Continuous Deployment workflow for automatically deploying. By default, it is configured to deploy to GitHub Pages. Deployment is disabled by default, just remove the `false &&` in the file to enable it.

`app/`

Application bootstrap files:

- `bootstrap.php`: The main bootstrap file that initializes the application runtime.

`assets/`

Frontend assets organized by type:

- `css/`: CSS stylesheets.
- `js/`: JavaScript files.
- `img/`: Images and other media files.

`config/`

Configuration files:

- `routes.yml`: Route definitions for your application. It imports the routes of [derafu/foundation](#) and the site adds its own.
- `services.yml`: Service container configuration (dependency injection). It imports the services

of `derafu/foundation` and the site declares its controllers, its translation provider and whatever it replaces.

`public/`

Web server document root:

- `index.php`: Application entry point with Front Controller.
- `static/`: Compiled assets (generated by the build process).

Note: If your application doesn't need a Front Controller, by default, it's used only for documentation.

`src/`

Application source code. Organize your PHP classes here according to their namespace. By convention, but not enforced out of Derafu ORG, use:

- `Abstract/`: Abstract classes, with `Abstract` prefix.
- `Contract/`: Interfaces for your application, with `Interface` suffix.
- `Controller/`: Controller classes, with `Controller` suffix.
- `Service/`: Service classes.

`templates/`

Template files for rendering HTML:

- `components/`: Reusable template components (header, footer, etc.).
- `layouts/`: Layout templates that define page structure.
- `pages/`: Page-specific templates.
- `base.html.twig`: Base template that defines the HTML structure for all layouts.
- `error.html.twig`: Error page template.
- `html.html.twig`: HTML wrapper template, used when rendering a markdown o php template.

`tests/`

Test files:

- `fixtures/`: Test fixtures and sample data.
- `src/`: Tests for your source code. You can organize your tests by tests suits, mirroring the structure of the `src/` directory, by features or any other criteria (but not enforced out of Derafu ORG).

`var/`

Temporary files and caches:

- `cache/`: Cache files.

- `logs/`: Log files.
- `tmp/`: Temporary files.

Key Files

Configuration Files

- `.gitignore`: Specifies files that Git should ignore.
- `LICENSE`: MIT license file by default.
- `package.json`: NPM configuration for frontend assets.
- `php-cs-fixer.php`: PHP CS Fixer configuration.
- `phpstan.neon`: PHPStan configuration.
- `phpunit.xml`: PHPUnit configuration.
- `vite.config.js`: Vite build configuration.

Application Files

- `public/index.php`: Main entry point that bootstraps the application.
- `app/bootstrap.php`: Initializes the runtime environment.

Everything Is in the Repository

All the files of the structure are in the repository of `derafu/website-skeleton`: there is no installer that copies files during `composer install` or `composer update`. When you start a site, the files are yours, and to get a change of the template you compare with it (or with `git`) and apply what you want.

Extending the Structure

When creating a new site based on `derafu/website-skeleton`, you should:

1. Keep the existing directory structure.
2. Add your own directories as needed.
3. Follow PSR-4 autoloading standards for your PHP classes.
4. Place tests in the corresponding structure within the `tests/` directory.

The structure is designed to be flexible while providing a consistent organization pattern across projects.

Last updated on 08/10/2026

Assets

Frontend Asset Management

Anonymous · 08/10/2026

Frontend Asset Management

derafu/website-skeleton includes a pre-configured asset build system using [Vite](#). This document explains how to work with frontend assets in your project.

Overview

The asset management system handles:

- CSS compilation.
- JavaScript bundling.
- Image optimization.
- Source maps generation.
- Asset versioning.

Directory Structure

```
project_dir/
├── assets/
│   ├── css/
│   │   └── app.css           # Main CSS file.
│   ├── img/                 # Image files.
│   └── js/
│       ├── app.js           # Main JavaScript entry point.
│       └── images.js        # Image import handling.
├── public/
│   └── static/               # Output directory (generated).
│       ├── css/
│       ├── img/
│       └── js/
└── vite.config.js           # Vite configuration.
```

Configuration

The Vite configuration is defined in `vite.config.js`. Key features include:

- Multiple entry points: `app.js`, `app.css`, and `images.js`.
- Output directory: `public/static/`.
- Custom file naming with `.min` suffix.
- Image optimization using `vite-plugin-sharp`.

Entry Points

- `app.js`: Main JavaScript code.
- `app.css`: Main CSS styles.
- `images.js`: Used for processing and importing images.

Image Optimization

Images are automatically optimized during the build process using the Sharp image processing library. The configuration includes:

- PNG compression: Quality 85%, maximum compression level.
- JPEG compression: Quality 85%, progressive encoding.
- Automatic resizing of large images to a maximum of 2000×2000 pixels.

Usage

Adding Assets

1. **CSS**: Add your CSS files to the `assets/css/` directory.
2. **JavaScript**: Add your JavaScript files to the `assets/js/` directory.
3. **Images**: Add your images to the `assets/img/` directory.

Importing Assets

CSS

In your main `app.css` file:

```
/* Import other CSS files */
@import './components/buttons.css';
```

JavaScript

In your JavaScript files:

```
// Import other JavaScript files.
import './components/slider.js';

// Import CSS from JavaScript.
```

```
import '../css/components/modal.css';

// Import images.
import logo from '../img/logo.png';
```

Images

For processing multiple images at once, use the `images.js` file:

```
// Import all images in a directory.
import.meta.glob('../img/**/*');
```

Building Assets

To build assets for production:

```
npm run build
```

This will:

1. Compile and bundle all assets.
2. Optimize images.
3. Generate minified files with source maps.
4. Output everything to the `public/static/` directory.

Using Built Assets

In your HTML/Twig templates, reference the built assets:

```
<link rel="stylesheet" href="https://www.derafu.dev/static/css/app.min.css">
<script src="/static/js/app.min.js"></script>

```

Best Practices

1. **Organize by component:** Group related CSS, JS, and images by feature or component.
2. **Optimize images before adding:** While the build process optimizes images, starting with optimized images is better.
3. **Use CSS imports:** Keep CSS modular by using imports.
4. **Lazy load images:** For image-heavy pages, consider implementing lazy loading.
5. **Watch file size:** Monitor the size of your built assets to ensure optimal loading times.

Advanced Configuration

The Vite configuration can be extended to support:

- CSS preprocessors (Sass, Less, etc.).
- TypeScript.
- Additional plugins for specific needs.
- Custom output paths and formats.

To modify the configuration, edit the `vite.config.js` file as needed.

Last updated on 08/10/2026

Docs » Base for Applications Category » Website Skeleton » GitHub Actions

GitHub Actions

GitHub Actions

Anonymous · 08/10/2026

GitHub Actions

derafu/website-skeleton includes pre-configured GitHub Actions workflows for continuous integration (CI) and continuous deployment (CD). This document explains how these workflows work and how to customize them.

derafu/library-skeleton (libraries) has only the CI, the same one without the deployment, and derafu/foundation (a package with no PHP code) has a CI that validates `composer.json` and installs the dependencies.

Overview

The `.github/workflows/` directory contains two main workflow files:

- `ci.yml`: Continuous Integration workflow.
- `cd.yml`: Continuous Deployment workflow.

Continuous Integration (CI)

The CI workflow runs automatically on each push and pull request to validate code quality and ensure tests pass.

What It Does

The CI workflow:

1. Sets up PHP (8.5) with the `mbstring` and `xdebug` extensions. Every Derafu package uses this same line, including the ones that require `ext-intl`.
2. Installs dependencies using Composer.
3. Validates Composer configuration.
4. Runs PHP CS Fixer to check code style.
5. Runs PHPStan for static analysis.
6. Runs PHPUnit tests.
7. Generates test coverage reports.

PHPUnit is strict (see [Code Quality](#)), so a warning or a deprecation fails the workflow.

The workflow also runs every Monday at 06:00 UTC, and that run does a `composer update` before the checks, so it uses the latest version of every dependency and not the ones of `composer.lock`. It is how a change in `derafu/foundation`, or in any of its packages, that breaks a site is noticed without having to push anything.

Customization

To customize the CI workflow, edit the `.github/workflows/ci.yml` file. Common customizations include:

- Changing PHP versions to test against.
- Adding or removing validation steps.
- Modifying test coverage thresholds.
- Changing notification settings.

Example of adding a new PHP version to test against:

```
# In .github/workflows/ci.yml
jobs:
  tests:
    strategy:
      matrix:
        php-version: ['8.5'] # Add or remove versions as needed.
```

Continuous Deployment (CD)

The CD workflow automatically builds and deploys your project documentation to GitHub Pages when changes are pushed to the main branch.

Note: Deployment is disabled by default, just remove the `false` `&&` in the file to enable it.

What It Does

The CD workflow:

1. Checks out the repository.
2. Sets up Node.js.
3. Installs frontend dependencies.
4. Builds assets using Vite.
5. Sets up PHP.
6. Installs Composer dependencies.
7. Generates a static website from your project documentation.
8. Deploys the static website to GitHub Pages.

GitHub Pages Setup

To use the CD workflow, you need to:

1. Enable GitHub Pages in your repository settings.
2. Set the source branch to `gh-pages` and the directory to `/` (root).
3. Ensure the workflow has permission to write to the repository.

Customization

To customize the CD workflow, edit the `.github/workflows/cd.yml` file. Common customizations include:

- Changing the deployment branch.
- Adding custom build steps.
- Configuring environment variables.
- Adding additional deployment targets.

Example of changing the deployment branch:

```
# In .github/workflows/cd.yml
- name: Deploy to GitHub Pages
  uses: peaceiris/actions-gh-pages@v3
  with:
    github_token: ${ secrets.GITHUB_TOKEN }
    publish_dir: ./public
    publish_branch: documentation # Change from gh-pages to another branch name.
```

Workflow Badges

You can add workflow status badges to your README.md to show the current status of your workflows:

```






```

Replace `derafu/library-skeleton` with your actual GitHub username and repository name.

Best Practices

1. **Keep workflows fast:** Optimize workflows to complete quickly.
2. **Use caching:** Cache dependencies to speed up builds.
3. **Secure secrets:** Use GitHub Secrets for sensitive information.
4. **Test workflow changes:** Test workflow changes on a feature branch before merging to main.
5. **Monitor workflow runs:** Regularly check workflow runs for issues.

Troubleshooting

If a workflow fails:

1. Check the workflow run logs in the GitHub Actions tab.
2. Verify that all dependencies are correctly installed.
3. Ensure environment variables are correctly set.
4. Check if tests are failing locally as well.
5. Look for GitHub Actions service status issues.

If needed, you can run the workflow manually from the Actions tab by selecting the workflow and clicking “Run workflow”.

Last updated on 08/10/2026