Continuous Integration and Continuous Deployment

Continuous integration and **continuous deployment**, commonly known as **CI/CD**, is a computing and IT philosophy whereby code is continuously tested and deployed/released automatically with as little manual handling as possible. This requires extra tools, usually a specialized platform that handles testing newly committed code as well as other tools that can take that code and package it for release, or upload it to a new version of the platform of website. However, if you're using GitLab, you have all the tools you need to get up and running and practicing CI/CD in no time.

In this chapter, we'll cover the following topics:

- How to install GitLab Runner
- Setting up tests in our GitLab example project
- The `.gitlab-ci.yml` file
- Setting up a `.gitlab-ci.yml` file in our GitLab example project
- Breaking down advanced `.gitlab-ci.yml` files
- The CI/CD interface in GitLab

By the end of this chapter, you'll have a good idea of how to navigate GitLab's CI/CD features and be ready to start integrating it into your own project.

How to install GitLab Runner

GitLab's implementation of a continuous integration and continuous deployment platform works off the idea of **Runners**, which are programs that are installed on other servers and can connect to the GitLab instance to receive jobs to run. These jobs can involve building, testing, and deploying code. In this section, we'll explore how to install GitLab Runner on Ubuntu, CentOS, and manually via binaries.

Ubuntu

To easily install GitLab Runner on Ubuntu, make sure that you have a recent version installed, as GitLab only provides packages for the currently supported versions. Next, open a Terminal session on your Ubuntu box and run the following command:

```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash
cat > /etc/apt/preferences.d/pin-gitlab-runner.pref <<EOF
Explanation: Prefer GitLab provided packages over the Debian native ones
Package: gitlab-runner
Pin: origin packages.gitlab.com
Pin-Priority: 1001
EOF
sudo apt-get install gitlab-runner Copy
```

Let's break this down a bit.

The first command downloads and executes a script from GitLab that checks if your operating system is compatible, sets up your local package manager (`apt`) with the right URLs to get packages and lists from, and then updates those lists. This is a security anti-pattern, though, as the user isn't sure what they're executing and it could contain malicious code. I'd recommend downloading the script to your machine and then executing it yourself after having a read of the code so that you know what it will do.

The next section pins the package to the GitLab origin. This is necessary because Ubuntu introduced a GitLab Runner package as well. However, it can be a lot more out of date than the GitLab version and so it's recommended to pin it to the GitLab download origin.

Lastly, we install the package so that it's ready to go.

Now, we need to register the Runner. To do that, go to the *How to register Runners* section in this chapter.

CentOS

To easily install GitLab Runner on Ubuntu, make sure that you have a recent version installed, as GitLab only provides packages for the currently supported versions. Next, open a terminal session on your Ubuntu box and run the following command:

```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh | sudo bash

sudo yum install gitlab-runner Copy
```

Let's break this down a bit.

The first command downloads and executes a script from GitLab that checks if your operating system is compatible, sets up your local package manager (`yum`) with the right URLs to get packages and lists from, and then updates those lists. This is a security anti-pattern, though, as the user isn't sure what they're executing and it could contain malicious code. I'd recommend downloading the script to your machine and then executing it yourself after having a read of the code so that you know what it will do.

Lastly, we install the package so that it's ready to go.

Now, we need to register the Runner. To do that, go to the *How to register Runners* section in this chapter.

Binaries

If you don't want to use the DEB or RPM packages to install GitLab Runner, or are using an operating system without those package managers available, you can still install GitLab Runner manually using the binary downloads.

Run the following commands to download the Runner, give it permissions to execute, create a user for it, and then install and run it:

```
sudo wget -O /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64

sudo chmod +x /usr/local/bin/gitlab-runner

sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell
```

```
/bin/bash
sudo gitlab-runner install --user=gitlab-runner --working-
directory=/home/gitlab-runner
sudo gitlab-runner start Copy
```

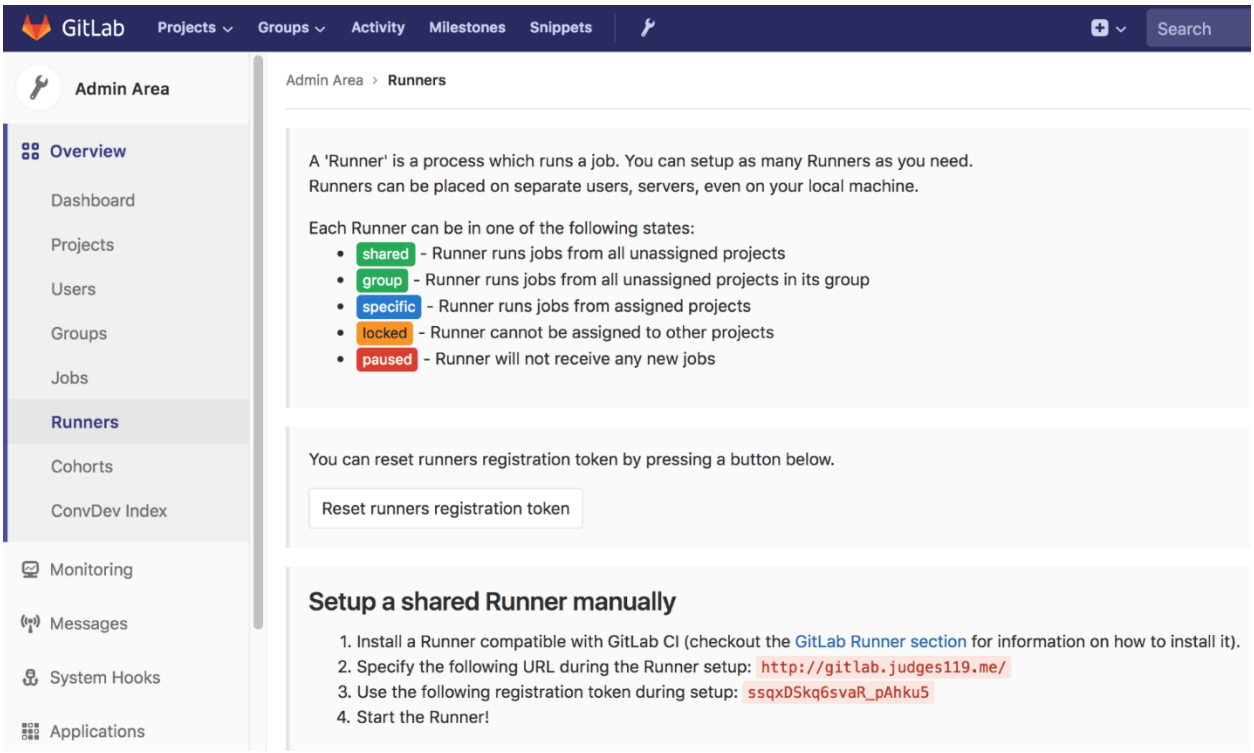
If you're using an ARM platform such as a Raspberry Pi or other lightweight device, you can change the `amd64` to `arm` in the download link.

How to register Runners

Once you have your Runner client installed, you can go through the process of registering it. There are two ways of doing this: in the administrator panel (only in self-hosted GitLab) or in project settings. The former creates a global shared Runner that can be assigned to multiple different projects or even made available to any project on the instance that needs a Runner. The latter method creates Runners that are tied specifically to a particular project/repository. There are also group runners, which can be assigned to a group of projects, but they are beyond the scope of this book.

Admin panel

To do this, you first need to log in to your GitLab instance as an account with administrator permissions. Once you're logged in, you need to click the spanner/wrench icon up the top at the end of the left-hand side menu. From here, go to Overview | Runners to be greeted by a scene like the following:



Finally, copy the value listed as the registration token, and move on to the next step.

Project settings

From your project home page, go to Settings | CI/CD. On this page, expand the section titled Runners. After the explanatory text, you'll be faced with an information box that looks like this:

Specific Runners

Setup a specific Runner automatically

You can easily install a Runner on a Kubernetes cluster.
Learn more about Kubernetes

1. Click the button below to begin the install process by navigating to the Kubernetes page
2. Select an existing Kubernetes cluster or create a new one
3. From the Kubernetes cluster details view, install Runner from the applications list

Install Runner on Kubernetes

Setup a specific Runner manually

1. Install a Runner compatible with GitLab CI (checkout the GitLab Runner section for information on how to install it).
2. Specify the following URL during the Runner setup: `http://gitlab.judges119.me/`
3. Use the following registration token during setup: `Krr66ucS7zLtiDKz_uAz`
4. Start the Runner!

You'll need to copy the value for the registration token and then move on to the next step.

Registering the CLI

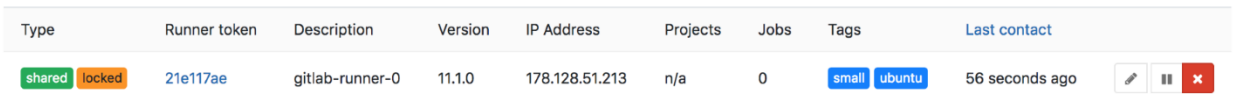
Jump into a terminal session on your GitLab Runner machine.

Run the following command:

```
sudo gitlab-runner register Copy
```

This should lead you to an interactive prompt. First, you'll need to input the URL of your GitLab instance (make sure that it's reachable from the Runner). Next up, it will ask for the shared token that you copied earlier. Then, enter a name for the Runner and lastly a series of comma-separated tags. You can use these tags to describe or group the Runner, maybe by OS, capability, pre-installed frameworks, and so on. Lastly, you have to pick the GitLab Runner executor; this is how GitLab Runner will execute any tests. In most circumstances, you should select `docker`, but there are some occasions where you might want to select other options, such as `kubernetes` or `shell`. The latter is really helpful for building OS-specific projects, such as iOS apps that can't be easily built in a Docker container. If you do choose `docker` as your executor, you'll also be asked to provide a default Docker image for projects that do not specify one in their `.gitlab-ci.yml` file. You'll also need to make sure that Docker is installed on the Runner; it is not bundled with the Runner package.

Now, your runner should be ready to go. You can verify this in GitLab in the Administration | Overview | Runners or project home page by going to | Settings | CI/CD | Runners. For example, here's what it should look like if you set up a global runner in the administration panel:



Adding to our example project

To get started with testing, let's make an issue on our GitLab project and then create a branch, starting with the issue number.

In our example project, we'll need to update our `composer.json` file to look like the following:

```
{
    "name": "judges119/monorot13",
    "description": "A MonoLog formatter that puts logs through the ROT13
transformation before logging them.",
    "type": "library",
    "license": "BSD",
    "authors": [
        {
            "name": "Adam O'Grady",
            "email": "adam.ogrady@gmail.com"
        }
    ],
    "require": {
        "monolog/monolog": "^1.23"
    },
    "require-dev": {
        "phpunit/phpunit": "~4.5"
    },
    "autoload": {
        "psr-4": {
            "Judges119\\Monolog\\": "src"
        }
```

```
    }
} Copy
```

After you've done that, run the following command:

```
composer install Copy
```

This will install the PHPUnit testing framework and set up the PSR-4 autoloading for our project. After you've run this, let's create a folder called `tests` and in that, a file called `ROT13FormatterTest.php`. In that file, put the following code:

```
<?php
require_once 'vendor/autoload.php';

use PHPUnit\Framework\TestCase;
use Judges119\Monolog\Formatter\ROT13Formatter;

final class ROT13FormatterTest extends TestCase
{

    public function testFormatReturnsString()
    {
        $formatter = new ROT13Formatter();
        $this->assertInternalType(
            'string',
            $formatter->format(["message" => "Changed to ROT13"])
        );
    }

    public function testROT13IsAccurate()
    {
        $formatter = new ROT13Formatter();
```

```
        $record = ['message' => 'ABC'];
        $transformed = $formatter->format($record);
        $this->assertEquals(
            $transformed,
            'NOP'
        );
    }

    public function testROT13TextCanBeReversed()
    {
        $formatter = new ROT13Formatter();
        $record = ['message' => 'Changed to ROT13'];
        $transformed = $formatter->format($record);
        $transformed2 = $formatter->format(['message' => $transformed]);
        $this->assertEquals(
            $transformed2,
            $record['message']
        );
    }

    public function testFormatBatchReturnsArrayOfStrings()
    {
        $formatter = new ROT13Formatter();
        $records = [
            ['message' => 'Changed to ROT13'],
            ['message' => 'From ancient Rome'],
            ['message' => 'Modified Caesar cipher']
        ];
```

```
        $transformed = $formatter->formatBatch($records);
        $this->assertInternalType(
            'array',
            $transformed
        );
        $this->assertInternalType(
            'string',
            $transformed[0]
        );
    }
} Copy
```

If you want to run the tests manually, you can change into your project directory in a terminal session and run the following command:

```
vendor/bin/phpunit tests/ROT13FormatterTest Copy
```

You should see our tests pass, four for four. If your tests don't, you should examine the output of the logs to help you find and fix any errors. I'd recommend adding and committing the work you've done so far before we move on to learning about the `.gitlab-ci.yml` file.

Breaking down .gitlab-ci.yml

Continuous integration and continuous deployment in GitLab is described and defined by a project's `.gitlab-ci.yml` file. The file format is **YAML** (an acronym of **yet Another markup language**), which is a human-readable text file that's used for storing data, and can be converted into digital representations by a computer. This document is stored in the root directory of your repository and outlines all of the stages and work required for the CI/CD to run.

A basic `.gitlab-ci.yml` file might look something like this:

```
before_script:
  - apt-get update -qq && DEBIAN_FRONTEND=noninteractive apt-get install -y
-qq ca-certificates git php php-xml
  - php -r "copy('https://getcomposer.org/installer', 'composer-
setup.php');"
  - php composer-setup.php
  - php composer.phar install

phpunit:
  script:
    - vendor/bin/phpunit tests/ROT13FormatterTest Copy
```

Because of the plain text format of YAML, along with the descriptive names, you can read the preceding code and have a rough idea of what happens. We can guess that `before_script` runs first, which runs a number of shell commands, followed by something called `phpunit`, which runs a script as well. However, we're still a bit in the dark about it all, so let's explore GitLab CI jobs, as well as some of the keywords.

Jobs

A *job* in `.gitlab-ci.yml` is a task or set of tasks that must be completed by a Runner. They are top-level entities within the `.gitlab-ci.yml` file, which are given an arbitrary name that's decided on by the author. It's a good practice to make the name descriptive to help summarize the purpose of jobs and make them easy to understand by others who might need to review or work with your CI/CD pipeline.

One thing that's important to note is that jobs are run independently of each other within the Runner, and thus different jobs can be picked up by different Runners. Jobs in the same stage (discussed as follows) can also be run in parallel on other Runners, allowing large pipelines to be broken down and completed faster than having to run everything in a sequential manner.

Under the job key/name, there are a number of different parameters/keywords that can be used to help define the job.

The script parameter

The most common parameter to a job, `script` is used to define any commands that should be run in the job. In our preceding example, we have the following:

```
phpunit:
  script:
    - vendor/bin/phpunit tests/ROT13FormatterTest Copy
```

In the `phpunit` job, we have the `scripts` parameter, which tells GitLab to execute `vendor/bin/phpunit tests/ROT13FormatterTest`. It's worth noting that commands with special characters (such as a colon, brackets, ampersand, greater/less than, and so on) could be interpreted as part of the YAML, so it's prudent to wrap any such commands in single or double quotes to ensure that they are run as a command.

The before_script and after_script parameters

These are scripts that you choose to be run before the job is executed or after the job is executed. These can also be defined at the top level of the YAML file (where jobs are defined) and they'll apply to all jobs in the `.gitlab-ci.yml` file. `before_scripts` is executed before all jobs, including deploy tasks, but is run after artefacts have been restored (more on that shortly). `after_script` is run at the end of the job, even after failed jobs. This can be handy for cleaning up after a job has run.

The stage parameter

Stages are defined at the top level of the YAML file and are used to define separate blocks of jobs, which can be executed in parallel. They also define the order in which stages are run. In each job, the `stage` parameter can be used to define which build stage

a job is in, thus grouping together similar jobs and allowing for jobs to depend on other jobs having finished. The following is an example of defining and using stages:

```
stages:
  - build
  - test
  - deploy

job 1:
  stage: build
  script: npm run build dependencies

job 2:
  stage: build
  script: npm run build artifacts

job 3:
  stage: test
  script: npm run test

job 4:
  stage: deploy
  script: npm run deploy
```
Copy

In this case, we have defined three stages: `build`, `test`, and `deploy`. All of the jobs attached to the `build` stage can be run in parallel, and if they all succeed, the jobs with the `test` stage that have been identified will be run. Assuming they pass, the `deploy` jobs will then be run. If any job fails at an earlier stage, no further stages will be executed and the pipeline will be marked as failed.

The image parameter

The `image` parameter is used to specify which Docker image should be used when running a job. This can be specified as a parameter within a job, or at the top level to indicate a Docker image to be used by all jobs. Note that `image` parameters specified within each job will override the global `image` definition.

The services parameter

This parameter is used to specify extra Docker containers that can be connected to the test image to provision services. This is very handy for setting up databases to connect to, rather than needing to install your database engine on each build run. You can also specify this parameter in the top level (but it will be overriden by local definitions).

The only and except parameters

The `only` and `except` job parameters are used to limit when a specific job is run. `only` is used to limit a job being executed to a specified branch or tag names, while `except` is the opposite: the job will always be run unless it's on a specified branch or tag. The value for this parameter can be defined with a regular expression or a special keyword such as *branches* to refer to all branches, *tags* to refer to all tags, and so on. For more reserved keywords for `only` and `except`, please refer to the GitLab CI documentation online.

Please note that `only` and `except` don't have to be mutually exclusive either; you can use a combination of them to have more fine-grained control over when a job will be executed. An example of this is as follows:

```
job:
  only:
    - /^iss-.*$/
  except:
    - tags Copy
```

In this case, the job will only be run on references where the label starts with `iss-`, and will not run on any tags that have been pushed to the repository.

There are more complex ways to use `only` and `except`, but they are beyond the scope of this quick start guide. More information can be found in the GitLab CI documentation (https://docs.gitlab.com/ee/ci/).

The tags parameter

You can specify a tag or tags in this parameter, which will limit this job to only being executed on Runners that also have the same tag. Please note that the tags are additive, so if you specify multiple tags, the job will only be executed on a Runner that has all of those tags present:

```
job:
  tags:
    - php
    - postgres
```
Copy

The preceding job will only be executed on Runners that have both the `php` and the `postgres` tag.

The allow_failure parameter

This parameter requires a Boolean response, either true or false. When false, the pipeline will execute as normal and any failures in that job will halt the rest of the pipeline. However, when set to true, that particular job can fail without stopping later tasks. The pipeline will still show green, but will have a *CI build passed with warnings* message displayed to alert users to the failed stage.

The when parameter

The `when` parameter can be one four values, and controls under which conditions a job is run. The possible values are as follows:

- `on_success`: The default behavior; a job will only be executed if the preceeding jobs/stages have passed

- `on_failure`: The job will only run if at least one job earlier in the pipeline has failed
- `always`: The task will always be executed, on both success and failures
- `manual`: A task that requires manual intervention to be started, such as from the GitLab web UI

One example use case of `when` is as follows:

```
stages:
  - build
  - cleanup_build

build_job:
  stage: build
  script:
    - webpack

cleanup_build_job:
  stage: cleanup_build
  script:
    - rm /dist/*
  when: on_failure
```
Copy

In the preceding case, the `build` job is always executed and attempts to use webpack to build our JavaScript assets, but if it fails, the `cleanup_build_job` task will be run, deleting any files that were created by the former task.

The environment parameter

The environment parameter is used to specify a particular environment to which a job will be deployed. Environments are discussed in the *Environments* section later in this chapter. The environment key can contain multiple other keys, most commonly `name` and `url`. The `name` sub-key defines a name for an environment to which the code will be deployed. You can track this using the GitLab web user interface by

going to Project | Operations | Environments. While you can create new environments in the web user interface, it's recommended that you define them first in the `.gitlab-ci.yml` file, which will automatically create them in the web UI on its first run.
The `url` parameter will be exposed in multiple parts of the GitLab web user interface as links and buttons that can be used to access the environment.

The cache parameter

To speed up build processes, you can cache certain files and directories between jobs and between different pipeline executions using the `cache` keyword. This is a more advanced strategy, but is really useful to reduce turnaround time in your CI/CD pipelines if you want builds to be tested, packaged, or deployed rapidly. The `cache` parameter can take a few different parameters itself to help define the caching rules. Please note that caching can be defined locally (per-job) or globally, and that local caches will override any global declarations.

To manually clear caches, you can open the GitLab web user interface for your project and navigate to CI/CD | Pipelines using the menus on the left. From here, click the Clear Runner Caches button.

cache – paths

This is an array of paths to files and/or directories that should be cached. You can also use the asterisk wildcard character (*****). This is demonstrated in the following code snippet:

```
build:
  cache:
  paths:
    - binaries/*.apk
    - .config Copy
```

In the preceding example, we cache the `.config` file as well as any `.apk` files in the `binaries` directory.

cache – key

This takes a string that can be used to create separate caches for different jobs or branches, like so:

```
test:
  cache:
    key: "$CI_COMMIT_REF_SLUG"
    paths:
      - binaries/ Copy
```

In this example, we use an inbuilt default variable provided by GitLab CI – `CI_COMMIT_REF_SLUG` – that is equal to the branch/tag name of the commit. In this case, GitLab CI will maintain a separate cache for each branch. Next up, let's look at jobs with different paths cached:

```
stages:
  - build
  - test

build_job:
  stage: build
  script: npm run build
  cache:
    key: build-key
    paths:
      - public/

test_job:
  stage: test
  script: npm run test
```

```
  cache:
    key: test-key
    paths:
     - vendor/ Copy
```

Without a `key` value, this would mean that the second job to run (`test_job`) would reuse the cache from `build_job` and at the end of its run would cache the `vendor/` directory. This cache will overwrite the existing cache from `build_job`, which means that next time the pipeline runs, the cache will only contain the `vendor/` directory (as that was the only thing defined to be cached by the last running job) and the cache will be useless to `build_job`. To prevent this happening, we use a separate cache key, which means that each time the pipeline is run, `build_job` will grab the cache stored under the `build-key` value and `test_job` will use the cache stored under `test-key`.

cache – untracked

This is a simple helper option that you can set to `true` in order to cache any files that are untracked by git at the end of a job run. This is handy for caching the output left over from package installations or program compilation.

cache – policy

Under default circumstances, the cache is downloaded before every job and then re-uploaded at the end of the job. This is equivalent to setting the following:

```
job:
  cache:
    policy: push-pull Copy
```

You may have a pipeline with jobs that would use a cache but not alter the contents of the cache. A good example is having one job that downloads all of the required packages and builds assets, and then subsequent jobs that simply run tests. In these cases, you can set `policy: pull`, which will download the cache at the start of the job but skip the uploading phase to save time. Conversely, if you have a job that always

creates the contents of a cache, you can set `policy: push`, which will skip the download phase but always run the upload step at the end of the job.

The artifacts parameter

The `artifacts` parameter is used to define a list of files and directories that should be attached to a job after success. They are packaged and sent to the GitLab instance (either your self-hosted one or GitLab.com) and become available for download through the web user interface.

artifacts – paths

Much like with caching, you specify an array of paths, which can include wildcards like so:

```
package:
  artifacts:
    paths:
      - public/
      - tests/*.html
```
Copy

In this case, everything in the `public/` directory as well as any HTML files in the `tests/` directory will be zipped, uploaded, and made available for download in the web user interface.

artifacts – name

By default, any uploaded files will be stored in a `.zip` archive titled `artifacts.zip`. You can use the `name` parameter to change the default filename (it will still end with `.zip`), as demonstrated in the following code:

```
package:
  artifacts:
    name: cool_project_name
```
Copy

In the preceding example, every time this job is completed, a zipped file called `cool_project_name.zip` will be uploaded to GitLab and made available for download in the web UI. However, this is not the limit of the naming parameter. You can use built-in GitLab variables such as `CI_JOB_NAME` (the name of the currently running GitLab CI job) or `CI_COMMIT_REF_NAME` (the name of the branch/tag) to create dynamically named artifacts like in this example:

```
package:
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
```
Copy

The preceding example will label every artifact with the current job name (in this case, `package`) and the current git branch name, separated by a hyphen and followed by `.zip`.

artifacts – untracked

Much like the `untracked` key in the `cache` section, `artifacts:untracked` is a boolean variable that can be set to `true` in order to collect all untracked files and upload them as artifacts.

artifacts – when

The `when` key can have one of three different values and controls under what circumstances artifacts are uploaded:

- `on_success`: Only upload artifacts when the job is successful (default).
- `on_failure`: Only upload artifacts when the job fails. This is useful if you want to examine failed builds.
- `always`: Upload artifacts, regardless of whether the job succeeds or not.

artifacts – expire_in

The `expire_in` sub-parameter is provided with a string defining how long artifacts should be kept for in the GitLab instance. If this is not specified, it defaults to the instance setting, which is 30 days by default for self-hosted GitLab or forever for GitLab.com projects. Some examples of strings that you can provide include the following:

- 42 weeks
- 42 days
- 42 mins 42 secs
- 2 mos 1 day
- 42 years 6 mos 6 hrs
- 1 week and 2 days

Keep in mind that the job that deletes old artifacts is run on an hourly basis, so units of time less than one hour might not be effective. There is also a Keep button in the GitLab web user interface that can be used to keep an artifact forever and override the specified time value.

The variables parameter

You can store variables per job or globally using the `variables` keyword. The value for `variables` should be an array of key/value pairs that can be represented by strings or integers (for both key and value), although typically the key will be an all-capitalized string for ease of recognition. Any variables used should be of a non-sensitive nature; they are not considered an appropriate method for storing secrets. Locally defined variables will override their globally defined counterparts, but to have a job run with no access to globally defined variables, you should redeclare it with an empty array like so:

```
cool_job_name:
  variables: {}
```

Copy

Other parameters

So far, we've covered the majority of the parameters that are available to jobs, including a bunch of ones that can be used at the top level to apply to all jobs. It's worth noting that there are other ones such as dependencies, coverage, and so on that are beyond the scope of this book, but can be found in the GitLab CI documentation online. Let's move on to creating a `.gitlab-ci.yml` file for our project so that we can start testing our commits.

Adding .gitlab-ci.yml to our example project

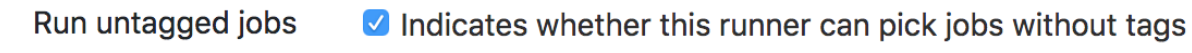
We've added tests to our `ROT13Formatter` project, but now we need to get those tests to be automatically executed in GitLab (either GitLab.com or our own hosted instance). To do this, let's create a file called `.gitlab-ci.yml` in our project and add the following to it:

```
before_script:
  - apt-get update -qq && DEBIAN_FRONTEND=noninteractive apt-get install -y
-qq ca-certificates git php php-xml
  - php -r "copy('https://getcomposer.org/installer', 'composer-
setup.php');"
  - php composer-setup.php
  - php composer.phar install

phpunit:
  script:
    - vendor/bin/phpunit tests/ROT13FormatterTest Copy
```

This exact file was discussed in the previous section, so we know that it simply executes a series of commands to configure the Runner and then runs one task that executes our tests. Now save, commit, and push this file to GitLab, and then we'll almost be ready to have our pipeline running. If you've set up a shared Runner in GitLab (as discussed earlier in this chapter), you'll need to go to Admin Area | Overview | Runners. From here, edit your Runner and tick Run untagged jobs;

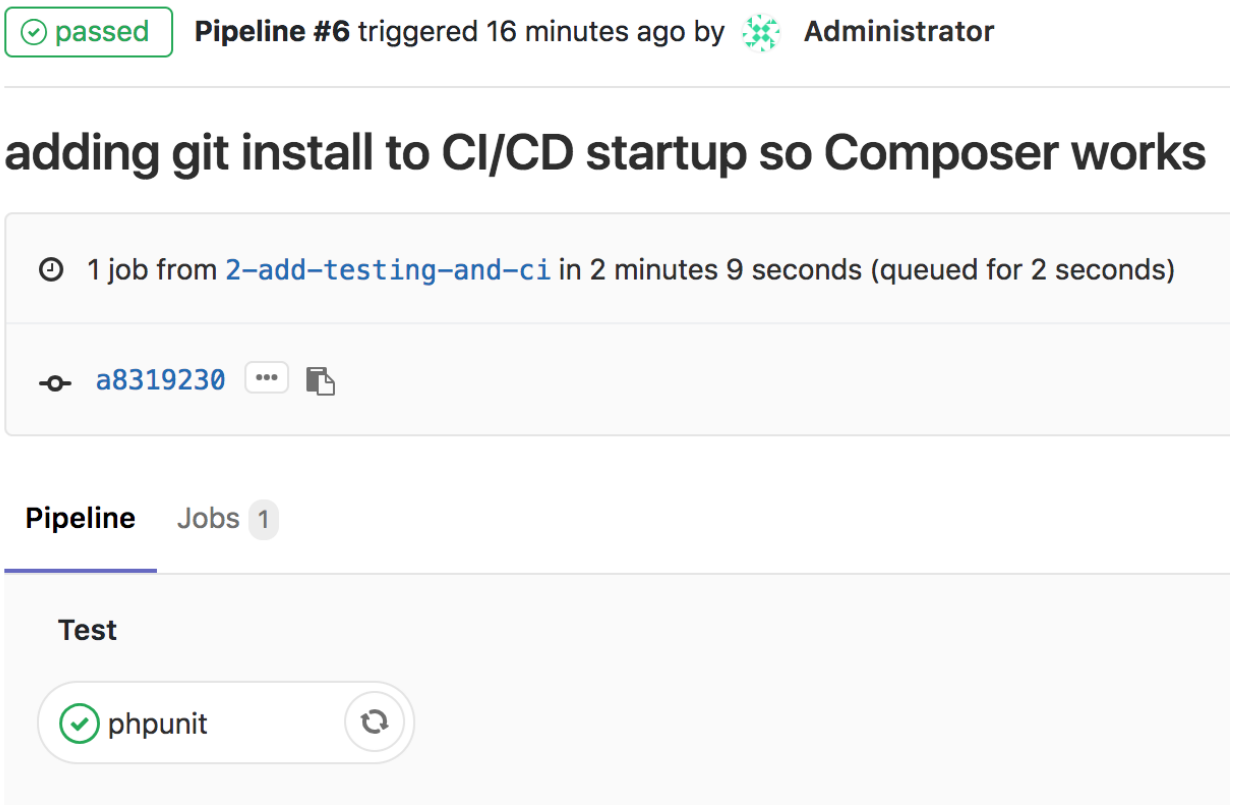
this will allow your Runner to pick up jobs that don't have tags (which our new `.gitlab-ci.yml` file lacks):



Now, when you go to Project | CI/CD | Pipelines, you should see your new pipeline there, ready to go:

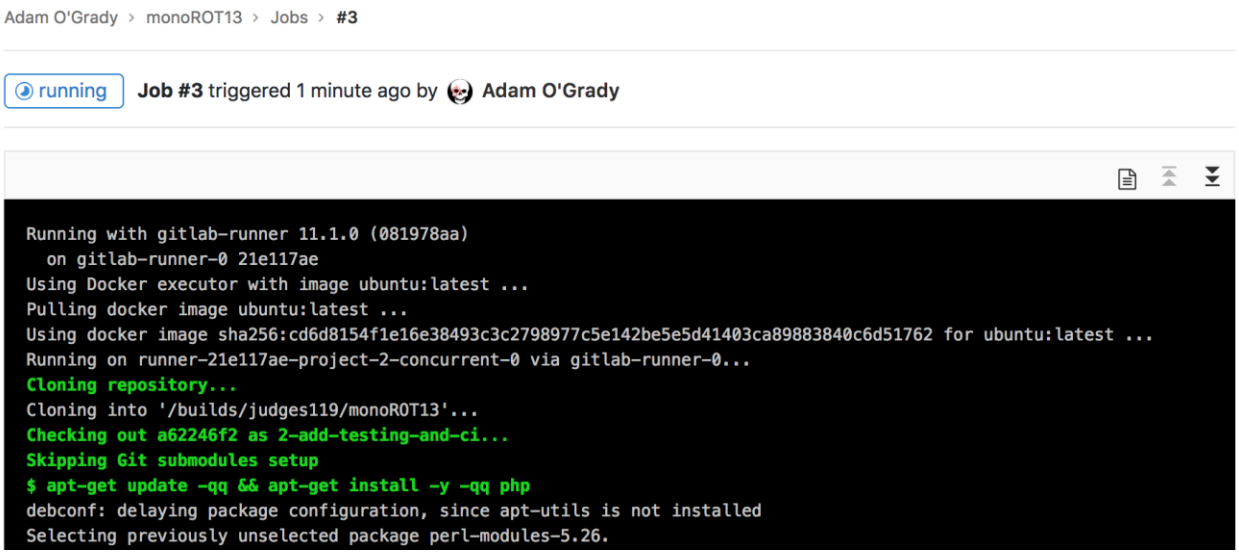
The status may also be running if the pipeline is still processing, pending if you have no free Runners, or failed if the pipeline failed somewhere along the pipeline.

You can click the status (passed, in the preceding screenshot) to receive a breakdown of the pipeline. Yours should look similar to this:



The commit name and branch might be different, but the main area under the Pipeline heading should look the same. It's pretty bare, but that's only because we have a single stage in our CI/CD pipeline, and within that stage we have a single task, `phpunit`. In more complex pipelines, you'll have multiple different stages shown, and each stage might have many different jobs within it.

You can click the Jobs heading to view the status of the individual jobs within the pipeline. This view will be pretty bare because we have such a simple pipeline, but you should still see the status, job ID, and name. By clicking on the status, you'll be taken to that individual job, which will show the current progress of it. The following is an example from an in-progress job execution:



As you can see, we have the job status and some information, followed by a large block of text indicating the output from any scripts being executed in the course of the job.

Deconstructing an advanced .gitlab-ci.yml file

Now that we've configured GitLab to run continuous integration and continuous deployment with our example project, it's worth exploring a more advanced `.gitlab-ci.yml` file and breaking it down to make sure that we understand what all of the components do. To this end, here's an example file that uses more jobs and more advanced parameters:

```
image: bitnami/laravel:latest

services:
  - postgres:9.6

variables:
  POSTGRES_DATABASE: postgres
  POSTGRES_PASSWORD: password
  DB_HOST: postgres
  DB_USERNAME: root

stages:
  - test
  - package
  - deploy

php_unit_test:
  stage: test
  script:
    - cp .env.example .env
    - composer install
    - php artisan key:generate
    - php artisan migrate
    - vendor/bin/phpunit
  cache:
    key: composer
    paths:
      - vendor/

js_unit_test:
```

```
  stage: test
  script:
    - npm install
    - npm run test

package_upload:
  stage: package
  script:
    - composer install
    - npm install
    - webpack
  cache:
    key: composer
    paths:
      -vendor/
    policy: pull
  artifacts:
    paths:
      - public/
    expire_in: 1 week
    only:
      - tags

deploy_production:
  stage: deploy
  script:
    - composer install
    - npm install
```

```
    - webpack
    - .composer/vendor/bin/envoy run deploy
  environment:
    name: production
    url: http://192.168.0.1
  when: manual
  only:
    - master
```
Copy

Let's break some parts of this down:

1. The header of the `.gitlab-ci.yml` file defines an image to use from the Docker Registry (`image: bitnami/laravel:latest`), as well as defining a `service` (in this case, the `postgres:9.6` image, also from the Docker registry). This service container will be attached directly to our Laravel image to provide a database without needing to set up a database as part of a `before_script` each time a job is run:

```
    image: bitnami/laravel:latest

    services:
      - postgres:9.6
```
Copy

2. The variables that are declared all relate to our database and will be used in the jobs so that we can connect to our database service image. Of particular note is the `DB_HOST` parameter, which simply specifies `postgres` as the hostname. This is because with GitLab CI, any services that are connected are referred to by their image name, in this case, `postgres`:

```
    variables:
      POSTGRES_DATABASE: postgres
      POSTGRES_PASSWORD: password
      DB_HOST: postgres
      DB_USERNAME: root
```
Copy

3. Next, we define three stages that we want: `test`, `package`, and `deploy`. These stages will be run sequentially (and any errors in an earlier stage will cause the whole pipeline to be canceled):

```
    stages:
      - test
      - package
      - deploy
```
Copy

4. During the `test` stage, we'll have two separate jobs that can be run simultaneously (provided there are enough available Runners). `php_unit_test` and `js_unit_test` don't depend on one another, so they can safely be run together without causing problems. Both jobs are quite similar in that they have mostly the same parameters, although their scripts each perform very different functions. The big difference between the two is that `php_unit_test` also contains a `cache` section, which means that after the job is finished, the entire `vendor/` directory will be zipped up and stored under the `composer` key for reuse in future jobs and pipelines. This helps reduce the time taken to build and execute subsequent runs of this job and any other jobs that use the same cache. Once both jobs are finished (and both are successful), the next stage can begin:

```
    php_unit_test:
      stage: test
      script:
        - cp .env.example .env
        - composer install
        - php artisan key:generate
        - php artisan migrate
        - vendor/bin/phpunit
      cache:
        key: composer
        paths:
          - vendor/
```

```
    js_unit_test:
      stage: test
      script:
        - npm install
        - npm run test
```
Copy

5. The `package_upload` job is the only component of the `package` stage, and consists of a script that installs all of the required dependencies and compiles all the front-nd elements. To speed up this process, we use the `cache` key, which should download the cache used in the earlier `php_unit_test` if it's available. For this example, we've also set the `policy` to `pull` to demonstrate it. This means that while it will download any available cache stored under the `composer` key, it will not re-upload any files once the job has finished. This task will also only operate on tagged commits, which is handy if you're doing versioned releases of software. Once all of the scripts have been executed, all of the files in the `public/` directory will be packaged and uploaded as artifacts and stored for up to a week:

```
    package_upload:
      stage: package
      script:
        - composer install
        - npm install
        - webpack
      cache:
        key: composer
        paths:
          -vendor/
        policy: pull
      artifacts:
        paths:
          - public/
```

```
          expire_in: 1 week
          only:
            - tags Copy
```

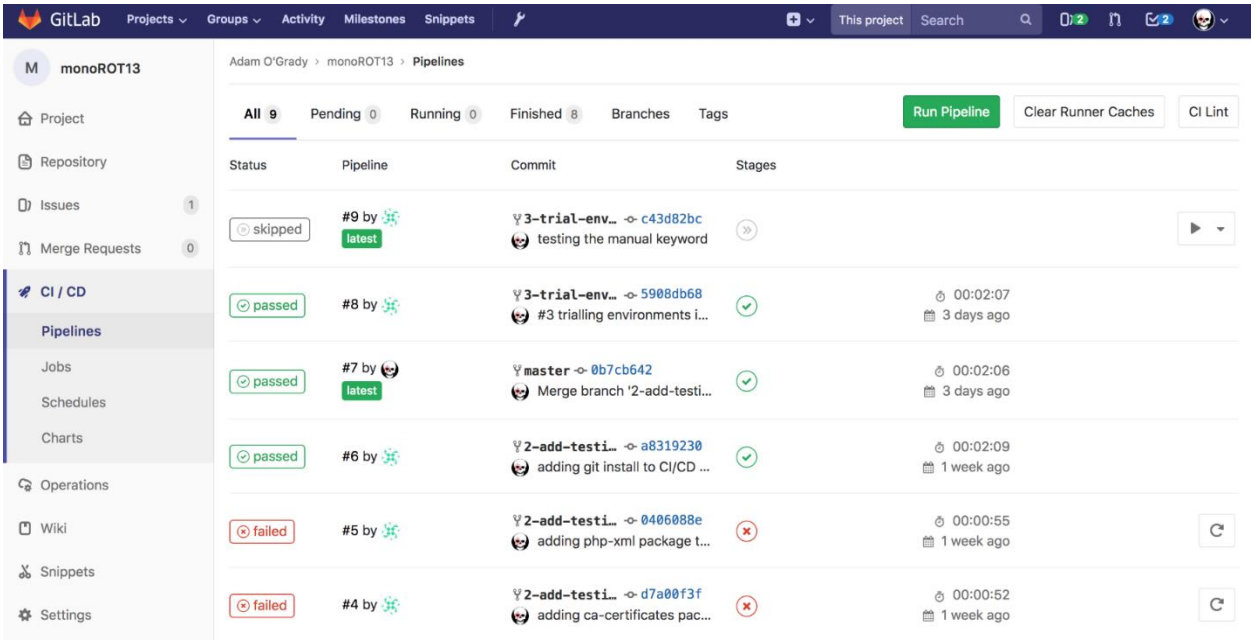
6. The last stage to run is `deploy_production`, which will only be available if the preceding stages have completed successfully. Note that if a job (such as `package_upload`) doesn't run because it isn't a tagged release, the stage can still be considered to have run successfully. The `deploy_production` task will run a script that installs all of the dependencies and prepare any necessary files and then deploy itself (the last command in the `script` array). Given that we are provided with a name and a URL under the `environment` parameter, you can find the status of the deployment in the GitLab web user interface for the project under Operations | Environments, and then under the name provided (`production`). There will also be links to the deployed application in the project user interface so that you can inspect it once it's live. Please note that this job will only run when code is pushed to the master branch:

```
    deploy_production:
      stage: deploy
      script:
        - composer install
        - npm install
        - webpack
        - .composer/vendor/bin/envoy run deploy
      environment:
        name: production
        url: http://192.168.0.1
      when: manual
      only:
        - master Copy
```

GitLab CI/CD web UI

While we've explored how to get GitLab CI up and running by creating a `.gitlab-ci.yml` file and looked at many of the configuration options available, we've only taken a brief look at the GitLab web user interface for continuous integration and continuous deployment. Let's break down the GitLab web user interface and look at how we can manage and inspect pipelines, jobs, charts, environments, and other common tasks.

The pipelines screen can be accessed under your project by going to CI/CD and then Pipelines through the left-hand side menu. You should then be faced with a screen that looks something like the following:



Here, you can view a list of the pipelines that have run or are currently running, as well as their status (failed, passed, skipped, and so on), who organized them, what branches and commits they are for (including the commit message), the success/failure of each stage (represented by the tick boxes), and the runtime and when it completed. You'll notice across the top left that there are also some buttons that provide extra functionality. You can click Run Pipeline to organize a manual, immediate run of the pipeline against a preferred branch. There's also the aforementioned Clear Runner Caches button, which will immediately clear the cache on all connected Runners. The last button, CI Lint, takes you to a small web application that can be used to lint a `.gitlab-ci.yml` file to ensure that the syntax is correct.

The output of a linted `.gitlab-ci.yml` file looks like this:

Check your .gitlab-ci.yml

Content of .gitlab-ci.yml

```yaml
image: bitnami/laravel:latest

services:
  - postgres:9.6

variables:
  POSTGRES_DATABASE: postgres
  POSTGRES_PASSWORD: password
  DB_HOST: postgres
  DB_USERNAME: root

stages:
  - test
  - package
  - deploy

php_unit_test:
  stage: test
  script:
    - cp .env.example .env
    - composer install
    - php artisan key:generate
    - php artisan migrate
    - vendor/bin/phpunit
  cache:
```

Validate

Clear

Status: syntax is correct

If the syntax passes its validation, you'll be shown the preceding message, as well as a table (visible in the following screenshot) that breaks down the keys and values provided in the script. You can use these to sanity check your work:

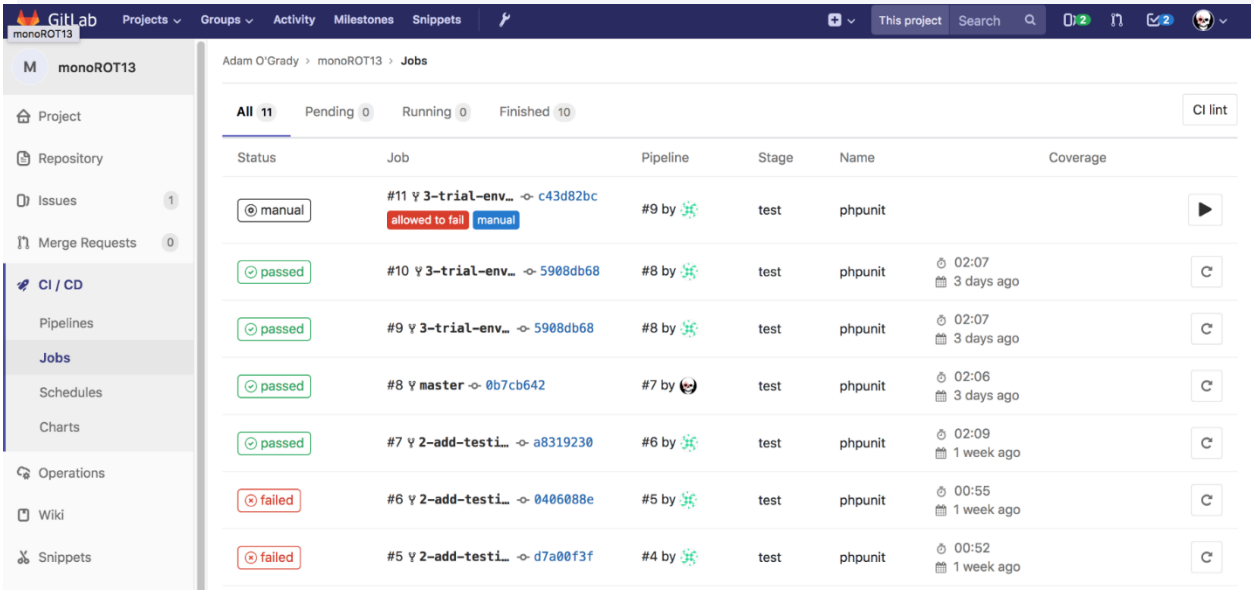
Parameter	Value

Test Job - php_unit_test	`cp .env.example .env`
	`composer install`
	`php artisan key:generate`
	`php artisan migrate`
	`vendor/bin/phpunit`
	
	Tag list:
	Only policy:
	Except policy:
	Environment:
	When: on_success

Test Job - js_unit_test	`npm install`
	`npm run test`
	
	Tag list:
	Only policy:
	Except policy:
	Environment:
	When: on_success

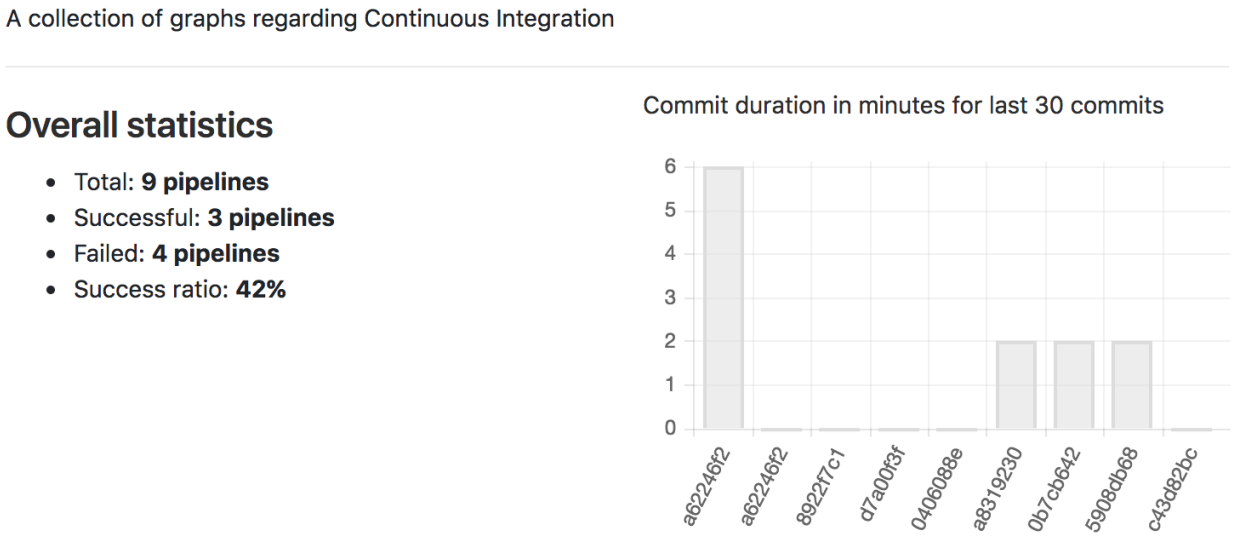
From the left menu under CI/CD, you can click Jobs to get a listing of all of the running, failed, and succeeded jobs. This view (shown in the following screenshot) provides information about each job, including the following:

- Status
- Branch
- Commit
- Pipeline
- Stage
- Name
- Run time
- Finish time

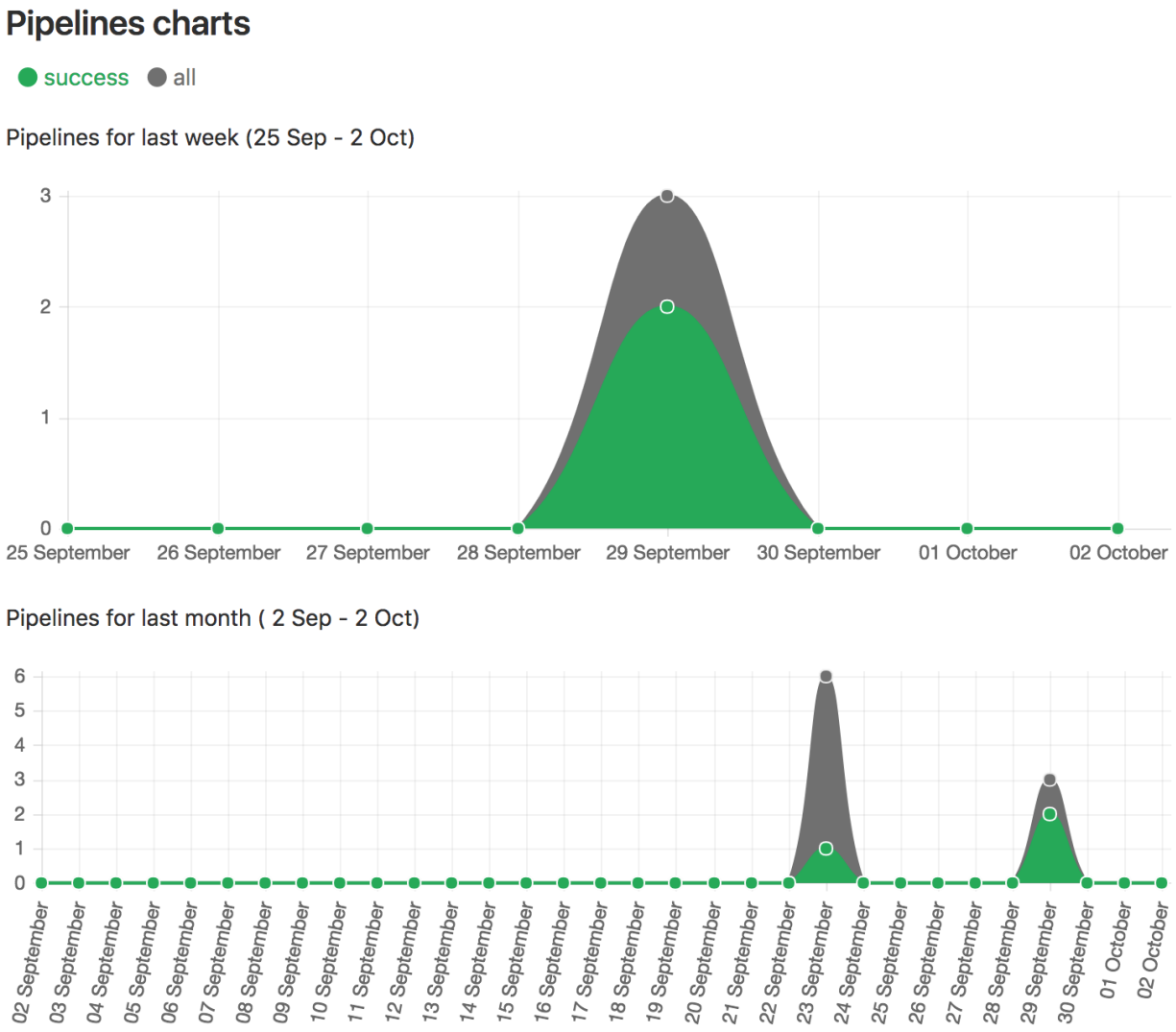


Looking at the preceding screenshot, you'll will be able to see a play button that can be clicked to initiate a job that requires manual involvement. Completed jobs also have a Re-Run button, which can be used to run the job again. Please note that these buttons are only available to people with the proper permissions in the GitLab instance. If there are artifacts to be downloaded, you'll also see a download button, which can be clicked on to begin downloading the zipped file containing all of the artifacts that were built in that job. If you only want to view the jobs relating to a specific pipeline, from the pipelines screen you can click the status of a pipeline and then click the Jobs tab to find similar information. However, this is limited to jobs that have been attached to that pipeline run.

From the menu on the left, under CI/CD, you can click Charts to get some graphical information about the CI/CD process attached to this project:

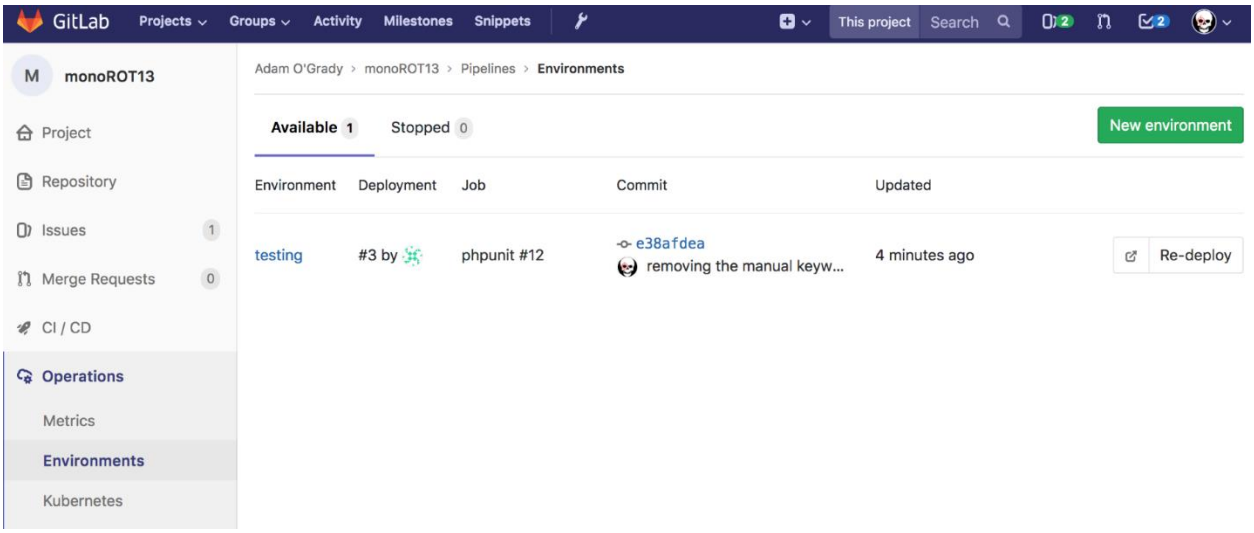


The first section contains some data breakdowns next to a chart of the times required for the pipelines to run. This can be useful for diagnosing issues with your pipelines and analyzing when pipelines need to be optimized to reduce time-to-completion and speed up the development/test/release life cycle. Next up, we have charts showing the breakdown of pipelines over the past week, month, and year:



You can see that it outlines both the total pipelines and the subset of successful pipelines in the charts. This can give good metrics on the active times for a project, as well as show trends relating to work that has been completed.

The last section of interest to us is Environments. By clicking on Operations | Environments through the menu on the left ,you'll be faced with a screen that looks similar to the following:



This shows all of the environments that your app has been deployed to which have been defined in your `.gitlab-ci.yml` file. It shows the name of the environment, the deployment and job that deployed to it, as well as the particular commit that has been loaded on it. Next up is the last updated time, followed by two important buttons. The first one links directly to the environment (opening it in a new tab) and is only available if you specified a URL parameter when defining an environment in your `.gitlab-ci.yml` file. The last button allows you to redo the deployment process in case anything went awry the first time.

Summary

By now, you should have a firm grip on the basics of continuous integration and continuous delivery, as done by GitLab. We started out in this chapter by exploring the concept of CI/CD and giving you a crash course in it.

Next up, we ran through the process of installing a GitLab Runner. Don't forget that the Runner is the platform where your CI/CD stages are actually executed; you can have many of them per project or per GitLab installation to help parallelize the work. We looked at their installation on Ubuntu and CentOS, and manually installing one via a binary. This was followed up by configuring the Runner in the GitLab web UI and then registering it on the Runner host so that it knew which GitLab URL to connect to and set up the registration token it would need.

After we finished configuring Runners, we updated our sample project with some tests to give it a reason to need a CI/CD platform. This was followed by a dive into the required `.gitlab-ci.yml` file, which outlines how the pipelines work for your project. We looked at jobs and a lot of the parameters you can supply to properly configure jobs in GitLab CI. We then set up our own `.gitlab-ci.yml` file for use with our sample project and pushed it to GitLab. From here, we investigated some of the web UIs for GitLab's CI/CD to help familiarize the user with how they can keep an eye on their projects.