Documentation Tool: PHPDoc

March 2025

Contents

PHP Documentation

1. Purpose

This document provides an in-depth exploration of PHP documentation using PHPDoc, focusing on its purpose, syntax, best practices, and integration with documentation tools. It aims to offer a comprehensive framework for developers to generate and maintain high-quality documentation, suitable for both general and project-specific contexts.

2. What is PHPDoc?

PHPDoc is a standard for documenting PHP code using specially formatted comments. These comments, enclosed in `/** ... */`, contain tags that provide metadata about code elements such as classes, functions, parameters, return values, and exceptions. Documentation tools can parse these comments to generate human-readable documentation in formats like HTML, PDF, or XML, automating the process and ensuring consistency.

- **Purpose:** PHPDoc helps developers understand code intent, usage, and structure, reducing cognitive load and improving collaboration. It is particularly useful for large codebases, APIs, and open-source projects where documentation is critical for users and contributors.
- **Relation to Tools:** While PHPDoc is the commenting standard, phpDocumentor is the tool that interprets these comments to generate documentation, offering flexibility in output and customization.

3. Writing PHPDoc Comments

PHPDoc comments are written using the `/** ... */` syntax and placed immediately above the code element they describe. The general structure includes:

- A brief description of the element.
- Tags starting with `@`, each on a new line, providing specific metadata.

Below is a detailed list of common PHPDoc tags, their purposes, and examples:

Tag	Purpose	Example
`@package`	Specifies the package the element belongs to	`@package MyProject`
`@author`	Identifies the author of the element	`@author Jane Doe <jane@example.com>`
`@version`	Specifies the version of the element	`@version 2.1.0`
`@since`	Indicates introduction version	`@since 1.0.0`
`@param`	Describes a parameter (type, name, desc)	`@param int $id User ID`
`@return`	Describes the return value (type, desc)	`@return string User name`

@throws	Specifies exceptions thrown	@throws RuntimeException
@see	References related elements	@see User::getName()
@link	Provides an external URL	@link https://example.com
@var	Describes a property or constant type	@var int for a property
@todo	Indicates tasks to be done	@todo Implement caching
@deprecated	Marks an element as deprecated	@deprecated Use newMethod instead
@internal	Indicates internal use only	@internal For internal use only
@example	Provides an example of usage	@example See example.php

Examples

Here are practical examples of PHPDoc comments for different code elements:

Class Documentation:

```
/**
 * Manages user data in the application.
 *
 * @package MyProject\Models
 * @author John Doe <john.doe@example.com>
 * @version 1.0.0
 * @since 1.0.0
 */
class User {
   private $id;

   /**
    * Constructs a new User instance.
    *
    * @param int $id The user's unique identifier.
    */
   public function __construct(int $id) {
      $this->id = $id;
   }
}
```

Function with Exception Handling:

```
/**
 * Performs a division operation.
 *
 * @param float $dividend The number to divide.
 * @param float $divisor The number to divide by.
 * @return float The result of the division.
 * @throws DivisionByZeroException If the divisor is zero.
 */
```

```
function divide(float $dividend, float $divisor): float {
    if ($divisor == 0) {
        throw new DivisionByZeroException("Division by zero is not allowed.");
    }
    return $dividend / $divisor;
}
```

Property and Constant Documentation:

```
class MyClass {
    /**
     * The user's ID.
     *
     * @var int
     */
    private $id;

    /**
     * Maximum number of items allowed.
     *
     * @var int
     */
    const MAX_ITEMS = 100;
}
```

Inheritance with @inheritDoc:

```
class ParentClass {
    /**
     * Does something.
     *
     * @return void
     */
    public function doSomething() {
        // implementation
    }
}

class ChildClass extends ParentClass {
    /**
     * {@inheritDoc}
     */
    public function doSomething() {
        // overridden implementation
    }
}
```

4. Using Documentation Tools with PHPDoc

To generate documentation from PHPDoc comments, developers can use various tools. Below is a detailed guide for using phpDocumentor, the most standard tool, followed by a brief overview of alternatives.

4.1 Installation

To use phpDocumentor, installation is typically done via Composer, the standard package manager for PHP, ensuring compatibility with modern development workflows. The command is:

```
composer require --dev phpdocumentor/phpdocumentor
```

This installs phpDocumentor as a development dependency, allowing us to run it from the `vendor/bin` directory. Verify installation with:

```
vendor/bin/phpdocumentor --version
```

4.2 Usage

The basic usage involves generating documentation from your PHP source code. The command-line interface is straightforward:

```
vendor/bin/phpdocumentor generate --directory=/path/to/your/project --
target=/path/to/output/directory
```

- `--directory`: Specifies the path to the PHP source code to document.
- `--target`: Specifies where to save the generated documentation.

For example, to document a project in `./src` and save output to `./docs`, use:

```
vendor/bin/phpdocumentor generate --directory=./src --target=./docs
```

4.3 Configuration

For more customized documentation, create a configuration file, typically `phpdoc.dist.xml`, which allows fine-tuning of the generation process. An example configuration is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpdocumentor>
```

```
  <parser>
    <target>./docs</target>
    <directory>./src</directory>
    <ignore>
      <directory>./src/tests</directory>
    </ignore>
  </parser>
  <transformer>
    <template name="clean" />
  </transformer>
</phpdocumentor>
```

Run with this configuration using:

```
vendor/bin/phpdocumentor generate -c phpdoc.dist.xml
```

This file can specify which directories to include or exclude, set templates, and define other options, enhancing control over the output.

4.4 Features

phpDocumentor offers a range of features to support comprehensive documentation:

- **Parsing PHPDoc Comments:** Supports a wide range of PHPDoc tags, including `@param`, `@return`, `@throws`, `@deprecated`, `@todo`, and more, ensuring detailed documentation.
- **HTML Generation:** Produces well-structured HTML documentation with navigable class hierarchies, method details, and indices, optimized for web browsing.
- **Customizable Templates:** Allows use of different templates (e.g., `clean`, `responsive`) or creation of custom templates to tailor the look and feel, supporting branding or specific project needs.
- **Multiple Output Formats:** Generates documentation in HTML (default), PDF, and XML, providing flexibility for different use cases (e.g., HTML for web, PDF for print).
- **Code Coverage Integration:** Can integrate with code coverage tools to show which parts of the code are tested, enhancing documentation with quality metrics.
- **Support for PHP 8 Features:** As of 2025, supports PHP 8 attributes, union types, and other modern features, ensuring compatibility with contemporary codebases.
- **IDE Integration:** Works with IDEs like PhpStorm, allowing live documentation viewing within the development environment, improving developer experience.
- **Performance Optimization:** Designed to handle large codebases efficiently, with caching and parallel processing for faster generation.

4.5 Best Practices

To maximize the effectiveness of phpDocumentor, follow these best practices:

- **Write Clear and Concise Comments:** Ensure PHPDoc comments are accurate, informative, and not redundant with code. For example, describe the purpose of a method, not just what it does (e.g., "Calculates the sum for billing" vs. "Adds numbers").
- **Keep Comments Updated:** Regularly update comments as code changes to prevent documentation drift, ensuring accuracy with each commit.
- **Use Standard Tags:** Stick to standard PHPDoc tags for compatibility with tools, avoiding custom tags unless necessary for project-specific needs.
- **Exclude Sensitive Information:** Avoid including sensitive data (e.g., API keys, passwords) in comments, as they may appear in generated documentation. Use `--ignore` to exclude files containing such data.
- **Document All Elements:** Include docblocks for classes, methods, properties, constants, and functions, ensuring comprehensive coverage. For inherited methods, use `@inheritDoc` to include parent documentation.
- **Handle Edge Cases:** For parameters that can be null, use `type|null` (e.g., `string|null`); for arrays, use `array<type>` or `type[]` (e.g., `array<int>` or `int[]`), ensuring clarity in type documentation.
- **Use Formatting:** Include Markdown or lightweight markup in descriptions for better readability in generated docs (e.g., bold, italic, lists).
- **Automate Documentation Generation:** Integrate phpDocumentor into CI/CD pipelines (e.g., GitHub Actions) to automatically generate and update documentation with each code change, ensuring timeliness.
- **Customize Appearance:** Use templates or CSS to tailor the documentation's look, aligning with project branding or user preferences.
- **Handle Deprecated Code:** Use `@deprecated` to mark obsolete elements, guiding users to alternatives, and `@todo` for planned features, enhancing future planning.

Examples

Simple Class with Method:

```
/**
 * Represents a user in the system.
 *
 * @package MyProject\Models
 * @author John Doe <john.doe@example.com>
 */
class User {
    /**
     * Gets the user's name.
     *
     * @return string The user's name.
     */
    public function getName(): string {
        return $this->name;
    }
}
```

Running phpDocumentor on this will generate HTML with a class page for `User`, showing the description, author, and method details, including the return type and description.

Method with Parameters and Exceptions:

```
/**
 * Performs a division operation.
 *
 * @param float $dividend The number to divide.
 * @param float $divisor The number to divide by.
 * @return float The result of the division.
 * @throws DivisionByZeroException If the divisor is zero.
 */
function divide(float $dividend, float $divisor): float {
    if ($divisor == 0) {
        throw new DivisionByZeroException("Division by zero is not allowed.");
    }
    return $dividend / $divisor;
}
```

This will generate documentation showing the function's parameters, return value, and potential exceptions, aiding users in understanding usage and error handling.

Inheritance with @inheritDoc:

```
class ParentClass {
    /**
     * Does something.
     *
     * @return void
     */
    public function doSomething() {
        // implementation
    }
}

class ChildClass extends ParentClass {
    /**
     * {@inheritDoc}
     */
    public function doSomething() {
        // overridden implementation
    }
}
```

Here, `@inheritDoc` ensures the child class's documentation includes the parent's description, maintaining consistency and reducing redundancy.