A continuous integration system for MPD Root: Deployment and setup in GitLab

G. Fedoseev, A. Degtyarev, O. Iakushkin[a], V. Korkhov

Saint-Petersburg State University (SPbU)

E-mail: [a]o.yakushkin@spbu.ru

The paper is focused on setting up a system of continuous integration within the available infrastructure of MPD Root project. The system's deployment and debugging for MPD Root purposes are considered; the installation and setup of the required tools using GitLab, GitLab CI Runner and Docker are described.

The deployment of various MPD Root configurations includes four major steps: installation of environmental components; creation of Root and FairRoot dependencies; building MpdRoot; and MpdRoot testing.

The build scripts and the container image configuration were stored in a repository. This allowed to alter the configuration of a build's dependencies without the developer having to directly access GitLab CI Runner service.

Keywords: Docker, GitLab, CI, caching dependencies, MPD Root

The work was supported by RFBR research project № 16-07-011113 and SPBU projects 0.37.155.2014 and 9.37.157.2014

Introduction

The development of modern software requires flexible tools employed to monitor and analyze changes made by developers [Abrahamyan 2016, Shichkina 2016, Iakushkin 2016]. The data on such changes is collected and analyzed by means of continuous integration (CI) systems. The deployment of such systems is often complicated by the long time needed to build external libraries and components—that is, the project's dependencies. This paper is focused on incorporating CI processes into the MPD Root project.

MPD Root is a set of programs and libraries that allows to simulate and analyze the data obtained on the Multi Purpose Detector (MPD). The MPD Root project stores and manages source code versions in GitLab, a repository system. GitLab also offers GitLab CI, a CI system integrated to the repository. However, the project now uses Kitware Cdash—a different system for automated building and testing.

Contrary to GitLab CI, Kitware Cdash runs builds and tests according to a schedule and does not allow developers to identify bugs in the code at the moment of submitting changes to the repository. Conversely, GitLab CI enables users to obtain information on building errors via the web interface of GitLab's repository.

GitLab allows to create a homogeneous isolated environment for building and testing. That means there is a separate container allocated to each of CI tasks. The environment in a separate container includes only those components that are required for the software in question. This approach prevents any interference of unrelated processes. The container system is implemented on Linux using Docker technology. For a detailed description, please refer to "Docker" and "CI setup" sections.

The automated building and testing returns either a positive or negative result. The former means that the software was built successfully and all the tests also completed successfully. The latter means there was an error during one of build stages.

The goal of our study was to set up a continuous integration system in the existing infrastructure of MPD Root.

The system we developed is different from the existing one in the following ways:

- software is tested more often
- builds are isolated in containers
- the system is integrated to the source code repository.

Problem statement

There were a number of smaller tasks to be resolved in order to attain the goal of our study:

1. to replicate MPD Root's organisation of source code storage on a separate server: to set up GitLab CE (the version control server) and to clone the MPD Root repository thereto;
2. to deploy and set up GitLab Runner (the build and test server) and to connect it to the version control server;
3. to develop the scripts that define building in the container format
4. to test and optimize the performance of the resulting system

The system we chose to develop was to meet the following criteria:

- Building and testing must take place after every change of the project's source code—i.e., after every commit.
- The entire build and test process must be defined in text files stored in the project's repository and incorporated into the version control system. This is especially important, because it allows to run every build according to a separate rule [Iakushkin 2015, Bogdanov 2015].
- The average build and test time must not exceed ten minutes [Elbaum 2014].
- Every build must take place in a separate container.

- A report on every build must be generated. The reports must be archived.
- The system must be easily set up and executed on Linux servers.

Overview of technology

The following hardware and software were used to achieve the goals of our study:

Servers

We selected two Windows Azure virtual machines to install the CI system and to run the tests. The machines have an operating system with the Linux kernel. One of them, with the instance size of D4V2, was used to deploy the server with the version control system. The other, with the instance size of D15V2, was used for continuous integration.

GitLab CI Runner

The software market has plentiful solutions for continuous integration—e.g., Shippable, Travis-CI, AppVeyor, etc. For our purpose, we selected GitLab CI Runner, a server-based software with open source code. GitLab CI Runner was created by GitLab developers to run continuous integration tasks and can be directly integrated to GitLab repository manager.

The GitLab repository allows to connect many remote execution servers with GitLab CI Runner. Each server (runner) will execute build and test tasks. There are two types of runners: Shared and Specific.

- Specific runners are typically used to build projects that have non-standard requirements to the environment; such runners can be linked to one project only.
- Shared runners can be linked to many projects.

Docker

We used Docker to isolate parallel CI tasks from each other. Docker is an open source virtualization framework specifically designed to launch software applications. It is one of the most popular and technologically advanced solutions for software containerization [Bogdanov 2014], which means there is detailed documentation and solid community support. Docker provides an additional layer of abstraction at the operating system (OS) level. It allows to run multiple isolated environments on a single OS to perform simultaneous execution of applications (containers). Compared to virtual machines, Docker allows to avoid the overhead of launching a full-fledged guest OS.

Docker containers are created based on images which document the state of the container. To run a container, you need to pick up the relevant image. The images, in their turn, consist of layers—i.e., objects that characterise the changes made to the file system. Each layer has a unique ID similar to a commit in version control.

Layers are stacked on top of each other to form a base for a container's file system.

When you create a new container, a new writable layer is added on top of the underlying stack. All changes made to the running container—such as writing new files, modifying existing files, and deleting files—are written to this new container layer.

Deployment and setup

The system under consideration requires two servers: one for version control and the other, for building and testing. These servers must have a network connection between them. In addition, there must be scripts defining the IC process to ensure the system's correct operation.

Setup of version control server

The GitLab Community Edition package was installed on the version control server. A web interface was available right after the installation and was used for further setup. First, we added a new project from the existing repository located at https://git.jinr.ru/nica/mpdroot.git. Then we opened port 80 to allow access to the repository manager from the Internet—this was needed to connect to the build and test server.

Build and test server

The gitlab-ci-multi-runner package was downloaded and installed on the build and test server. After installation, we used a special instruction to register CI Runner with the version control server. We also used a character-based authorization key (token) required for registration—the token is available via the web interface of the version control server. GitLab CI Runner used this token to request another token from the version control server. The second token was used for further interaction.

GitLab CI Runner can use a variety of executors to run CI scripts in different scenarios: directly on the OS (shell, ssh), inside containers (docker, docker-ssh), or on virtual machines (virtualbox, parallels). We selected the shell executor to launch build and test tasks, because it allowed to manage Docker containers with a CI script. In other scenarios, the CI script could only access the environment within a container. This approach enabled us to cache a separate image that includes only those dependencies that were installed at the relevant build stages.

After registration, we set up CI Runner's parameters in /etc/gitlab-runner/config.toml according to the requirements of MPD Root build. The standard restrictions did not work: the time limits placed on tasks did not allow to complete the build process, the massive text output was interrupted, and there were errors in CI. The default time limits resulted in the interruption of the process. Therefore, we increased the limits placed on a single task. We also increased the size of log files. CI Runner was given additional privileges to execute sudo commands without a password.

Docker was installed at the final stage of setting up the build and test server. The setup and use of Docker containers is described below.

Setup of CI process

We created the .gitlab-ci.yml file to configure CI stages and to define the steps taken at each of them. The image of Docker container that runs the CI process is described in Dockerfile. Both files are placed in the root of MPD Root's repository.

File .gitlab-ci.yml defines three CI stages: build, test, and cleanup.

The build stage was divided in two parts: compiling the image of build and test system using Dockerfile, and compiling MPD Root inside a container that uses the previously built image.

After MPD Root was compiled, a new layer was saved to the image and then was used by another container for testing.

The cleanup stage released any resources allocated to containers and deleted redundant images. Only one image was saved: the image that contained the build and test system with FairSoft.

Our Dockerfile had instructions for the following components to be installed to the container: Ubuntu 14.04, FairSoft dependencies and FairSoft package. At the end of Dockerfile, we added an instruction to copy all source files of the project from the repository to the container. Further they were to be built inside the container. All of the steps mentioned here were formulated as separate commands. Docker images are designed so that every instruction creates a new layer on top of the previous one. In other words, a successful execution of each instruction creates a new control point. The first compilation of an image from Dockerfile involves the execution of all its commands in succession. But after the Dockerfile is modified, the layers are rewritten starting from the commands that underwent changes.

There are ADD instructions that always result in rewriting of all layers in the image. These instructions are used to copy files to the container. That is why we placed the ADD instruction at the end of Dockerfile to copy MPD Root's source files to the container.

The layer system of Docker image allowed to drastically reduce the time of dependencies compilation. The dependencies are compiled only once and are stored in Docker image thereafter. That is, they are already installed, and a layer with updated main software can be written right on top of them.

Conclusion

This paper describes the possibility of transferring the automated build and test functionality from CDash to GitLab CI, a continuous integration system. It contains a detailed account of deploying and debugging a continuous integration system for the MPD Root project. It also describes the setup of the required tools using GitLab, GitLab CI Runner, and Docker.

References

Elbaum, Sebastian, Gregg Rothermel, and John Penix. "Techniques for improving regression testing in continuous integration development environments." Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014.

Iakushkin, Oleg. "Intellectual scaling in a distributed cloud application architecture: A message classification algorithm." In " Stability and Control Processes" in Memory of VI Zubov (SCP), 2015 International Conference, pp. 634-637. IEEE, 2015.

Bogdanov A.V., Gankevich I.G., Gayduchok V.Yu., Yuzhanin N.V. Running applications on a hybrid cluster // Computer Research and Modeling. — 2014. Vol 7, No. 3. – P. 475-483

Iakushkin, Oleg, Yulia Shichkina, and Olga Sedova. "Petri Nets for Modelling of Message Passing Middleware in Cloud Computing Environments." In International Conference on Computational Science and Its Applications, pp. 390-402. Springer International Publishing, 2016.

Shichkina, Yulia, Alexander Degtyarev, Dmitry Gushchanskiy, and Oleg Iakushkin. "Application of Optimization of Parallel Algorithms to Queries in Relational Databases." In International Conference on Computational Science and Its Applications, pp. 366-378. Springer International Publishing, 2016.

Abrahamyan, Suren, Serob Balyan, Avetik Muradov, Vladimir Korkhov, Anna Moskvicheva, and Oleg Jakushkin. "Development of M-Health Software for People with Disabilities." In International Conference on Computational Science and Its Applications, pp. 468-479. Springer International Publishing, 2016.

Bogdanov, A., A. Degtyarev, V. Korkhov, V. Gaiduchok, and I. Gankevich. "Virtual supercomputer as basis of scientific computing." Horizons in Computer Science Research 11 (2015): 159-198.