CodeMonitoring.Framework Documentation

Release dev-master

The author

Jul 20, 2016

Contents

This documentation covering version dev-master has been rendered at: Jul 20, 2016

This is the documentation for the whole project.

Part I

Introduction

PHP Project to help developers maintain there code quality by using specialized tools for analyzing, but a single tool to visualise and monitor the results.

In general this tool can be used to monitor everything. A CLI call is provided to parse and import output. Different parser can be provided, also different importer can be provided.

Part II

Target Group of this project

Developers at first, with easy to use CLI access. In the future also Companies to monitor the overall code quality across multiple projects, or to monitor anything else.

Part III

Current state

It's a very early state, mostly about documenting ideas and concepts.

Also some parts are kick started to be implemented. Take a look at roadmap.

Part IV

Goal

The target of this project is to provide a well designed web GUI for Code monitoring written in PHP. While this projects is based on PHP and will itself support PHP out of the box, further languages will be possible by extensions.

The project will parse output of further software and display the results as a web GUI. By providing the understandable output you can integrate everything. Also it will be possible to write own parsers to support further formats in the future.

All results will be parsed via CLI and persisted. Code Monitoring will analyze the data and display it with filter and graphs.

Also, due to the concepts and code, it's possible to do anything you want with the information, e.g. send email, twitter, instead of persisting and displaying the information.

Displaying information as a website is just a package itself and not needed. You can use Codemonitoring to monitor access logs, other logs, phpunit results or whatever and do whatever you want with the output, like "if this then that". All you have to do is to provide something to parse, a parser to parse the file, and to provide an importer to do anything with the parsed information.

Some parser and importer will be provided, some will hopefully be developed by other developer in the future, and the rest is up to you.

CHAPTER 1

Install

Inside your shell, run:

```
composer update && \
./flow doctrine:migrate
```

Todo

Update once a first alpha is finished. We should provide an install documentation for the user at this point.

CHAPTER 2

Usage

To import some results, run:

```
./flow import:importfiles <files>
```

E.g. run it with provided examples:

```
./flow import:importfiles Packages/Application/CodeMonitoring.Framework/Resources/
↪Private/Example/checkstyle-psr2.xml Packages/Application/CodeMonitoring.
↪Framework/Resources/Private/Example/phpmd.xml
```

To ease development we provide a `Makefile`.

Todo

Document priority for importer and parser

Todo

Document that first importer and parser will be used.

CHAPTER 3

Features

3.1 Implemented

The following features are currently implemented.

3.1.1 Parse file based information

Todo

Add ref to documentation here.

Parsing is possible and documented at ... . It's possible to write custom parser.

Currently a parser for `checkstyle` is provided inside `CodeMonitoring.Parser.Checkstyle`. The parser is registered and configured to parse `.xml`-files with Checkstyle 2.5.x, like generated through PHP CodeSniffer.

Nothing is done with the parsed information yet, as no importer exists, see *Import file based information*.

3.2 Future

The following features are currently planned for the future. With our current knowledge, that's everything needed at the moment to expand and built arbitrary useful software based on our framework.

3.2.1 Import file based information

Parse different file based outputs like phpunit, phpmd, phpcs. At beginning, only import "file based information", which means information like the above tools generate. Each information is related to a specific file and mostly a line.

Output like phploc is project based and provides overall information not related to specific files. This will be another feature.

3.2.2 Parse and import project based information

Parse output of tools like phploc, which is project based and provides information about the complete project, not specific files.

3.2.3 Display gathered information

First basic display via charts inside a web gui. This should be done in a new package that will grow and provide an API for further development. The API should ease the integration of "widgets" and such, so packages will be possible that will provide a parser, importer and widgets and provide full integration for things like Wordpress, TYPO3, Drupal, Apache Logs, Github Issues, Git, etc.

3.3 Overall *features* to keep in mind during development

The following features can not be implemented, but should be kept in mind during development.

In general, everything that is provided as a feature, should also be expendable by 3rd parties. This is especially true for:

- Allow imports through 3rd parties
- Allow parser from 3rd parties
- Allow graphs from 3rd parties

But also for things like:

The following should already be possible, once the dependencies are implemented, as we use Flow Framework. All the developers have to do, is to provide a package implementing the features. All Information are already available.

- Possibility to provide arbitrary statistics, we don't have in mind yet. Things that might look out of scope like comparison between commits and projects tasks.
- Provide access to all existing information inside the system, so 3rd parties can process them as they wish. E.g. create reports via E-Mail. Provide new displayed information aggregated from existing ones.

CHAPTER 4

Processes

There are many processes involved in this project. To allow you to get an overview how they are organized and work together, we will document them in this section.

Todo

Add all processes

Once a process will be started or finished, it has to be documented. Only this way other developers and 3rd parties are able to understand and extend.

4.1 Importing

To get started, you need to import data from some format like *checkstyle*.

The import process will use an importer together with a parser. The parser will parse the given file, while the importer will use the result for further process, e.g. calculations and persistence.

The parser will return the parsed information in a defined format to the importer. The importer will get the configuration by user and handle the information. E.g. he will raise severity for some issues, or ignore some issues. He will also, depending on configuration, persist the parsed results without modifications. This allows to reimport parsed information onces the config changes. So you can compare the history of a file or project with adjusted configuration and allows comparison. So you can test a new configuration before applying it.

It's up to the `Importer` how to handle the delivers information from parser and what configuration options are available.

At some point we will introduce a queue to process the files. To speed up the CLI Access and use a non blocking process.

4.1.1 Possible Improvements

To ease integration of new importers, we can provide the following:

A **trait** providing the priority, with property and getter. The package can attach it to parser *and* importer and configure priority via `Objects.yaml`.

Also provide **Eel** for detecting whether file can be parsed or imported? Provide some language like `lineXContains("string")`, `lineXStartsWith("string")`, or `fileNameMatches("string")`, stuff like that.

Use different **Backends** attachable to Importer? Things like "mail", "log", "db", ...

Also provide some **Events**, either via AOP or Signal Slots. Enabling developers to register things like an email if a specified conditions occurs, e.g. more then x vulnerabilities against CGL.

CHAPTER 5

Development

At this very early state everything is one repository, even if multiple packages already exist. As soon as first packages get a good state and work together, they will be splitted into there own repositories.

This allows to handle them different and independent. Also they will get their own documentation.

5.1 Extending

The following sections provide documentation how to extend the framework by providing your own parser, importer and such.

In general you have to build a custom Flow package containing your code like parser. Generating this packages is not part of this documentation, but can be found at Flow Docs.

5.1.1 Custom parser

To provide your own parser, you have to create a new PHP Class implementing `CodeMonitoring\Framework\Parse\ParserInterface`. That's all you have to do.

The interface defines the following methods:

```
rface ParserInterface extends CanHandleFileInterface, PriorityInterface

/**
 * This method will be called once the parser will be used.
 * Due this method the file to parse will be provided.
 *
 * Most likely you will "persist" the file in a property for later usage in
 * getData method.
 *
 * @param Resource $file
 *
 * @return void
 */
public function setFileToParse(Resource $file);

/**
 * Will provide the parsed information to the world.
 *
 * This method should return whatever the parser has parsed. Most likely
 * this method will be called by the importer.
 *
 * @return mixed
 */
public function getData();
```

Use Eel expression for `canHandle`

In addition you can use the trait `CodeMonitoring\Framework\Parse\EelParsingDetectionTrait` to implement the `canHandle` method. The trait will implement this method and provide the property `$eelExpression` which can be configured through `Objects.yaml`.

An example can be found in *CodeMonitoring.Parser.Checkstyle*. Add the following to your parser class:

```
use EelParsingDetectionTrait;
```

And the following to your `Object.yaml`:

```
CodeMonitoring\Parser\Checkstyle\Parser\CheckstyleParser:
    properties:
        eelExpression:
        value: 'file.fileExtension == "xml" && String.substr(lines[1], 1, 10) ==
↪"checkstyle" && String.substr(lines[1], 21, 3) == "2.5"'
```

Adjust the first line to match your fully qualified class name. The expression will be parsed through the trait and result in `true` or `false` depending on expression and file.

The above expression will check that the extension of file is equal to `xml` and that the second line contains `checkstyle` in version `2.5`.

Currently the following context is provided to parse the expression:

For more information about *Eel* check out the official documentation about Eel.

5.1.2 Custom Importer

Todo

Document once first version is stable enough.

To provide your own importer, you have to create a new PHP Class implementing `CodeMonitoring\Framework\Import\ImpoterInterface`. That's all you have to do.

5.2 Roadmap

There is no timeline, just the order in which features will, at this moment, be implemented.

1. Implement first example parser and importer, to kick start framework to deal with these tasks and provide an example implementation. Also provide documentation. After this is implemented, there is already a single standard and others can work with imported information.
2. Implement first output based on imported information. Provide first Views and GUI.
3. Improve eel parsing trait:
4. Provide further example packages to show what is possible and how to implement it. E.g.:
   (a) Provide Package to import Github issues and display them via Web GUI Package.
   (b) Provide E-Mail importer with rule set via configuration to allow sending e-mails for all kind of imports.

Once we have the first two points finished, an alpha version 1.0.0 will be released that should be ready to use.