Gitlab CI Course Notes

Release June 2019

Jun 20, 2019

Contents:

CHAPTER 1

About

I have created these course notes for the students of the "Gitlab CI: Pipelines, CI/CD and DevOps for Beginners" online course on Udemy. If you are not a student yet, consider registering so that you can take full advantages of this material.

It is additional material for the students that prefer to take notes. You can easily print the entire course content and add your notes.

I did my best to put this together, but it is not perfect. If you spot any spelling mistakes or would like to contribute and make this better, feel free to send me a Pull Request on github.com

Enjoy!

- *Valentin*

CHAPTER 2

Introduction

This section provides a short introduction to Gitlab CI.

2.1 Your first pipeline in Gitlab CI

- In this first video, we are going to create a very simple pipeline in Gitlab
- what is a pipeline anyway?
- the analogy to an assembly line needed to build a car
- **notice a few characteristics:**
  - a series of steps that need to be done on a certain order
  - the steps are connected
  - the output from the previous step is the input for the next step
  - some steps could be done in parallel (wheels)
- many more steps are required to build the final product
- not only the production part is important but also the final testing part before the car goes to the consumer
- the goal is to get the product out of the factory
- you will later see that building and delivering software is, in some regard, quite similar to the step needed to manufacture a car
- **let's try to build something similar with Gitlab CI and have two major stages:**
  - build & test
  - Build a car assembly line using Gitlab CI
  - Add chassis
  - Add engine

    - Add wheels
- later in the test phase, we will test it.
- to get started with Gitlab CI you do not need to download any software or anything similar
- create a free account at gitlab.com
- create a new project. This will we a normal Git repository with nothing inside
- new file .gitlab-ci.yaml
- as in the car assembly, in a software pipeline, we also have a series of steps or jobs what we need to perform
- always use spaces for indentation, not tabs.
- after committing, Gitlab will create the pipeline for us
- every time we make a change in the repository, the pipeline will start
- GitLab will execute your scripts with the tool called GitLab Runner, which runs similarly to your terminal
- Gitlab does not save anything unless told so
- we will use artifacts to save the car.txt file
- inspect final artifact
- where is the build folder in the repository?
- nothing is automatically committed to the repository

2.1.1 YOUR NOTES

2.2 Gitlab architecture

- you need at least one Gitlab Server and one Runner
- the server will provide the interface, store the repositories
- the execution of the pipeline will be delegated to the Gitlab Runner
- so the Gitlab Server does not run the jobs
- this allows for a very scalable architecture
- see the Runners under the project Settings > CI/CD > Runners
- we are using Shared Runners provided by Gitlab.com
- you can create your own runners on your own IT infrastructure and still use gitlab.com
- you can assign runners for specific projects

2.2.1 YOUR NOTES

2.3 Why Gitlab / Gitlab CI?

- Gitlab is a modern tool, and Gitlab will probably become one of the market leaders in the next years
- **Gitlab offers:**
  - a modern, scalable architecture
  - you can easily work with Docker
  - pipeline as a code
  - partially open source
- you need to try it on your own and see if it solves YOUR problems

2.3.1 YOUR NOTES

2.4 How much does Gitlab cost?

- **there are two ways to run Gitlab**
  - gitlab.com
  - self-hosted on your server infrastructure
- **gitlab.com**
  - has a free package
  - 2000 pipeline minutes
  - easy to start and try it out
- **self-hosted**
  - has a free option as well (Community Edition)
  - you need to take care of running Gitlab (installation, updates, infrastructure, backups, . . .)
  - have control over your data

2.4.1 YOUR NOTES

CHAPTER 3

Basic CI/CD workflow with Gitlab CI

This section provides a short introduction to the CI/CD workflow.

3.1 What is CI / CD?

- **CI stands for Continuous Integration**
  - CI is a practice
  - code is integrated with other developers
  - most commonly the way to integrate code is to check if the build step is still working
  - a common practice is to also check if unit tests still work
  - coding guidelines should also be considered (can be mostly automated)
  - work is integrated all the time, multiple times per day
  - most of the time the goal of the CI pipeline is to build a package that can be deployed
- **CD can stand for Continuous Delivery**
  - CD is done after CI
  - you cannot do CD without first doing CI
  - **the goal of CD is to take the package created by the CI pipeline and to test it further**
    - making sure it can be installed (by actually installing the package on a system similar to production)
    - run additional tests to check if the package integrates with other systems
  - after a manual check/decision, the package can be installed on a production system as well
- **CD can also stand for Continuous Deployment**
  - CD goes a step further and automatically installs every package to production

  - the package must first go through all previous stages successfully
  - no manual intervention is required

YOUR NOTES

- **the advantages of CI**
  - detecting errors early in the development process
  - reduces integration errors
  - allow developers to work faster
- **the advantages of CD**
  - ensure that every change is releasable
  - reduces the risk of a new deployment
  - delivers value much faster (changes are released more often)

3.2 Short introduction to Node.js and npm

- Node.js is a runtime environment for executing JavaScript
- initially, JavaScript was only executed within a browser
- Node.js opened the possibility of running JavaScript without a browser.
- we will need Node.js for running some tools
- npm is the Node Package Manager
- npm is used to install new tools/libraries

3.2.1 YOUR NOTES

3.3 Creating a new project

- to build a simple static website we will use a tool called Gatsby
- Gatsby runs on Node.js and we need to use npm to install it
- first, check that Node.js / npm are installed.
- open a terminal and run *node –version* and *npm –version*

- in order to install simply gatsby, run *npm install -g gatsby-cli*
- *-g* instructs npm to globally install the tool on your computer (so that you can use it from any folder)
- to create a new website run *gatsby new static-website*
- instead of static-website you can use any name you like
- inspect the website by starting the local development server: *gatsby develop*
- open the website from the address http://localhost:8000

3.3.1 YOUR NOTES

3.4 Building the project locally

- *gatsby develop* can only be used for local development
- we need to create a production-ready version of the website
- to create a build from gatsby we use: *gatsby build*
- the output is inside the public folder

3.4.1 YOUR NOTES

3.5 Short introduction to images and Docker

- Docker is a tool that allows virtualization
- there is a large collection of images that have many software configurations "pre-installed"
- docker works with images
- an image is a file with a set of instructions on how to package code or tools and all the dependencies
- once an image is executed, it becomes a container
- a container has some similarities with a virtual machine (but it isn't one)
- traditional CI server require to install all the tools/dependencies on the server
- Gitlab breaks this pattern (see the architecture lecture as well) and works with Docker images which contain all the dependencies
- the Gitlab CI Runner with be using the specified Docker images

3.5.1 YOUR NOTES

3.6 Building the project using Gitlab CI

- I recommend you use a code editor to edit the project files
- I am using Visual Studio Code which is free to use and can be downloaded from https://code.visualstudio.com/
- we now need to replicate the steps that we have done on our computer in Gitlab
- we first need to create a new file for defining the pipeline: .gitlab-ci.yml
- we use *npm install* to install all the project dependencies
- we also need to install Gatsby: *npm install -g gatsby-cli*
- with *gatsby build* we will build the website on Gitlab
- we prefer to use a Docker image instead of manually installing node & npm
- the reason for using a Docker image with pre-installed node & npm is speed and a simpler pipeline configuration file
- **use Docker Hub (https://hub.docker.com/) to search for images**
  - make sure you are using official images
  - make sure the image that you plan to use is updated regularly
  - usually a large number of downloads in a good indication that the image is popular and up-to-date
  - we will use the "node" image without specifying a version
  - if the job fails, try using node in a specific version:
- due to the architecture of Gitlab, executing the jobs may seem very slow (compared to other CI servers)
- we will improve the speed of the jobs during the course
- the output from the job is not saved anywhere, so we need to define an artifact
- the artifact contains only the public folder (no other files)

3.6.1 YOUR NOTES

3.7 Adding a test stage

- **why do jobs fail?**
  - all command have an exit code (between 0 and 255) that is returned
  - 0 means successfully executed
  - >0 means that is failed

- we will add a simple test
- the test will check if a string is found inside the index.html file (the start page of the website)
- we use the command *grep* like this: *grep "Gatsby" index.html*
- use *echo $?* will give you the last exit code (on Unix-like machines)
- it is essential to find a way to test your assertions, to make sure that the pipeline fails

3.7.1 YOUR NOTES

3.8 Running jobs in parallel

- we can use the alpine Docker image to optimize our build speed
- alpine images are only 5MB in size and are faster to download and start
- we can use Gatsby to start a server from the production build (the public folder)
- when we start a server with the website, we can run other kinds of tests
- we use *gatsby server* to start a local server
- we want to start a server with our website and check if it works
- we can use curl to download a copy of the website using HTTP (similar to what a standard browser does)
- pipes (|) are used to use the output from one command as the input for another
- assigning two jobs to the same stage makes them run in parallel
- when planning to run parallel jobs, you need to make sure there are no dependencies between them

YOUR NOTES

3.9 Running jobs in the background

- gatsby serve is a never-ending process; it will run forever
- such commands will block the pipeline as the runner will wait for them to complete (which they don't)
- all jobs have a default timeout (most commonly 1h)
- if a job is not done in 1 hour, Gitlab will terminate it and the pipeline will be marked as failed
- adding & after a command will release the terminal and run the command in the background
- with the sleep command, you can add a delay between commands
- using sleep is considered rather a bad practice, but in some cases, it is acceptable (especially if we are talking about a few seconds)
- if you are adding sleeps that exceed 30 seconds, it might be a sign that you are doing something working
- pipelines can run in parallel (if the previous pipeline did not finish before the next pipeline is triggered)
- if you want to stop a running job, you can simply open the job and click on Cancel
- we have used the tac command to fix an error that has occurred as a result of using curl

YOUR NOTES

3.10 Deployment using surge.sh

- surge is a cloud platform for hosting static websites
- surge is easy to use & configure
- install surge on your computer using *npm install –global surge*
- to create an account/project simply run *surge* and follow the instructions

YOUR NOTES

3.11 Using environment variables for managing secrets

- do not store any credentials (username, passwords, tokens) in your pipeline OR project files
- with Gitlab you can define environment variables that contain secrets
- environment variables will be available when running the pipeline
- in order to deploy with surge from Gitlab, we need to generate a token: *surge token*
- we do not want to give Gitlab our username (email) and password, so a token is a better alternative
- to create an environment variable from your project to Settings > CI/CD
- the variables that we will use are SURGE_LOGIN and SURGE_TOKEN
- the name of the variables is given by Surge, as Surge will automatically detect them

YOUR NOTES

3.12 Deploying the project using Gitlab CI

- every time you want to use a tool, you need to make sure it is installed
- the node Docker image that we use does not include surge, so we use npm to install surge
- alternatively, we could use a Docker image that already has surge installed
- to deploy a project run: *surge –project ./public –domain SOMENAME.surge.sh*

YOUR NOTES

CHAPTER 4

Gitlab CI Fundamentals

This section provides an overview of the most essential features available in Gitlab CI.

4.1 Overview

- we will go over the most critical features in Gitlab
- **we will continue to improve the existing pipeline**
  - execution speed
  - general Git workflow
  - add environments
  - add manual steps
- we will go over the fundamentals but not always get into details
- consider reviewing the resources which often point to the official documentation
- assignments will help you get more practice and explore additional feature of Gitlab

YOUR NOTES

4.2 Predefined environment variables

- let's try to add a version to your website so that we know which version is currently deployed
- Gitlab comes with a large list of predefined variables
- Full list at https://docs.gitlab.com/ee/ci/variables/predefined_variables.html
- Try the following in your pipeline to see how variables look like: *echo $CI_COMMIT_SHORT_SHA*
- the dollar sign $ indicates that this is a variable
- we will add a "version" to the website by replacing a marker
- edit the file *src/pages/index.js*
- we will use %%VERSION%% as a marker, but you can use whatever you like
- **sed tool - stream editor**
  - *sed -i 's/word1/word2/g'* inputfile
  - s is for substitute
  - use /g at the end for a global replace
  - -i option for edit in place
  - *sed -i "s/%%VERSION%%/$CI_COMMIT_SHORT_SHA/" ./public/index.html*
  - don't forget to use double quotes when using variables, as they won't be replaced
- adapt jobs / tests to reflect the new information
- *curl -s "instazone.surge.sh" | grep "$CI_COMMIT_SHORT_SHA"*

YOUR NOTES

4.3 Pipeline triggers / Retrying failed jobs / Pipeline schedules

- sometimes jobs fail for no apparent reason
- especially if your pipeline takes a long time to build and one of the last jobs failed, it might be worth retrying it
- **pipelines can be triggered manually without commits**
  - manually click on "Run Pipeline"
  - can select the branch
  - define variables
- **you can set a schedule when your pipeline should run**
  - from your project to go CI/CD > Schedules
  - click on New Schedules

  - you can use one of the predefined options or define your own using the cron syntax
  - you can wait for the pipeline to run or you can manually run it (in case you want to test the configuration)
  - you can run some jobs using the condition *only/except: - schedules*
  - see the full documentation at https://docs.gitlab.com/ee/user/project/pipelines/schedules.html

YOUR NOTES

4.4 Using caches to optimize the build speed

- you probably have noticed that some of the jobs do need a lot of time to run
- especially the build job which needs to download some dependencies before it can run
- if you are used with other more "traditional" CI servers like Jenkins, this extra time might seem like "forever"
- this behavior occurs because each job is started using a clean environment and only the code within Git is available
- rest assured, there is a solution for this and it is called "cache"
- using caches it is possible to speed up the execution of the job by instructing Gitlab to hold onto some files that we might need
- **What to cache?**
  - ideal candidates for caching are the external project dependencies that are not stored in Git and that need to be downloaded
  - in our case, the project dependencies are defined in the packages.json file as npm dependencies
  - the folder that npm uses is called *node_modules*
- Usage in .gitlab-ci.yml:

```
cache:
    key: ${CI_COMMIT_REF_SLUG}
    paths:
        - node_modules/
```

- the cache can be used locally (on a job level) or globally
- you should notice that each job now has an overhead of downloading the cache (pull) and re-uploading the cache (push)
- **troubleshooting: Clearing caches**
  - sometimes caches misbehave
  - go from your project to Pipelines
  - click on the button "Clear Runner Caches" to delete the cache
- fine-tuning the cache is a more advanced topic, but for the moment we are good to go

YOUR NOTES

4.5 Cache vs Artifacts

- let's clarify one thing: the difference between cache and artifacts
- they might seem very similar but they are not the same thing and serve different purposes
- **artifacts**
  - is usually the output from the build process (the package that we want to deploy)
  - an artifact can be partial (if the final package is built across multiple stages)
  - artifacts can be used to pass data between jobs/stages
- **cache**
  - should not be used for storing artifacts (even if technically possible)
  - should only be used as temporary storage for project dependencies
- read the official documentation: https://docs.gitlab.com/ee/ci/caching/#cache-vs-artifacts

YOUR NOTES

4.6 Environments

- currently, we are directly deploying to master (which is not optimal)
- look at the CI/CD diagram we can notice that we are a few systems short
- even if we do Continuous Deployment, we rarely want to deploy to the production system directly
- adding a pre-production or testing stage and running some tests there is a desirable approach
- this allows us to run different kind of tests which require the whole system to respond (usually called integration or acceptance tests)
- it also allows us to test the deployment process before doing this on the production system
- our scenario is very simplistic, but the same idea applies even to large and complex systems
- Gitlab has the concept of environments
- environments allow you to control the continuous deployment of your software
- allows you to track your deployments, so that you know what is currently installed, on which systems and in which version
- environments let you simply tag your jobs and in this way Gitlab knows what you are doing
- you can do this inside a deployment job with:

```
environment:
name: staging
url: http://somedomain.surge.sh
```

- you can view your environments from your project page by going to Operations > Environments

YOUR NOTES

4.7 Defining variables

- it is not a good idea to duplicate information that can change (for example the domain name)
- you can define variables in the jobs or globally
- you can specify a variable like this:

```
variables:
  STAGING_DOMAIN: somedomain.surge.sh
```

- now if you need to change the domain name, you only have to do it in one place

YOUR NOTES

4.8 Manual deployments / Manually triggering jobs

- we are revisiting the Continuous Delivery strategy
- when doing Continuous Delivery, we want to have a manual review step before going to production
- Gitlab offers the possibility of manually triggering jobs
- add *when:manual* to the jobs that need this
- now you will need to view the pipeline and manually click on the "play" button associated with the job
- if there are additional stages after the manual job, they will still be executed
- if this is not desired, additionally configure the manual job with: *allow_failure: false*
- *allow_failure: false* combined with a manual job will set the pipeline in the status "Blocked"

YOUR NOTES

4.9 Merge Requests: Using Branches

- right now everything is pushed to master
- while we have in place tests to make sure nothing goes to production that does not work, it still breaks the pipeline
- a broken pipeline means other developers cannot continue working => costs time & money
- so we want to avoid breaking the master, as much as possible
- we could use branches for each new feature, task or bugfix
- once the work was reviewed and the pipeline is successful, the branch can be merged back to the master branch
- this also ensures that the master branch is all the time deployable (which is an essential aspect of CD)
- there are many strategies for dealing with branches
- one of the most known branching models is Gitflow
- you are free to use which model works best for you
- just avoid working only with the master branch
- we will simply create a new branch for each new change and stop pushing directly to the master branch
- if we create a branch, all the jobs will run as normal
- but we do not want to deploy to staging or production from a branch
- we can set a job policy to run the deploy jobs only for the master branch:

```
only:
  - master
```

4.9.1 YOUR NOTES

4.10 Merge requests: Configuring Gitlab

- to implement our new workflow, we need to do a few settings for your project
- **no longer allow pushing to master**
  - go to Settings > Repository > Protected branches
  - set Allow to push to "No one"
  - nobody will be able to push a change directly to master
  - all the changes must go through the process of creating a Merge Request
- **configuring Merge Requests**
  - go to Settings > General > Merge Requests
  - set Merge method to Fast-forward merge
  - under Merge checks, check Pipelines must succeed

YOUR NOTES

4.11 Merge requests: Your first merge request

- to create a Merge Request (MR) we first need to create a branch
- make a change inside your branch (add some text or something to the website)
- on top of Gitlab you should see Gitlab inviting you to create a Merge Request
- you can also do this from Merge Requests > New merge request
- the Title of the MR will be pre-filled with the commit message you have given
- you can select to delete the branch after the Merge Request was accepted (merged)
- after the branch is merged, the master pipeline will start

YOUR NOTES

4.12 Dynamic environments

- right now we don't have an environment where we can inspect the merge requests
- we can automatically spin up a dynamic environment for each merge request
- this allows us to review the changes on an actual system
- we can also run more advanced tests if we want to
- this can not only be a good thing for developers but for testers or product owners/project managers and so on
- we can make a dynamic environment by using predefined Gitlab variables
- **we can use the following variables**
  - $CI_COMMIT_REF_NAME to have the branch name as the environment name
  - $CI_ENVIRONMENT_SLUG for a url-friendly environment name

```
environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://instazone-$CI_ENVIRONMENT_SLUG.surge.sh
```

YOUR NOTES

4.13 Destroying environments (Clean-up after the Merge Request)

- with so many potential branches, once they are merged, the environments that were created are no longer needed
- we need to tell surge to delete the environments when we don't need them anymore
- surge documentation: https://surge.sh/help/tearing-down-a-project
- by setting the variable GIT_STRATEGY to none inside a job, you will disable git cloning for that job
- this is needed for the "stop review" which needs to run even if the branch was deleted
- if the branch is deleted, it does not make sense to clone the repository and try to open that branch
- "deploy review" needs to have a link to the "stop review" job
- the link is setting on_stop
- "stop review" will be automatically triggered by Gitlab one the branch was merged

YOUR NOTES

CHAPTER 5

Gitlab CI Online course

This course notes have been created for the students of the "Gitlab CI: Pipelines, CI/CD and DevOps for Beginners" online course on Udemy. If you are not a student yet, consider registering so that you can take full advantages of this material.

Here to sign-up to the course on >> .. _Udemy: https://www.udemy.com/gitlab-ci-pipelines-ci-cd-and-devops-for-beginners/?couponCode=COURSE_NOTES

CHAPTER 6

Indices and tables

- genindex
- modindex
- search