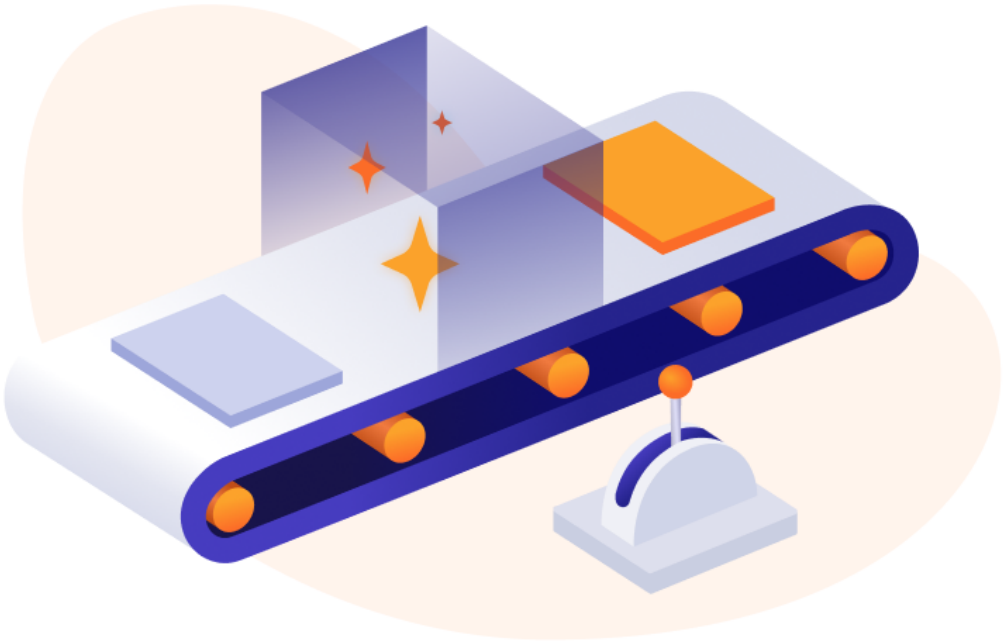
Continuous Integration in Gitlab CI/CD with Drupal 8

A tutorial showing you how to set up continuous integration in Gitlab CI/CD to automate your tests and builds.

ADCI Solutions Follow
Nov 6, 2018 · 5 min read



By ADCI Solutions

A release of new versions of programming software is fraught with considerable difficulties—there always may be unexpected errors in the process of deploying a web application to the environment which leads to a non-working application in production. To help minimize these risks, the continuous deployment methodology is used. The basis of continuous is the **deployment pipeline**, which allows us to automate testing and deployment of the application, thereby increasing the speed of the deployment and reducing the risks in a process of releasing a new version.

If you are using the Gitlab platform for your project, it already has a built-in CI/CD support. Together with free private repositories, a bug tracker, and other features, it makes the cloud-based service useful for both storing code and also for testing and deploying. You can also learn how to use the Gitlab pages service for your project documentation in one of our previous articles.

In this article, we will look at the configuration of the pipeline using Gitlab, configure necessary files, learn how to build application and run unit tests on a separate docker image, as well as automatically send the built application to a staging or production server. And all of this we can do just using a couple of lines in the configuration file! Sounds interesting? Let's start!

. . .

Configuring GitLab continuous delivery

First of all, you need 3 things to start using Gitlab CI:

1. The working instance of gitlab or gitlab.com.
2. The configuration file with the description of the pipeline `.gitlab-ci.yml` in the root directory of your project.
3. The configured gitlab task runner, which executes the commands described in the pipeline.

Gitlab Runner is the project that is used for execution of jobs described in the `.gitlab-ci.yml` file and also it communicates with Gitlab using its own API.

In addition to installing our own runner on a custom hosting, you can use ready shared runners hosted on Digital Ocean. To do this, on a Gitlab project page go to settings-CI/CD-runner settings-enable shared runners in the settings of your project. There you can also find the documentation on installing your own runner on a dedicated server.

After we've finished with the runner, let's take a closer look at the deployment process. The pipeline of Gitlab consists of several stages, by default it is build, test, and deploy, each of them consists of 1 or more CI **jobs**. Jobs in Gitlab are independent tasks that can be performed in parallel. After completing all the jobs at one stage, the runner starts to execute the next stage.

. . .

Build and test stage

Let's create the `.gitlab-ci.yml` file in the root of the project and specify our custom stages:

```
stages:
  - build
  - deploy
```

For the build stage, we create a build job that installs our Drupal 8 project vendor dependencies using Composer and launch our custom tests.

To do this, we need to specify a **Docker image for the Gitlab runner**, for the brevity we will use the tetraweb/php image which already contains the installed PHP and Node.js in this example.

```
build:
  stage: build
  image: tetraweb/php:7.1
```

We can also specify several jobs with different versions of PHP and test our application for different versions.

In the `before_script:` section which is performed before the main script execution, we install Composer and necessary packages.

```
 before_script:
   - apt-get update

   #install additional gd extension, that is required b
   - apt-get install libpng-dev -y
   - docker-php-ext-install gd

   #install composer
   - apt-get install zip unzip
   - php -r "copy('https://getcomposer.org/installer',
     php composer-setup.php
```

Now after we have installed Composer, in the script section we run the building of our project.

```
  script:
    - php composer.phar install

    #remove unnecessary files
```

After the successful completion of the job, we can attach resulting files to it, this is called an **artifact**.

For the artifact, we can specify its name, duration of storage and other settings, as well as a list of files to be added.

`{CI_COMMIT_SHA}` is a built-in placeholder, in the future, we will also be able to define our own constants.

```
  artifacts:
    name: "test_project_{CI_COMMIT_SHA}"
    expire_in: '1 week'
    paths:
```

In case we want to run the job only for commits from a specific branch, you can specify it in the section directive ‘only’.

```
  only:
    - master
```

In the end, our config file will look like this:

```
build:
  stage: build
  image: tetraweb/php:7.1

  before_script:
    - apt-get update

  #install additional gd extension, that is required b
  - apt-get install libpng-dev -y
  - docker-php-ext-install gd

  #install composer
  - apt-get install zip unzip
  - php -r "copy('https://getcomposer.org/installer',
  - php composer-setup.php
  - php -r "unlink('composer-setup.php');"

  #install composer package to run parallel tasks
  - composer -n global require -n "hirak/prestissimo"

  script:
    - php composer.phar install
```

If we want to run a unit test for the built application, let's add the following line to the end of the script directive:

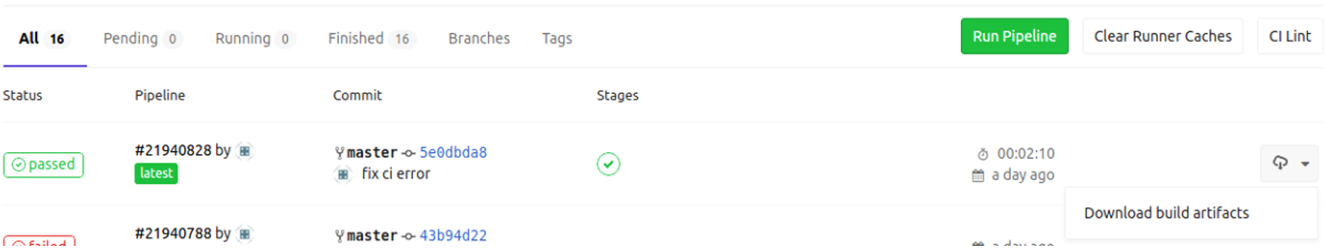
```
    # run phpunit tests
    - ../web/vendor/bin/phpunit -c core --group our_tes
```

`--testsuite = unit` flag indicates that we are running unit tests only, which do not require the installed Drupal.

So, after pushing the configuration file into the master branch of your repository, go to the CI/CD-pipelines at the project page and you will see that our task runner automatically began to perform the pipeline. If we do not want to run it automatically, we can add the following directive.

```
when: manual
```

If the job is finished without errors, we can download the resulting artifact.



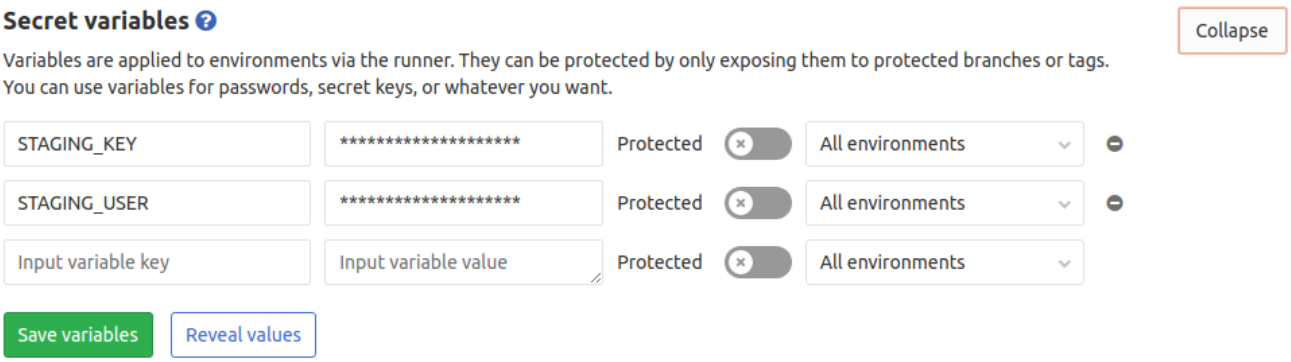
. . .

Deploy stage

Okay, we've installed dependencies for our application and have run out some of the unit tests, let's try to deploy it into the stage environment, for example, for the user acceptance testing.

We need to create a new `deploy_stage` job, which is related to the stage deploy, that we declared above.

First, we need to create a variable inside of Gitlab where we will store a private ssh key for access to our server. Go to the settings-CI/CD-secret variables and enter necessary values.



```
deploy_dev:
 stage: deploy
 environment:
   name: dev
 only:
   - master
image: tetraweb/php:7.1
before_script:
  #install ssh agent
  - 'which ssh-agent || ( apt-get update -y && apt-get
  - mkdir -p ~/.ssh
  - eval $(ssh-agent -s)
  - '[[ -f /.dockerenv ]] && echo -e "Host *\n\tStrict



 script:
   - ssh-add <(echo "$STAGING_KEY")
   - ssh -p22 server_user@server_host " <<EOF
   - cd /var/www/html/my-drupal-site/web/
   - drush sql:dump --result-file=../../backups/${CI_B
   - mkdir /var/www/html/_tmp
   - EOF
```

In this script, we've installed the ssh-agent on the Docker container and disabled `StrictHostKeyChecking` in the ssh configs in the `before_script` block. After that, we can log in to the stage environment server via the ssh.

There are several ways to copy project files to the server, in this case, we copy them to the tmp folder via SCP, then we replace the folders and move the config file `settings.env.php` to the working directory. We also created a backup of a database, ran import of the Drupal config files and database updates.

. . .

Afterword

In this article, we've learned how to use Gitlab CI/CD in our project and wrote the necessary config files. Of course, for the real project, you will want to add more stages and jobs, this is just a very basic example how to use this service. I hope you will start using a continuous delivery approach in your project and make your deploy process easier.

. . .

Originally posted at the ADCI Solutions website.

. . .

The author is Dmitry Romanovsky, Web Developer at ADCI Solutions

Dmitry is famous for finding beautiful and optimized solutions to solve a client's problem. As a hobby Dmitry studies machine learning and reads books. Also, in his spare time he watches TV series, listens to music, and plays video games sometimes.

. . .

Thanks for the claps!

Follow us on social networks: Twitter | Facebook | LinkedIn