# Software Release Document

# Group 1: Cooperatio

Robert Kraaijeveld, Steven Schenk, Robin Bakker, Cees-Jan Nolen

Table of Contents

[Software Release Document 0](#_Toc441174733)

[Group 1: Cooperatio 0](#_Toc441174734)

[Introduction 2](#_Toc441174735)

[Our systems’ design 3](#_Toc441174736)

[The system-environment 4](#_Toc441174737)

[Hardware: 4](#_Toc441174738)

[Software: 4](#_Toc441174739)

[The deployment pipeline 6](#_Toc441174740)

[JENKINS 6](#_Toc441174741)

[DOCKER 6](#_Toc441174742)

[JENKINS JOBS AND THEIR STEPS FOR INPUT/PROCESSING 7](#_Toc441174743)

[Build-job 7](#_Toc441174744)

[Test-job 7](#_Toc441174745)

[Release-job 7](#_Toc441174746)

[Deployment-job 7](#_Toc441174747)

[JENKINS JOBS AND THEIR STEPS FOR INPUT/PROCESSING 8](#_Toc441174748)

[Dependency grabbing-job 8](#_Toc441174749)

[Build and test-job 8](#_Toc441174750)

[Deployment-job 8](#_Toc441174751)

[UPGRADING THE SOFTWARE 9](#_Toc441174752)

[The upgrade process 9](#_Toc441174753)

[Steps: 9](#_Toc441174754)

# Introduction

This document has been created in order to communicate our project56 Continuous Integration setup to our product owner.

In this document we describe:

- Our projects’ structure; The components and the way they are divided.

- The dependencies and configuration of our software.

- The setup of our server-environment

- The setup of our continuous integration system; Our build/test/deployment-pipeline.

- How our system should handle upgrades.

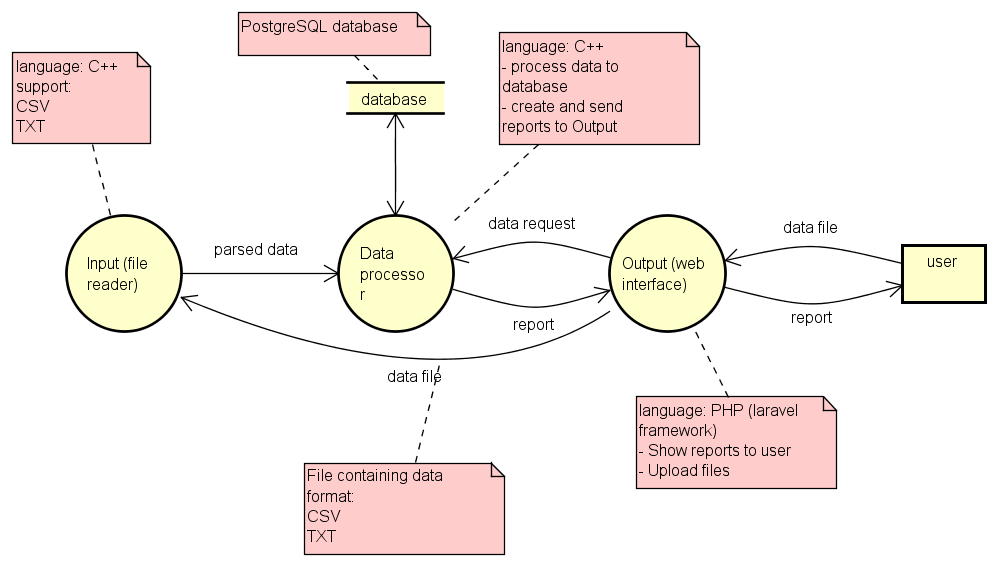
We hope that this document will provide a clear view of the measures we have taken as a team to create a high-quality software system, including its’ (extensive) deployment pipeline.

# Our systems’ design

Our system is divided into these 3 subsystems:

1. Input-system.
2. Data-processing system.
3. Output-providing system.

These 3 subsystems all have their own processes and functionalities which create our entire system. The graphic visualization below gives you some more insight into our systems’ structure.



**Graphic 1: Context diagram and system boundary**

The system-environment

Our system will be hosted at a Hogeschool Rotterdam virtual machine, located on an actual (physical) server. Below you will find a list of our virtual machines’ hardware, and below that the software that our system depends on.

Hardware:

* CPU: Intel Xeon e5-2650 @2.00 Ghz
* Memory: 3953 Megabytes
* Disk: 15 Gigabytes
* OS: Ubuntu 14.04.3

Software:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Version number** | **Short description** | **Preferred download method** |
| PHP\*  programming language | 5.6.16 | PHP server-side programming language. | Archive download (<http://php.net/downloads.php>) or Aptitude install on Linux. |
| C++ and GNU compiler | Unavailable | Backend programming language + compiler. | Available out of the box in most OS’s. |
| Postgres\* | 9.3.10 | SQL-style DBMS. | Aptitude install on Linux. |
| Nginx\* | 1.4.6 | Lightweight webhost. | Aptitude install on Linux. |
| Git | 1.9.1 | Repository manager | Aptitude install on Linux. |
| Composer\* | 1.0-dev | PHP package manager. | Curl-install script. (https://getcomposer.org/download) |
| The Laravel Framework | 5.1.26 | MVC-style framework for PHP. | Composer:  ‘composer global require "laravel/installer=~1.1" ‘ |
| Phpunit\*\* | 5.1.3 | PHP unit-testing framework. | Composer:  ‘composer global require "phpunit/phpunit=5.1.\*" ’ |
| Apache Ant\* | 1.9.3 | Building engine. | Archive download (<https://ant.apache.org/bindownload.cgi>) or Aptitude install on Linux. |
| PHPDox\*\* | 0.8.0 | PHP XML/HTML  Documentation generator. | Phar file: ‘https://github.com/theseer/phpdox ‘ |
| PHPLoc\* \* | 2.5.1 | PHP static analyzer. | Composer:  ‘composer global require 'phploc/phploc=\*' ‘ |
| PHPMailer\* | 5.2 | PHP mailing lib. | Composer‘composer require phpmailer/phpmailer’ |
| Lpqxx\*\* | Unavailable | Postgres to C++ API. | Tarball/Zip download: http://pqxx.org/development/libpqxx/wiki/DownloadPage |
| UnitTest++\*\* | Unavailable | C++ unit-testing framework. | Source code download: https://github.com/unittest-cpp/unittest-cpp |
| Cmake | Unavailable | C/C++ Compiler. | Tarball/Zip download: https://cmake.org/download/ |
| CppCheck\*\* | Unavailable | C/C++ static code analyzer. | Tarball/Zip download: http://cppcheck.sourceforge.net/ |

\*: These software-systems are used to accommodate development and building in the Laravel framework.

\*\*: These systems are downloaded in our deployment pipeline and need not be installed manually.

The deployment pipeline

JENKINS

To build, test and deploy (which shall from now on shall be called ‘deploying’) we use Jenkins. Jenkins allows you as a programmer to create ‘Jobs’: A job may build a project using ant or it may just execute a shell script. A job can be ran automatically (periodically on or certain triggers) and can also be configured to provide reports covering things like test code coverage or compiler warnings. The amount of successful/failed builds/jobs is also listed in Jenkins, enabling us to create trend graphs and the like.

DOCKER

We use the Docker system in order to manage Postgres and our Nginx webserver, since they are both dependencies that need to be running all the time. Docker is a pretty unique concept: It is a bit like a Virtual Machine, but with much, much less overhead.

The reason that a Docker instance (known as a container) is much faster than a VM is because each container shares its operating system with the main Docker process.

Our two containers both have a synchronized folder: The /var/www/laravel folder on each container points to the /var/www/laravel directory on the server, as explained below.

uses

Synchronized to

Synchronized to

/var/www/laravel/

/var/www/laravel/

Uses

Webserver container

Postgres Container

/var/www/laravel/

Server

JENKINS JOBS AND THEIR STEPS FOR INPUT/PROCESSING

**Note**: Even though in this document we treat the two pipelines as being separate, they should both be executed when new code is being pushed to git. We treat the two pipes separately because they do different things for different code.

Build-job

1. Change directory to (home/USER/)
2. Git clone the project56 repo in (cooperatio/Project56)
3. Change directory to the repo location (/home/USER/project56)
4. Compile the source code, both the main processes and the Unit tests (g++ command with flags: -std=c++11 –pthread –lpqxx –lpq –lUnitTest++)

Test-job

1. Go to the repo directory if we’re not there yet (/home/USER/project56)
2. Execute the pre-compiled Unit tests.
3. Execute CppCheck on both the input and processing subsystems.

Release-job

1. Configure and run the postgres and webserver docker containers.
2. Create our PostgreSQL database.

3. (re)Create our Database tables using an Artisan Migrate snapshot.

Deployment-job

1. Copy the website-files from the http branch to the webhost directory (/var/www/laravel)
2. Run a composer update in (/var/www/laravel)
3. Copy any necessary C/C++ files to the webhost directory if necessary.
4. Start our input background-process.

JENKINS JOBS AND THEIR STEPS FOR INPUT/PROCESSING

**Note**: Even though in this document we treat the two pipelines as being separate, they should both be executed when new code is being pushed to git. We treat the two pipes separately because they do different things for different code.

Dependency grabbing-job

1. Git clone the project56 repo in (cooperatio/Project56) if it isn’t there yet.

2. Change directory to the repo location (/home/USER/project56)

3. Download the PHP tools used during the build.

Build and test-job

1. Go to the repo directory if we’re not there yet (/home/USER/project56)
2. Build using Apache Ant. The build steps Ant takes are listed below.
3. Execute PhpLoc, PhpUnit and PhpDox.
4. Generate HTML/XML reports.

Deployment-job

1. Make sure the Postgres and Webserver docker containers are running, start them if they aren’t running.
2. Copy the website-files from the http branch to the webhost directory (/var/www/laravel)
3. Run a composer update in (/var/www/laravel)

UPGRADING THE SOFTWARE

The upgrade process

The best option for handling upgrades to our system would be to have 2 servers/Vm’s running in parallel: One server containing the old version, One server actually serving the users. In that case, an update could be deployed on server B whilst server A takes over server B’s users until the update is done.

However, because of a lack of time and resources we only have one Vm available. Therefore we will have to take some measures to ensure that system-downtime during an update is as low as possible.

Our system is set up in such a way that some processes can be ran twice at the same time without interrupting each other. Because of this we can easily compile a new version of a process and start running that process right away, whilst the old process still has time to finish its’ job.

Steps:

1. Pull the new code from the github repository.
2. Compile the new source code.
3. Position the new source code where it is needed.
4. Start the new process.
5. Stop the old process as soon as it is no longer being used.

MAKING SURE AN UPDATE DOESN’T HAVE UNforeseen consequences

To make sure an update doesn’t break other code (especially code that is dependent on the updated code) an update has to be properly tested in a testing environment that replicates the actual production environment as closely as possible, so that any unintended consequences will not be overlooked. In other words; an upgrade should always be prepared for unforeseen consequences.