# Development B

# Continuous Delivery

# Assignment C: Release-documentation output-subsystem

By Robin Bakker en Robert Kraaijeveld, INF2C

Table of contents

[Stakeholders 3](#_Toc440647831)

[Asset and configuration management 4](#_Toc440647832)

[Deployment steps 6](#_Toc440647833)

[Release plan-timeline and team responsibilities 9](#_Toc440647840)

[Environments and upgrades 10](#_Toc440647845)

[Upgrading live software to newer versions 11](#_Toc440647847)

# Stakeholders

Within this project, the following key stakeholders can be identified:

- The CityGis company: CityGis has a stake in this project because (within the ‘scenario’) they commissioned and provide the resources for this project. They want their data to be presented to the users and other stakeholders in a way that will reflect the quality of their work.

- The Car maintainer: The vehicles that use CityGis-equipment are maintained by a maintainer. This company/person would obviously like to know which cars have defects or exhibit strange behavior as soon as possible, since vehicles for emergency services are very scarce.

- The data-analyst/Commercial-user: The data-analyst or commercial user, from here on referred to as ‘the user’ wants to have the clearest and most complete (over)view of the CityGis data as possible to get his/hers moneys’ worth.

# Asset and configuration management

The Project56 output-subsystem has been programmed in PHP5, combined with the usage of the Laravel-framework. Below you will find all the assets, dependencies and libraries being used within our output subsystem together with their version numbers.

**Software: dependencies necessary for running and/or building.**

|  |  |  |  |
| --- | --- | --- | --- |
| **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. |
| 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 code analysis interface. | Composer:  ‘composer global require 'phploc/phploc=\*' ‘ |
| PHPMailer | 5.2 | PHP mailing library. | Composer  ‘composer require phpmailer/phpmailer’ |

\*= These dependencies should only be downloaded within the Jenkins shell script mentioned later.

**Hardware: hardware requirements recommended for running the system.**

|  |  |  |  |
| --- | --- | --- | --- |
| **CPU** | **Memory** | **Hard disk-space** | **Operating System** |
| Intel Xeon e5-2650 @2.00 Ghz per core | 3953 Megabytes | 15 Gigabytes free without OS | Ubuntu 14.04.3 |

Note that these hardware requirements are the hardware requirements that we (I.E: Team Cooperatio) use; The system could also be run with less RAM and about +- 10 gigabytes hard disk-space excluding the OS. A Linux OS is not mandatory for the output-subsystem, but it is required to run the Input- and processing-subsystems.

# Deployment steps

# Tech usage

In order to continuously build, test and deploy our subsystem we use the Jenkins Continuous Integration system. Jenkins runs locally on our Ubuntu server and is accessible from our servers’ IP-address. Jenkins builds and tests our source files and also creates (or rather, displays already created) HTML reports from PHPDox, one of the dependencies mentioned in the previous chapter. More on that subject later.

# (Jenkins) Deployment pipeline steps

Note: In Jenkins, we make the distinction between Build actions and Post-build actions. Build actions are any kind of actions that create output (usually in the form of usable software + reports) and Post-build actions use that output. A build can be triggered by a certain time-interval or by a git push.

# 1. Build actions:

**-** Within Jenkins, run a shell script consisting of the following commands:

cd /

cd /home/username/preferredProjectLocation/

sudo ./Buildscript.sh

This shell script (Buildscript.sh) should:

**Repo initialization:**

- Initialize a new local Git repository in the current directory and clone the Project56 git repository if it is not already present. If the repo is already present, the script should only try to pull any changes to the repository using git pull.

**Dependency downloading:**

- Proceed to download several PHP-phar archives (containing the dependencies marked with an asterisk in the last chapter.) and unpack them in the usr/bin/ directory, so these dependencies can be used directly via their name.

**Building through ant; analyzing, testing and reporting:**

- Proceed to call ant on the build.xml file located in preferredProjectLocation/Project56/http/Laravel. This build file is too long to fully disclose in this document, but it should do the following things:

Run PHPLoc on all the PHP files within project56/Project56/http/public. PHPLoc produces human readable output that lists a static code analysis of all the PHP source code it finds. PHPLoc lists mostly simple statistics like the amount of methods and classes etc.

Run PHPUnit; PHPUnit runs 2 simple unit-tests contained in the project56/Project56/http/Laravel/tests directory. The result of these tests is displayed in the console output. Note that the build will not pass unless all tests pass. More unit-tests may be added freely.

Run PHPDox; PHPDox saves the contents and structure of all the source files in preferredProjectLocation /Project56/http/Laravel/public and saves the output in an index.xhtml file in the preferredProjectLocation /Project56/http Laravel/build/api/html directory.

# 2. Post-build actions:

**Publishing reports:**

- After the build steps above are complete, publish the index.xhtml file that PHPDox generated. Jenkins has an HTML-report plugin for this, which takes a path and a file as input. The provided file will be displayed when you click on the “HTML Report” button at each build-attempt.

# 3. Release:

**Preparing server:**

**-** Nginx should be configured in order to actually deploy the output website and make it publicly visible from our servers’ IP-address. An nginx.conf file containing the configuration for the server should be created and should overwrite the standard nginx.conf file in the nginx installation directory.

*Note that you could potentially use a different webhosting-service than Nginx, such as Apache. However, we found that nginx is much more lightweight than heavier webhosting-services like Apache and also works better with Laravel in some regards. The choice is yours.*

- PostgreSQL should be properly configured. This includes creating a user (Linux only) to contain the Postgres processes, creating a Database for the output-subsystem to use and filling this database with tables. The Artisan system, included in the Laravel framework enables “Migration”; migration basically means inserting a table-structure-snapshot into a database. You can use this feature to have your tables created much quicker than they could ever be created by hand. The migration snapshots are contained within the Laravel/migrate/ directory of the subsystem.

# 4. Deployment:

**Making files publicly visible:**

After the Jenkins build, post-build and server preparation steps are complete, the (sub) system should be made public. To make sure that deployment goes smoothly and all dependencies are up-to-date, the following steps should be taken:

*Note that for this part of the CI-pipeline, having a Linux OS makes everything significantly easier. The var/www/ directory is always connected to the systems’ webhost, so no advanced configuration is needed. Windows user should think about hosting the output-subsystem on an (external) Linux server.*

- All the files under the preferredProjectLocation/Project56/http/Laravel/ directory should be copied to the /var/www/Laravel/ directory.

**Updating dependencies and starting background processes:**

- In the /var/www/Laravel/ directory, composer update should be run. This command ensures that all dependencies that are managed by composer in this directory are updated.

- The Postgres and Nginx processes should be manually started if they haven’t started running already.

- For all the output-features to run correctly, the input- and processing-subsystem should also be properly installed and configured. The details for installing and configuring in these subsystems are described in a different document.

# Release plan-timeline and team responsibilities

# Week 2-6: Research and design

In the first weeks of the project we will focus on trying to get a better understanding of our stakeholders’ wishes. As mentioned earlier, each stakeholder has a different vision for this product; this fact must be incorporated into our research. Week 2 to 6 are also going to be used to create initial visual designs and implement some simple functionality into our output-subsystem. UX-research will also take place, since the output-subsystem is the main visual “hook” for potential CityGis customers.

# Week 6-16: Implementation with weekly prototype

The biggest part of the project timeline is taken by the actual implementation of the (sub)-system. In these weeks we will start with a very simple prototype of the several subsystems and gradually build more advanced prototypes. Each prototype will be presented to our Product owner weekly. The prototypes will all be deployed in our testing environment, and be moved to production when their correctness is ensured.

# Week 16-18: Extra time to work on deployment pipeline and testing

The last few weeks will be treated as extra time, in which we will be able to perfect our deployment pipeline and create more (unit) tests.

# Team responsibilities:

|  |  |
| --- | --- |
| **Team member** | **Task / Main focus** |
| Robin Bakker | Output-subsystem |
| Robert Kraaijeveld | Processing-subsystem |
| Steven Schenk | Processing-subsystem, Team Leader |
| Cees-Jan Nolen | Input-subsystem |
| William de Visser\* | Output-subsystem |

\*= This team member has resigned from the team.

# Environments and upgrades

# Development, testing and production environments

For the project we are using 3 environments, an environment where can be developed, a testing environment and an deployment environment.

The development environment is largely on every team members local machine. This also means that every environment differs significantly. With the help of GitHub we try to maintain up to date software on every local machine, so that the code under developing doesn’t differ from the other team members. For the front-end part of the project we are using a Vagrant Homestead virtual machine on our machines. This means we can simulate a server environment. Despite that fact the environments can still differ slightly.

The testing environment is a virtual Ubuntu server. Also all the software needed to run our project is installed and configured. Our test environment is the same as the deployment environment, which is simply the /var/www/ location on the virtual Ubuntu server. The test environment is the same as the deployment environment because the test environment has to be a reflection of the deployment environment so that the test results will be as realistic as possible.

As the deployment environment is the same as the testing environment, there is nothing more to explain about the environment.

Below, the extra software dependencies needed for running the entire Project56 system is listed. This software is necessary for running the test environment accurately.

* G++ 4.8.4
* UnitTest++ 1.5.0
* CMake 2.8.12.2
* CppCheck 1.61
* LibPqxx 4.0

# Upgrading live software to newer versions

When upgrading the software on the server which is live and used by users, it is importing to maintain as least downtime as possible. On just one server this is really hard or maybe even impossible, but it is possible on multiple servers. Then you would direct the user traffic to the other server and upgrade the first one. When the upgrade is done you redirect the users to the upgraded server and then you update the other server.

The deployment environment we are using with Project56 is a virtual machine on a server owned by school. Because we only have one machine, it is hard to maintain zero downtime deployment. When we would have had two virtual machines it would have been able to upgrade the second virtual machine and when that’s done, shut down the first one and direct all the traffic to the second already upgraded machine.

The data migration between the new and old version will be done by ODB. This software compares the deltas between the databases and when a database is upgraded, it can still insert old data into it. This way you don’t lose the data in the old database.

Steps:

1. Create backup of database with ODB
2. Get the new source code from GitHub.
3. Compile the new source code
4. Put the compiled software on the right place
5. Let ODB create the database and migrate the data
6. Start the new compiled software
7. Close the old software process as soon as this one isn’t used anymore