Docs

Foundation Project

Derafu Foundation

Anonymous · 21/08/2026 · #php

Table of Contents

Foundation Project

Derafu Foundation

Anonymous · 21/08/2026 · #php

Derafu Foundation

Last updated on 21/08/2026

Introduction

Base for Derafu's Projects

Anonymous · 21/08/2026

Base for Derafu’s Projects

Overview

Derafu Foundation is a standardized base for creating consistent PHP projects. It provides a pre-configured environment with best practices for code organization, documentation, testing, and code quality assurance. This foundation is designed to streamline the creation of new projects while ensuring they follow consistent patterns and standards.

Key Features

- **Standardized Structure**: Consistent file and directory organization across all projects.
- **Code Quality Tools**: Pre-configured PHP CS Fixer, PHPStan, and PHPUnit setups.
- **Asset Management**: Built-in Vite configuration for CSS, JavaScript, and image optimization.
- **Documentation**: Automatic website generation from README.md with GitHub Pages deployment.
- **CI/CD Integration**: GitHub Actions workflows for continuous integration and deployment.
- **Middleware-based HTTP Layer**: Ready-to-use PSR-compliant HTTP handling.
- **Routing System**: Flexible routing with support for static routes and filesystem-based routes.
- **Templating**: Twig and Markdown rendering support.

Quick Start

Installation

Create a new project based on this foundation:

```
composer create-project derafu/project project_dir --stability=dev
```

Asset Compilation

If your project uses CSS, JavaScript, or images, add them to the `assets` directory and compile them:

```
npm install
npm run build
```

Development Server

Start the built-in PHP development server:

```
php -S localhost:9000 public/index.php
```

Then visit http://localhost:9000 to see your project.

Project Structure

```
project_dir/
├── .github/             # GitHub configurations and workflows.
├── app/                 # Application bootstrap files.
├── assets/              # Frontend assets (CSS, JS, images).
├── config/              # Configuration files.
│   ├── routes.yaml      # Route definitions.
│   └── services.yaml    # Service container configuration.
├── public/              # Web server root directory.
│   ├── index.php        # Application entry point.
│   └── static/          # Compiled assets (generated).
├── src/                 # Source code.
├── templates/           # Template files.
│   ├── components/      # Reusable template components.
│   ├── layouts/         # Layout templates.
│   └── pages/           # Page templates.
├── tests/               # Test files.
│   ├── fixtures/        # Test fixtures.
│   └── src/             # Source code tests.
├── vendor/              # Composer dependencies.
├── var/                 # Temporary files and caches.
├── .gitignore           # Git ignore configuration.
├── LICENSE              # MIT License.
├── package.json         # NPM configuration.
├── php-cs-fixer.php     # PHP CS Fixer configuration.
├── phpstan.neon         # PHPStan configuration.
├── phpunit.xml          # PHPUnit configuration.
├── README.md            # Project documentation.
└── vite.config.js       # Vite build configuration.
```

Customization

After creating a project based on this foundation, you can:

1. Modify `README.md` with your project-specific details.
2. Update `package.json` with your project information.
3. Add your routes in `config/routes.yaml`.
4. Customize services in `config/services.yaml`.
5. Create controllers and other classes in the `src/` directory.
6. Add templates to the `templates/` directory.
7. Write tests in the `tests/` directory.

Last updated on 21/08/2026

Project Structure

Project Structure

Anonymous · 21/08/2026

Project Structure

Derafu Foundation provides a standardized directory structure for your projects. 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.yaml`: Route definitions for your application.
- `services.yaml`: Service container configuration (dependency injection).

`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.

File Copying Mechanism

The `Installer` class in `Installer.php` is responsible for copying foundation files to new projects during Composer installation. This class:

1. Defines a list of files to copy in the `FILES` constant.
2. Copies each file from the foundation package to the project directory.
3. Creates necessary directories if they don't exist.
4. Handles file overwrite rules (some files are configured to be overwritten in updates, others not).

Files marked with `true` in the `FILES` constant will be overwritten if they already exist:

```
private const FILES = [
    // Files that will not be overwritten if they exist.
    '.github/workflows/cd.yml',
    '.github/workflows/ci.yml',
    // Files that will be overwritten even if they exist.
    'app/bootstrap.php' => true,
    // ...
];
```

Note: The idea behind overwriting files is to make it easier to update the foundation. But, you can disable it by removing the `Derafu\\Foundation\\Installer::copyFiles` from the `post-install-cmd` and `post-update-cmd` scripts in your `composer.json` file.

Extending the Structure

When creating a new project based on this foundation, 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 21/08/2026

Assets

Frontend Asset Management

Anonymous · 21/08/2026

Frontend Asset Management

Derafu Foundation 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>
<img src="/static/img/logo.png">
```

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 (remember to deactivate the `Derafu\\Foundation\\Installer::copyFiles` from the `post-install-cmd` and `post-update-cmd` scripts in your `composer.json` file).

Last updated on 21/08/2026

Code Quality

Code Quality Tools

Anonymous · 21/08/2026

Code Quality Tools

Derafu Foundation comes with pre-configured code quality tools to ensure consistent code style and identify potential issues early in development. This document explains how to use these tools effectively.

PHP CS Fixer

PHP CS Fixer is a tool that automatically fixes PHP coding standards issues in your code according to rules you define.

Configuration

The configuration is defined in `php-cs-fixer.php` at the root of your project. Key features include:

- PSR-12 coding standards by default.
- Strict type declarations enabled.
- Automatic array syntax conversion to short syntax.
- Ordered imports.
- Modern PHP features optimization (e.g., arrow functions).
- PHPUnit strict assertions.

Usage

Run PHP CS Fixer to check code style issues:

```
vendor/bin/php-cs-fixer fix --dry-run --diff
```

Fix code style issues automatically:

```
vendor/bin/php-cs-fixer fix
```

PHPStan

PHPStan is a static analysis tool that finds bugs in your code without running it. It's focused on finding errors in code logic.

Configuration

The configuration is defined in `phpstan.neon` at the root of your project. By default, it:

- Sets analysis level to 5 (out of 9)
- Analyzes code in the `src` and `tests` directories

Levels Explained

- Level 1: Basic checks.
- Level 2: Possibly undefined variables.
- Level 3: Return types, phpdocs.
- Level 4: Type hints.
- Level 5: Basic dead code detection.
- Level 6: Detecting unreachable code.
- Level 7: Union types.
- Level 8: More precise analysis
- Level 9: Mixed type detection

Usage

Run PHPStan analysis:

```
vendor/bin/phpstan analyse
```

PHPUnit

PHPUnit is the standard testing framework for PHP applications.

Configuration

The configuration is defined in `phpunit.xml` at the root of your project. Key features include:

- Test coverage reporting enabled.
- Strict mode enabled (fails on warnings, notices, etc.).
- Test results output to `var/tests-coverage.txt` and `var/tests-coverage.xml`.
- Test documentation output to `var/tests-testdox.txt`.

Directory Structure

- `tests/src/`: Unit tests for your source code.
- `tests/fixtures/`: Test data and fixtures.

Usage

Run the test suite:

```
vendor/bin/phpunit
```

Generate test coverage reports:

```
vendor/bin/phpunit --coverage-html var/coverage
```

Integration with CI/CD

These tools are integrated into the CI workflow (`.github/workflows/ci.yml`), which automatically runs on each push to the repository. This ensures that code quality standards are maintained throughout development.

Best Practices

1. **Run tools locally before committing**: This helps catch issues before they enter the codebase.
2. **Gradually increase PHPStan level**: Start with level 5 and work toward higher levels as your project matures.
3. **Aim for high test coverage**: Write tests for all critical code paths.
4. **Update rules as needed**: Customize tool configurations to match your project's specific requirements (remember to deactivate the `Derafu\\Foundation\\Installer::copyFiles` from the `post-install-cmd` and `post-update-cmd` scripts in your `composer.json` file).

Last updated on 21/08/2026

GitHub Actions

GitHub Actions

Anonymous · 21/08/2026

GitHub Actions

Derafu Foundation 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.

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 multiple PHP versions for testing.
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.

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: ['8.3', '8.4']  # 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:

```
![GitHub last commit](https://img.shields.io/github/last-commit/derafu/project/main)
![CI
Workflow](https://github.com/derafu/project/actions/workflows/ci.yml/badge.svg?branch=
main&event=push)
![GitHub code size in
bytes](https://img.shields.io/github/languages/code-size/derafu/project)
![GitHub Issues](https://img.shields.io/github/issues-raw/derafu/project)
![Total Downloads](https://poser.pugx.org/derafu/project/downloads)
![Monthly Downloads](https://poser.pugx.org/derafu/project/d/monthly)
```

Replace `derafu/project` 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 21/08/2026