

Learning DevOps tools with World IT Experts

by Artemii Kropachev and Denis Zuev

Table of contents

Table of contents	2
Intro	6
Authors	6
What is this book about?	6
Why Jenkins and GitLab	8
Structure of this book	8
About DevOps	9
Business challenges	9
Waterfall, Agile, DevOps	9
CI/CD	10
DevOps tools	11
Requirements	12
You as a reader	12
Hardware requirements	12
Software requirements	12
VirtualBox installation	13
Vagrant installation	14
Installing Git client	14
Verification	14
Lab environment overview	15
Getting started with GitLab	20
Git repository management	20
GitHub	20
GitLab	21
BitBucket	21
GitLab, GitHub, and BitBucket comparison	22
Git	22
GitLab installation	23
Connecting to GitLab	25
Git basics	33
Using Git help	34
Working with Git	35
Sharing git projects	41
Working with merge requests	50

Getting started with Jenkins	59
About Jenkins	59
Installing Jenkins	59
Pipelines overview	66
Creating Jenkins Pipeline	67
Groovy - Basics	70
Node	71
Stage	72
Step	73
Input	76
Pipelines - advanced syntax	80
Variables	80
Functions	82
Variable + Function() == LOVE	92
For loops	93
Try, catch, finally	96
If, else if, else	98
Switch case	102
Retry	105
Parallel	108
Configuring Jenkins	109
Installing plugins	110
Updates	111
Available	112
Installed	115
Adding new users	116
Jenkins and GitLab API	120
Why API?	120
API access tools	120
Using curl to access APIs	120
Formatting output using jq	122
Using GitLab API	124
GitLab API	124
Authentication	124
API Status codes	127
Managing GitLab projects	127
List projects	127
Get single project	130

Create a project	131
Update project settings	133
Delete project	134
Other project-related activities	135
Managing GitLab users	135
Managing branches	139
Managing merge requests	141
Managing other GitLab entities	146
Using Jenkins API	147
Jenkins API	147
Authentication	149
List jobs	152
Create a Job	153
Trigger a build	156
Making GitLab and Jenkins to work together	157
Why we need it	157
What we want to achieve	157
Moving Jenkins pipelines to GitLab	157
Create a GitLab repo with pipeline script	158
Install git on Jenkins	160
Create Jenkins job	161
Triggering Jenkins job automatically	168
Configuring Jenkins build triggers	168
Using GitLab Jenkins plugin	171
Install GitLab Jenkins plugin	171
Configuring Jenkins to GitLab authentication	173
Configuring GitLab to Jenkins authentication	174
Automatic job triggering via GitLab	177
Allowing remote API calls to Jenkins	177
Configuring GitLab Webhooks	178
Update repository files	181
Updating GitLab progress from Jenkins	182
gitlabBuilds	182
gitlabCommitStatus	183
Integrating Jenkins pipelines with merge requests	186
Application	186
Configure GitLab WebHook	186
Configure GitLab merge settings	187

Configure Jenkins job parameters	188
Creating a merge request	189
More options of integration	194
Conclusions	195

Intro

Authors

Artemii Kropachev is a worldwide IT expert and international consultant with more than 15 years of experience. He has trained, guided and consulted hundreds of architects, engineer, developers, and IT experts around the world since 2001. Artemii's architect-level experience covers Solutions development, Data Centers, Clouds, DevOps and automation, Middleware and SDN/NFV solutions built on top of any Red Hat or Open Source technologies. I also possess one of the highest Red Hat Certification level in the world - Red Hat Certified Architect in Infrastructure Level XX. Artemii's life principles can be defined as "never stop learning" and "never stop knowledge sharing". Artemii would love to share some of his experience and knowledge via this book.

Denis Zuev. A lot of people in the industry know Denis for his Certification Achievements. He is holding a lot of expert-level certifications from the leading IT companies in the industry, such as Cisco Systems, Juniper Networks, Vmware, RedHat, Huawei, and many others. Currently, he holds the following ones: RHCI, RHCX, RHCA Level VII, 6xCCIE, 4xJNCIE, CCDE, HCIE, VCIX-NV. You won't be able to find a lot of people with such a diverse set of certification achievements. Denis doesn't have a specific area of expertise and worked in different technologies including Networks, Servers, Storage, Cloud, Containers, DevOps, SDN/NFV, Automation, Programming, and Software Development. He can easily describe himself with a single sentence - "**Give me a screwdriver and I will build you a rocket**". Denis worked with leading IT companies around the globe including Cisco Systems, Juniper Networks, Red Hat, and ATT. He is a trainer and a mentor for thousands of people around the globe, and I know how to deliver any information in an easy manner. These days Denis truly enjoys DevOps, Automation, Containers, Software Development, and Professional Training.

Reviewers

Andrey Bobrov started his career as a network engineer and during 10 years covered a lot of enterprise technologies in routing, switching, security, voice, and wireless directions. He has had a lot of network certificates: CCNP, CCDP, HCNP, CCSA, and others. Several years ago he opened for myself DevOps world and changed his career to this way. Currently, he is certified as RHCA Level VIII and continues his work for his Red Hat certifications, because this vendor is the leader of the market nowadays. He works as a team leader of the DevOps team and everyday uses tools, described in this book.

Roman Gorshunov is an IT architect and engineer with over 13 years of experience in the industry, primarily focusing on infrastructure solutions running on Linux and UNIX systems for Telecoms. He is currently working on delivering automated OpenStack on Kubernetes – resilient

cloud deployment (OpenStack Helm) and CI/CD for the AT&T Network Cloud platform based on OpenStack Airship; he is a core reviewer in Airship project and also a developer in other Open Source projects.

What is this book about?

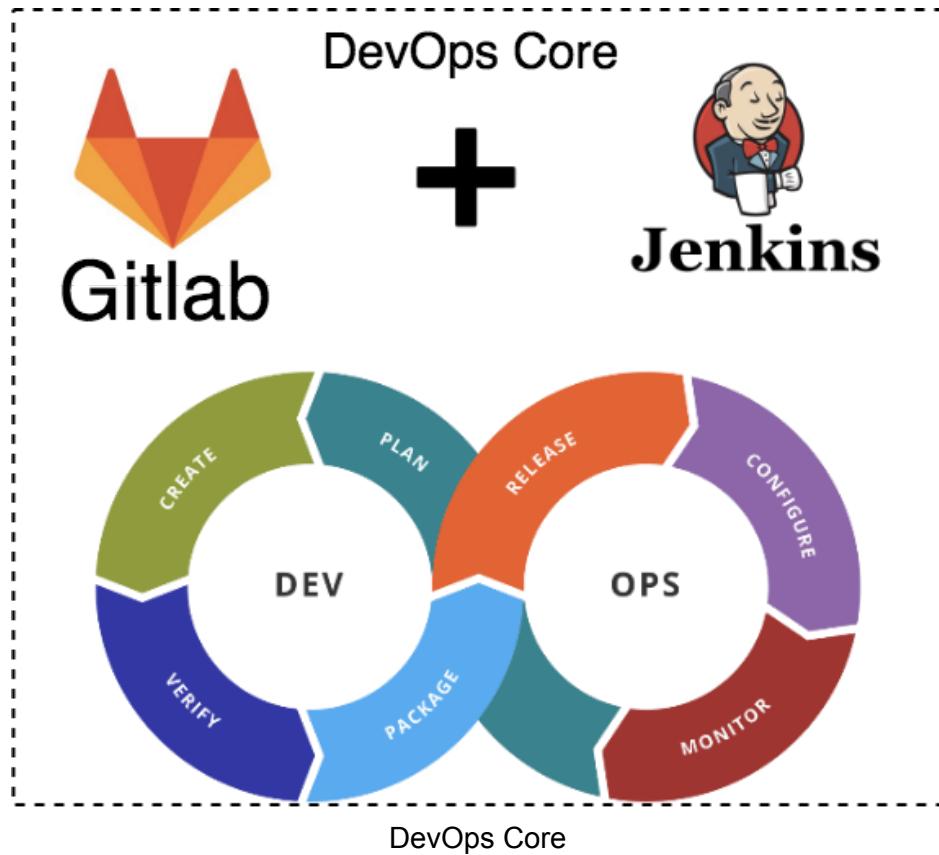
Artemii Kropachev. I implemented many projects in different areas and all of them required a certain level of automation including continuous delivery and even continuous deployment which sound like something impossible and something absolutely new for most of the companies I worked with. I was surprised that even very large corporation with 50+ thousands of servers and hundreds of applications don't follow DevOps best practices and don't understand what it means. I want to share some knowledge to allow make you familiar with common DevOps processes and tools.

Denis Zuev. Just recently, after taking a countless number of interviews in different companies across the USA, I got on a call with my friends and told them this sad story that there is little to no Automation within the largest companies out here when it comes to Network and Infrastructure teams. I got the same feedback from my friends who work in the leading IT companies. I have a lot of requests from my former students to teach them DevOps and Automation. This is why we started this book.

There is quite a lot to learn. Most of these components are quite important and critical for every modern Enterprise, DataCenter or Service Provider Environment. It is not possible to cover everything in a single book or video class, otherwise, it would be a book with 10 000 pages or 1000 hours of videos. This is why we broke it down into multiple books and video classes.

In this book, we cover a tiny but very important piece of the whole DevOps ecosystem, and we call it "**DevOps Core**".

This book is your easy way to start learn famous DevOps tools and describe why you should choose it and how you can use it together. We are positive that this book helps to make the first step for engineers who think about changing their career as a lot of our students did.



Why Jenkins and GitLab

As we have mentioned previously, that we have taken over 100 interviews for DevOps positions in different areas, that includes Microservices, Networks, Security, Server, and Infrastructure Automation, Cloud and Application Development. And the pay rates were just great. We are not telling you that once you are done with this book, that will immediately get you a well-paid job. No! But if you want to work in these areas, this book will get you one step closer.

One of the things that a lot of experts in the fields are missing nowadays, it is a tiny but the very important piece of the puzzle called "**DevOps Core**" skill set. If you are new to Jenkins and GitLab and want to learn how to deliver your code to the final destination in an automated manner, then this book is for you.

GitLab functionality is very similar to a well-known repository management public service GitHub. We didn't choose GitHub since it cannot be installed in a local isolated environment which we will be using in this book. In most cases, everything you will learn about GitLab is applicable to GitHub too.

Structure of this book

The structure of this book is quite simple. We have the following chapters for you:

- [**Intro**](#). We introduce you to DevOps and explain what that buzzword is really about. We also use Vagrant here to build the lab and then start learning GitLab and Jenkins.
- [**Getting started with GitLab**](#). We introduce you to GitLab as a main Version Control tool. We tell you what GitLab is and why we need it. We show you how to install, configure and work with GitLab, that includes creating merge requests.
- [**Getting started with Jenkins**](#). We introduce you to Jenkins as the main CI/CD tool. We explain to you what Jenkins and CI/CD are. We show you how to install and configure Jenkins. One of the most important topics where we focus and spend most of our time is **Jenkins Pipelines**. Finally, we do a bidirectional integration between GitLab and Jenkins. This is where you will see the real power of DevOps automation. Trust us, this part of the book you will need to understand 100%. This will be our first integration point with GitLab as well. We learn basic and advanced pipeline syntaxes along with other topics such as Jenkins shared libraries and Jenkins API. API is everywhere, and it is not yet another buzzword.
- [**Jenkins and GitLab APIs**](#). You will learn how to manage GitLab repositories and work with GitLab API. Yes, we will tell you what API is and how to use it in real life. We learn how to use Jenkins API. API is everywhere, and it is not yet another buzzword.
- [**Making GitLab and Jenkins to work together**](#). We go through mutual integration between GitLab and Jenkins step by step. This is where you will see the real power of DevOps automation. The goal is to trigger an automation pipeline by a Git event - merge request which we will explain in details.

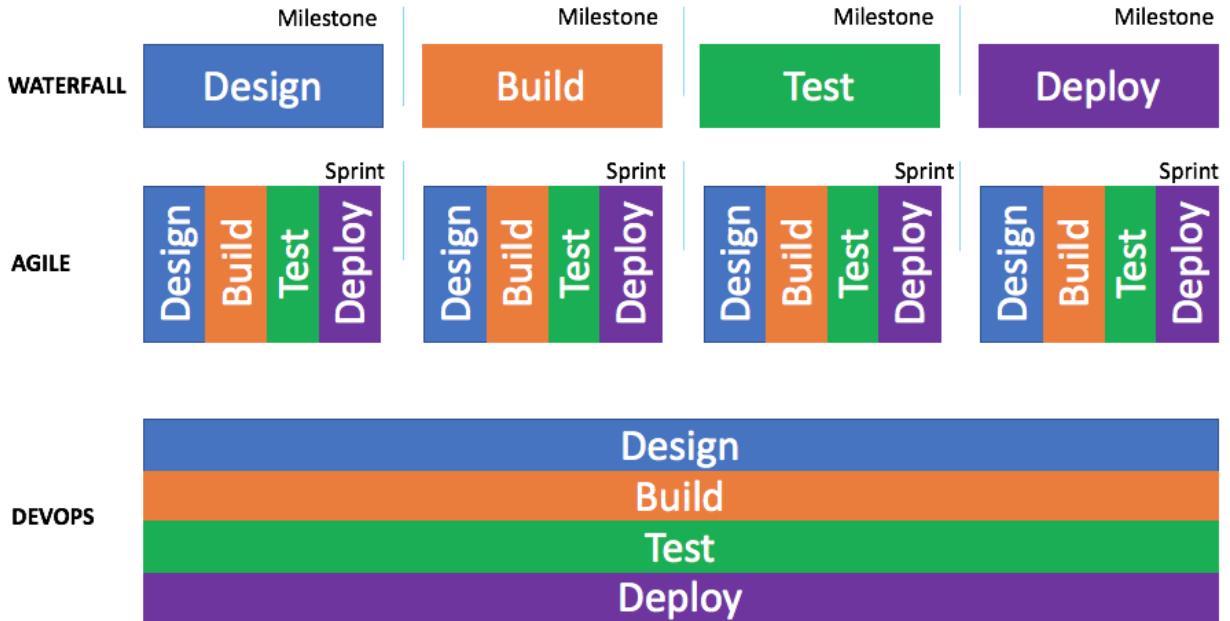
About DevOps

Business challenges

These days IT departments are challenged by business to be able to compete with other companies. It is important to have the required features as soon as they are developed. All companies are interested to reduce time and efforts to deliver change to production. DevOps methodology allows achieving that by the highest level of automation.

Waterfall, Agile, DevOps

DevOps is an evolution of older development methodologies like Waterfall and Agile. You can easily feel the difference between all three by taking a look at the diagram below.



Waterfall vs Agile vs DevOps

Waterfall approach focuses on releasing the product when it is ready for production after it's been designed, built, tested and finally deployed with all the features and functionality that end-user needs. It usually means that the product is stable but it comes at a price of delayed release. It may take one, two or even 3 years to release it. It is really hard to think 3 years ahead.

This is why **Waterfall** evolved into the **Agile** methodology, to release the product in smaller cycles called **sprints**. It can be a feature or functionality, something that makes your product better with every **sprint**. And **sprints** are somewhere in between 2 week and 2 months. It is much better than 1 to 5 years for a full cycle as in Waterfall approach. **Agile** brings a concept of Minimum Viable Product (MVP). MVP is a product with bare minimum functionality pushed to production and ready to be used.

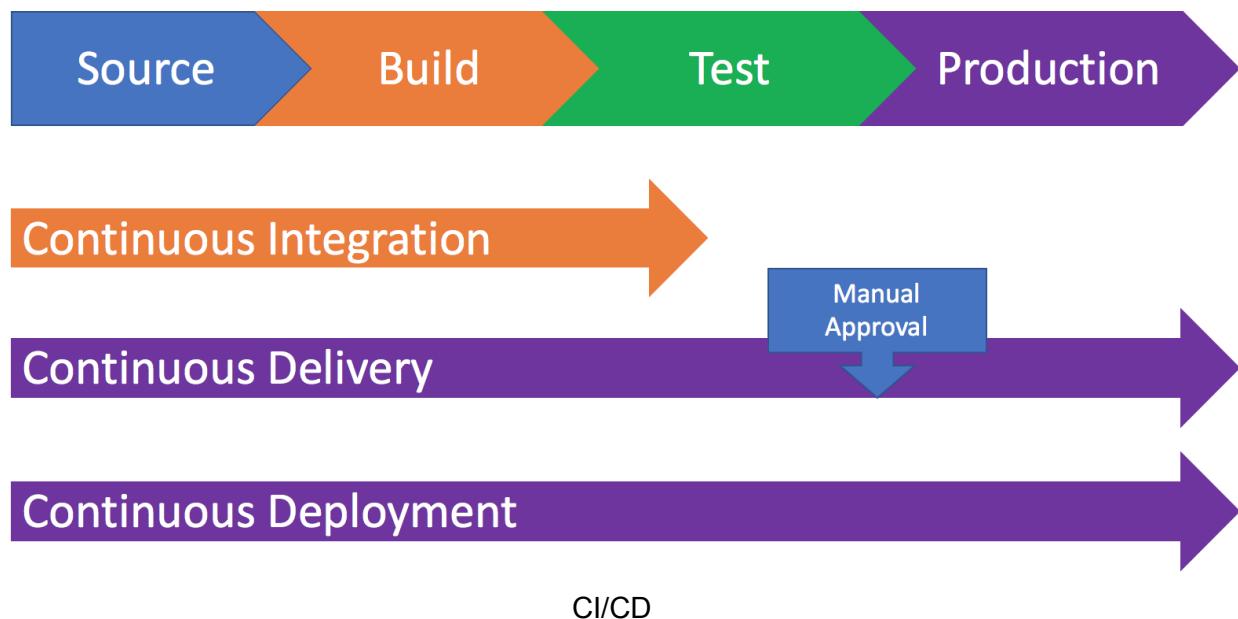
DevOps is another step forward in continuous deployment, where product development lifecycle is continuously happening in an endless loop. Here is one simple example: while you are developing your code, all the other steps like build, test and deploy phases are happening automatically at the moment you are committing your code to the software repository. Using other words, every commit produces a tested and certified micro release of the product which is delivered to production automatically. It sounds impossible, but this is how **DevOps** works. And it is not the only use case, **DevOps** made possible for Development teams to work seamlessly with Operation teams, which was impossible with **Waterfall** and **Agile** approaches.

DevOps also enhanced the MVP concept where it is built with Agile and continuously improved with **DevOps**.

Imagine that you need to automate your network deployment, server or application provisioning from the ground up. The first thing you do, is you build a bunch of scripts that do part of the job one by one developing these scripts as you progress. That is your **MVP**, and then you keep integrating these scripts together into a single self-sufficient product that does the job. At the same time, you keep designing, testing and deploying this product. That is **DevOps**. And in this book, we are going to teach you core tools that allow you to do so.

CI/CD

DevOps often refers to CI/CD which stands for Continuous Integration/Continuous Delivery/Deployment. Take a look at the diagram below.

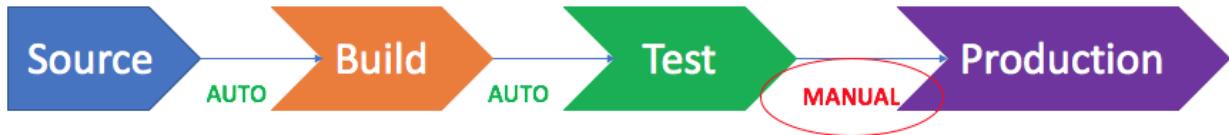


Let's talk briefly about these terms briefly:

- **Continuous Integration** - primarily focused on creating, building and running automation tests for your applications. That allows to save time and find bugs quicker.
- **Continuous Delivery** - Extends **Continuous Integration** by pushing new code to the next stages through manual approval procedures.
- **Continuous Deployment** - extends **Continuous Delivery** by automating code delivery procedures and push it all the way up to production.

The main difference between continuous delivery and deployment is a manual approval which is shown in the diagram:

Continuous Delivery



Continuous Deployment



Continuous Delivery vs Continuous Deployment

DevOps tools

DevOps is widely used and is applicable to many aspects of any business. And because of this, there are so many tools being used to achieve the ultimate goal and automate product delivery process. And by product, we mean anything and everything. It can be a software development or network deployment, or infrastructure support. Most of the work that you do on a daily basis can be taken cared of by using DevOps techniques. As in many other automation or software development processes, you will be required to develop your code/scripts and store it in the source code repository, as well as test, build, deploy and use that code in an automated fashion. GitLab and Jenkins fit perfectly for these tasks. This is why we focus on them in this book.

Requirements

You as a reader

We assume that our readers are proficient in working with their main Operating System, whether is it Windows, Linux or MacOS and know how to use Google Search. We also assume that you have basic Linux experience since we use CentOS/RHEL 7 heavily in this book.

Hardware requirements

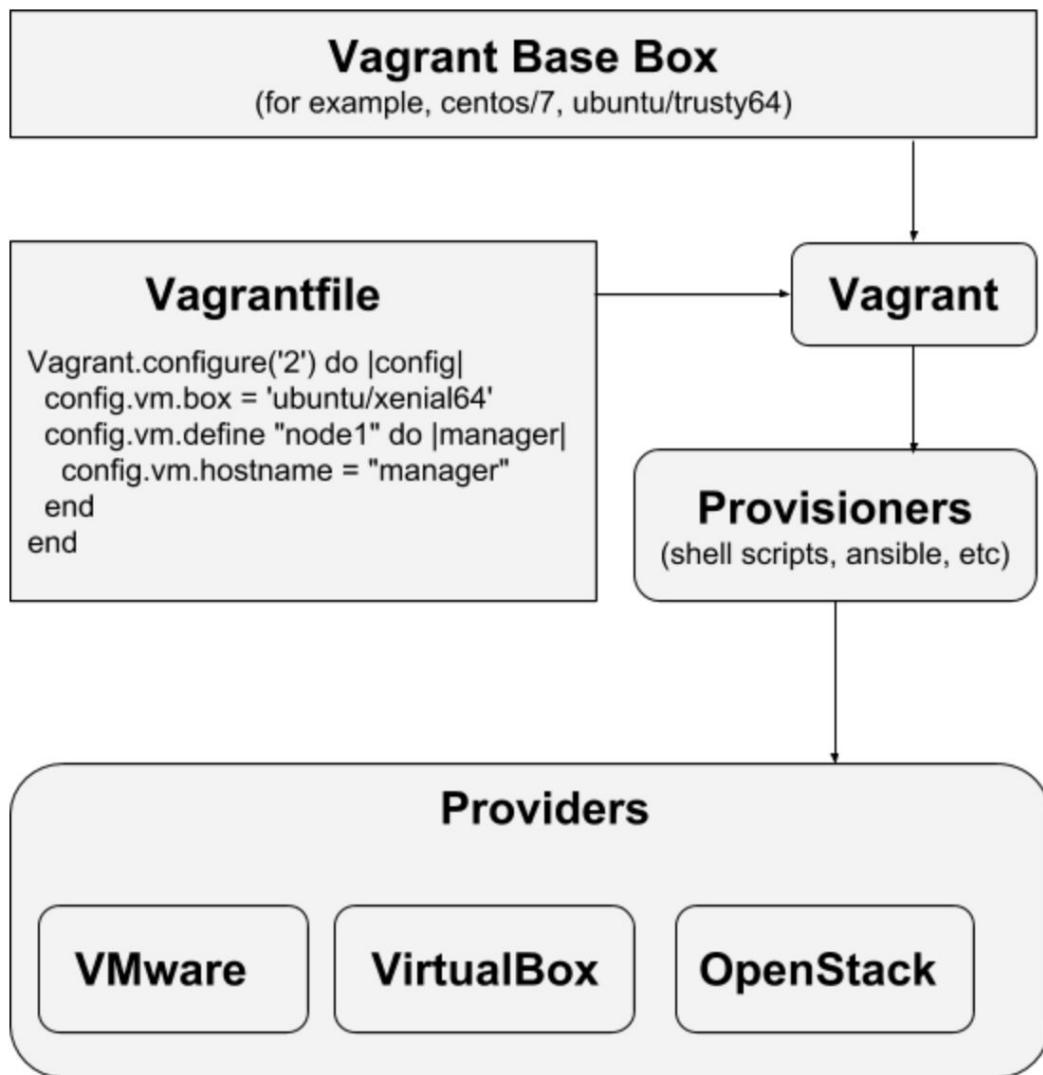
We won't need a lot of computing and memory resources for our lab, so you will be able to run pretty much everything on your laptop if you have at least 8GB, or better if 16GB RAM. CPU is

not that very important, so whatever CPU you have should be enough. We are going to deploy a number of virtual machines and that is why CPU virtualization support is required.

Software requirements

We use **Vagrant** as the main lab provisioning tool along with other scripting tools. If you are not familiar with Vagrant, it allows for simplifying the deployment and the initial configuration of virtual machines, by using different underlying technologies. In most cases, a plain text file called **Vagrantfile** describes the parameters of the VM. Having Vagrantfile allows you to create, stop, and destroy your virtual environment with a single command. The beauty of using Vagrant is that we can redeploy your lab as many times as we need to, and each time, we will have the same result.

This is how Vagrant architecture looks like at a high level.



Vagrant Architecture

Vagrant is an orchestration tool that uses different virtualization providers such as VMware, VirtualBox or OpenStack behind the scenes. We use VirtualBox as the main virtualization provider in this book because it's free and runs on all of the most popular Operating Systems whether it is Windows, Linux or MacOS.

VirtualBox installation

As already mentioned Vagrant is using virtualization providers behind the scene to run Vagrant Machines. One of the most popular and free providers available in the internet is VirtualBox. You can find all the required code and instruction at <https://www.virtualbox.org/wiki/Downloads> to download and install VirtualBox. Depending on OS you are using, you might be required to reboot your OS.

Vagrant installation

The Vagrant instructions and the software to download are available at <https://www.vagrantup.com/docs/installation/>. Just download the package for your OS, and then install it. Vagrant also requires a virtualization platform, like VMware, KVM, VirtualBox, AWS, Hyper-V, or Docker. You will be required to install the appropriate virtualization platform, depending on your operating system.

Installing Git client

The last thing you will need for this book is to install a Git client from <https://git-scm.com/downloads>. Most of our Git-related labs may be performed inside GitLab or Jenkins instances. However, you may be required to use Git throughout the rest of this book and pool all the source code from the source code repository. You are free to use other installation methods, like Brew for MacOS, or yum/dnf/apt for Linux distros. Once you have Git client installed, navigate to your working directory using your favorite CLI.

Verification

Let's check that we have everything in place by, firstly, cloning your gitlab repo:

```
$ git clone https://github.com/flashdumper/Jenkins-and-GitLab.git  
Cloning into 'Jenkins-and-GitLab'...
```

If you get any errors, they should be self-explanatory and easy to fix.

Otherwise, navigate to the cloned directory:

```
$ cd Jenkins-and-GitLab
```

Check that you have 2 main directories for the following chapters:

```
$ ls -l  
total 0  
drwxr-xr-x@ 2 vagrant vagrant 64 Aug 22 02:00 Vagrantfile
```

Before we move any further, It is a good idea to check that you can bring your VMs up using “**vagrant**” command. First, verify that Vagrant have found Vagrantfile and read the configuration.

```
$ vagrant status  
Current machine states:  
  
gitlab           not created (virtualbox)  
jenkins          not created (virtualbox)  
  
This environment represents multiple VMs. The VMs are all listed  
above with their current state. For more information about a specific  
VM, run `vagrant status NAME`.
```

Bring VMs up using “**vagrant up**” command.

```
$ vagrant up  
Bringing machine 'default' up with 'virtualbox' provider...  
...  
<output omitted>  
...
```

Check that Vagrant VM is running.

```
$ vagrant status  
Current machine states:  
  
gitlab           running (virtualbox)  
jenkins          running (virtualbox)  
  
This environment represents multiple VMs. The VMs are all listed  
above with their current state. For more information about a specific  
VM, run `vagrant status NAME`.
```

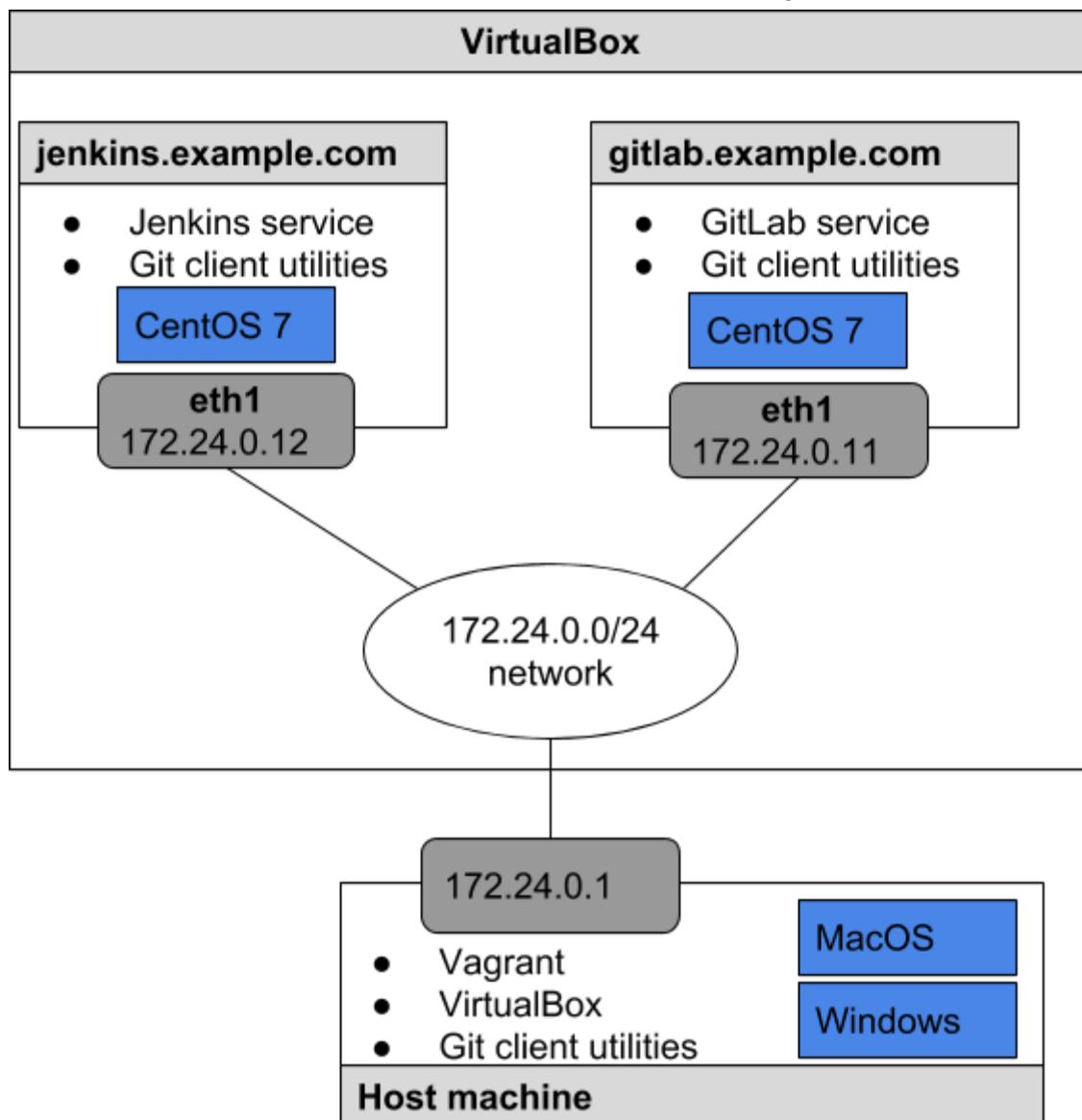
If everything goes well, you should see the output similar to the one above.

Now, destroy the vagrant environment we just created by running “**vagrant destroy**” command.

```
$ vagrant destroy -f
==> jenkins: Forcing shutdown of VM...
==> jenkins: Destroying VM and associated drives...
==> gitlab: Forcing shutdown of VM...
==> gitlab: Destroying VM and associated drives...
```

Lab environment overview

The lab environment is built on top of VirtualBox virtualization software using Vagrant. We need Vagrant to have a repeatable way to provision required virtual machines. You may use Windows, MacOS or Linux host operating system and still be able to host the lab environment required for this book. The lab environment is shown in the diagram:



We will be using the following Vagrantfile:

```
Vagrant.configure(2) do |config|  
  
  config.vm.define "gitlab" do |conf|  
    conf.vm.box = "centos/7"  
    conf.vm.hostname = 'gitlab.example.com'  
    conf.vm.network "private_network", ip: "172.24.0.11"  
    conf.vm.provider "virtualbox" do |v|  
      v.memory = 2048  
      v.cpus = 2  
    end  
    conf.vm.provision "shell", inline: $inline_script  
  end  
  
  config.vm.define "jenkins" do |conf|  
    conf.vm.box = "centos/7"  
    conf.vm.hostname = 'jenkins.example.com'  
    conf.vm.network "private_network", ip: "172.24.0.12"  
    conf.vm.provider "virtualbox" do |v|  
      v.memory = 2048  
      v.cpus = 2  
    end  
    conf.vm.provision "shell", inline: $inline_script  
  end  
end  
  
$inline_script = <<SCRIPT  
# Inline Script starts  
cat <<EOF > /etc/hosts  
127.0.0.1 localhost localhost.localdomain  
172.24.0.11 gitlab.example.com gitlab  
172.24.0.12 jenkins.example.com jenkins  
EOF  
  
yum install -y git  
git config --global user.name "Root User"  
git config --global user.email admin@example.com  
git config --global push.default simple  
# Inline script ends  
SCRIPT
```

Note! This Vagrantfile is stored in our GitHub repository at <https://github.com/flashdumper/Jenkins-and-GitLab>. Use “git clone” command to copy the repository to your local machine.

This Vagrantfile defines 2 VMs (gitlab and jenkins). Both VMs have predefined IP addresses and hostnames. Vagrantfile configures host resolution using records at /etc/hosts inside VMs. However, it may be required to add the following records to your local hosts file:

```
172.24.0.11 gitlab.example.com gitlab  
172.24.0.12 jenkins.example.com jenkins
```

Hosts file location is dependant on your operating system:

OS	Hosts file location
Linux	/etc/hosts
MacOS	/etc/hosts
Windows	C:\Windows\System32\Drivers\etc\hosts

Here is an example how hosts file looks like on MacOS:

```
$ cat /etc/hosts|grep 172.24.0  
172.24.0.11 gitlab.example.com  
172.24.0.12 jenkins.example.com
```

Start gitlab VM and verify connectivity using the commands provided below.

```
$ vagrant up gitlab  
==> gitlab: Importing base box 'centos/7'...  
==> gitlab: Matching MAC address for NAT networking...  
==> gitlab: Checking if box 'centos/7' is up to date...  
...  
<OUTPUT OMITTED>  
...  
==> gitlab: Running provisioner: shell...  
    gitlab: Running: inline script  
  
$ vagrant ssh gitlab  
[vagrant@gitlab ~]$  
  
[vagrant@gitlab ~]$ hostname  
gitlab.example.com
```

```
[vagrant@gitlab ~]$ exit
logout
Connection to 127.0.0.1 closed.

$ ping gitlab.example.com
PING gitlab.example.com (172.24.0.11): 56 data bytes
64 bytes from 172.24.0.11: icmp_seq=0 ttl=64 time=0.361 ms
64 bytes from 172.24.0.11: icmp_seq=1 ttl=64 time=0.369 ms
^C
--- gitlab.example.com ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.361/0.365/0.369/0.004 ms
```

Finally, exit out of gitlab VM and destroy vagrant environment.

```
$ exit
$ vagrant destroy gitlab -f
==> gitlab: Forcing shutdown of VM...
==> gitlab: Destroying VM and associated drives...
```

Do not hesitate to play with Vagrant and practice some basic Vagrant commands. Here is the table of most often used commands:

Command	Description
vagrant up	Create and start all instances defined in Vagrantfile
vagrant destroy	Destroy all instances defined in Vagrantfile
vagrant status	Display status of the instances defined in Vagrantfile
vagrant up gitlab	Create and start gitlab instance, Other instances will stay untouched
vagrant up jenkins	Create and start jenkins instance, Other instances will stay untouched
vagrant destroy gitlab	Destroy gitlab instance
vagrant destroy jenkins	Destroy jenkins instance
vagrant ssh gitlab	Do ssh connection to gitlab instance
vagrant ssh jenkins	Do ssh connection to jenkins instance

vagrant halt jenkins	Stop jenkins instance without destroying
vagrant ssh-config	Display parameters for SSH configuration

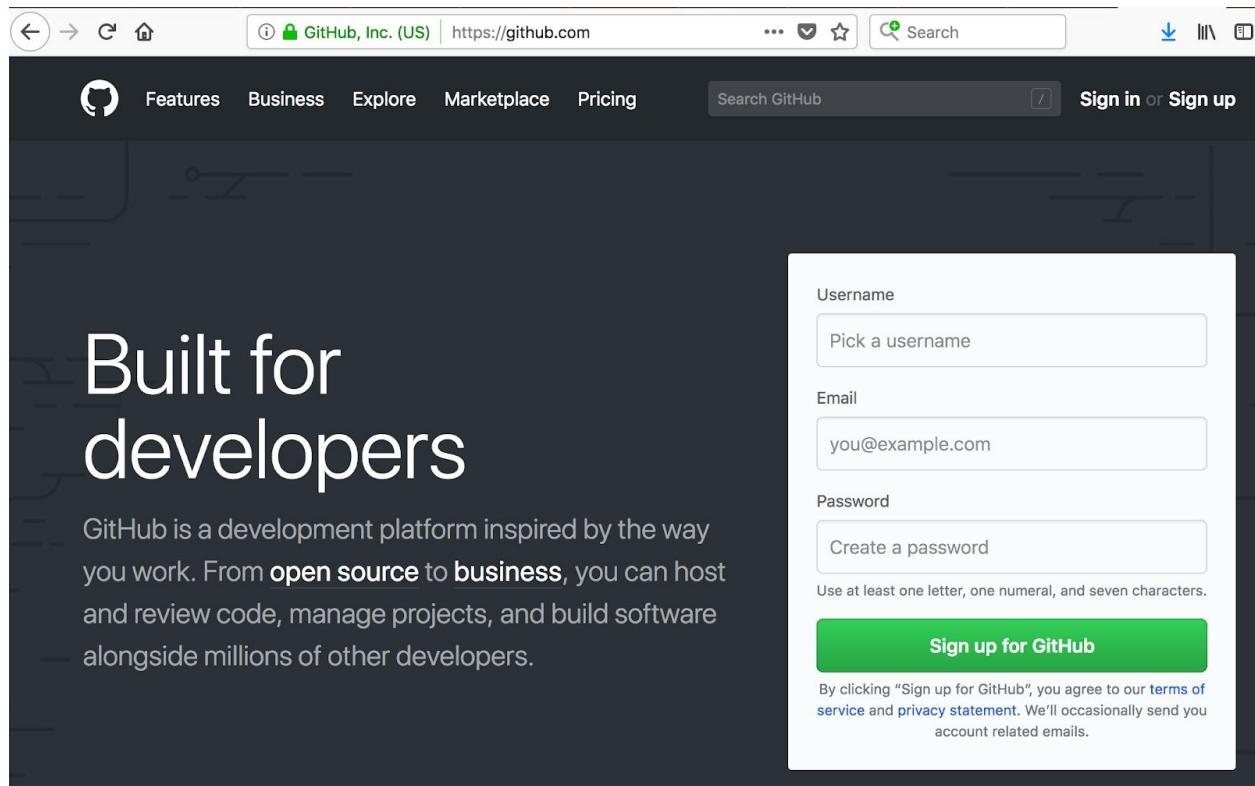
Getting started with GitLab

Git repository management

You may have probably heard of GitHub, GitLab and Bitbucket already. If not, all three are the most popular repository management services. It means that these services allow a developer to store and manage all the development work in a central repository system and keep it under version control. What is the difference between all 3 and where should we use them? We will try to give a short introduction to all of them.

GitHub

GitHub is the largest hosted repository service in the world. You can go and freely create your own account at <https://github.com/>, and create any number of public repositories that are going to be available for everyone. You can also create private repositories but you will need to pay for it.



GitHub.com sign-up interface

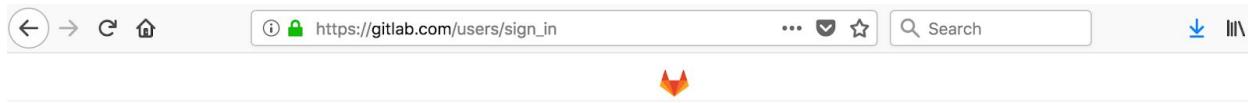
GitHub is widely used in the Open Source world to share the code and work together. Many people publish their work and allow others to freely use it. A lot of other people, use it for storing code sample for demos or even publish the code using in the book as we are doing in the book.

GitHub is awesome and if you have something to share with the rest of the world, do not hesitate and do that right now.

Note: A code repository is a collection of files and directories combined under a single name.

GitLab

Is an Open Source project that allows you to use a publicly available version at <https://gitlab.com>. **GitLab** is not the same as **GitHub**. These are two different services. While both **GitHub** and **GitLab** are repository services and have hosted version, only **GitLab** has an option to be installed on a local server. That is one of the reasons why it is so popular. Not always you want or can keep all your work on a publicly hosted servers. GitLab has two versions - **Enterprise** and **Community** editions. **The community** edition is available to download and install on your server. **Enterprise** edition is a paid version with additional features on top of **Community** edition features. Great service when you want to bring repository management service into your data center and host internally.



GitLab.com

GitLab.com offers free unlimited (private) repositories and unlimited collaborators.

- [Explore projects on GitLab.com \(no login needed\)](#)
- [More information about GitLab.com](#)
- [GitLab.com Support Forum](#)
- [GitLab Homepage](#)

By signing up for and by signing in to this service you accept our:

- [Privacy policy](#)
- [GitLab.com Terms.](#)

A screenshot of the GitLab.com sign-in interface. It features a 'Sign in' button at the top left, a 'Register' link at the top right, and input fields for 'Username or email' and 'Password'. Below the password field is a 'Remember me' checkbox and a 'Forgot your password?' link. At the bottom is a large green 'Sign in' button. To the right of the sign-in form is a smaller box containing social media sign-in options (Google, Twitter, GitHub, Bitbucket) and a 'Remember me' checkbox. Below the sign-in form is a link to request a new confirmation email.

GitLab.com sign-in interface

BitBucket

BitBucket is one of the Atlassian services that offers the similar repository management services as GitHub and GitLab with additional integration features to other Atlassian services. Other Atlassian products and services that you may have heard are Jira and Confluence.

The screenshot shows the BitBucket homepage. At the top, there's a navigation bar with icons for back, forward, search, and other site functions. The URL is https://bitbucket.org. Below the navigation is the BitBucket logo and links for Features, Enterprise, and Pricing. On the right are Log in and Get started buttons. A banner at the top says "6 steps away from importing your GitHub repositories. Import now". The main heading "Built for professional teams" is displayed in a large, bold font. Below it, a subtext reads "Bitbucket is more than just Git code management. Bitbucket gives teams one place to plan projects, collaborate on code, test and deploy." There are two calls-to-action: "Get started for free" and "Or host it yourself with Bitbucket Enterprise →". Below these are snippets of code and user profiles.

BitBucket interface

GitLab, GitHub, and BitBucket comparison

The comparison of repository management solution is done using the table below:

	GitHub	GitLab	BitBucket
Public Repos	Yes (Free)	Yes (Paid and Free)	Yes (Paid)
Private Repo	Yes (Paid)	Yes (Paid and Free)	Yes (Paid)
Hosted Service	Yes	Yes	Yes
On-premise	No	Yes CE (Free), EE(Paid)	Yes (Paid)

Git

Though all three repository management systems are different they have things in common, and one of them is Git. **Git** is a simple code versions tracker. It allows you to work with different repository management systems including GitHub, GitLab, and BitBucket. Git is available on all the most popular desktop and server OS including Windows, MacOS, and Linux. You can

download the **Git** client at <https://git-scm.com/downloads>. You should already have Git client installed on your PC.

GitLab installation

Before you start, we need to navigate to Github repo folder and run “**vagrant up gitlab**” command to start a single VM called gitlab:

```
$ vagrant up gitlab
Bringing machine 'gitlab' up with 'virtualbox' provider...
...
<output omitted>
...
gitlab: SSH username: vagrant
gitlab: SSH auth method: private key
```

Note! The “vagrant” directory and Vagrantfile need to be created in advance

Check if VM is running.

```
$ vagrant status gitlab
```

Current machine states:

```
gitlab           running (virtualbox)
```

The VM is running. To stop this VM, you can run `vagrant halt` to shut it down forcefully, or you can run `vagrant suspend` to simply suspend the virtual machine. In either case, to restart it again, simply run `vagrant up`.

Make sure that gitlab.example.com is accessible from your host machine. You will need to access GitLab service via Web Browser from your host machine.

```
$ ping gitlab.example.com -c 3
PING gitlab.example.com (172.24.0.11): 56 data bytes
64 bytes from 172.24.0.11: icmp_seq=0 ttl=64 time=0.358 ms
64 bytes from 172.24.0.11: icmp_seq=1 ttl=64 time=0.517 ms
64 bytes from 172.24.0.11: icmp_seq=2 ttl=64 time=0.368 ms

--- gitlab.example.com ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.358/0.414/0.517/0.073 ms
```

*Note! You need to configure your local hosts file and add custom hosts entries. Please refer “**Lab environment overview**”.*

Finally, ssh into VM running “vagrant ssh” command and switch to **root** user.

```
$ vagrant ssh gitlab
[vagrant@gitlab ~]$ sudo -i
[root@gitlab ~]#
```

Now we are ready to install our GitLab from scratch. Install gitlab repo to your VM by running the following command:

```
# curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.rpm.sh | bash

% Total    % Received % Xferd  Average Speed   Time   Time   Time  Current
                                         Dload  Upload Total Spent   Left  Speed
100  6463    0  6463    0    0  9635      0 --:--:--:--:--:--:--:--  9631
Detected operating system as centos/7.
...
<output omitted>
...
The repository is setup! You can now install packages.
```

Note! We will use “#” to indicate that command must be run under “root” account.

Install GitLab and Git packages.

```
# yum -y install git gitlab-ce
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
...
<output omitted>
...
Installed:
  gitlab-ce.x86_64 0:11.0.4-ce.0.el7
```

Start autoconfiguration of gitlab server running **gitlab-ctl reconfigure** command.

```
# gitlab-ctl reconfigure
Starting Chef Client, version 13.6.4
resolving cookbooks for run list: ["gitlab"]
...
<output omitted>
...
Chef Client finished, 427/611 resources updated in 03 minutes 38 seconds
gitlab Reconfigured!
```

Note: In case if you are running into “check LANG and LC_ environment variables”, just run the following commands:*

```
# export LC_ALL="en_US.UTF-8"  
# export LC_CTYPE="en_US.UTF-8"
```

By running “`gitlab-ctl reconfigure`” it takes settings from ruby file `/etc/gitlab/gitlab.rb` and automatically configures the server. For our book, default settings are enough. The installation process takes anywhere from 1 to 10 minutes.

Note! GitLab installer uses Chef configuration management tool. Each time you change GitLab configuration you need to execute “`gitlab-ctl reconfigure`” to apply configuration changes.

Verify that the gitlab server is up and running by running curl command

```
# curl localhost  
<html><body>You are being <a  
href="http://localhost/users/sign_in">redirected</a>.</body></html>
```

Exit from GitLab VM.

```
# exit  
$ exit
```

Connecting to GitLab

Open your browser and navigate to “<http://gitlab.example.com/>”. You should be able to see GitLab welcome page. By default, gitlab server is configured to respond to gitlab.example.com.

Please create a password for your new account.

GitLab Community Edition

Open source software to collaborate on code

Manage Git repositories with fine-grained access controls that keep your code secure. Perform code reviews and enhance collaboration with merge requests. Each project can also have an issue tracker and a wiki.

Change your password

New password

Confirm new password

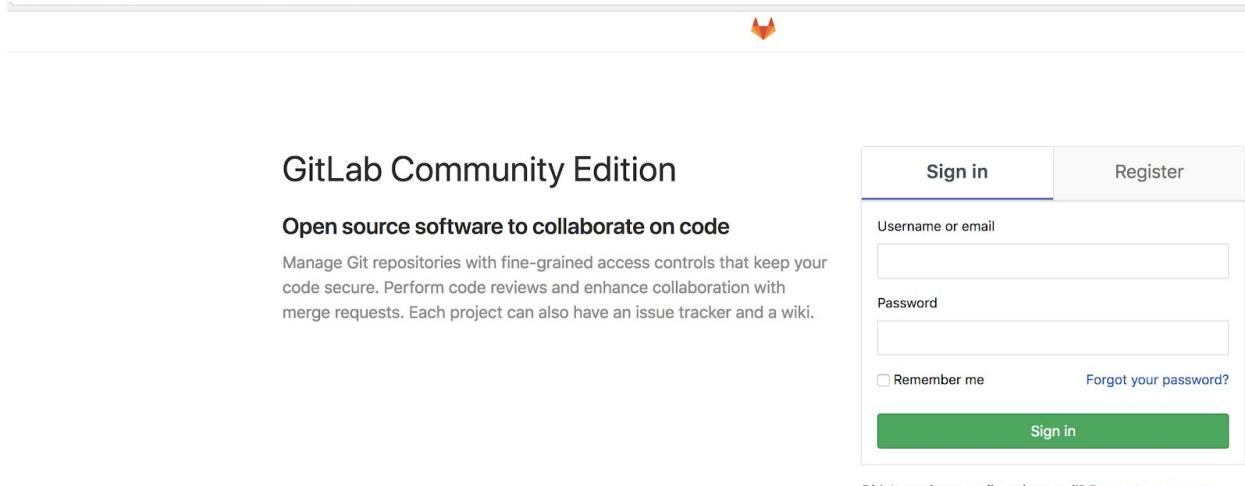
Change your password

Didn't receive a confirmation email? Request a new one

Already have login and password? Sign in

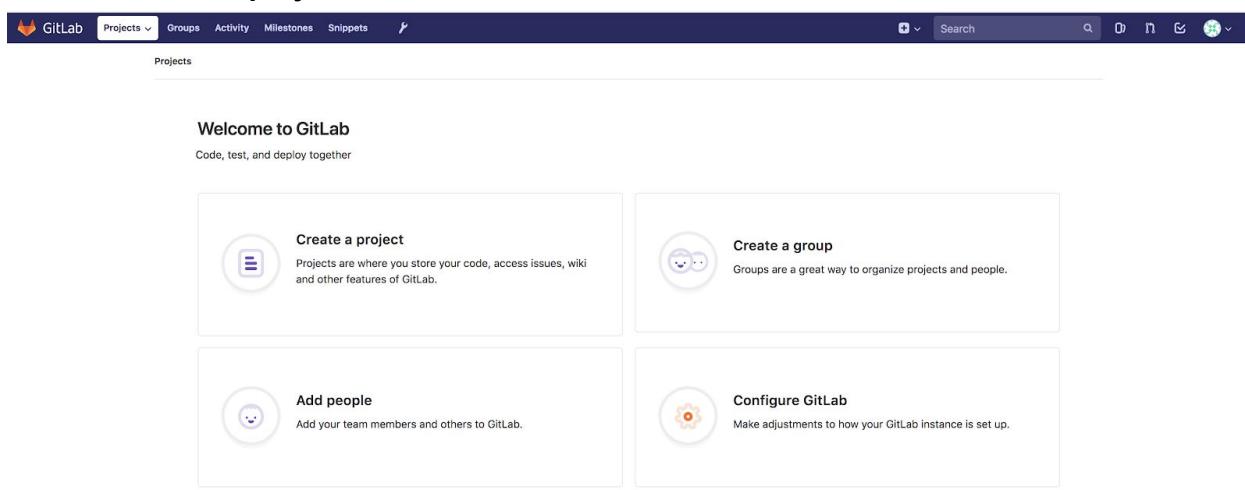
Note! It is a general recommendation and best practice to use DNS names over IP addresses. You can use IP addresses instead of DNS names, but there is no guarantee that everything is going to work as expected.

Set your new password under **Change your password** tab. We use “**DevOps123**” as a standard password everywhere. Again, no paranoia about security in this book, and more focus on DevOps. Once you change your password, it redirects you to a login page. Use “**root**” as username and the password you just set “**DevOps123**” in our case and then click on “Sign In”.



The image shows the GitLab Community Edition login page. At the top, there is a navigation bar with links for "Sign in" and "Register". Below the navigation bar, there are fields for "Username or email" and "Password". There is also a "Remember me" checkbox and a link for "Forgot your password?". A green "Sign in" button is located at the bottom of the form. At the very bottom of the page, there is a small note: "Didn't receive a confirmation email? Request a new one."

And we are logged in to our personal GitLab server. That was quite easy we hope. Now, what do we do with all this beauty? Right, let's create our first repository and upload some code. Click on “**Create a project**”.



The image shows the GitLab dashboard. At the top, there is a navigation bar with links for "Projects", "Groups", "Activity", "Milestones", "Snippets", and "Search". Below the navigation bar, there is a "Welcome to GitLab" section with the tagline "Code, test, and deploy together". This section features four cards: "Create a project" (with a brief description and a document icon), "Create a group" (with a brief description and a user icon), "Add people" (with a brief description and a user icon), and "Configure GitLab" (with a brief description and a gear icon).

Specify repo name, “**first_repo**” in our case, make it “**public**” and press “**Create project**” button.

Blank project	Create from template	Import project
<p>Project path</p> <div style="display: flex; align-items: center;"> <input style="width: 150px; margin-right: 10px;" type="text" value="http://gitlab.example.com/"/> root <div style="margin-left: auto;"> Create a group </div> </div> <p>Project name</p> <div style="display: flex; align-items: center;"> <input style="width: 200px; margin-right: 10px;" type="text" value="first_repo"/> G </div> <p>Want to house several dependent projects under the same namespace? Create a group</p> <p>Project description (optional)</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> The first GitLab Repo </div> <p>Visibility Level</p> <ul style="list-style-type: none"> <input type="radio"/> 🔒 Private Project access must be granted explicitly to each user. <input type="radio"/> 🛡️ Internal The project can be accessed by any logged in user. <input checked="" type="radio"/> 🌐 Public The project can be accessed without any authentication. <p><input type="checkbox"/> Initialize repository with a README Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.</p> <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div style="flex-grow: 1; text-align: right;"> Create project </div> <div style="text-align: right;"> Cancel </div> </div>		

Our first repo is created.

The screenshot shows the GitLab interface for the 'first_repo' project. The top navigation bar includes 'Dashboard', 'Projects', 'Groups', 'Activity', 'Milestones', 'Snippets', and a search bar. The left sidebar has sections for Project (Details, Activity, Cycle Analytics), Issues, Merge Requests, CI / CD, Operations, Wiki, Snippets, and Settings. The main content area displays a success message: 'Project 'first_repo' was successfully created.' It also shows a summary: 'You won't be able to pull or push project code via SSH until you add an SSH key to your profile' and 'The Auto DevOps pipeline has been enabled and will be used if no alternative CI configuration file is found. [More information](#)'. Below this, there's a summary card with 'Project ID: 1', a star icon (0 stars), and links for 'HTTP' and 'Global'. A note says 'The repository for this project is empty'. It also mentions 'Auto DevOps' and 'Protected branches'. At the bottom, there are buttons for 'New file', 'Add Readme', and 'Add Kubernetes cluster'. A 'Command line instructions' section contains a 'Git global setup' command:

```
git config --global user.name "Administrator"
git config --global user.email "admin@example.com"
```

We have two ways of working with our “**first_repo**” repository via HTTP and SSH.

The screenshot shows a GitLab repository named "first_repo". At the top, there is a purple circular icon with a white letter "F". Below it, the repository name "first_repo" is displayed with a small gear icon. Underneath the name, the text "The first GitLab Repo" and "Project ID: 1" are shown. A navigation bar at the top includes "Star" (0), "HTTP" (selected), "http://gitlab.example.com/roo", a copy icon, a plus icon, and "Global". A tooltip for "HTTP" explains: "You won't be able to pull or push project code via SSH until you add an SSH key to your profile". Another tooltip for "HTTP" says: "Create a personal access token on your account to pull or push via HTTP." Below the repository name, the text "The repository for this project" is visible, followed by some truncated text.

The difference between both is simple:

1. **HTTP** - In general, we can pull the information from the repository using HTTP transport protocol. Once we pull the files via HTTP and make changes, we can push the changes back using HTTP again.
2. **SSH** - this method uses SSH as the transport protocol. For some Linux-based environments, this is a more secure way of GIT communications.

We will use both methods when we get to Jenkins part. Try the **SSH** method first.

The screenshot shows the same "first_repo" repository page. The "SSH" button in the navigation bar is now selected, and the URL "git@gitlab.example.com:root/first_repo" is displayed. A "Copy URL to clipboard" button is highlighted with a black arrow. The rest of the interface is identical to the previous screenshot.

Login to gitlab VM:

```
$ vagrant ssh gitlab
[vagrant@gitlab ~]$ sudo -
[root@gitlab ~]#
```

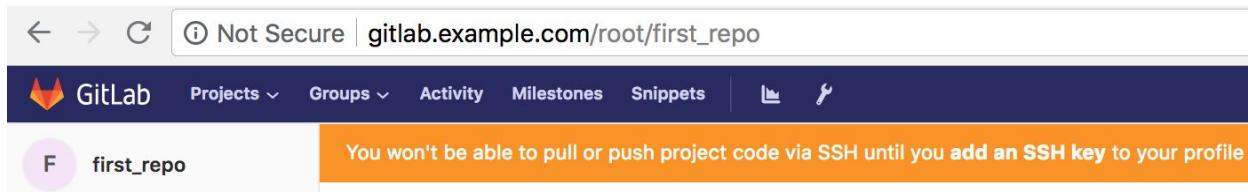
Try to clone our first repo to gitlab VM using "**git clone**" command using **SSH**.

```
[vagrant@gitlab ~]$ sudo -i
[root@gitlab ~]#
[root@gitlab ~]# git clone git@gitlab.example.com:root/first_repo.git
Cloning into 'first_repo'...
The authenticity of host 'gitlab.example.com (127.0.0.1)' can't be
established.
ECDSA key fingerprint is
SHA256:24E8qBAxWVZMWF+cIuc05LzGBKlUq+1N684IkF/wZns.
ECDSA key fingerprint is
MD5:6d:37:cb:12:41:10:6e:17:d3:45:ed:a2:df:b2:14:b5.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'gitlab.example.com' (ECDSA) to the list of
known hosts.
Permission denied (publickey,gssapi-keyex,gssapi-with-mic).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Note! The git command above was run under “**root**” user inside the gitlab host.

We got an error indicating that we do not have **SSH Keys** added to **GitLab** user profile. In addition to this, in your web browser, it gives you the same information.



Time to adjust GitLab configuration to make sure we can work with it. Use **ssh-keygen** command to generate SSH Keys.

```
[root@gitlab ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa): <ENTER>
Enter passphrase (empty for no passphrase): <ENTER>
Enter same passphrase again: <ENTER>
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:Fe4b+8a5y5wiZ/mummNxirQa/Wdq9Il2UKL8IBXPYmE root@gitlab.example.com
The key's randomart image is:
+---[RSA 2048]---+
```

```
| .E..  
| ..+=...  
| =o+=o  
| o.=++ o  
| oS=oO .  
| o O++  
| =o+ .  
| +.=++.  
| .o*.*Xo  
+---[SHA256]---
```

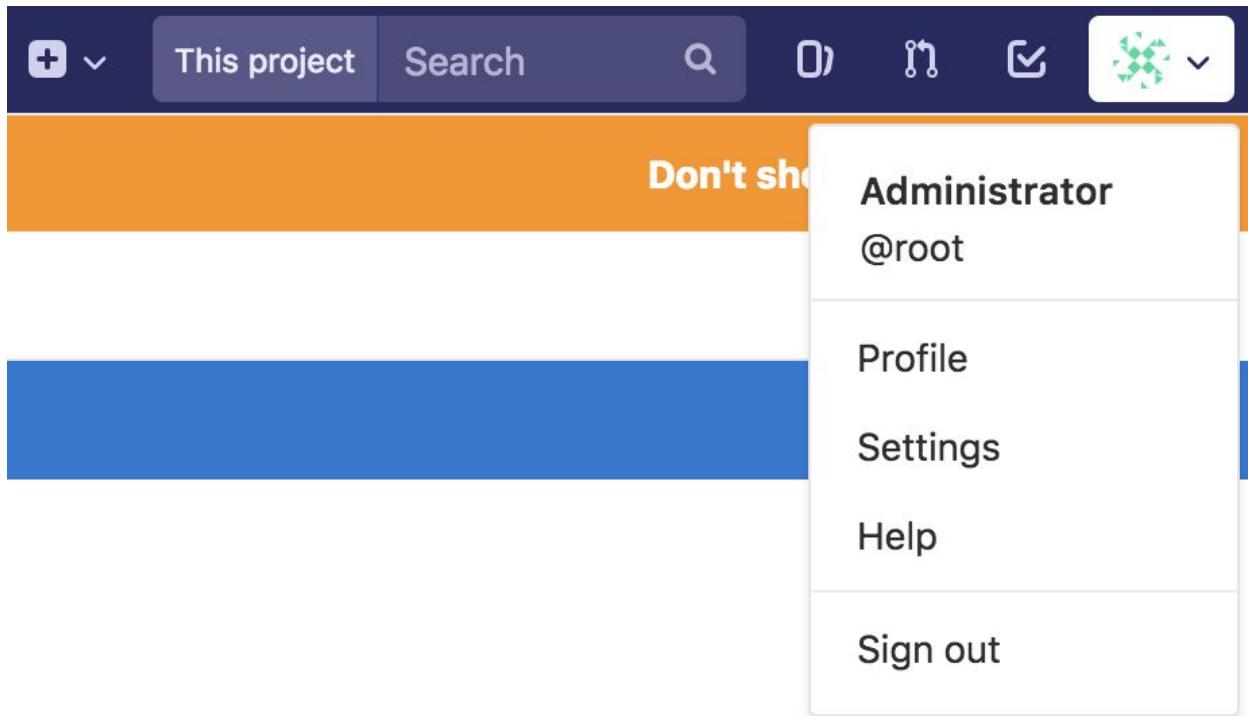
This generates a pair of public and private keys that are being saved in the hidden “**.ssh**” directory inside your user home folder.

```
[root@gitlab ~]# ls -l ~/.ssh/  
total 12  
-rw----- 1 root root 1675 Sep 16 19:30 id_rsa  
-rw-r--r-- 1 root root 405 Sep 16 19:30 id_rsa.pub  
-rw-r--r-- 1 root root 180 Sep 16 19:27 known_hosts
```

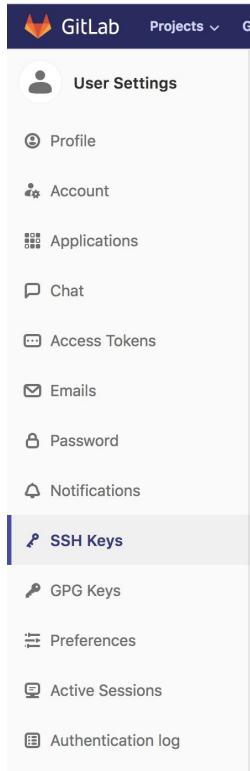
We have two files:

- **id_rsa** - It is your private key. This key is private, you should not show it to anyone.
- **id_rsa.pub** - it is a public key. Public keys are saved to send to remote parties. We need to upload the public key to GitLab.

In your Browser, navigate to GitLab Admin Settings by clicking on the icon at the top-right corner, then click => Settings.



On the left sidebar navigate to “SSH Keys”.



Finally, paste your public key from “`/root/.ssh/id_rsa.pub`” file to the “**Key**” text area and click on “**Add Key**”.

User Settings > SSH Keys

SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

To add an SSH key you need to generate one or use an [existing key](#).

Key

Paste your public SSH key, which is usually contained in the file ‘`~/.ssh/id_rsa.pub`’ and begins with ‘`ssh-rsa`’. Don’t use your private SSH key.

```
ssh-rsa  
AAAAB3NzaC1yc2EAAAQABAAQDKEcRAEaw8nzH8+IYrMfsDbCSFBQIVs+EZqlPnZ  
83PNfvvJsjgT3TLkxu9GQ/SbGTNbOQT0o4NfTtdG8wlCol5GrXX02PMYR/DnZ1gVaTqm0C5  
eTG67vfT2YafeVzLapt2RP0Oh3PpW//f3XDLx45W1c/sTt/loURxBg/4iulueO+i3LaAvLV3LU  
xPFMapbUIDYgTjPFk+rLegsUdigN7DPmvnLL0kBpCmwX8ca5FYiDwOMPkWrcf2ZnGqO+e  
5ijpCJOLz7V/ubV9PMbdenEz27Wfxv12sFi34mB0nUzLzKhu381ldlAwL9klkl+8OZbAeKnEb  
MQ26X5DbJZ4R root@gitlab.example.com
```



Title

Name your individual key via a title

Add key

You will see something like that:

User Settings > SSH Keys > `root@gitlab.example.com`

SSH Key
Title: <code>root@gitlab.example.com</code>
Created on: Sep 16, 2018 7:38pm
Last used on: N/A

Fingerprint: d2:3c:68:77:ee:69:f6:94:c8:01:24:3e:18:45:ed:19

```
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDKEcRAEaw8nzH8+IYrMfsDbCSFBQIVs+EZqlPnZ83
```

Remove

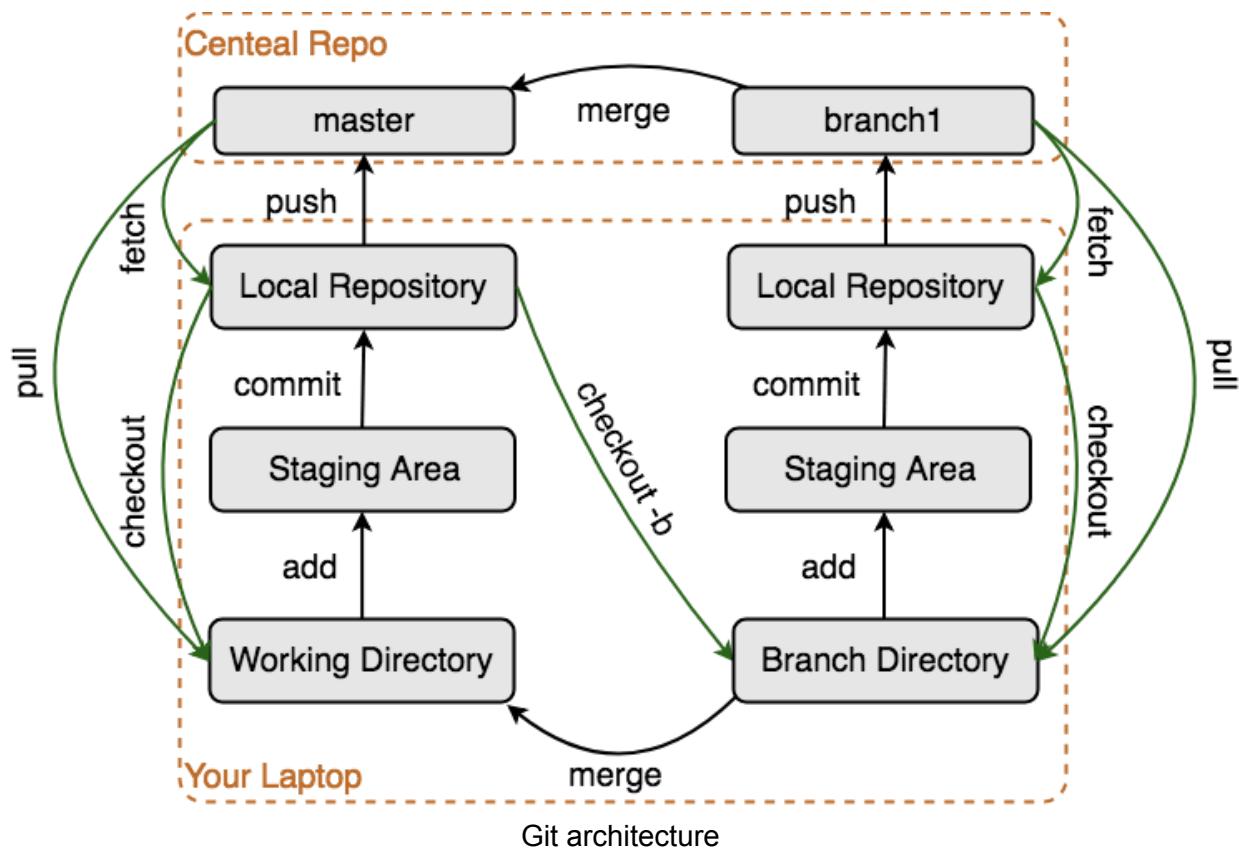
Under “root” user, clone the “`first_repo`” one more time and see how it works

```
[root@gitlab ~]# git clone git@gitlab.example.com:root/first_repo.git  
Cloning into 'first_repo'...  
warning: You appear to have cloned an empty repository.
```

Perfect, our first GitLab repo is cloned and we are ready to move forward

Git basics

Let's start with taking a look at Git Architecture and what we use in this book. It is okay that the diagram below does not make much sense to you, if are looking at it for the first time. Do not worry, you will be able to understand everything as we go through this book. When confused, refer to this diagram, it will help you to understand what we are doing in the following labs.



Git consists of 4 main components:

- **Central Repository** - it is a metadata and object database shared with other users. Central repository examples are Github, Bitbucket, GitLab, etc.
- **Local Repository** - complete clone of your central repository that is being copied to your local PC.
- **Staging Area** - is an abstraction layer where, where Git starts tracking the changes.
- **Working directory** - is the directory where you make all the modifications on your local PC. Changes are not under version control and untracked.

The most common operations which performed while working with files under Git control are:

- add, delete, update files to the staging area
- commit files to the local repository

- Pushing, fetching and pulling changes while working with the central repository
- Rolling back the changes

Using Git help

At any point, do not be shy to use git help if you do not understand git command, subcommand or key option.

```
[root@gitlab ~]# git --help
usage: git [--version] [--help] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

The most commonly used git commands are:
add          Add file contents to the index
bisect       Find by binary search the change that introduced a bug
branch       List, create, or delete branches
checkout     Checkout a branch or paths to the working tree
clone        Clone a repository into a new directory
commit       Record changes to the repository
...
output truncated for brevity
...
'git help -a' and 'git help -g' lists available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.

[root@gitlab ~]# git commit -h
usage: git commit [options] [--] <pathspec>...
      -q, --quiet            suppress summary after successful commit
      -v, --verbose           show diff in commit message template

Commit message options
      -F, --file <file>      read message from file
...
output truncated for brevity
...

Commit contents options
      -a, --all               commit all changed files
      -i, --include           add specified files to index for commit
```

```
...
output truncated for brevity
...
```

Working with Git

When you begin a new project it is a good idea to create a git repository first and create new files already in. But if you already have a project, it is fine just move or copy the existed files in the git repository. Let's create a new file README.md and some text there.

Let's switch to **first_repo** directory before we start working with Git.

Navigate to "first_repo" directory

```
[root@gitlab ~]# cd first_repo/
```

And create README file

```
[root@gitlab first_repo]# echo This is our first project > README.md
```

*Note: md file extension is a special extension for text files written in Markdown language. You can learn more about **markdown** formatting at*

<https://guides.github.com/pdfs/markdown-cheatsheet-online.pdf>.

Check the status of **README.md** file

```
[root@gitlab first_repo]# git status .
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README.md
nothing added to commit but untracked files present (use "git add" to track)
```

You can see that **README.md** is under **Untracked files**. It basically means that we this file is NOT under git version control.

Add **README.md** file for tracking by git.

```
[root@gitlab first_repo]# git add README.md
```

Or

```
#adds all the new or changed files in this repo  
[root@gitlab first_repo]# git add .
```

Check the status of your git repo

```
[root@gitlab first_repo]# git status  
  
# On branch master  
#  
# Initial commit  
#  
# Changes to be committed:  
#   (use "git rm --cached <file>..." to unstage)  
#  
#       new file: README.md  
#
```

Now **README.md** appears under “Changes to be committed”. The “**git add**” command just updates the repository index and prepares the content for next commit. At this point **README.md** file is not saved to the repository on your PC yet.

Let's commit **README.md** to the repo

```
[root@gitlab first_repo]# git commit -m 'First Commit'  
[master (root-commit) 5d62cb9] First Commit  
 1 file changed, 1 insertion(+)  
 create mode 100644 README.md
```

Note! You need to configure git username/email

Check repo status

```
[root@gitlab first_repo]# git status  
# On branch master  
nothing to commit, working directory clean
```

Git status command tells us that there is nothing to commit because we just pushed **README.md** to the local repository, it is still on our PC and not being pushed to the central repository yet, **gitlab.example.com** in our case. Open your browser and navigate to http://gitlab.example.com/root/first_repo. You won't see any files listed in there.

The screenshot shows a GitLab repository named 'first_repo'. The page title is 'first_repo' with a 'fork' icon. Below it, the text 'The first GitLab Repo' and 'Project ID: 1' are displayed. A navigation bar includes 'Star' (0), 'SSH', 'git@gitlab.example.com:root/first_repo', a copy icon, a plus icon, and 'Global' dropdown.

The repository for this project is empty

If you already have files you can push them using the [command line instructions](#) below.

Note that the master branch is automatically protected. [Learn more about protected branches](#)

You can automatically build and test your application if you [enable Auto DevOps](#) for this project. You can automatically deploy it as well, if you [add a Kubernetes cluster](#).

Otherwise it is recommended you start with one of the options below.

[New file](#) [Add Readme](#) [Add License](#) [Enable Auto DevOps](#) [Add Kubernetes cluster](#)

Command line instructions

Git global setup

```
git config --global user.name "Administrator"  
git config --global user.email "admin@example.com"
```

Use **git log** command to verify the changes in our local repository.

```
[root@gitlab first_repo]# git log  
commit bc58a6532c4139b6beb8c0110758dc356c96d54f  
Author: root <root@gitlab.example.com>  
Date: Sun Sep 16 19:47:55 2018 +0000  
  
First Commit
```

That is absolutely normal, and in order to push the **README.md** file to our GitLab server you need to run '**git push**' command.

```
[root@gitlab first_repo]# git push  
Counting objects: 3, done.  
Writing objects: 100% (3/3), 238 bytes | 0 bytes/s, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To git@gitlab.example.com:root/first_repo.git  
 * [new branch]      master -> master
```

Open your browser and navigate to “http://gitlab.example.com/root/first_repo” again. README.md file should be there.

The screenshot shows a GitLab repository page for 'first_repo'. At the top, there's a purple circle with a white letter 'F'. Below it, the repository name 'first_repo' is displayed with a small '0' icon next to it. The description 'The first GitLab Repo' and 'Project ID: 1' are shown. A navigation bar includes 'Star' (0), 'Fork' (0), 'SSH' (git@gitlab.example.com:root/first_repo), and other project management buttons like 'Add Changelog', 'Add License', 'Add Contribution guide', 'Add Kubernetes cluster', and 'Set up CI/CD'. A 'Auto DevOps' section is present with a cloud icon, a brief description, and a link to documentation. Below this, the commit history shows a single commit from 'root' authored 8 minutes ago, with a commit message 'First Commit' and a hex SHA bc58a653. The file list shows a single file, 'README.md', which contains the text 'This is our first project'. Navigation buttons for 'History', 'Find file', 'Web IDE', and a dropdown are also visible.

Perfect, we just made our first push to gitlab.example.com server. But there is still a lot more things to learn.

Let's edit README.md and add a new line

```
[root@gitlab first_repo]# echo README.md File is modified >> README.md
[root@gitlab first_repo]# cat README.md
This is our first project
README.md File is modified
```

Check repo status

```
[root@gitlab first_repo]# git status
# On branch master
```

```
# Changes not staged for commit:  
#   (use "git add <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working  
directory)  
#  
#       modified:   README.md  
#  
no changes added to commit (use "git add" and/or "git commit -a")
```

You see that Git identified the changes in README.md, but these changes are not under version control.

To apply the changes in the repo we need to run “git add” and “git commit” again.

```
[root@gitlab first_repo]# git add README.md  
[root@gitlab first_repo]# git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       modified:   README.md  
#  
[root@gitlab first_repo]# git commit -m 'Second Commit'  
[master ee46d78] Second Commit  
 1 file changed, 1 insertion(+)
```

Let's check what git know about our modifications

```
[root@gitlab first_repo]# git log  
commit ee46d786f849a9f3aad287ed469f26d4b72dc07  
Author: Root User <root@example.com>  
Date:   Sun Sep 16 19:58:42 2018 +0000  
  
        Second Commit  
  
commit bc58a6532c4139b6beb8c0110758dc356c96d54f  
Author: root <root@gitlab.example.com>  
Date:   Sun Sep 16 19:47:55 2018 +0000  
  
        First Commit
```

Here you can see that git knows about the changes made to **README.md**. In our case, it says that there were two commits with comments “First commit” and “README.md modified”. You can also see who has done these changes and what time.

You can also check the list of files stored in your local repo with the following command

```
[root@gitlab first_repo]# git ls-files -s  
100644 4acbdbe4e8d334035c9bb94cdf7a4c2ce945e7fb 0 README.md
```

Just to note that README.md file now stored in 3 different places:

1. Working directory (local directory)
2. In local git repo (local repository)
3. On GitLab (upstream repository)

Now let's remove README.md from **Working directory**.

```
[root@gitlab first_repo]# rm -f README.md  
[root@gitlab first_repo]# ls -l  
total 0
```

README.md has been deleted from the Working directory and staging area (**do not forget to refer to “Git Architecture” diagram if confused**), but it is still in your local git repo.

Now check git status to see the changes

```
[root@gitlab first_repo]# git status  
# On branch master  
# Your branch is ahead of 'origin/master' by 1 commit.  
#   (use "git push" to publish your local commits)  
#  
# Changes not staged for commit:  
#   (use "git add/rm <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working  
directory)  
#  
#       deleted:    README.md  
#  
no changes added to commit (use "git add" and/or "git commit -a")
```

It says that the status of README.md is **deleted**. But it also says that to revert the changes we can use “git checkout -- <file>” command.

Run “git checkout” command to recover the file from **local repository**:

```
[root@gitlab first_repo]# git checkout -- README.md  
[root@gitlab first_repo]# ls -l  
total 4  
-rw-r--r--. 1 root root 53 Sep 16 20:02 README.md  
[root@gitlab first_repo]# cat README.md
```

```
This is our first project  
README.me File is modified
```

As you can see **README.md** with all its content is back. Isn't it cool?

Sharing git projects

At this point, we used a single account in GitLab. In reality, you work in the group of people managing one or several git projects. In order to simulate this behavior, we need to create one more user and add him to our repository.

The easiest way would be to open another browser and navigate to http://gitlab.example.com/users/sign_in and click on Register. E.g. if you are currently using Chrome, then open Firefox and register a new user filling out all required fields. Once you are done, click **Register** button.

The screenshot shows the GitLab Community Edition registration page. At the top, there is a header with navigation icons (back, forward, search, etc.) and the URL gitlab.example.com/users/sign_in. Below the header, the page title is "GitLab Community Edition" and the sub-headline is "Open source software to collaborate on code". A brief description follows: "Manage Git repositories with fine-grained access controls that keep your code secure. Perform code reviews and enhance collaboration with merge requests. Each project can also have an issue tracker and a wiki." On the right side, there is a registration form. The "Register" tab is selected. The form fields are: "Full name" (value: "user"), "Username" (value: "user", status: "Username is available."), "Email" (value: "user@example.com"), "Email confirmation" (value: "user@example.com"), "Password" (value: "*****", note: "Minimum length is 8 characters"). At the bottom of the form is a blue "Register" button. Below the form, a small note says: "Didn't receive a confirmation email? Request a new one."

Create a new username: **user** with password: **DevOps123**. Click on **Register** and It should create a new user, log you in and redirect to the main dashboard.

The screenshot shows the GitLab dashboard at `gitlab.example.com/dashboard/projects`. At the top, there's a navigation bar with links for Projects, Groups, Activity, Milestones, and Snippets. A search bar is also present. Below the navigation, a blue banner displays the message "Welcome! You have signed up successfully." The main content area is titled "Welcome to GitLab" with the subtitle "Code, test, and deploy together". It features four cards: "Create a project" (with a document icon), "Create a group" (with a user icon), "Explore public projects" (with a person icon), and "Learn more about GitLab" (with a lightbulb icon). Each card has a brief description below its title.

Quite similar to what we have seen when we logged in as `root` user. Since our `first_repo` repository is public, this new user will be able to have read-only access to it.

Navigate to “**Explore your public projects**” -> **Explore projects** -> **All**. You should be able to see `first_repo` we have recently created.

The screenshot shows the GitLab homepage with the URL "gitlab.example.com/explore/projects". The top navigation bar includes links for "Projects", "Groups", "Activity", "Milestones", and "Snippets". Below the navigation is a section titled "Projects" with three tabs: "Your projects", "Starred projects", and "Explore projects" (which is underlined). Further down are filters for "Trending", "Most stars", and "All". A specific project card for "Administrator / first_repo" is displayed, featuring a profile picture with a letter 'F', the repository name, and a description "The first GitLab Repo".

Click on **Administrator/first_repo** to navigate to repo project.

The screenshot shows the "first_repo" project details page. The top header includes a message about SSH key requirements. The main content area shows the project's name, description, and various statistics like star and fork counts. Below this is a file list with one item: "First Commit" (root authored 26 minutes ago). At the bottom, there's a table showing file details: "README.md" with its last commit being "First Commit" at 26 minutes ago.

The same message at the top of the screen telling us that we won't be able to pull or push repo code until we add SSH keys.

Go back to CLI and ssh into gitlab VM using "**vagrant ssh gitlab**" if you happen to logout. Make sure that you are in the directory where you have Vagrantfile for our project.

We need to generate SSH keys inside gitlab VM and upload them to GitLab “**user**” profile.

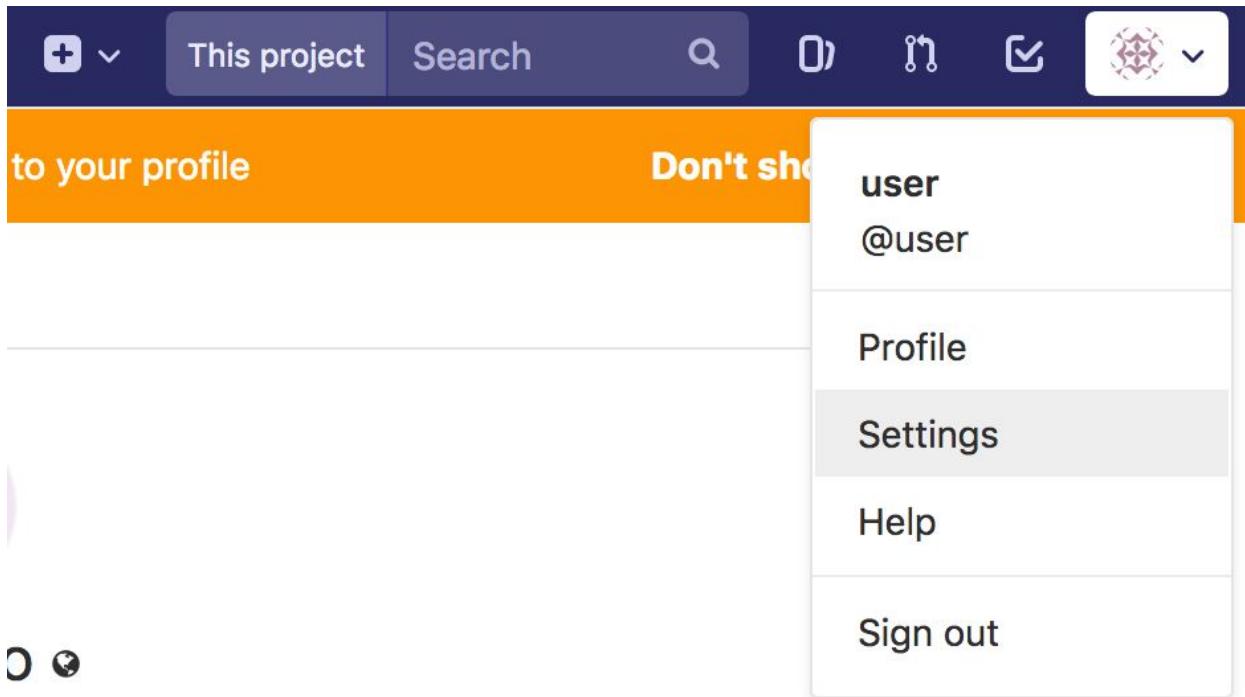
```
[vagrant@gitlab ~]$ sudo useradd user
[vagrant@gitlab ~]$ sudo su - user

[user@gitlab ~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa): <ENTER>
Created directory '/home/user/.ssh'.
Enter passphrase (empty for no passphrase): <ENTER>
Enter same passphrase again: <ENTER>
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.

<OUTPUT OMITTED>
...

[user@gitlab ~]$ cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQC300D9No1DN1L2P3e+JS6zDRwoVMQGNBgnB2P72H9sRaW
wDAvXVhtsH5nbtSxfJJ1p8/MNBvx+j73kmbCCAyfoB9bcI9ox40uJf8j40NZgBYYXVz32FFGHhI
6VUS0whP/FKeox0f/JE1I1RRZ4XA7f4gYee0FCpWNL5pNevyramo/N23JAT2Ko0VOXgCuHOLnIK
cGQbodcB00vfz1/sFJxFrQEGNUHJ9Dydzaa3/uY3H5kr/coFSRPYLTQdISNks8H78R0rYiE7/e/
KKF2M8N3H0hMabxGQ5ubmLyK/cIhRXZx0p38YB9ePs9he1v1B07YFvULi19F1erPUT52JfIR
user@gitlab.example.com
```

Navigate to **User Settings => SSH Keys**



Copy and paste private keys to **Key** text-area, and then press **Add key**

User Settings > SSH Keys

SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

To add an SSH key you need to generate one or use an [existing key](#).

Key

Paste your public SSH key, which is usually contained in the file '~/.ssh/id_rsa.pub' and begins with 'ssh-rsa'. Don't use your private SSH key.

```
ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQC3O0D9N0lDNl2P3e+JS6zDRwoVMQGNBgnB2
P72H9sRaWwDAvXVhtsH5nbSxfJJ1p8/MNBvx+j73kmbCCAYfoB9bcI9ox4OuJf8j4ONZgBY
YXVz32FFGHhI6VUSoWhP/FKeox0f/JEl1RRZ4XA7f4gYeo0FCpWNL5pNevyramo/N23JAT2
KoVOXgCuHOLnIKcGQbodcB00vfz1/sFJxFrQEGNUHJ9Dydzaa3/uY3H5kr/coFSRPYLTQdl
SNKS8H78ROrYiE7/e/KKF2M8N3H0hMabxGQ5ubmLyK/clhRXZx0p38YB9ePs9he1v1B07YF
vULi19F1erPUt52JfIR user@gitlab.example.com
```

Title

Name your individual key via a title

Add key

Once Keys are added, Navigate go to "http://gitlab.example.com/root/first_repo" and copy repo URL to the clipboard using **SSH** method.

F

first_repo

The first GitLab Repo

Project ID: 1



Go back to vagrant VM CLI and clone this repo.

```
[user@gitlab ~]$ git clone git@gitlab.example.com:root/first_repo.git
Cloning into 'first_repo'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

Configure your git with basic settings:

```
[user@gitlab ~]$ git config --global user.name "User"
[user@gitlab ~]$ git config --global user.email user@example.com
```

Then change directory to first_repo and create a new file

```
[user@gitlab ~]$ cd first_repo/
[user@gitlab first_repo]$ touch user.txt
[user@gitlab first_repo]$ ls -l
total 4
-rw-rw-r--. 1 user user 26 Sep 16 20:11 README.md
-rw-rw-r--. 1 user user 0 Sep 16 20:13 user.txt
```

The new file is in there, let's try to submit this new file to GitLab repo.

```
[user@gitlab first_repo]$ git add .; git commit -m 'creating user.txt file'; git push
...
<OUTPUT OMITTED>
...
```

GitLab: You are not allowed to push code to this project.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

Since **first_repo** was created by another user, by default all other users have read-only access to this repository. Let's go back to our repository under **root** user.

Login as root user to <http://gitlab.example.com/> and navigate to **Projects => Your projects => first_repo** or simply paste “http://gitlab.example.com/root/first_repo” in your browser. And from there to go **Settings => Members**.

The screenshot shows the 'new_repo' project page on GitLab. The left sidebar has sections for Project, Repository, Issues (0), Merge Requests (0), CI / CD, Operations, Wiki, and Snippets. The 'Settings' section is expanded, showing options for General, Members (which is selected and highlighted in grey), Badges, Integrations, Repository, and CI / CD.

In “Select members to Invite” box, type “user” and press **Enter**.

In “Choose a role permission” box choose “**Maintainer**” and finally press **Add to project**.

Project members

You can add a new member to **new_repo** or share it with another group.

Add member

Select members to invite

user x

Choose a role permission

Maintainer

[Read more](#) about role permissions

Access expiration date

Expiration date

Add to project Import

Perfect, now your user should appear as a member of **first_repo**.

Existing members and groups	
Members of first_repo 2	<input type="text" value="Find existing members by name"/> Name, ascending
 Administrator @root It's you Given access 1 hour ago	Maintainer
 user @user Given access 7 minutes ago	Maintainer Expiration date Delete

Go back to vagrant VM cli and try to push the changes one more time.

```
[user@gitlab first_repo]$ git push
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 278 bytes | 0 bytes/s, done.
```

```
Total 3 (delta 0), reused 0 (delta 0)
To git@gitlab.example.com:root/first_repo.git
  bc58a65..88ca656  master -> master
```

We can see that some data was pushed to first_repo at gitlab.example.com. Navigate back to "http://gitlab.example.com/root/first_repo" and you should be able to see **user.txt** file to appear over there.

The screenshot shows a GitLab repository interface. At the top, there's a dropdown menu set to 'master' and a breadcrumb navigation bar showing 'first_repo /'. To the right are buttons for 'History', 'Find file', 'Web IDE', and a search icon. Below this is a commit card for 'creating user.txt file' by 'user' 9 minutes ago, with a commit hash '88ca656' and a copy icon. A table lists files with their last commit details:

Name	Last commit	Last update
README.md	First Commit	36 minutes ago
user.txt	creating user.txt file	9 minutes ago

Below the table is a preview of the README.md file content: "This is our first project".

Now you should be able to pull these changes to under **root** user running “**git pull**” command. Exit out from vagrant VM, ssh into GitLab server and pull changes from GitLab Server VM.

```
[vagrant@gitlab ~]$ sudo -i
[root@gitlab ~]# cd first_repo/
[root@gitlab first_repo]# git pull
...
<OUTPUT OMITTED>
...
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 user.txt
```

Note: If you are redirected to an editor asking to enter a reason for merge request. Just type :wq to exit the vi editor. Vi is an editor by default on all Linux distros and MacOS. And if you do not have any experience with vi editor, it is going to be hard to deal with. Some people even reboot their servers in order to get out of vi. This is how tough vi is. Use the power of Google if you got lost at any point in time.

Check that you have both files pulled from **first_repo** repository.

```
[root@gitlab first_repo]# ls -l
```

```

total 4
-rw-r--r--. 1 root root 53 Sep 16 20:02 README.md
-rw-r--r--. 1 root root 0 Sep 16 20:19 user.txt

```

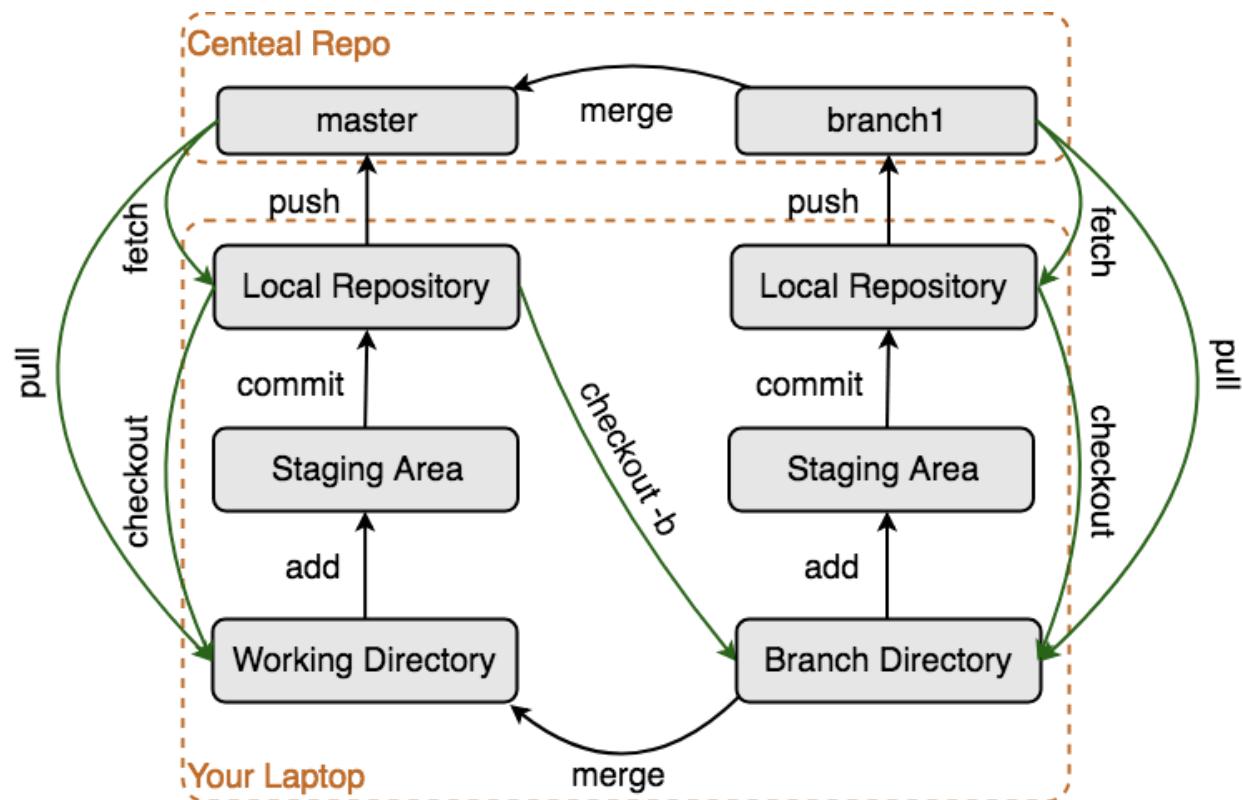
This what allows you to work hand-by-hand with your team members and share your work and eventually work more efficiently. Whether you have a simple bash script, complex ansible playbook or whole application is written in Python, you can share it with your colleagues. Or even better, revise and update these script to do a better job.

We will work more with GitLab and you will see how it becomes one of your main DevOps tools.

Working with merge requests

We have seen how two people can collaborate together. In larger teams where we have software development aspect, and this approach is not going to work well, because we will be required to do such thing as code review, sanity checks, take care of repository structure and other related tasks. And if we have, let's say, 5-10 members in the **first_repo**, actively submitting different files to the repo, it is going to be a total mess in a month timeframe. This is where we need to introduce you to a concept of git branches and merge requests.

Let's take a look at our git architecture diagram one more time and learn some new components in there.



Git architecture

So far we have been working mostly in the working directory and everyone was happy. But imagine that **root** developed a script that does something cool and uploaded it to **first_repo**.

```
[root@gitlab first_repo]# echo "echo $USER" > cool_script.sh
[root@gitlab first_repo]# bash cool_script.sh
root
[root@gitlab first_repo]# git pull; git add .; git commit -m "Checkout my
awesome script"; git push
...
<OUTPUT OMITTED>
...
To git@gitlab.example.com:root/first_repo.git
  88ca656..59fe035  master -> master
```

Now imagine that vagrant VM **user** has taken a look at that script and thought that he can make it better. So he pulled it from GitLab repo and wants to change that script.

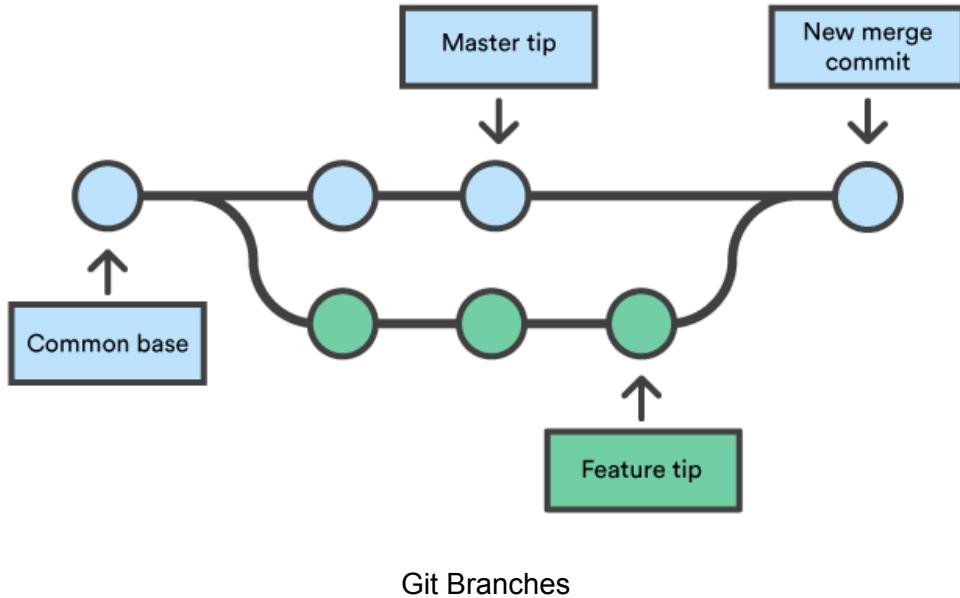
```
[root@gitlab first_repo]# su - user
Last login: Sun Sep 16 20:08:12 UTC 2018 on pts/0
[user@gitlab ~]$ cd first_repo/
[user@gitlab first_repo]$ git pull
...
<OUTPUT OMITTED>
...
create mode 100644 cool_script.sh
```

Note! We work under “user” account on the GitLab server

So how would you improve coll_script.sh?

1. Make a copy of cool_script.sh and name it cool_script_v2.sh and push it to gitlab.
2. Make the changes inside the script by commenting out the old lines and writing new ones inline.
3. Just rewrite the script and push it to the repo.

With Git, you have a better option by using branches.



The way branches work is you switch to the exact clone of your repository and start making changes in there. Once you are done you submit the changes back to the GitLab server in a different branch and create a merge request. And whoever responsible for the repo is going to review the code changes and going to approve, decline or comment on the changes you made. Let's see how it works in our case.

First, we need to switch to a new branch, let's call it `cool_script_changes`. Verify that there is no other branch with such name first

```
[user@gitlab first_repo]$ git branch
* master
```

Now create a new branch and switch to it using **git branch** and **git checkout** subcommands.

```
[user@gitlab first_repo]$ git branch cool_script_changes
[user@gitlab first_repo]$ git branch
  cool_script_changes
* master
```

The asterisk (*) specifies the currently checked out branch of your working directory. Now let's switch to `cool_script_changes` branch.

```
[user@gitlab first_repo]$ git checkout cool_script_changes
Switched to branch 'cool_script_changes'

[user@gitlab first_repo]$ git branch
* cool_script_changes
  master
```

First, take a look at the content of `cool_script.sh` and verify that it has not been changed since we have switched to the new branch.

```
[user@gitlab first_repo]$ cat cool_script.sh
echo root
```

Perfect, now let's make some changes in that `cool_script.sh`.

```
[user@gitlab first_repo]$ echo 'echo $USER is awesome' > cool_script.sh
[user@gitlab first_repo]$ bash cool_script.sh
user is awesome
```

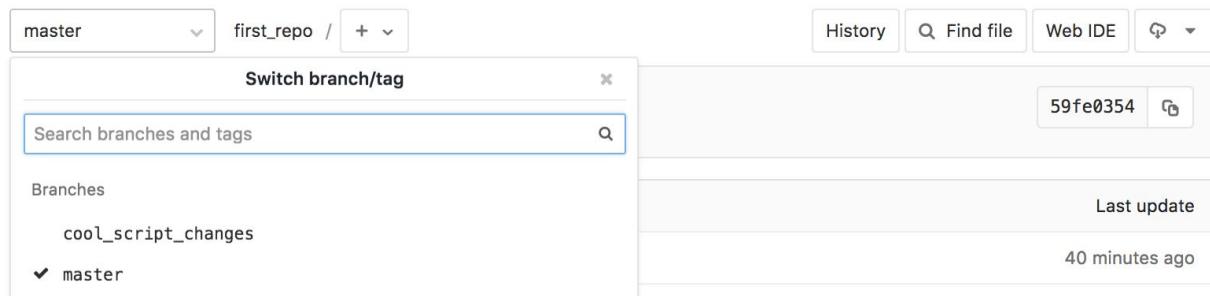
Now we are ready to add, commit and push the changes back to gitlab in `cool_script_changes` branch.

```
[user@gitlab first_repo]$ git add .
[user@gitlab first_repo]$ git commit -m 'I have modified your script and it
looks much better now'
[cool_script_changes 78031cd] I have modified your script and it looks much
better now
 1 file changed, 1 insertion(+), 1 deletion(-)

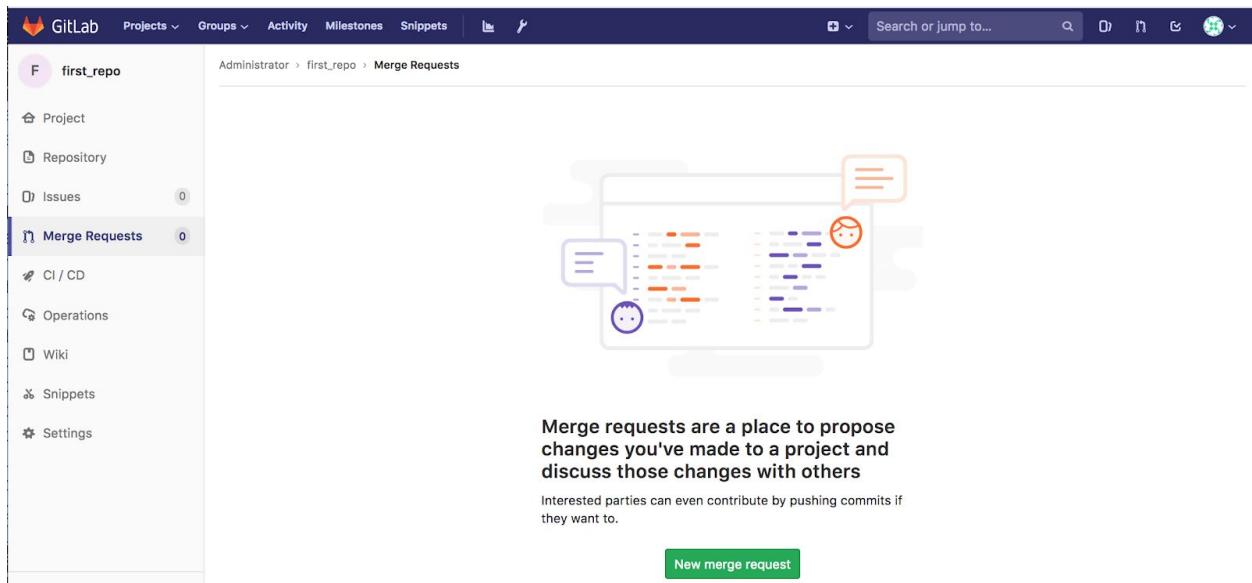
[user@gitlab first_repo]$ git push origin cool_script_changes
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 367 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: To create a merge request for cool_script_changes, visit:
remote:
http://gitlab.example.com/root/first_repo/merge_requests/new?merge_request%
5Bsource_branch%5D=cool_script_changes
remote:
To git@gitlab.example.com:root/first_repo.git
```

```
* [new branch]      cool_script_changes -> cool_script_changes
```

Note that we provided a special keyword **origin** following by the name of the new branch. If we open http://gitlab.example.com/root/first_repo in our browsers where we were logged under “**user**”, we should see that there is now a new branch called **cool_script_changes**.



It means that the changes in the GitLab and we can create a merge request. Click on **Merge requests** on the sidebar and then **New merge request**.



Select **first_repo** and **cool_script_changes** in source branch block and **first_repo** and **master** as Target branch. Finally, click on “**Compare branches and continue**”

New Merge Request

The screenshot shows the 'New Merge Request' interface. On the left, under 'Source branch', it lists 'root/first_repo' and 'cool_script_changes'. Below this, a message from a user named 'root' says: 'I have modified your script and it looks much better now' (commit 78031cd6, authored 8 minutes ago). On the right, under 'Target branch', it lists 'root/first_repo' and 'master'. Below this, another message from 'root' says: 'Hey y'all, look at what an awesome script I have just written' (commit 59fe0354, authored 15 minutes ago). At the bottom, there is a green button labeled 'Compare branches and continue'.

Now let's make a quick write up for **root** user and explain what we did with the script. You may click on “**Remove source branch when merge request is accepted**” button, so when the changes are accepted by root, the branch is automatically deleted. This is very convenient, so you do not need to delete that branch yourself.

The screenshot shows the 'New Merge Request' form. The sidebar on the left lists project navigation items: Project, Repository, Issues (0), Merge Requests (0), CI / CD, Operations, Wiki, Snippets, and Settings. The main form area has a title 'I have modified your script and it looks much better now' and a detailed description starting with 'Hi root' and explaining changes. It includes a rich text editor toolbar. Below the description, there are fields for 'Assignee' (set to 'Assignee'), 'Milestone' (set to 'Milestone'), and 'Labels' (set to 'Labels'). Under 'Source branch' is 'cool_script_changes' and under 'Target branch' is 'master'. There are two checkboxes: one checked for 'Remove source branch when merge request is accepted' and one unchecked for 'Squash commits when merge request is accepted'. At the bottom are 'Submit merge request' and 'Cancel' buttons.

By doing this, we are basically asking **first_repo** owner to review the changes we made in the script and give us some feedback. So If we login as **root** and go to http://gitlab.example.com/root/first_repo we will be able to see that there is one new Merge request on the sidebar.

The screenshot shows the GitLab interface for a new repository named "new_repo". The sidebar on the left includes sections for Project, Details, Activity, Cycle Analytics, Repository, Issues (0), and Merge Requests (1). The Merge Requests section is highlighted.

Click on “I have modified your script and it looks much better now”.

The screenshot shows the GitLab interface for a repository named "first_repo". The Merge Requests page displays one open merge request. The title of the merge request is "I have modified your script and it looks much better now". The merge request was opened 5 minutes ago by a user and was last updated 5 minutes ago. The status bar indicates 0 comments and 0 reviews.

From there you can see all the comments and all the changes that we made by clicking on **Changes** tab next to **Discussion** and **Commits**. That will show you what exactly was changed.

The screenshot shows a GitLab interface for a project named 'first_repo'. On the left, a sidebar lists various project sections: Project, Repository, Issues (0), Merge Requests (1), CI / CD, Operations, Wiki, Snippets, and Settings. The 'Merge Requests' section is currently active, showing one open merge request. The main content area displays the merge request details:

- Administrator > first_repo > Merge Requests > !1**
- Open** | Opened 2 hours ago by Administrator
- Edit** | **Close merge request**
- I have modified your script and it looks much better now**
- Hi root I have modified your script and it looks much better now**
- Request to merge cool_script_changes into master** | **Open in Web IDE** | **Check out branch** | **P**
- Pipeline #4 pending for 747729e3 on cool_script_changes** | **P**
- Merge when pipeline succeeds** (radio button selected) | **Yc** | **Remove source branch** | **Modify commit message**
- Yc** | **Merge when pipeline succeeds manually using the command line** | **Merge immediately**
- 0** | **0** | **0**

*Note: pipeline currently in the **pending** state. It's going to change once we reach the integration part with Jenkins.*

If you think that there is something wrong with this new version of the script than you can click on **Discussion** and make a comment stating your concerns. You can also confirm the changes in the comment and finally click on **Merge Immediately** button.

The screenshot shows the GitLab interface for a project named 'first_repo'. The left sidebar has a 'Merge Requests' section with 0 items. The main area shows a merged merge request from 'Administrator' titled 'Request to merge cool_script_changes into master'. It includes a message: 'Hi root I have modified your script and it looks much better now'. Below the message are two pipeline status cards: 'Pipeline #4 pending for 747729e3 on cool_script_changes' and 'Pipeline #5 pending for 288ff21a on master'. The message was merged by Administrator 1 hour ago, and the source branch was removed.

Now if you navigate back to "http://gitlab.example.com/root/first_repo", choose **master** branch and check the content of **cool_script.sh** you should be able to see that the content was changed.

The screenshot shows the repository view for the 'first_repo' project. The left sidebar has a 'Repository' section with 'Files' selected. In the main area, the 'cool_script.sh' file is shown with its content: 'echo \$USER is awesome'. The file was last modified 2 hours ago by user 'Administrator' with commit hash '747729e3'.

This is just one of the use cases how you can use your GitLab with branches and merge requests. In the reality there are hundreds

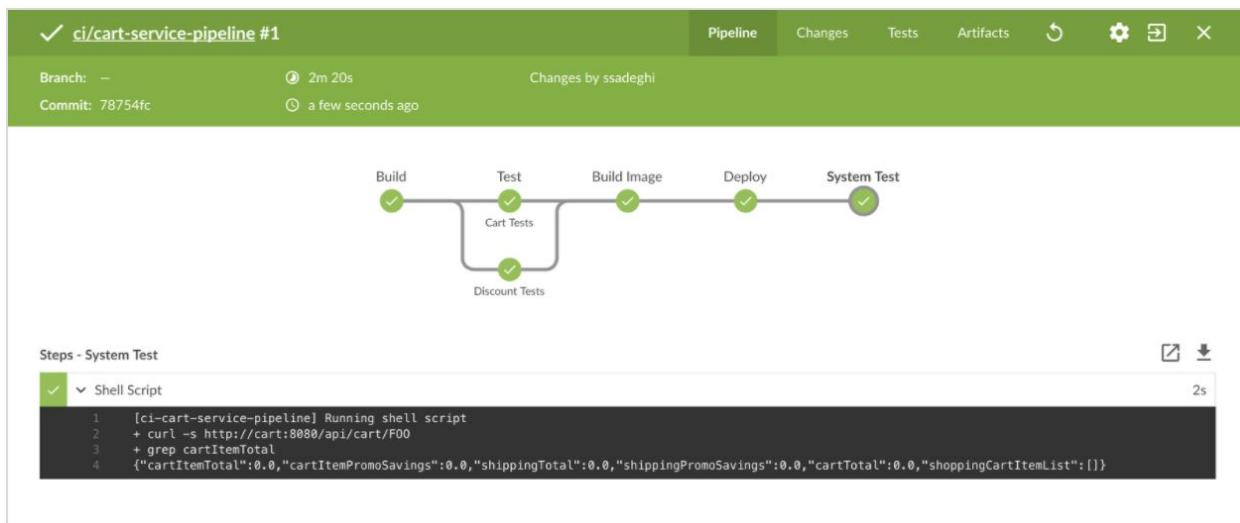
Getting started with Jenkins

In this Chapter, we talk about Jenkins as the main CI/CD tool to automate your main tasks. We admit that there is 1001 different tool available on the market which can help you to achieve your final goal. But we have chosen Jenkins for its popularity, flexibility and openness to the Open Source community. We have experience working with other tools, but Jenkins has proven to be a problem solver for most of the challenges we have faced in our carrier.

About Jenkins

So what is Jenkins and What is CI/CD? Let's do one by one:

1. Jenkins is a simple yet powerful automation Server written in Java that uses Groovy language to build Automation Pipelines. We give you enough examples later in this book so you fully understand what it means and how to use it.
2. CI/CD - Continuous Integration / Continuous Delivery / Continuous Deployment - while it sounds futuristic, CI/CD is an automated workflow on a high level.



Jenkins CI/CD Pipeline / Workflow

In other words, Jenkins is a simple yet powerful scheduler, that makes our life 10 times easier and you will see how and why later in this book.

Installing Jenkins

Installing Jenkins is very straightforward and does not take much time. Just follow the instruction below and you have it up and running shortly.

First, we need to navigate to the GitHub repository we have cloned previously. The repo contains a **Vagrantfile** which we described at the beginning of this book.

Start the Jenkins machine:

```
$ vagrant up jenkins
Bringing machine 'jenkins' up with 'virtualbox' provider...
...
<OUTPUT OMITTED>
...
jenkins: Complete!
```

Note: Our github repo is located at <https://github.com/flashdumper/Jenkins-and-GitLab>.

Once VM is up and running, you can ssh into it.

```
$ vagrant ssh jenkins
[vagrant@jenkins ~]$ hostname
jenkins.example.com

[vagrant@jenkins ~]$ sudo -i
[root@jenkins ~]#
```

Jenkins is written in **Java** and heavily dependent on it. So the first thing we need to do is to install java development kit.

```
[root@jenkins ~]# yum install java-1.8.0-openjdk -y
...
<OUTPUT OMITTED>
...
Complete!
```

Install Jenkins repo and gpg keys, and then install Jenkins.

```
[root@jenkins ~]# curl http://pkg.jenkins-ci.org/redhat/jenkins.repo >
/etc/yum.repos.d/jenkins.repo

[root@jenkins ~]# rpm --import
https://jenkins-ci.org/redhat/jenkins-ci.org.key

[root@jenkins ~]# yum install -y jenkins
...
<OUTPUT OMITTED>
...
Complete!
```

And finally, start Jenkins service and make it persistent throughout the reboots.

```
[root@jenkins ~]# systemctl start jenkins.service
[root@jenkins ~]# systemctl enable jenkins.service
jenkins.service is not a native service, redirecting to /sbin/chkconfig.
Executing /sbin/chkconfig jenkins on
```

Verify that Jenkins service is running and listening on port 8080.

```
[root@jenkins ~]# curl localhost:8080
<html>
...
<OUTPUT OMITTED>
...
Authentication required
...
<OUTPUT OMITTED>
...
</body></html>
```

Great job. Now open your browser at “<http://jenkins.example.com:8080/>” and you should be redirected to Jenkins initialization page.

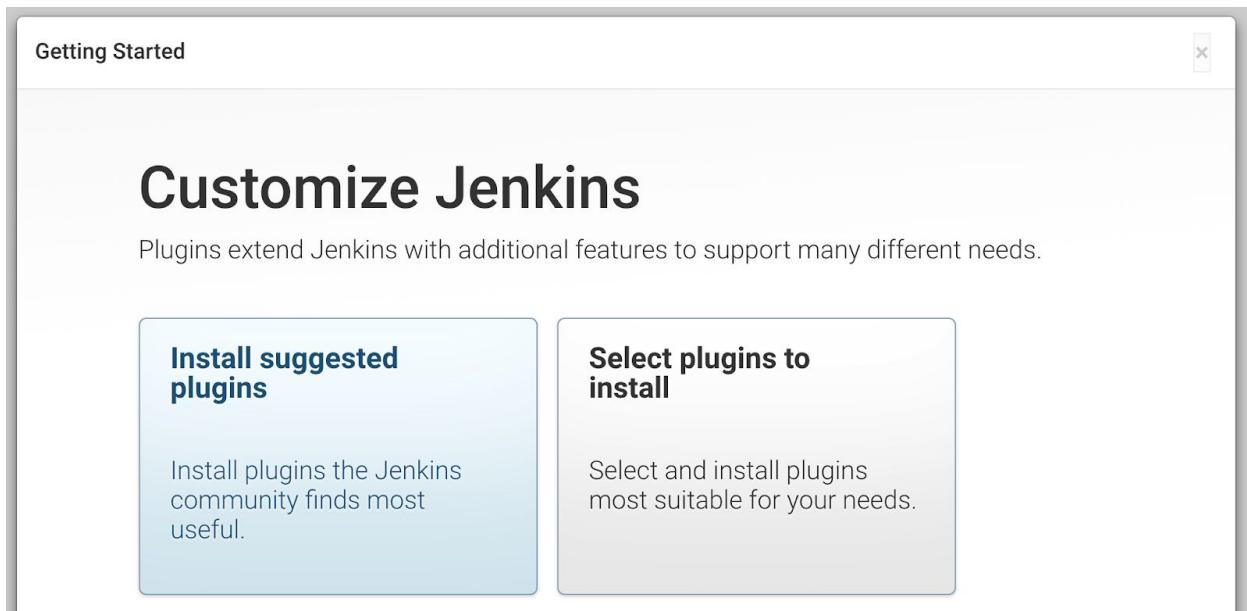


You need to provide a randomly generated Admin password from **/var/lib/jenkins/secrets/initialAdminPassword** in order to initialize Jenkins.

Copy the password from the path above and paste into the “Administrator password” text area and press **Continue**.

```
[root@jenkins ~]# cat /var/lib/jenkins/secrets/initialAdminPassword  
c713bfe0e3d74853b2df070b0f74dd7e
```

Jenkins has a concept of using different plugins to do the job. These plugins are written by the Jenkins community and available at <https://plugins.jenkins.io/>. In our case, we install suggested plugins to get started. Later in this book, we will learn how to install additional plugins with Jenkins. Click on “**Install Suggested Plugins**” to proceed with the initialization.



It will start plugins installation process.

Getting Started

Getting Started

✓ Folders	✓ OWASP Markup Formatter	✓ Build Timeout	✓ Credentials Binding	** JDK Tool ** Script Security ** Command Agent Launcher Folders ** bouncycastle API ** Structs ** Pipeline: Step API ** SCM API ** Pipeline: API ** JUnit OWASP Markup Formatter ** Token Macro Build Timeout ** Credentials ** SSH Credentials ** Plain Credentials
✓ Timestamper	✓ Workspace Cleanup	✓ Ant	✓ Gradle	
⌚ Pipeline	⌚ GitHub Branch Source	⌚ Pipeline: GitHub Groovy Libraries	⌚ Pipeline: Stage View	
⌚ Git	⌚ Subversion	⌚ SSH Slaves	⌚ Matrix Authorization Strategy	
⌚ PAM Authentication	⌚ LDAP	⌚ Email Extension	⌚ Mailer	

Once finished you will be redirected to the user creation page. Create a user named “**jenkins**” as on the picture below and click “**Save and Continue**”.

Getting Started

Create First Admin User

Username:	<input type="text" value="jenkins"/>
Password:	<input type="password" value="....."/>
Confirm password:	<input type="password" value="....."/>
Full name:	<input type="text" value="Jenkins User"/>
E-mail address:	<input type="text" value="jenkins@example.com"/>

Jenkins 2.134

[Continue as admin](#)

[Save and Continue](#)

Note: It is also a common practice to create “admin” user first. But it can be anything else. In our case, we use “jenkins” as our first user.

On the next page just click on “**Save and Finish**”.

Getting Started

Instance Configuration

Jenkins URL:

The Jenkins URL is used to provide the root URL for absolute links to various Jenkins resources. That means this value is required for proper operation of many Jenkins features including email notifications, PR status updates, and the `BUILD_URL` environment variable provided to build steps.

The proposed default value shown is **not saved yet** and is generated from the current request, if possible. The best practice is to set this value to the URL that users are expected to use. This will avoid confusion when sharing or viewing links.

Jenkins 2.134 Not now [Save and Finish](#)

Then click on **Start using Jenkins**.

Getting Started

Jenkins is ready!

Your Jenkins setup is complete.

[Start using Jenkins](#)

And we should be finally redirected to Jenkins dashboard.

The screenshot shows the Jenkins dashboard. At the top, there's a navigation bar with links like 'New Item', 'People', 'Build History', 'Manage Jenkins', 'My Views', 'Credentials', and 'New View'. Below the navigation bar, the main content area has a heading 'Welcome to Jenkins!' and a message 'Please [create new jobs](#) to get started.' On the left, there are two sections: 'Build Queue' (which says 'No builds in the queue.') and 'Build Executor Status' (which shows '1 Idle' and '2 Idle').

That was not really hard at all.

Pipelines overview

As we mentioned before, Jenkins works mainly with pipelines. And a pipeline is just a sequence of steps that are executed in a specific order. It is a workflow like shown below:



Imagine that you need to automate the process of uploading and applying configuration settings to a set of servers or networking devices. Or another scenario where you want to add a new server, VM or networking device. One more example, if you need to deliver application from source code to production systems where only RPM changes are allowed. How would you do it without Jenkins? We can see two options here:

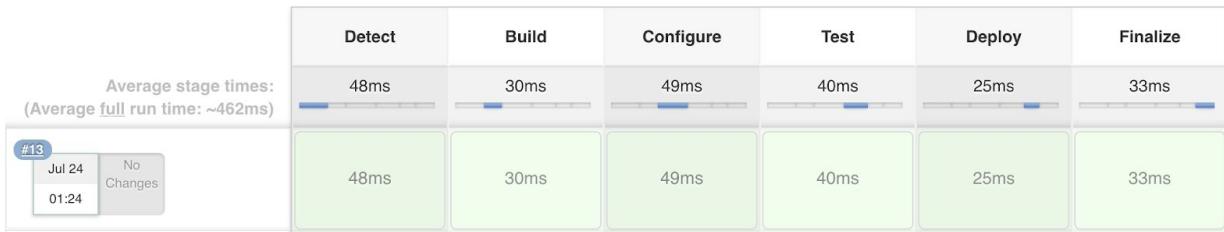
1. Do it all manually, meaning that you need to develop the configuration files or settings and apply them one by one verifying that it all works as expected.
2. Another option is to develop one or a set of scripts to do the same job.

The second option, of course, is preferred and you do not need Jenkins for this. And you can execute these scripts one by one to do the job.

This is where Jenkins comes into the picture. Jenkins will take all these scripts, form a workflow and run them in the order you specify. And when you have new configuration available, or a new device in your network. Jenkins will start the pipeline and provision, configure and deploy and configure it automatically. Isn't this great? And this is what we will teach you.

Creating Jenkins Pipeline

This is how Jenkins pipeline looks like. A series of steps completed in sequence or parallel.



Let's create a pipeline ourselves. Open Jenkins dashboard page at ["http://jenkins.example.com:8080/"](http://jenkins.example.com:8080/) and click on "New Item"

New Item

People

Build History

Manage Jenkins

My Views

Credentials

New View

Welcome to Jenkins!

Please [create new jobs](#) to get started.

Build Queue

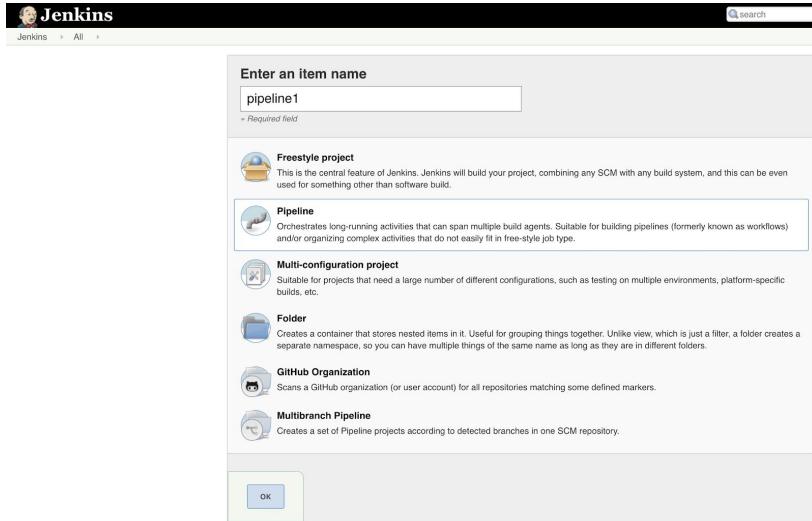
No builds in the queue.

Build Executor Status

1 Idle

2 Idle

Type "**pipeline1**" in pipeline text area and choose **Pipeline**, and click **OK**. We are not interested in all other options in this book, simply because they are not that popular in comparison to **Pipeline**.



That will redirect you to **pipeline1** settings. There is a lot of settings, but you may safely ignore them for now. We cover some of them as we go.
Scroll all the way down to Pipeline section, click on “try sample Pipeline...”, choose “Hello World” and click “Save”.

```
node {  
    echo 'Hello World'  
}
```

And your first pipeline is created. It was not that hard, right? Now start your pipeline clicking on “Build now”.

The screenshot shows the Jenkins interface for a pipeline named 'pipeline1'. The top navigation bar includes links for Back to Dashboard, Status, Changes, Build Now, Delete Pipeline, Configure, Full Stage View, Rename, and Pipeline Syntax. Below this is a 'Build History' section with a search bar and RSS feed links for all and failures.

Once you start your first build, you should be able to see a new build appeared in Build History on the mid-left side of the screen.

The screenshot shows the Jenkins interface for pipeline 'pipeline1'. The main title is 'Pipeline pipeline1'. The sidebar contains links for Back to Dashboard, Status, Changes, Build Now, Delete Pipeline, Configure, Full Stage View, Rename, and Pipeline Syntax. The 'Stage View' section displays a message: 'This Pipeline has run successfully, but does not define any stages. Please use the stage step to define some stages in this Pipeline.' Below this is a 'Permalink' section listing four build links. The 'Build History' section shows one build entry for '#1' on Jul 24, 2018 at 12:58 AM, with RSS feed links for all and failures.

You are also going to see the warning message telling that we need to define stage step in the Pipeline. We will talk about this more in a moment. For now, click on the #1 in the build history.

The screenshot shows the Jenkins build history for build '#1'. The build was run on Jul 24, 2018 at 12:58 AM. It includes a search bar and RSS feed links for all and failures. To the right of the build history is a vertical scroll bar.

Take a look at the pipeline sidebar and its different options, we are going to use them later in this chapter.



One of the most commonly used menus is “**Console Output**”.

Console Output

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] echo
Hello World
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

If you examine output, you see “**Hello World**” message that was generated by our example pipeline. Before we build something cool that you can actually use in real life, we need to learn Groovy. Groovy is a scripting language that Jenkins Pipelines are using to build a workflow.

Groovy - Basics

The Groovy scripting language is the very essential piece of Jenkins automation. Whether you want to make a simple-to-use one-step-pipeline or a complex 100+ steps pipeline running different tasks on different machines, you use Groovy. If you are new to Groovy, no worries, you will learn Jenkins pipeline syntax with Groovy step by step.

There are two ways to describe write Jenkins pipelines. Using **Scripted** and **Declarative** syntax. **Declarative** syntax is rather new and not widely used yet, that is why we focus on **Scripted** syntax. Just to give you an idea about the difference between both syntaxes, let's take a look at the simple

Scripting syntax	Declarative syntax
<pre>node { stage('Hello') { echo "Hello World" } }</pre>	<pre>pipeline { agent any }</pre>

```

        }
    }

    stages {
        stage("Hello") {
            steps {
                sh "echo Hello World"
            }
        }
    }
}

```

You can see the difference between just looking at these simple pipelines. Now imagine that your pipeline consist of 50+ steps and even more conditions. This complexity will multiply by the number of steps, stages and different condition you have.

As you may notice, Jenkins pipeline is hierarchical, structured and described within curly braces `{ }`. It is hard to keep track of all this hierarchy when you have a complex pipeline, so we recommend you to use a code editor such as Atom or Visual Studio Code. They highlight code syntax. Here is how the same code above looks like with Visual Studio Code.

The screenshot shows two side-by-side code editors. The left editor has a dark theme and displays the Jenkins Pipeline code with syntax highlighting: 'node' is blue, 'stage' is orange, and 'sh' is red. The right editor has a light theme and displays the same code with syntax highlighting: 'pipeline' is blue, 'agent' is orange, 'stages' is green, 'stage' is orange, 'steps' is green, and 'sh' is red. Both editors show the same code structure with nested braces and line numbers from 1 to 10.

```

1 node {
2     stage('Hello') {
3         echo "Hello World"
4     }
5 }
6
7
8
9
10 }

```

```

1 pipeline {
2     agent any
3     stages {
4         stage("Hello") {
5             steps {
6                 sh "echo Hello World"
7             }
8         }
9     }
10 }

```

Note! Different color themes have been used.

It looks much nicer and you can actually keep track of nested braces `{ }` and not being lost while building your pipeline.

Note! You can download Visual Studio Code from <https://code.visualstudio.com/download>. You will be required to install groovy syntax plugin, but that is out of the scope of this book.

Node

Going back to scripted syntax. From the example above you can see that our very first block is called **node**. The **node** block describes where Jenkins is going to run this pipeline. By default, Jenkins server is being used and it's called **master** node.

In our case, two examples above are the same.

```

1  node {
2    stage('Hello') {
3      echo "Hello World"
4    }
5  }
6

```

```

1  node('master') {
2    stage('Hello') {
3      echo "Hello World"
4    }
5  }
6

```

This helps when we have a lot of tests or we need to distribute the workload between several Jenkins nodes that may be located in different places. For example, when you have several Data Center, you would want to jobs to be locally executed within a Data Center. Node is usually at the highest level of the hierarchy, but not always.

Stage

node usually consists of one or several stages, for example, build, configure, test, deploy. Each **stage** can have more steps. Let's build our first pipeline where we add a new switch or server to our network. First, we need to identify the steps.

Let's create a set of stages at a high level and define in the pipeline. We came up with 6 main stages in this project.

```

node('master') {
  stage('Detect') {
  }
  stage('Build') {
  }
  stage('Configure') {
  }
  stage('Test') {
  }
  stage('Deploy') {
  }
  stage('Finalize') {
  }
}

```

*Note that stage names are completely arbitrary, meaning you can **name** them as you are willing to.*

Complex pipelines may need to execute different stages in different locations. In that case, a stage will be the parent object.

Step

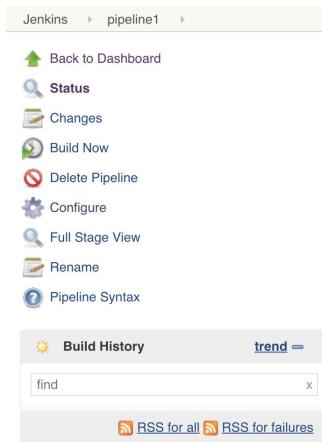
Each stage has one or more steps. Let's fill every stage with a set of steps using Jenkins pipeline built-in **echo** command. As you may have guessed it echoes some message to pipeline output. This is a perfect tool to make a placeholder for pipeline stages.

Navigate back to Job Configuration at <http://jenkins.example.com:8080/job/pipeline1/configure> and update our pipeline with the following content.

```
node('master') {  
    stage('Detect') {  
        echo "Detect a new switch in the network"  
        echo "Check DHCP leases for a new IP address from a pool of management IPs"  
        echo "use SNMP or ssh with default credentials to get switch/server info"  
    }  
    stage('Build') {  
        echo "Generate switch/server configuration using jinja2 templates"  
        echo "Run syntax checks and lint tests"  
    }  
    stage('Configure') {  
        echo "Upload configuration files"  
        echo "Install required packages"  
  
    }  
    stage('Test') {  
        echo "Run smoke tests"  
        echo "Run functionality tests"  
        echo "Run regression tests"  
        echo "Run compatibility tests"  
    }  
    stage('Deploy') {  
        echo "Update device inventory"  
        echo "Add device to monitoring tool, security center and syslog server"  
    }  
    stage('Finalize') {  
        echo "Create a PDF report based on the pipeline workflow and its results"  
        echo "Send an email, text and message to Slack to notify the interested  
parties"  
    }  
}
```

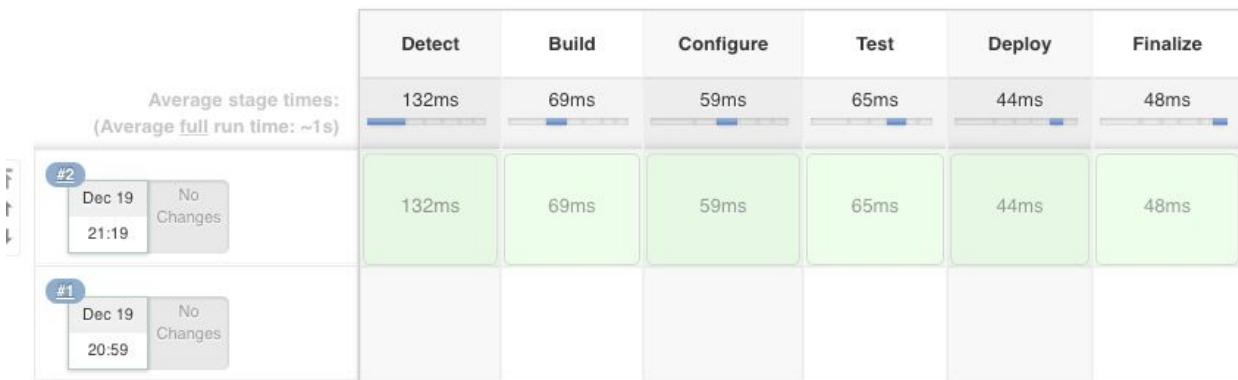
}

It looks much better now. It's time to test it out. Open your browser at <http://jenkins.example.com:8080/job/pipeline1/> and click **Build Now**.



As a result, you should be able to see the pipeline represented in graphical format. These are the steps we have just defined.

Stage View



You can check build logs by clicking on build number. In our case **#2** and then **Console Output**. That will show the fully consolidated log for this build.



Console Output

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Detect)
[Pipeline] echo
Detect a new switch in the network
[Pipeline] echo
Check DHCP leases for a new IP address from a pool of management IPs
[Pipeline] echo
use SNMP or ssh with default credentials to get switch/server info
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] echo
Generate switch/server configuration using jinja2 templates
[Pipeline] echo
Run syntax checks and lint tests
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Configure)
[Pipeline] echo
Upload configuration files
[Pipeline] echo
Install required packages
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] echo
Run smoke tests
[Pipeline] echo
Run functionality tests
[Pipeline] echo
Run regression tests
[Pipeline] echo
Run compatibility tests
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] echo
```

Executing shell scripts using “sh”

This is all good, but **echo** command does not really do anything. We can fix it with pipeline **sh**. **Sh** runs commands you specify in the Linux shell. It can be pretty much anything, whether you want to run ping command, or get information from the file, or even run bash or ansible-playbook, **sh** command is to the rescue. It makes automation simple and elegant.



Console Output

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Detect)
[Pipeline] echo
Detect a new switch in the network
[Pipeline] echo
Check DHCP leases for a new IP address from a pool of management IPs
[Pipeline] echo
use SNMP or ssh with default credentials to get switch/server info
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] echo
Generate switch/server configuration using jinja2 templates
[Pipeline] echo
Run syntax checks and lint tests
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Configure)
[Pipeline] echo
Upload configuration files
[Pipeline] echo
Install required packages
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] echo
Run smoke tests
[Pipeline] echo
Run functionality tests
[Pipeline] echo
Run regression tests
[Pipeline] echo
Run compatibility tests
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] echo
```

Now you can replace all the **echo** and **sh** commands with the scripts that you may already have as a part of your day-to-day job. If you do not have any, it is going to be a good exercise to develop them and see how Jenkins Pipelines make your job easier.

Input

Another basic command that we want to show you is **input**. The purpose of this command is to stop pipeline execution and wait for user input. In our example, we inject **Approval** stage in between **Test** and **Deploy** stages. The purpose of it is to review all the logs and make sure that our Jenkins pipeline works as expected and all the scripts passed executed and gave us positive results.

```
...
stage('Test') {
```

```

echo "Run smoke tests"
echo "Run functionality tests"
echo "Run regression tests"
echo "Run compatibility tests"
}

stage('Approval') {
    input "Please check if we are good to go before we push into production."
}

stage('Deploy') {
    echo "Update device inventory"
    echo "Add a device to monitoring tool, security center, and syslog server"
}

...

```

Now let's update our pipeline and **Save** the changes.

The screenshot shows the Jenkins Pipeline configuration screen. At the top, there's a title bar with 'Pipeline' and tabs for 'Definition' and 'Pipeline script'. Below that is a code editor area with line numbers (23-37) and Groovy script content. A 'try sample Pipeline...' button is visible. Below the editor are 'Use Groovy Sandbox' and 'Pipeline Syntax' links. At the bottom are 'Save' and 'Apply' buttons.

```

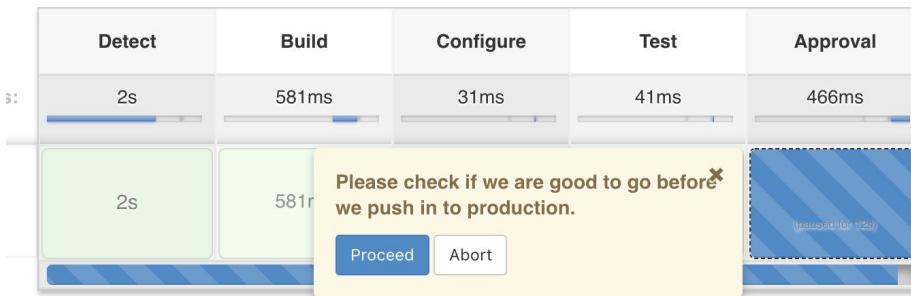
Script 23 | echo "Run smoke tests"
           24 | echo "Run functionality tests"
           25 | echo "Run regression tests"
           26 | echo "Run compatibility tests"
           27 |
           28 | } stage('Approval') {
           29 |     input "Please check if we are good to go before we push in to production."
           30 | }
           31 | } stage('Deploy') {
           32 |     echo "Update device inventory"
           33 |     echo "Add device to monitoring tool, security center and syslog server"
           34 | }
           35 | } stage('Finalize') {
           36 |     echo "Create a PDF report based on the pipeline workflow and its reults"
           37 |

```

Finally, press **Build Now** to start the new build.

The screenshot shows the Jenkins interface for a pipeline named 'pipeline1'. At the top, there's a navigation bar with links like 'Back to Dashboard', 'Status', 'Changes', 'Build Now', 'Delete Pipeline', 'Configure', 'Full Stage View', 'Rename', and 'Pipeline Syntax'. Below this is a 'Build History' section with a search bar and RSS feed links. The main part of the screen displays a pipeline timeline with five stages: Detect, Build, Configure, Test, and Approval. The Approval stage is currently active, indicated by a yellow background and a modal dialog. The dialog contains the text: 'Please check if we are good to go before we push in to production.' with a red 'X' icon. It also has two buttons: 'Proceed' and 'Abort'. The 'Approval' stage has a duration of 466ms.

Once build is started you should be able to see **Approval** stage appeared in the pipeline workflow. It is waiting for our Approval to proceed further.



Before we do that, navigate to pipeline output and verify that we have our scripts executed and working properly with no unexpected behavior.

```

[pipeline1] Running shell script
+ cat /etc/hosts
127.0.0.1      jenkins.example.com      jenkins
127.0.0.1      localhost localhost.localdomain localhost4 localhost4.localdomain4
::1            localhost localhost.localdomain localhost6 localhost6.localdomain6
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] echo
Generate switch/server configuration using jinja2 templates
[Pipeline] sh
[Pipeline] Running shell script
+ echo Generate switch/server configuration using jinja2 templates
Generate switch/server configuration using jinja2 templates
[Pipeline] echo
Run syntax checks and lint tests
[Pipeline] sh
[Pipeline] Running shell script
+ echo jenkins
jenkins
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Configure)
[Pipeline] echo
Upload configuration files
[Pipeline] echo
Install required packages
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] echo
Run smoke tests
[Pipeline] echo
Run functionality tests
[Pipeline] echo
Run regression tests
[Pipeline] echo
Run compatibility tests
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Approval)
[Pipeline] input
Please check if we are good to go before we push in to production.
Proceed or Abort


```

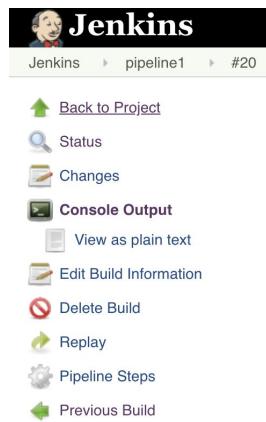
Note that you can also move forward by clicking **Proceed**. Once you Approve the stage.

```

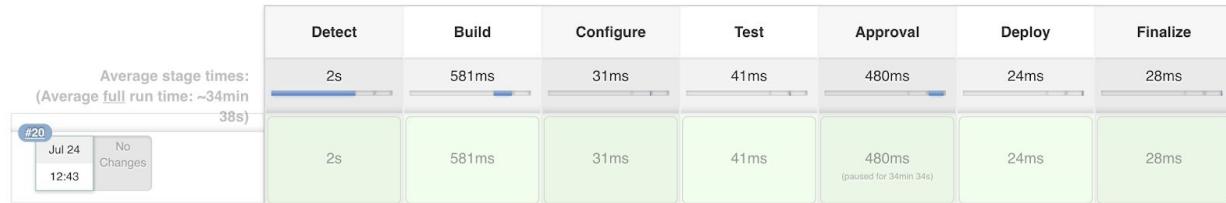
Please check if we are good to go before we push in to production.
Proceed or Abort
Approved by Jenkins User
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] echo
Update device inventory
[Pipeline] echo
Add device to monitoring tool, security center and syslog server
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Finalize)
[Pipeline] echo
Create a PDF report based on the pipeline workflow and its results
[Pipeline] echo
Send and email, text and message to Slack to notify the interested parties
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

Go back to by clicking on “**Back to Project**” in the top-left corner of the screen.



Take a look at the pipeline. You should be able to see **Approval** stage in between **Test** and **Deploy** stages.



You will learn more about pipeline syntax in the following section.

Pipelines - advanced syntax

We have taken a look at the very basics of pipeline syntax in Jenkins. In this section, we want to go over advanced topics that you will most likely use when building Jenkins pipelines.

Variables

Quite often we need to work with dynamic data in Jenkins and variables are very useful in this way. Let's take a look at the example where we need to run a simple pipeline. And at the end of the pipeline, we need to create a file and put there date and time the job was started and finished.

And it has to be done only if the pipeline was approved. The best way to do that is going to be using variables. Creating a variable is easy. Usually, the following structure is being used:

```
<variable> = <value>
```

A simple example can be

```
var1 = "value1"
```

This command creates a variable called “**var1**” with a value of “**value1**”. There is not much that we can do with this because this value is static and we can just use “**value1**” directly.

Instead of this, we will run “**date**” command and store its value in a variable that we can later use. We will also create a variable called **timedate_start** and **timedate_end** and store the results in different variables. There is a caveat though, we will need to access specific **sh** parameter called “**returnStdout**” and set its value to true. This will slightly change the syntax.

```
timedate_start = sh script:"date", returnStdout:true
```

We do not discuss all the different attributes at this moment, but we will get back to this topic later in this book.

Next step is to access the data stored in **timedate** variable. In different situations, we use different ways to extract the data from variables. In this case, we use the following structure “**“\${VARIABLE}”**”. In our case it looks like the following:

```
echo "${timedate}"
```

Note: double quotes are required and it won't work with single quotes or without them.

Let's see this in action. We modify our pipeline to make it shorter and easier to understand.

```
node('master') {
    stage('Detect') {
        // Making timestamp at the beginning of the job.
        timedate_start = sh script:"date", returnStdout:true

        // There are placeholders for real tasks
        sh "echo Detect a new switch in the network"
        sh "echo Check DHCP leases for a new IP address from a pool of management
IPs"
    }
    stage('Approval') {
        input "Please check if we are good to go before we push in to production."
    }
    stage('Finalize') {
        // There are placeholders for real tasks
        echo "Create a PDF report based on the pipeline workflow and its results"
    }
}
```

```

    echo "Send an email, text, and message to Slack to notify the interested
parties"

    // Taking timestamp at the end of the job
    timedate_end = sh script:"date",returnStdout:true

    //echoing the results
    sh "echo Job started at ${timedate_start}"
    sh "echo Job ended at ${timedate_end}"
}

}

```

Note: double forward-slashes (//) are using to make comments in the code. It is a best practice to explain your code as you work on it. In the future, it will help you to avoid confusions and allow others to use your work.

Save the results and start the build. Once you approve it, then navigate to build logs and check the results. You should see the following:

```

[pipeline1] Running shell script
+ echo Job started at Wed Jul 25 00:37:12 UTC 2018
Job started at Wed Jul 25 00:37:12 UTC 2018
[Pipeline] sh
[pipeline1] Running shell script
+ echo Job ended at Wed Jul 25 00:37:19 UTC 2018
Job ended at Wed Jul 25 00:37:19 UTC 2018
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

You see that the job started at 00:37:12 and ended 7 seconds later. You can achieve great things with variables and we use variables more and more as we go.

There is also a list of Jenkins built-in variables such as JENKINS_URL, JOB_NAME, BUILD_NUMBER. We use them later in our book. The full list of built-in variables is available at <https://wiki.jenkins.io/display/JENKINS/Building+a+software+project>.

Functions

Next important topic is a function. The main purpose of a function is to make your code reusable and more readable.

For example, you want to send messages to your Slack channel and notify project users on the progress. Before we jump into building notification with slack, we need to discuss how to build a function. Function usually has the following structure:

```
def <function_name>(<parameter1>,<parameter2>,...) {  
    <place your code here>  
}
```

It's confusing we know, so let's replace it with the actual example, see how it works and then figure out how it works step by step.

Example1:

```
def run_ls() {  
    ls = sh script:"ls -l /",returnStdout:true  
    echo "${ls}"  
}
```

Function definition starts with **def** (short for define), following by function name and round braces (). Round braces are used to define **parameters**. We work with parameters in just a moment. The last and final part is function body enclosed with curly braces {}, where we define our code for the function. So the structure is very easy.

To execute/call this function we just need to specify its name with round braces **run_ls()**. Round braces () basically tell Jenkins that run_ls is a function with no parameters.

Let's update our pipeline to use this new function we have just created.

```
def run_ls() {  
    ls = sh script:"ls -l /",returnStdout:true  
    echo "${ls}"  
}  
  
node('master') {  
    stage('Init') {  
        // Making timestamp at the beginning of the job.  
        timedate_start = sh script:"date",returnStdout:true  
        run_ls()  
    }  
    stage('Finalize') {  
        // Taking timestamp at the end of the job  
        timedate_end = sh script:"date",returnStdout:true  
    }  
}
```

```

    //echoing the results
    sh "echo Job started at ${timedate_start}"
    sh "echo Job ended at ${timedate_end}"
}
}

```

Save the changes in Jenkins pipeline, run the job and check Console Output.

Console Output

```

Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Detect)
[Pipeline] sh
[pipeline1] Running shell script
+ date
[Pipeline] sh
[pipeline1] Running shell script
+ ls -l /
[Pipeline] echo
total 20
lrwxrwxrwx.  1 root      root      7 May 12 18:50 bin -> usr/bin
dr-xr-xr-x.  5 root      root     4096 May 12 18:55 boot
drwxr-xr-x. 18 root      root     2960 Jul 23 17:59 dev
drwrxr-xr-x. 82 root      root     8192 Jul 23 18:24 etc
drwxr-xr-x.  3 root      root     21 May 12 18:54 home
lrwxrwxrwx.  1 root      root      7 May 12 18:50 lib -> usr/lib
lrwxrwxrwx.  1 root      root      9 May 12 18:50 lib64 -> usr/lib64
drwxr-xr-x.  2 root      root      6 Apr 11 04:59 media
drwxr-xr-x.  2 root      root      6 Apr 11 04:59 mnt
drwxr-xr-x.  2 root      root      6 Apr 11 04:59 opt
dr-xr-xr-x. 120 root     root      0 Jul 23 17:59 proc
dr-xr-x---.  3 root      root     170 Jul 24 03:36 root
drwxr-xr-x.  25 root     root     860 Jul 24 22:09 run
lrwxrwxrwx.  1 root      root      8 May 12 18:50 sbin -> usr/sbin
drwxr-xr-x.  2 root      root      6 Apr 11 04:59 srv
dr-xr-xr-x. 13 root     root      0 Jul 23 17:59 sys
drwxrwxrwt. 12 root     root     4096 Jul 25 02:45 tmp
drwxr-xr-x. 13 root     root     155 May 12 18:50 usr
drwxr-xr-x.  2 vagrant   vagrant   25 Jul 23 17:59 vagrant
drwxr-xr-x. 18 root      root     254 Jul 23 17:59 var

```

Good, now we can call this function at any point in the pipeline workflow. There is a problem though.

Whenever we call `run_ls()` we end up seeing the content of the `/` directory with the same `ls` options. What can we do to change this behavior? Function parameters to the rescue. We will modify `run_ls()` function and add two parameters, one of them is going to represent `ls` command options and another is going to represent one or more arguments.

*Note: Linux command syntax usually looks like **command option argument**, where:*

- **Command** - executable (`ls`, `cd`, `echo`, `kill`, etc)
- **Option** - changes command behavior and starts with one or two dashes (-) e.g. `-l -a`
- **Argument** - something command operates on, e.g `/tmp ~ /var/log`

In our example “**ls -l /**”:

- **ls** - is a command
- **-l** - an option
- **/** - an argument

Now let's see how we can use parameters

```
def run_ls(options, arguments) {  
    ls = sh script:"ls ${options} ${arguments}", returnStdout:true  
    echo "${ls}"  
}  
  
node('master') {  
    stage('Init') {  
        // Making timestamp at the beginning of the job.  
        timedate_start = sh script:"date", returnStdout:true  
        // checking /tmp directory content  
        run_ls('-l','/tmp')  
  
        // checking /home and /var/tmp/ directories content  
        run_ls('-la','/home/ /var/tmp')  
  
    }  
    stage('Finalize') {  
        // Taking timestamp at the end of the job  
        timedate_end = sh script:"date", returnStdout:true  
  
        //echoing the results  
        sh "echo Job started at ${timedate_start}"  
        sh "echo Job ended at ${timedate_end}"  
    }  
}
```

In this example, we are calling **run_ls('-l','/tmp')** function and specifying two parameters separated by comma (,). These two parameters are passed to the function we defined at the very beginning and they become variables within this **run_ls()** function. This will be equal to: **options = “ls -l”, arguments = “/tmp”**. And then we are using these variables in **sh** command. Now save the pipeline, start the build and check Console Output.

Console Output

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Init)
[Pipeline] sh
[Pipeline] { Running shell script
+ date
[Pipeline] sh
[Pipeline] { Running shell script
+ ls -l /tmp
[Pipeline] echo
total 3332
-rw-r--r--. 1 jenkins jenkins 19673 Jul 23 18:25 akuma7977688014197832363jar
drwxr-xr-x. 2 jenkins jenkins 18 Jul 23 18:25 hsperfdata_jenkins
drwxr-xr-x. 2 root root 18 Jul 23 18:09 hsperfdata_root
drwxr-xr-x. 2 jenkins jenkins 6 Jul 23 18:25 jetty-0.0.0-8080-war-_any-4722210350448256014.dir
drwxr-xr-x. 2 jenkins jenkins 6 Jul 23 18:25 jna-1712433994
-rw-r--r--. 1 jenkins jenkins 1137285 Jul 23 18:25 jna3728572508927751622jar
-rw-r--r--. 1 jenkins jenkins 45 Jul 25 00:35 job_timer.txt
drwx-----. 3 root root 17 Jul 23 17:59 systemd-private-92af49aeeae445208a043cffc329cc90-chrony.service-g13RzX
-rw-r--r--. 1 jenkins jenkins 2245752 Jul 23 18:25 winstone1899653194133477298.jar

[Pipeline] sh
[Pipeline] { Running shell script
+ ls -la /home/ /var/tmp/
[Pipeline] echo
/home/:
total 0
drwxr-xr-x. 3 root root 21 May 12 18:54 .
dr-xr-xr-x. 18 root root 239 Jul 23 18:00 ..
drwx-----. 3 vagrant vagrant 95 Jul 24 03:36 vagrant

/var/tmp/:
total 0
drwxrwxrwt. 3 root root 85 Jul 25 02:45 .
drwxr-xr-x. 18 root root 254 Jul 23 17:59 ..
drwx-----. 3 root root 17 Jul 23 17:59 systemd-private-92af49aeeae445208a043cffc329cc90-chrony.service-IepRXl
```

This is just perfect. Exactly what we wanted to see. And our code is clean and reusable. Perfect! Now we are ready to write something really cool that will help you on a daily basis. We will write a function that does an API call to Slack and sends the progress on our pipeline progress. Slack is a popular online messenger. You can learn more at <https://slack.com/>. Before we create a function that sends a message to the Slack channel, we need to create a new workspace and generate API token. Navigate to <https://slack.com/create> and create your slack workspace. First, you will need to provide your valid email and click **Next**

Create a new workspace

Your email address

you@example.com

Please fill in your email.

Enter confirmation code that you got on the email.

Check your email

We've sent a six-digit confirmation code to slack@example.com. It will expire shortly, so enter your code soon.

Your confirmation code

Keep this window open while checking for your code.
Haven't received our email? Try your spam folder!

Specify your name and continue further.

What's your name?

This is how your teammates in Slack will see and refer to you.

Full name

Jenkins

Display name (optional)

Display name

By default, Slack will use your full name — but you can choose something shorter if you'd like.

It's ok to send me email about the Slack service.

Continue to Password →

Set your password.

Set your password

Choose a password for signing in to Slack.

Password

••••••••••

Passwords must be at least 6 characters long, and can't be So-so things like "password", "123456" or "abcdef".

Continue to Workspace Info →

Answer some silly questions if any.

Tell us about your team

What will your team use Slack for?

Shared interest group

How big is your shared interest group?

4000 or more people

Continue to Group Name →

Name your group. We will be able to change it later if needed.

What's your group called?

Group name

Jenkins

We'll use this to name your Slack workspace, which you can always change later.

Continue to Workspace URL →

And specify your workspace name. It has to be unique, so be creative.

What URL do you want
for your Slack workspace?

Choose the address you'll use to sign in to Slack.

Your workspace URL (letters, numbers, and dashes only)

jenkins-and-gitlab .slack.com

Create Workspace →

On the following step press “**Skip For Now**”

Send Invitations

Your Slack workspace is ready to go. Know a few friends or coworkers who'd like to explore Slack with you?

Email address [+ Add another invitation](#)
 jame@example.com
 name@example.com
 name@example.com

Or, you can [get an invite link](#) to share with other people.

[Skip For Now](#)

Send Invitations

On the following step click on “**continue in browser**”. You can Download slack for your platform as well, but this is not necessary.

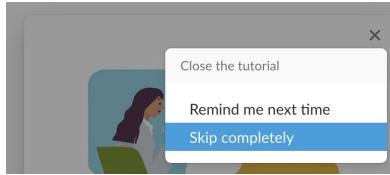
Get the Slack desktop app

Install the Slack app for Mac to get better notifications, and to launch Slack right from your Dock.

Download the Mac App

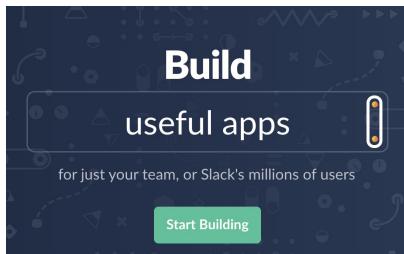
or [continue in the browser](#)

It is going to redirect you to your workspace and start the tutorial. You may skip tutorial it if you want. That is what we did in any case.



Now we have our first channel up and running and ready to generate API token and make our first API call with Jenkins. Leave this browser tab open, we will need it later.

Open another tab, navigate to <https://api.slack.com/>, and click on **Start Building**.



In the pop-up menu name your App and choose group we specified earlier.



The next step is to enable incoming hooks. We use hooks more often with different applications Including Jenkins and GitLab later in the book.

Basic Information

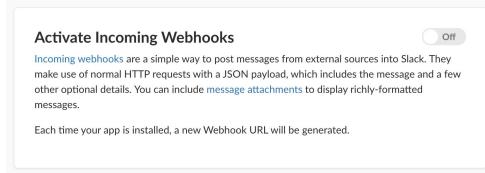
Building Apps for Slack
Create an app that's just for your workspace (or following the steps below.)

Add features and functionality
Choose and configure the tools you'll need to create your app.

Incoming Webhooks
Post messages from external sources into Slack.

Activate incoming webhooks and then **add new webhook to workspace**.

Incoming Webhooks

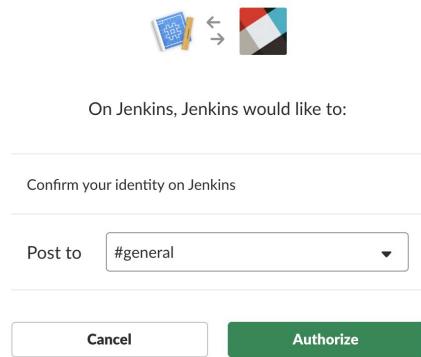


Activate Incoming Webhooks Off

Incoming webhooks are a simple way to post messages from external sources into Slack. They make use of normal HTTP requests with a JSON payload, which includes the message and a few other optional details. You can include [message attachments](#) to display richly-formatted messages.

Each time your app is installed, a new Webhook URL will be generated.

Authorize the App we have created to post **#general** channel in our Slack workspace.



Finally we will have our API token Generated.

Webhook URLs for Your Workspace

To dispatch messages with your webhook URL, send your [message](#) in JSON as the body of an [application/json](#) POST request.

Add this webhook to your workspace below to activate this curl example.

Sample curl request to post to a channel:

```
curl -X POST -H 'Content-type: application/json' --data '{"text":"Hello, World!"'}  
https://hooks.slack.com/services/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
```

Webhook URL	Channel	Added By
1M/BBXR2NS0N/1Xvox150V1ke83TW0YBdem68	#general	Jenkins Jul 24, 2018

Copy the test Url and put it to Vagrant VM cli to check that we can now communicate with Slack.

```
# curl -X POST -H 'Content-type: application/json' --data '{"text":"Hello, World!"'}  
https://hooks.slack.com/services/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
```

Note: that you will be given a URL with a different token when you create your slack app.

Go back to slack channel you are created, You should be able to see the test message.

 Jenkins 2:06 AM
joined #general.
 Jenkins 2:40 AM
added an integration to this channel: Jenkins
 Jenkins APP 2:44 AM
Hello, World!

Hooray, you just made an API call. It is time to integrate it with Jenkins. Create a new function and then update pipeline to send messages on pipeline progress.

```
def run_ls(options, arguments) {  
    ls = sh script:"ls ${options} ${arguments}", returnStdout:true  
    echo "${ls}"  
}  
  
def slack_notification(step) {  
    progress = '='*(25*step)  
    // YOU WILL HAVE TO USE YOUR OWN TOKEN IN URL BELOW  
    sh """curl -X POST -H 'Content-type: application/json' --data '{"text":" Job  
progress: ${progress} ${25*step}%"}'  
https://hooks.slack.com/services/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"""  
}  
  
node('master') {  
    stage('Init') {  
        // Making timestamp at the beginning of the job.  
        timedate_start = sh script:"date", returnStdout:true  
        // checking /tmp directory content  
        run_ls('-l','/tmp')  
  
        // Sending pipeline progress to slack  
        slack_notification(1)  
    }  
    stage('Build') {  
        // Just a placeholder message.  
        echo "We are building something awesome here"  
  
        // Sending pipeline progress to slack  
        slack_notification(2)
```

```

}

stage('Test') {
    // Making timestamp at the beginning of the job.
    input "Please check if we are good to go before we push in to production."


    // Sending pipeline progress to slack
    slack_notification(3)
}

stage('Finalize') {

    // Taking timestamp at the end of the job
    timedate_end = sh script:"date", returnStdout:true

    //echoing the results
    sh "echo Job started at ${timedate_start}"
    sh "echo Job ended at ${timedate_end}"

    // Sending pipeline progress to slack
    slack_notification(4)
}

}

```

We defined a function called **slack_notification()** that takes one parameter called **step** and then it just makes an API call with **slack.com** to send a custom message. Save this code in our Jenkins job, start the build and navigate back to the Slack workspace you have created.



Jenkins APP: 11:53 AM
Job progress: ===== 25%
Job progress: ====== 50%
Job progress: ===== 75%
Job progress: ===== 100%

Now you can invite your friends or colleagues to your slack workspace and start building great automation tools and work together.

Variable + Function() == LOVE

You think we are kidding you, right? Let us prove you wrong.
When you define a variable and set its value to a called function, you can achieve even better results. In the example with Slack messages, we are making an API call to our Slack channel

and we do not know whether the API call was successful or not. This is where can benefit from **variable + function** combination. Take a look at the example below

```
def slack_notification(message) {
    // YOU WILL HAVE TO USE YOUR OWN TOKEN IN URL BELOW
    result = sh script:"""curl -i -X POST -H 'Content-type: application/json' --data
'{"text":"' ${message}"'}'
https://hooks.slack.com/services/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
2>/dev/null | grep HTTP""", returnStdout:true
    return result
}

node('master') {
    stage('Init') {
        // Sending pipeline progress to slack
        response = slack_notification('We are at Init stage')
        echo "${response}"
    }
}
```

You can see that we have a special keyword “**return**” within a function. It returns a value of variable “**result**” back to where we call this function. And we are calling this function and storing its result in “**response**” variable. And then we can do whatever we want with the results.

Now save the pipeline, start the build and check Console Output.

Console Output

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Init)
[Pipeline] sh
[Pipeline] Running shell script
+ grep HTTP
+ curl -i -X POST -H 'Content-type: application/json' --data '{"text":"' We are at Init stage"}'
https://hooks.slack.com/services/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
[Pipeline] echo
HTTP/1.1 200 OK

[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

For loops

Loops are one of the essential components of every programming or scripting language, and groovy is not an exception. What is the use case you may ask? The simplest one would be to update a configuration on all your devices in the network. We need to create a simple pipeline

that simulates this behavior. There are several ways how you can construct a **for loop** in Jenkins. Let's take a look at these examples, so you can pick the one you like and use that structure throughout the book.

Option 1:

```
// Define a list of hosts to work with
list_of_hosts = ['1.1.1.1','8.8.8.8','google.com','redhat.com','cisco.com']

for(host in list_of_hosts) {
    sh "ping -c3 ${host}"
}
```

Note: This is not a pipeline!

This option syntax is quite easy to understand. We have a list of hosts and we want to iterate over it and just ping every host in that list. The syntax of this **for loop** is close to a function syntax thus the easiest to understand. It basically says, “ For every host in list_of_hosts ping that host.

Option 2:

```
// Define a list of hosts to work with
list_of_hosts = ['1.1.1.1','8.8.8.8','google.com','redhat.com','cisco.com']

list_of_hosts.each { item ->
    sh "ping -c3 ${item}"
}
```

Note: This is not a pipeline!

Option 2 has slightly different syntax, that is easy to understand but hard to remember especially with all these arrows **->**.

Option 3:

```
// Define a list of hosts to work with
list_of_hosts = ['1.1.1.1','8.8.8.8','google.com','redhat.com','cisco.com']

for(String host: list_of_hosts) {
    sh "ping -c3 ${host}"
}
```

Note: This is not a pipeline!

Another option that is easy to understand, and close to Option 1, to the exception that we have this “**String**” statement that puts a lot of people to confusion if you have no prior experience with Java.

Option 4:

```
// Define a list of hosts to work with
list_of_hosts = ['1.1.1.1','8.8.8.8','google.com','redhat.com','cisco.com']

for (int i=0 ; i < list_of_hosts.size() ; i++) {
    sh "ping -c3 ${list_of_hosts[i]}"
}
```

Note: This is not a pipeline!

This is the most flexible way. It has a C language syntax to form a **for-loop**. This may be difficult to understand if you've just started learning programming languages.

This is why we stick with **Option 1**, but if you are comfortable with Other options, feel free to use them.

Now modify our pipeline and add **for loops**.

```
// Define a list of hosts to work with
list_of_hosts = ['1.1.1.1','8.8.8.8','google.com','redhat.com','cisco.com']

node('master') {
    stage('Start') {
        // pinging every host in the list
        for(host in list_of_hosts) {
            sh "ping -c3 ${host}"
        }
    }
    stage('Finish') {
        // Generating a new message for every host.
        for(host in list_of_hosts) {
            sh "echo This message is generated for: ${host}"
        }
    }
}
```

Save the pipeline, start the build and check Console Output.

In the **Start** stage, we are running ping command for all the hosts in the list.

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Start)
[Pipeline] sh
[pipeline1] Running shell script
+ ping -c3 1.1.1.1
[Pipeline] sh
[pipeline1] Running shell script
+ ping -c3 8.8.8.8
[Pipeline] sh
[pipeline1] Running shell script
+ ping -c3 google.com
[Pipeline] sh
[pipeline1] Running shell script
+ ping -c3 redhat.com
[Pipeline] sh
[pipeline1] Running shell script
+ ping -c3 cisco.com
[Pipeline] sh
[pipeline1] Running shell script
```

And in the **Finish** stage, we generating messages for the same hosts.

```
[Pipeline] { (Finish)
[Pipeline] sh
[pipeline1] Running shell script
+ echo This message is generated for: 1.1.1.1
This message is generated for: 1.1.1.1
[Pipeline] sh
[pipeline1] Running shell script
+ echo This message is generated for: 8.8.8.8
This message is generated for: 8.8.8.8
[Pipeline] sh
[pipeline1] Running shell script
+ echo This message is generated for: google.com
This message is generated for: google.com
[Pipeline] sh
[pipeline1] Running shell script
+ echo This message is generated for: redhat.com
This message is generated for: redhat.com
[Pipeline] sh
[pipeline1] Running shell script
+ echo This message is generated for: cisco.com
This message is generated for: cisco.com
```

This pipeline does a great job automating routine tasks when you need to work with a large number of devices. But if one of the hosts is going to be unavailable, then everything breaks. How do we fix it? This is where error handling with **try, catch, finally** comes handy.

Try, catch, finally

The concept of **try, catch, finally** is not new as well and quite often used in conjunction with for loops and **if, else** statements that we cover in the following section. The structure of **try, catch, finally** is similar to what we have seen already:

```
try {
```

```

<INSERT YOUR CODE HERE>
} catch ( err ) {
    <INSERT YOUR CODE HERE>
} finally {
    <INSERT YOUR CODE HERE>
}

```

*Note: **catch** and **finally** blocks are optional.*

Let's try a simple example.

```

node('master') {
    stage('try_catch_fin') {
        try {
            echo "This code is being run"
        } catch ( err ) {
            echo "If an error is caught this block, this block is executed"
        } finally {
            echo "This code is always executed"
        }
    }
}

```

Save this code in our Jenkins job, start the build and check Console Output.

Console Output

```

Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] echo
This code is being run
[Pipeline] echo
This code is executed always
[Pipeline] End of Pipeline
Finished: SUCCESS

```

In this example only **try** and **finally** blocks are being executed because there are no errors. Let's update the code to simulate the failure.

```

node('master') {
    stage('try_catch_fin') {
        try {
            // simulating error
            sh "false"

            echo "This code is being run"
        }
    }
}

```

```

        } catch ( err ) {
            echo "If an error is caught this block, this block is executed "
        } finally {
            echo "This code is executed always"
        }
    }
}

```

Save this code, run the build and go to Console Output.

Console Output

```

Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (try_catch_fin)
[Pipeline] sh
[Pipeline] Running shell script
+ false
[Pipeline] echo
If error is caught this block, this block is executed
[Pipeline] echo
This code is executed always
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

In this output, we can see that the error is caught in **try** block, and then **catch** block is executed, and then **finally** block is executed in any case. This helps to handle any errors and change your code behavior.

If, else if, else

Try, catch, finally are good where we need to handle an error, but if we need to change code behavior, then we will need to use **if, else** statements. This is how the syntax looks like:

```

If ( condition) {
    <INSERT YOUR CODE HERE>
}

```

The code between “{“ and “}” will be executed if the condition is satisfied.

Here is a simple example:

```

node('master') {
    stage('if_simple') {
        status = sh script:"ping -c3 google.com",returnStatus:true
        if (status == 0) {
            echo "ping to host was successful"
        }
    }
}

```

```
        }
    }
}
```

Save this code, run the build and go to Console Output.

Console Output

```
Started by user Jenkins User
Replayed #3
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (if_simple)
[Pipeline] sh
[pipeline1] Running shell script
+ ping -c3 google.com
[Pipeline] echo
ping to host was successful
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Sometimes it is required to handle complex conditions. This can be achieved by “if-else”. The full syntax of it looks like below:

```
If ( condition) {
    <INSERT YOUR CODE HERE>
} else if (condition) {
    <INSERT YOUR CODE HERE>
} else {
    <INSERT YOUR CODE HERE>
}
```

Note: “else if” and “else” blocks are optional

A quick example below shows **if-else** statements in action

```
node('master') {
    stage('if_else') {
        status = sh script:"ping -c3 google.com",returnStatus:true
    }
}
```

```

        if (status == 0) {
            echo "ping to host was successful"
        } else if (status == 1) {
            echo "ping to host was NOT successful"
        } else {
            echo "There was some other error with status code: ${status}"
        }
    }
}

```

Save this code, run the build and go to Console Output.



Console Output

```

Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (if_else)
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 google.com
[Pipeline] echo
ping to host was successful
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

You see it says “**ping to host was successful**” because ping command returned status code 0, which means that the command ran successfully with no errors. Change google.com to something that does not exist (e.g. **google1.com**) and run the pipeline one more time. You should see the following.

```

Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (if_else)
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 google1.com
[Pipeline] echo
There was some other error with status code: 2
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

We can make **if else** to work with **for loop** statement.

```
list_of_hosts = ['1.1.1.1', 'juniper.net', 'google1.com', 'jenkins.io',
'google.com', 'redhat.com', 'cisco.com']

node('master') {
    stage('if_else') {
        for(host in list_of_hosts) {
            status = sh script:"ping -c3 ${host}",returnStatus:true
            if (status == 0) {
                echo "ping to ${host} was successful"
            } else if (status == 1) {
                echo "ping to ${host} was NOT successful"
            } else {
                echo "There was some other error while pinging ${host}. Exit status
code: ${status}"
            }
        }
    }
}
```

Save this code, run the build and go to Console Output.



Console Output

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (if_else)
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 1.1.1.1
[Pipeline] echo
ping to 1.1.1.1 was successful
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 juniper.net
[Pipeline] echo
ping to juniper.net was NOT successful
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 google1.com
[Pipeline] echo
There was some other error while pinging google1.com. Exit status code: 2
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 jenkins.io
[Pipeline] echo
ping to jenkins.io was NOT successful
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 google.com
[Pipeline] echo
ping to google.com was successful
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 redhat.com
[Pipeline] echo
ping to redhat.com was successful
[Pipeline] sh
[Pipeline] Running shell script
+ ping -c3 cisco.com
[Pipeline] echo
ping to cisco.com was successful
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

See that different hosts are getting different results and **if else** conditional statements are working properly.

Switch case

There is another way how to make conditional statements like **if else**. It is called **switch case** which in many cases better than **if else** statements. We are not going to deep dive into the difference on a low level, but rather show you how it works.

Switch case syntax looks like the following:

```
switch (<variable or value>) {
```

```

case <condition or value>:
    <PLACE YOUR CODE HERE>
    break
case <condition or value>:
    <PLACE YOUR CODE HERE>
    break
default:
    <PLACE YOUR CODE HERE>
    break
}

```

The syntax is very straightforward. In the beginning, we store variable or value inside round braces () of **switch** statement, and then we check if that value matches the conditions or values in **case** statements. If there is a match, the code inside the curly braces is executed, if there is no match, then **default** statement is executed.

Take a look at the simple example.

```

switch ('just_a_string') {
    case '1.1.1.1':
        echo "It is an IP address"
        break
    case ['cisco.com','juniper.net']:
        echo "It is cisco.com or juniper.net"
        break
    default:
        echo "It is something else"
        break
}

```

Note: *This is not a pipeline!*

In this example, we are matching “**just_a_string**” value inside **switch** statement with all the values inside the **case** statements, and since there is no match, then **default** statement is being used. Check out the Console Output once you save and run this code in Jenkins.



Console Output

```

Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] echo
It is something else
[Pipeline] End of Pipeline
Finished: SUCCESS

```

Check how it works with **for loop** statement.

```
list_of_hosts = ['1.1.1.1', 'juniper.net', 'google1.com', 'jenkins.io',
'google.com', 'redhat.com', 'cisco.com']

for(host in list_of_hosts) {
    switch (host) {
        case '1.1.1.1':
            echo "It is an IP address"
            break
        case ['cisco.com','juniper.net']:
            echo "It is cisco.com or juniper.net"
            break
        default:
            echo "It is something else"
            break
    }
}
```

Here we define a **for loop** statement that evaluates every element in the list **list_of_hosts** against the values inside of **case** statements. Console Output shows us the following.



Console Output

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] echo
It is an IP address
[Pipeline] echo
It is cisco.com or juniper.net
[Pipeline] echo
It is something else
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Though it is quite easy to use **switch case** statements, still **if else** statement is predominant in conditionals in pretty much any programming or scripting language out there.

Retry

Retry is a good option when you want to run some specific step or the whole stage several times before you consider it failed. It is quite useful when you are using unreliable links to execute some commands, or you are making an API call to a system that fails from time to time for an unknown reason (in most of the cases it means that their code suck, but no one never admits that). The structure of **retry** statement is quite simple.

```
retry (<max times to retry>){  
    <PLACE YOUR CODE HERE>  
}
```

First, create a simple pipeline that has a 33% chance to fail while running.

```
// doing some magic to enable randomizer  
Random rand = new Random()  
my_list = [true,true,false]  
  
list_of_hosts = ['1.1.1.1', 'juniper.net', 'google1.com', 'jenkins.io',  
'google.com', 'redhat.com', 'cisco.com']  
  
node('master') {  
    stage('no_retry') {  
        for(host in list_of_hosts) {  
            index = rand.nextInt(3)  
            echo "Checking ${host}"  
            sh "${my_list[index]}"  
            echo "${host} is alive"  
        }  
    }  
}
```

It may give us the following output.

Console Output

```
Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (retry)
[Pipeline] echo
Checking 1.1.1.1
[Pipeline] sh
[pipeline1] Running shell script
+ true
[Pipeline] echo
1.1.1.1 is alive
[Pipeline] echo
Checking juniper.net
[Pipeline] sh
[pipeline1] Running shell script
+ true
[Pipeline] echo
juniper.net is alive
[Pipeline] echo
Checking google1.com
[Pipeline] sh
[pipeline1] Running shell script
+ true
[Pipeline] echo
google1.com is alive
[Pipeline] echo
Checking jenkins.io
[Pipeline] sh
[pipeline1] Running shell script
+ true
[Pipeline] echo
jenkins.io is alive
[Pipeline] echo
Checking google.com
[Pipeline] sh
[pipeline1] Running shell script
+ false
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
ERROR: script returned exit code 1
Finished: FAILURE
```

Note: Since there is a 33% chance to fail, you may have the completely different result. In my case, I was extremely lucky to get 4 true results in a row.

Now we inject **retry** part.

```
// doing some magic to enable randomizer
Random rand = new Random()
my_list = [true,true,false]

list_of_hosts = ['1.1.1.1', 'juniper.net', 'google1.com', 'jenkins.io',
'google.com', 'redhat.com', 'cisco.com']
```

```

node('master') {
    stage('retry') {
        for(host in list_of_hosts) {
            // injecting retry
            retry(5) {
                index = rand.nextInt(3)
                echo "Checking ${host}"
                sh "${my_list[index]}"
                echo "${host} is alive"
            }
        }
    }
}

```

Run the pipeline.

Console Output

```

Started by user Jenkins User
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (retry)
[Pipeline] retry
[Pipeline] {
[Pipeline] echo
Checking 1.1.1.1
[Pipeline] sh
[Pipeline] Running shell script
+ false
[Pipeline] }
ERROR: script returned exit code 1
Retrying
[Pipeline] {
[Pipeline] echo
Checking 1.1.1.1
[Pipeline] sh
[Pipeline] Running shell script
+ true
[Pipeline] echo
1.1.1.1 is alive
[Pipeline] }
[Pipeline] // retry
[Pipeline] retry
[Pipeline] {
[Pipeline] echo
Checking juniper.net
[Pipeline] sh
[Pipeline] Running shell script
+ false
[Pipeline] }
ERROR: script returned exit code 1
Retrying

```

<OUTPUT IS OMITTED >

```

cisco.com is alive
[Pipeline] }
[Pipeline] // retry
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

Once we got a **false**, retry automatically tries to return the code in curly braces {} for the number of times we specify in the round braces () .

Parallel

So far we did all the works in a serial manner, which means that all the steps in any stage are going to be executed one after another. But sometimes we need to run several steps or stages in parallel. One of the examples is in testing environments you need to run traffic generator and check devices under the tests for specific output simultaneously. This is where parallel plays a great role. The syntax of **parallel** statement is easy to understand.

```

parallel(
    <stage_name>: {
        <PLACE YOUR CODE HERE>
    },
    <stage_name>: {
        <PLACE YOUR CODE HERE>
    }
)

```

In the pipeline below we run ping commands in parallel. The first section is going to run one ping command for 10 seconds, and another section is going to run 5 ping commands for ~2 seconds each.

```

list_of_hosts = ['1.1.1.1','8.8.8.8','google.com','redhat.com','cisco.com']

node('master') {
    stage('parallel') {
        parallel(
            'generate some traffic': {
                sh "ping -c 10 8.8.8.8"
            },
            'running tests': {

```

```

        for(host in list_of_hosts) {
            sh "ping -c 2 ${host}"
        }
    )
}
}

```

If we would be running these both sections in sequence, the total time would be ~20sec.
In our case, the whole job takes 10 seconds.

Stage View



Though in the job logs we see that a summary is more than 10sec.



Using a different combination of pipeline advanced syntax, you will be able to cover 75% of all the tasks in your day to day DevOps routine. So, where do you get the rest 25%? We are going to leave this for the books we are about to release because it goes beyond the basic and advanced techniques.

Configuring Jenkins

We know how to create and write powerful pipelines in Jenkins, but we have not touch Jenkins configuration at all. In this section we work with Jenkins Plugins, learn how to add users, and create new slaves.

Installing plugins

Jenkins is a powerful CI/CD tool that is being widely used all around the globe. While, in our opinion, the software itself is purely written, full of bugs and features, and does not have a comprehensive documentation, it's still being the most popular CI/CD tool in the world. It is all because of Jenkins community support and thousands of plugins available. Jenkins has plugins for pretty much everything you could imagine. And if there is no plugin you are looking for, you can always write your own. Jenkins plugins are written in Java, so this is required, but it is never too late to learn something new.

We are not going to teach you how to write plugins because it would be way too advanced, but rather show you how to install and use some of the plugins we use in this book.

Installing the plugins is actually the easiest part of all when it comes to Jenkins. Our very first plugin is going to be a one-click installation and going to bring huge Jenkins user experience, and you will see why in a moment. First.

From the Jenkins dashboard at <http://jenkins.example.com:8080/>, navigate to “**Manage Jenkins**”



And then go to “**Manage plugins**”

Manage Jenkins

- Configure System**
Configure global settings and paths.
- Configure Global Security**
Secure Jenkins; define who is allowed to access/use the system.
- Configure Credentials**
Configure the credential providers and types
- Global Tool Configuration**
Configure tools, their locations and automatic installers.
- Reload Configuration from Disk**
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly on disk.
- Manage Plugins**
Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
⚠ There are updates available

Here we have 4 mains tabs we work with:

- **Updates** - Here you can see newer versions available for the plugins that are already installed.
- **Available** - List of plugins available to install on your Jenkins server.
- **Installed** - List of all the plugins installed on your Jenkins Server.
- **Advanced** - Misc settings such as proxy setup.

The screenshot shows the Jenkins Update Center interface. On the left, there are navigation links: 'Back to Dashboard', 'Manage Jenkins', and 'Update Center'. The main area has tabs at the top: 'Updates' (which is selected), 'Available', 'Installed', and 'Advanced'. Below the tabs is a table with columns: 'Install', 'Name ↓', 'Version', and 'Installed'. There are two rows in the table:

Install	Name ↓	Version	Installed
<input type="checkbox"/> Git client	Utility plugin for Git support in Jenkins	2.7.3	2.7.2
<input type="checkbox"/> Pipeline: API	Plugin that defines Pipeline API.	2.29	2.28

A 'Filter:' search bar is located at the top right of the table area.

Now, let's take a closer look at the first 3 tabs.

Updates

On the tab called **Updates**, we can select and update currently installed plugins to the latest version. But do not run to update all your plugins if there is a new release available. A lot of these plugins are written by the community and have dependencies. So there is no guarantee that the update will work better. Luckily, we can rollback an upgrade, we are going to talk about this in a bit.

The screenshot shows the Jenkins Update Center 'Updates' tab. At the top, there are tabs: 'Updates' (selected), 'Available', 'Installed', and 'Advanced'. Below the tabs is a table with columns: 'Install', 'Name ↓', 'Version', and 'Installed'. There are two rows in the table:

Install	Name ↓	Version	Installed
<input type="checkbox"/> Git client	Utility plugin for Git support in Jenkins	2.7.3	2.7.2
<input type="checkbox"/> Pipeline: API	Plugin that defines Pipeline API.	2.29	2.28

At the bottom of the table area, there are three buttons: 'Download now and install after restart', 'Update information obtained: 2.7 sec ago', and 'Check now'.

Before you update any installed plugin, always check plugin changelog and release notes. You can do that by clicking on the plugin name. That will redirect you to the plugin page.

If it happens that you install a plugin and it does not work as expected or break something up, you can rollback to the old version on the **Installed** tab.

A screenshot of the Jenkins Plugin Manager interface. At the top, there's a search bar with the text 'Git client plugin'. Below it, a card shows the plugin details: 'Utility plugin for Git support in Jenkins' (version 2.7.3). There are buttons for 'Downgrade to 2.7.2' and 'Uninstall'. A checkmark icon is present on the left of the card.

Available

You can install all the plugins here. There is a quite a lot of plugins available, over 1500 at the time of writing this book. How do we find the plugin you need? We can use **filter** text area to narrow down your search or you can go to <https://plugins.jenkins.io/> and search a plugin by using different criteria available for you.

A screenshot of the Jenkins Plugins Index page. The header features a cartoon character and the title 'Plugins Index'. Below it, a sub-header reads 'Discover the 1000+ community contributed Jenkins plugins to support building, deploying and automating any project.' The page includes a 'Browse' dropdown, a search bar with a magnifying glass icon, and a sidebar with sorting options ('Sort relevance') and categories ('Categories'). The main content area lists various plugin categories with checkboxes.

Categories	Sub-Categories
<input type="checkbox"/> Platforms	<input type="checkbox"/> Administration
<input type="checkbox"/> iOS development	<input type="checkbox"/> Agent controllers
<input type="checkbox"/> .NET	<input type="checkbox"/> Page decorators
<input type="checkbox"/> Android development	<input type="checkbox"/> Users and security
<input type="checkbox"/> Ruby development	<input type="checkbox"/> Cluster management
<input type="checkbox"/> TestWeaver	<input type="checkbox"/> CLI extensions
<input type="checkbox"/> User interface	<input type="checkbox"/> Command Line Steps
<input type="checkbox"/> User Interface	<input type="checkbox"/> Configuration Management
<input type="checkbox"/> List view column plugins	<input type="checkbox"/> Source code management
<input type="checkbox"/> Our Gender Survey Standalone	<input type="checkbox"/> SCM connections
<input type="checkbox"/> Sonar Linter	<input type="checkbox"/> Application Security on Cloud
<input type="checkbox"/> AWS Global Configuration	<input type="checkbox"/> SCM related
	<input type="checkbox"/> Git
	<input type="checkbox"/> GitHub Notifications
	<input type="checkbox"/> Artifactory
	<input type="checkbox"/> Artifactory API
	<input type="checkbox"/> Artifactory Alternative E
	<input type="checkbox"/> Build reports
	<input type="checkbox"/> Artifact uploaders

We need to install a couple of plugins, we were able to find one by using **filter** text area in **Plugin Manager** and another use by sorting by the **relevance** in **user interface** plugins.

Filter:

Updates	Available	Installed	Advanced
Install ↓		Name	Version
<input checked="" type="checkbox"/> Blue Ocean BlueOcean Aggregator			1.7.1
<input checked="" type="checkbox"/> GitLab This plugin integrates GitLab to Jenkins by faking a GitLab CI Server.			1.5.9

Once you select both plugins, click on **Download now and install after restart**. You will be redirected to the installation progress page. Scroll all the way down and select **Restart Jenkins when installation is complete and no jobs are running**.

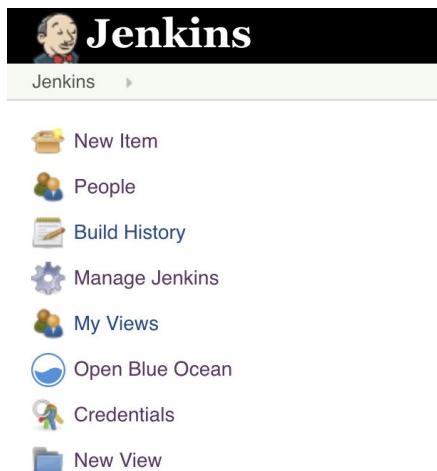
→ [Go back to the top page](#)
 (you can start using the installed plugins right away)

→ Restart Jenkins when installation is complete and no jobs are running

Wait for Jenkins to reboot and come back to the operational state.

Blue Ocean is the most popular Jenkins user experience plugin and GitLab plugin is a plugin that allows smooth integration with GitLab that we integrate within the next Chapter.

First, let us show you the difference between **Standard** and **Blue Ocean views**. Navigate back to Jenkins dashboard at <http://jenkins.example.com:8080/> and click on **Open Blue Ocean**.



You will see the Jenkins jobs dashboard.

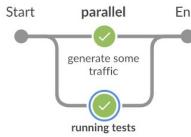
The screenshot shows the Jenkins Pipelines interface. At the top, there's a navigation bar with tabs for 'Pipelines' (which is active), 'Administration', and a user icon. Below the navigation is a search bar with the placeholder 'Pipelines' and a magnifying glass icon. To the right of the search bar is a 'New Pipeline' button. The main content area has a table with one row. The columns are labeled 'NAME', 'HEALTH', 'BRANCHES', and 'PR'. The row contains the value 'pipeline1' under 'NAME', a cloud icon under 'HEALTH', and two minus signs under 'BRANCHES' and 'PR'. There's also a star icon to the right of the row.

We have only one pipeline so far. Now click on **pipeline1**. That will redirect you to **pipeline1** jobs history.

The screenshot shows the Jenkins pipeline1 job history page. At the top, there's a header with the pipeline name 'pipeline1' and icons for a cloud, a star, and a gear. Below the header are three navigation links: 'Activity' (which is active), 'Branches', and 'Pull Requests'. The main content area has a table with eight rows of data. The columns are labeled 'STATUS', 'RUN', 'COMMIT', 'MESSAGE', 'DURATION', and 'COMPLETED'. Each row contains a status icon (green checkmark, green circle, red X, green circle, red X, green checkmark, red X), a run number (127, 126, 125, 124, 123, 122, 121), a commit hash ('-' for all), a message ('Started by user Jenkins User' for all), a duration ('10s', '4s', '2s', '5s', '2s', '3s', '<1s'), and a completed time ('7 hours ago', '21 hours ago', 'a day ago'). Each row also has a circular refresh icon to its right.

STATUS	RUN	COMMIT	MESSAGE	DURATION	COMPLETED
✓	127	—	Started by user Jenkins User	10s	7 hours ago
✓	126	—	Started by user Jenkins User	4s	21 hours ago
✗	125	—	Started by user Jenkins User	2s	a day ago
✓	124	—	Started by user Jenkins User	5s	a day ago
✗	123	—	Started by user Jenkins User	2s	a day ago
✓	122	—	Started by user Jenkins User	3s	a day ago
✗	121	—	Started by user Jenkins User	<1s	a day ago

What an improvement compared to the standard view. On this page, we can see the Jenkins jobs summary. Click on the very top one. This should be the job where we used **parallel** statement.



It looks way better than **Standard** view and increases user experience.

Blue Ocean plugin is very powerful, and it can be a separate book written just on Blue Ocean. So we are not going to cover all the functionality, but rather show you the power of Jenkins plugins. You can just click on this icon and that will get you back to the standard view.

We are also going to cover gitlab plugin configuration and integration pieces in the following Chapter.

Installed

Go back to the **Plugin Manager** at <http://jenkins.example.com:8080/pluginManager/> and choose Installed tab. This is where you will be able to see all the plugin installed on this Jenkins server. You can also disable and uninstall the plugins from here, and sometimes you can downgrade plugins to previous versions if there is a need.

Filter: <input type="text"/>					
Updates	Available	Installed	Advanced		
Enabled	Name ↓	Version	Previously installed version	Uninstall	
	Ant Plugin				
<input checked="" type="checkbox"/>	Adds Apache Ant support to Jenkins	1.8		Uninstall	
	Apache HttpComponents Client 4.x API Plugin				
<input checked="" type="checkbox"/>	Bundles Apache HttpComponents Client 4.x and allows it to be used by Jenkins plugins.	4.5.5-3.0		Uninstall	

You can see that some plugins you can disable, and some of them you can not. The reason being is that some plugins have dependants, meaning that some plugins are dependant on this plugin and you will need to uninstall them first. You can hover the mouse cursor over the checkbox and hold it for a couple of seconds before the help box appears on the screen.

[Apache HttpComponents Client 4.x API Plugin](#)

Bundles Apache HttpComponents Client 4.x and allows it to be used by Jenkins plugins. [4.5.5-3.0](#) [Uninstall](#)

This plugin cannot be disabled
It has one or more enabled dependants.

[GitHub Pipeline for Blue Ocean](#) [Bitbucket Branch Source Plugin](#)
[Git client plugin](#) [Bitbucket Pipeline for Blue Ocean](#)
[JIRA Integration for Blue Ocean](#) [Pipeline: Basic Steps](#) [JIRA plugin](#)

Adding new users

In Jenkins, we have two distinct entities - **users** and **credentials**.

- **Credentials** - Jenkins CI/CD communicates with different devices and application that usually require different types of credentials like username and password, SSH Keys, API tokens, CA certificates, etc. Credentials is that secret store to keep all the external credentials securely.
- **Users** - Internal store to manage Jenkins users. This is Jenkins' own user database.

We will work with credentials in the next Chapter.

In order to add a user form Jenkins Homepage at <http://jenkins.example.com:8080/>, navigate to **Manage Jenkins**. Scroll all the way down and then click on **Manage Users**.



Manage Old Data

Scrub configuration files to remove remnants from old plugins and earlier versions.



Manage Users

Create/delete/modify users that can log in to this Jenkins



In-process Script Approval

Allows a Jenkins administrator to review proposed scripts (written e.g. in Groovy) which run inside the Jenkins process and so could bypass security restrictions.

They have hidden it very well. Next click on **Create User**.

The screenshot shows the Jenkins dashboard with a dark header featuring the Jenkins logo and the word "Jenkins". Below the header, there is a breadcrumb navigation: "Jenkins" followed by a right-pointing arrow and "Jenkins' own user database". On the left side, there is a sidebar with three items: "Back to Dashboard" (with an upward arrow icon), "Manage Jenkins" (with a gear icon), and "Create User" (with a user icon).



Back to Dashboard



Manage Jenkins



Create User

That will give you a simple Jenkins user registration page. Create a user called **gitlab** filling all the fields and press **Create User**. Use “**DevOps123**” as password.

Create User

Username:	<input type="text" value="gitlab"/>
Password:	<input type="password" value="....."/>
Confirm password:	<input type="password" value="....."/>
Full name:	<input type="text" value="Gitlab User"/>
E-mail address:	<input type="text" value="gitlab@example.com"/>

Create User

Now we can log out and login as gitlab user to verify the credentials. Press **log out** at the top-right side of the screen.

Jenkins User | log out

Then log in back using **gitlab** username and password.



Welcome to Jenkins!

gitlab

.....

Keep me signed in

By default, there is no difference between users in Jenkins. They all have full control over Jenkins server. You can verify this by navigating to **Manage Jenkins** from the home page at <http://jenkins.example.com:8080/> and then choosing **Configure Global Security**.

Manage Jenkins



Configure System

Configure global settings and paths.



Configure Global Security

Secure Jenkins; define who is allowed to access/use the system.

You will see plenty of settings, but the default ones are to use **Jenkins' own user database** and **Logged-in users can do anything**.



Configure Global Security

Enable security



Disable remember me



Access Control

Security Realm

Delegate to servlet container



Jenkins' own user database



Allow users to sign up



LDAP



Unix user/group database



Authorization

Anyone can do anything



Legacy mode



Logged-in users can do anything



Allow anonymous read access



Matrix-based security



Project-based Matrix Authorization Strategy



Jenkins security goes beyond the scope of this book, and we will cover this topic in our future books and video classes.

Jenkins and GitLab API

This chapter explains how to work with GitLab and Jenkins APIs. We encourage you to study this Chapter, though you may skip it if it is not a subject of interest for you. This chapter allows you to understand Jenkins or GitLab better on a low level.

Why API?

Most of the modern applications provide HTTP-based APIs which allow communicating with application programmatically. It allows creating a Python-based application or even a simple bash script which will trigger Jobs in Jenkins or create/delete GitLab entities. API is a simple way of software integration. We are not going to cover everything, but rather give you a basic level knowledge which you can use in your current or future projects to build something awesome.

API access tools

Using curl to access APIs

We show you how to access HTTP-based APIs using CLI via **curl** utility, a well-known Linux tool which comes with every Linux installation. Curl accepts URL and returns server answer. Here is a curl example to display projects in GitLab.

```
[vagrant@jenkins ~]$ curl http://gitlab.example.com/api/v4/projects
[{"id":1,"description":"The first GitLab
Repo","name":"first_repo","name_with_namespace":"Administrator /
first_repo","path":"first_repo","path_with_namespace":"root/first_repo","cr
eated_at":"2018-09-16T19:21:50.385Z","default_branch":"master","tag_list":[]
],"ssh_url_to_repo":"git@gitlab.example.com:root/first_repo.git","http_url_
to_repo":"http://gitlab.example.com/root/first_repo.git","web_url":"http://
gitlab.example.com/root/first_repo","readme_url":"http://gitlab.example.com
/root/first_repo/blob/master/README.md","avatar_url":null,"star_count":0,"f
orks_count":0,"last_activity_at":"2018-09-16T20:24:52.208Z","namespace":{"i
d":1,"name":"root","path":"root","kind":"user","full_path":"root","parent_i
d":null}]}]
```

You can also access **API** using different tools and script languages. **Curl** is one of the simplest ways to send and receive data via HTTP/HTTPS.

The **curl** utility supports **GET**, **POST**, **PUT** and **DELETE** request types which are leveraged by GitLab and Jenkins APIs to manage entities. Usually, we use **GET** to receive information, **POST**

update and create entries and **DELETE** to delete an entity. Curl sets request type to **GET** by default (if no HTTP methods are provided). The following examples are equivalents:

```
$ curl http://gitlab.example.com/api/v4/projects  
$ curl -X GET http://gitlab.example.com/api/v4/projects
```

Both GitLab and Jenkins accept and return JSON objects. The output above is quite difficult to follow. For this reason, we recommend using **json.tool** python method to display output in a human-readable format like shown below:

```
[vagrant@jenkins ~]$ curl http://gitlab.example.com/api/v4/projects |  
python -m json.tool  
% Total    % Received % Xferd  Average Speed   Time     Time     Time  
Current                                         Dload  Upload   Total   Spent   Left  
Speed  
100    716  100    716      0       0  11866      0  --::--  --::--  --::--  
11933  
[  
 {  
   "avatar_url": null,  
   "created_at": "2018-09-16T19:21:50.385Z",  
   "default_branch": "master",  
   "description": "The first GitLab Repo",  
<OUTPUT OMITTED>  
   "web_url": "http://gitlab.example.com/root/first_repo"  
 }  
 ]
```

Now data is sorted and readable. Note that **json.tool** doesn't change the data, it rather represents json objects properly using indentations which is more appealing for a human eye.

The example above is well formatted but still contains some unwanted data - curl download statistics. You can remove that by using "**--silent**" curl option:

```
$ curl --silent http://gitlab.example.com/api/v4/projects | python -m  
json.tool  
[  
 {  
   "avatar_url": null,  
   "created_at": "2018-09-16T19:21:50.385Z",  
   "default_branch": "master",  
   "description": "The first GitLab Repo",
```

```
<OUTPUT OMITTED>
    "web_url": "http://gitlab.example.com/root/first_repo"
}
]
```

So, now the output is well formatted and doesn't contain any unwanted information. We will use option “**curl --silent**” in all examples to avoid showing curl download statistics.

Sometimes, we need to understand HTTP return code as well as some other important debug information. This is available using “**--verbose**” curl option:

```
$ curl --silent --verbose http://gitlab.example.com/api/v4/users
* Trying 172.24.0.11...
* TCP_NODELAY set
* Connected to gitlab.example.com (172.24.0.11) port 80 (#0)
> GET /api/v4/users HTTP/1.1
> Host: gitlab.example.com
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 403 Forbidden
< Server: nginx
<OUTPUT OMITTED>
<
* Connection #0 to host gitlab.example.com left intact
{"message":"403 Forbidden - Not authorized to access /api/v4/users"}
```

*Note: The example above returns **403 Forbidden - Not Authorized**.*

Formatting output using jq

As we mentioned before “**python -m json.tool**” allows formatting output and makes it human-readable. There is a tool named “**jq**” which can be used to format output and access JSON data in a smooth way.

The “**jq**” tool can be installed on CentOS 7 as shown below:

```
$ sudo yum install epel-release -y
<OUTPUT OMITTED>
$ sudo yum install jq -y
<OUTPUT OMITTED>
```

*Note: If you are using Mac, you can install running “**brew install jq**” command. Refer to <https://stedolan.github.io/jq/download/> for installing **Jq** on other operating systems.*

Here is the usage example:

```
$ curl --silent http://gitlab.example.com/api/v4/projects | jq
```

```
[  
  {  
    "id": 1,  
    "description": "The first GitLab Repo",  
    "name": "first_repo",  
    "name_with_namespace": "Administrator / first_repo",  
    "path": "first_repo",  
    "path_with_namespace": "root/first_repo",  
    "created_at": "2018-09-16T19:21:50.385Z",  
    "default_branch": "master",  
    "tag_list": [],  
    "ssh_url_to_repo": "git@gitlab.example.com:root/first_repo.git",  
    "http_url_to_repo": "http://gitlab.example.com/root/first_repo.git",  
    "web_url": "http://gitlab.example.com/root/first_repo",  
    "readme_url": "http://gitlab.example.com/root/first_repo/blob/master/README.md",  
    "avatar_url": null,  
    "star_count": 0,  
    "forks_count": 0,  
    "last_activity_at": "2018-09-16T20:24:52.208Z",  
    "namespace": {  
      "id": 1,  
      "name": "root",  
      "path": "root",  
      "kind": "user",  
      "full_path": "root",  
      "parent_id": null  
    }  
  }  
]
```

Note: **python -m json.tool** is not required anymore since **jq** formats output properly.

The “**jq**” utility allows to perform JSON queries and access only data items we need. This great feature treats JSON as a database and asks you to write a query to access data items. For example, if we need only `ssh_url_to_repo` property, we may access it using the following example:

```
$ curl --silent http://gitlab.example.com/api/v4/projects | jq  
".[0].ssh_url_to_repo"  
"git@gitlab.example.com:root/first_repo.git"
```

Do not memorize **jq** syntax, but rather try to understand the way it works. In short, it is similar to how you would parse lists and dictionaries in python, using index numbers and key names. Check **jq** documentation for more: <https://stedolan.github.io/jq/tutorial/>

Using GitLab API

GitLab API

GitLab comes with well-described API which allows performing all activities related to GitLab repository management. You can always refer to "<https://docs.gitlab.com/ce/api/>" or <https://docs.gitlab.com/ee/api/> (depending on GitLab version) to have the latest API documentation.

GitLab exposes an HTTP-based API which is usually accessible via "/api/v4/<ENTITY TYPE>", where "v4" is the currently supported API version.

Authentication

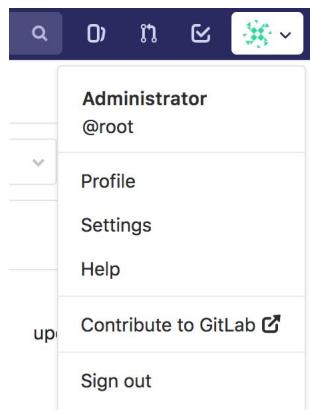
Most API requests require authentication, or will only return public data when authentication is not provided. For those cases where it is not required, this will be mentioned in the documentation for each individual endpoint. For example, the /projects/endpoint.

There are a few GitLab authentication methods. We will be using personal access tokens which is the most simple way to access GitLab API objects.

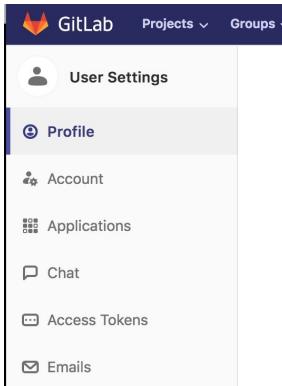
A token is a string which stores sets of characters (for example, "9koXpg98eAheJpvBs5tK"). This string can be used to perform authentication.

Note: **token IS NOT a password.** You still need a password to access GitLab UI but you may use token to work with API directly.

First, you need to access GitLab UI using your working account. We want to perform API activities using the root account. Once you are logged in, click on the user icon on the top right corner.



In the menu above click on Settings and then **Access Tokens**:



In the “**Personal Access Tokens**” window gives a new token a name and specify scope “**api**” and press “**Create personal access token**”.

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Add a personal access token

Pick a name for the application, and we'll give you a unique personal access token.

Name

Expires at

Scopes

api
Grants complete read/write access to the API, including all groups and projects.

read_user
Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.

sudo
Grants permission to perform API actions as any user in the system, when authenticated as an admin user.

read_repository
Grants read-only access to repositories on private projects using Git-over-HTTP (not using the API).

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Your New Personal Access Token

1X-TyHMnB5WFGdDxpwmr



Make sure you save it - you won't be able to access it again.

Add a personal access token

Pick a name for the application, and we'll give you a unique personal access token.

Once you clicked on “Create personal access token” GitLab shows the token on the screen. Don’t forget to store it somewhere since there will be no possibilities to access it again. In our example, “**1X-TyHMnB5WFGdDxpwmr**” is our personal access token for the user “**root**”.

Now we have the root personal access token. First, access some data available only for GitLab administrators without using token:

```
$ curl --silent http://gitlab.example.com/api/v4/users
{"message": "403 Forbidden - Not authorized to access /api/v4/users"}
```

This is an expected output because an unauthorized user is not allowed to access GitLab user information for security reasons.

Previously, we created a GitLab token. Now we can use it to access user information. There are 2 types how we can pass access token to GitLab:

- As part of URL
- As part of Headers

First, we want to pass token as a part of URL:

```
$ curl --silent
http://gitlab.example.com/api/v4/users?private_token=1X-TyHMnB5WFGdDxpwmr | jq
[
  {
    "avatar_url":
    "https://www.gravatar.com/avatar/b58996c504c5638798eb6b511e6f49af?s=80&d=identicon",
    "bio": null,
    "can_create_group": true,
    ...
<OUTPUT OMITTED>
  ...
]
```

Note: User “root” is a GitLab administrator. Once it is authentication against API, all information will be accessible using API. As you can see now, curl returned some information about users which tells us that token worked fine.

The next example demonstrates how to pass token in HTTP headers:

```
$ curl --silent --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/users | jq
[
  {
    "avatar_url":
    "https://www.gravatar.com/avatar/b58996c504c5638798eb6b511e6f49af?s=80&d=identicon",
    "bio": null,
    "can_create_group": true,
    ...
<OUTPUT OMITTED>
  ...
]
```

It worked again! All next examples will use private access tokens to work with API.

API Status codes

GitLab API returns various status codes depending on items accessed. The following table gives an overview of how the API functions generally behave.

Request type	Description
GET	Access one or more resources and return the result as JSON.
POST	Return 201 Created if the resource is successfully created and return the newly created resource as JSON.
GET / PUT	Return 200 OK if the resource is accessed or modified successfully. The (modified) result is returned as JSON.
DELETE	Returns 204 No Content if the resource was deleted successfully.

Refer to the official GitLab documentation for the full list of status codes at <https://docs.gitlab.com/ee/api/#status-codes>

Managing GitLab projects

We already know how to create GitLab projects using UI. In this example we will try to access project information, create and update a project and delete it once we've done. You may want to access additional information available via <https://docs.gitlab.com/ce/api/projects.html>

List projects

Using the root private access token we may receive full project information using "/api/v4/projects" API url.

```
$ curl --silent --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/projects | jq
[
  {
    "_links": {
      "events": "http://gitlab.example.com/api/v4/projects/1/events",
      "issues": "http://gitlab.example.com/api/v4/projects/1/issues",
      "labels": "http://gitlab.example.com/api/v4/projects/1/labels",
    }
  }
]
```

```

        "members":  

"http://gitlab.example.com/api/v4/projects/1/members",  

        "merge_requests":  

"http://gitlab.example.com/api/v4/projects/1/merge_requests",  

        "repo_branches":  

"http://gitlab.example.com/api/v4/projects/1/repository/branches",  

        "self": "http://gitlab.example.com/api/v4/projects/1"  

    },  

...  

<OUTPUT OMITTED>  

...  

        "printing_merge_request_link_enabled": true,  

        "public_jobs": true,  

        "readme_url":  

"http://gitlab.example.com/root/first_repo/blob/master/README.md",  

        "request_access_enabled": false,  

        "resolve_outdated_diff_discussions": false,  

        "shared_runners_enabled": true,  

        "shared_with_groups": [],  

        "snippets_enabled": true,  

        "ssh_url_to_repo": "git@gitlab.example.com:root/first_repo.git",  

        "star_count": 0,  

        "tag_list": [],  

        "visibility": "public",  

        "web_url": "http://gitlab.example.com/root/first_repo",  

        "wiki_enabled": true  

    }  

]

```

Too much output? Yes we think so too.

The output contains a list of projects. Each project has a number of properties. The data structure looks like the following:

```
[  

{  

    "property1": "value1",  

    "property2": "value2",  

    "property3": "value3",  

    <OTHER PROPERTIES>  

},
```

```
{  
    "property1": "value1",  
    "property2": "value2",  
    "property3": "value3",  
    <OTHER PROPERTIES>  
}  
]  
]
```

In order to display properties using **jq**, we can use project id and name:

```
$ curl --silent -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects | jq ".[] | { id, name}"  
{  
    "id": 1,  
    "name": "first_repo"  
}
```

Note! We used “-X GET” to access the data which is the default behavior of curl. We just wanted to highlight that project list can be obtained using GET HTTP request.

The “**jq**” utility allows us to display only **id** and **name** properties in the this example. You will learn the structure of the queries as we go.

In our example, we have only one project. In real-life, the number of projects can go up to thousand and the output will be too large. GitLab API has a capability to search objects using the “**search**” URL parameter. We just need to add it to our URL request:

Let's display properties using jq. We only need project id and name:

```
$ curl --silent -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects?search=first_repo | jq ".[] | { id, name}"  
{  
    "id": 1,  
    "name": "first_repo"  
}
```

Note: There is no difference in the output since we are searching for the project using specific name.

Most of the times, only basic information about project is required. You do not need the full list of properties on daily basis. GitLab helps to reduce output to a simple view using “**simple=true**” url parameter:

```
$ curl --silent -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects?simple=true | jq  
[
```

```
{
  "id": 1,
  "description": "The first GitLab Repo",
  "name": "first_repo",
  "name_with_namespace": "Administrator / first_repo",
  "path": "first_repo",
  "path_with_namespace": "root/first_repo",
  "created_at": "2018-09-16T19:21:50.385Z",
  "default_branch": "master",
  "tag_list": [],
  "ssh_url_to_repo": "git@gitlab.example.com:root/first_repo.git",
  "http_url_to_repo": "http://gitlab.example.com/root/first_repo.git",
  "web_url": "http://gitlab.example.com/root/first_repo",
  "readme_url":
  "http://gitlab.example.com/root/first_repo/blob/master/README.md",
  "avatar_url": null,
  "star_count": 0,
  "forks_count": 0,
  "last_activity_at": "2018-09-16T20:24:52.208Z",
  "namespace": {
    "id": 1,
    "name": "root",
    "path": "root",
    "kind": "user",
    "full_path": "root",
    "parent_id": null
  }
}
]
```

This time GitLab returned significantly less data, as it was shown during the first list example.

Get single project

A single project can be shown if ID or project name is known. Previously, we gathered project id “1” for the “first_repo” project.

You may access project data as follows using ID using “`/api/v4/projects/<ID>`” API object:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/projects/1 | jq
{
  "id": 1,
  "description": "The first GitLab Repo",
  "name": "first_repo",
```

```
"name_with_namespace": "Administrator / first_repo",
...
<OUTPUT OMITTED>
...
```

Same data can be retrieved using “URL” encoded project name. If using namespaced API calls, make sure that the NAMESPACE/PROJECT_NAME is **URL-encoded**. For example, / is represented by **%2F**. For our particular example, project “**first_demo**” is available in the “root” namespace. So, URL encoded project path will be “**root%2Ffirst_repo**”.

The full list of URL encoded characters can be found [here](#).

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/projects/root%2Ffirst_repo | jq
{
  "id": 1,
  "description": "The first GitLab Repo",
  "name": "first_repo",
  "name_with_namespace": "Administrator / first_repo",
  "path": "first_repo",
  "path_with_namespace": "root/first_repo",
  ...
<OUTPUT OMITTED>
...
```

Create a project

A user can create a GitLab project by accessing “**/api/v4/projects**” URL using POST method. Project API expects to have the name of project path as API parameters to be able to create a new project. At least one has to be provided.

Let’s create a new empty project in the root namespace:

```
$ curl -s -X POST -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/projects?name=demo1 | jq ".[] | {id,
name}"
{
  "id": 2,
  "name": "demo1"
}
```

Note: “-X POST” requests are required to submit new data to GitLab via API call and create a project.

You may want to verify that the project has been created from GitLab WebUI under root user:

The screenshot shows the 'Your projects' section of the GitLab WebUI. There are two projects listed:

- Administrator / demo1** (Maintainer): Updated 3 minutes ago. This project was created successfully.
- Administrator / first_repo** (Maintainer): Updated 1 hour ago. This project was created successfully.

The project has been created successfully. You could pass additional and not mandatory parameters like “description” but we want to modify them later in the book.

If we will try to create project again using the name which already taken GitLab to return the following error message:

```
$ curl -s -X POST -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/projects?name=demo1 | jq ".[] | {id,
name}"
{
  "id": null,
  "name": [
    "has already been taken"
  ]
}
```

Now we want to show you how to create a project with complex data:

```
$ curl -s -X POST -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr" -H
"Content-Type: application/json" -d '{"name": "demo2", "description":
"Demo project 2"}' http://gitlab.example.com/api/v4/projects | jq
{
  "id": 3,
  "description": "Demo project 2",
  "name": "demo2",
  "name_with_namespace": "Administrator / demo2",
  "path": "demo2",
  "path_with_namespace": "root/demo2",
<OUTPUT OMITTED>
```

We create a new project “**demo2**” with description “**Demo project 2**”. Instead of passing parameters as a part of URL we passed them as a JSON object. The following snippet shows how to pass a list of parameters using curl:

```
-H "Content-Type: application/json" -d '{"name": "demo2", "description": "Demo project 2"}'
```

GitLab WebUI now shows the following:

The screenshot shows the GitLab WebUI interface with the 'All' tab selected. There are three projects listed:

- D** Administrator / **demo2** Maintainer Demo project 2 ★ 0 updated 2 minutes ago
- D** Administrator / **demo1** Maintainer ★ 0 updated 8 minutes ago
- F** Administrator / **first_repo** Maintainer The first GitLab Repo ★ 0 updated 1 hour ago

Update project settings

Once project was created, API allows to update its properties using PUT request type at ["/api/v4/projects/<PROJECT_ID>](/api/v4/projects/<PROJECT_ID>).

Previously, we created a project “demo1” without description. Let’s verify that description property is empty.

```
$ curl -s -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects?search=demo1 | jq  
".[].description"  
null
```

The project demo2 has been created in the “root” namespace and URL encoded project name will be “root%2F**demo1**”.

Note: We may also use a numeric ID to work with project properties. GitLab API expects project id or URL encoded project path to be able to update project properties.

Let’s configure project description using a simple API call:

```
# curl -s -X PUT --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr" -H  
"Content-Type: application/json" -d '{"description": "Demo project 1 after  
API update"}' http://gitlab.example.com/api/v4/projects/root%2Fdemo1 | jq  
{  
  "id": 2,  
  "description": "Demo project 1 after API update",  
  "name": "demo1",  
<OUTPUT OMITTED>
```

Note: “-X PUT” is required to update a project. We also highlighted important data for better understanding.

Project information now shows the correct description value:

```
# curl -s -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects?search=demo1 | jq  
".[].description"  
"Demo project 1 after API update"
```

The same information is shown in GitLab Web UI:



Delete project

The project can be deleted by ID or URL encoded project name using DELETE request type at `/api/v4/projects/<PROJECT ID>`

Let's verify that project "demo2" exists:

```
$ curl -s -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects?search=demo2 | jq ". | {id,  
description, name}"  
{  
  "id": 3,  
  "description": "Demo project 2",  
  "name": "demo2"  
}
```

Note! The output above shows project demo2 properties. This means that the project exists.

We can delete the project using “curl -X DELETE” as show below:

```
$ curl -s -X DELETE --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects/root%2Fdemo2  
{"message": "202 Accepted"}
```

Note! 202 Accepted means that GitLab deleted the project.

Verify that the project doesn't exist anymore:

```
$ curl -s -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects?search=demo2  
[]
```

Note! GitLab was not able to find any projects with the name “demo2”.

Other project-related activities

We gave you an idea how to manage GitLab projects via API. API exposes a number of project activities which are briefly described in the table below:

Action	Type	URL
List project for a user	GET	/api/v4/users/:user_id/projects
Get project users	GET	/api/v4/projects/:id/users
Create a project for a user	POST	/api/v4/projects/user/:user_id
Fork project	POST	/api/v4/projects/:id/fork
List forks of a project	GET	/api/v4/projects/:id/forks
Share project with a group	POST	/api/v4/projects/:id/share

Note! This is not the full list of project activities.

Managing GitLab users

Let's work with another object to manage via GitLab API. First, we want to list all GitLab users. The output below shows id, name, and username for each user:

```
$ curl -s -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/users | jq ".[]" | {name, id, username}"  
{  
  "name": "user",  
  "id": 2,  
  "username": "user"  
}  
{  
  "name": "Administrator",  
  "id": 1,  
  "username": "root"  
}
```

Get details about the user named “user” (using /api/v4/users/<USER ID>)

```
$ curl -s -X GET --header "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/users/2 | jq ". | {name, id, username}"
{
  "name": "user",
  "id": 2,
  "username": "user"
}
```

We are ready to create a new user with the following parameters:

Parameter	Value
email	demouser1@example.com
username	demouser1
name	Demo User
password	DevOps123

The following example demonstrates user creation:

```
$ curl -s -X POST -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr" -H
"Content-Type: application/json" -d '{"email": "demouser1@example.com",
"username" : "demouser1", "name": "Demo User", "password": "DevOps123"}'
http://gitlab.example.com/api/v4/users | jq ". | {name, id, username}"
{
  "name": "Demo User",
  "id": 3,
  "username": "demouser1"
}
```

Now, make sure that new user has been created:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/users?search=demo | jq ".[] | {name, id,
username}"
{
  "name": "Demo User",
  "id": 3,
  "username": "demouser1"
}
```

It looks like the user “demouser1” exists but doesn't have any bio information which we are going to update:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/users?search=demo | jq ".[].bio"
null
```

We can update user information using its id (“3” in our example).

```
$ curl -s -X PUT -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr" -H
"Content-Type: application/json" -d '{"bio": "This user has been
updated"}' http://gitlab.example.com/api/v4/users/3 | jq ".bio"
"This user has been updated"
```

Note! In the example above, we used “jq” to display only the “bio” user property.

Finally, we want to grant user “demouser1” access permissions to work on a demo project we created recently (“demo1”). This is a part of project administration tasks and this feature is accessible using “/api/v4/projects” URL.

The project “demo1” is located under the “root” namespace and its URL encoded name is “root%2Fdemo1”. We can access the project users using “/api/v4/projects/<PROJECT ID>/users”:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/projects/root%2Fdemo1/users | jq
[
  {
    "id": 1,
    "name": "Administrator",
    "username": "root",
    "state": "active",
    "avatar_url":
      "https://www.gravatar.com/avatar/e64c7d89f26bd1972efa854d13d7dd61?s=80&d=id
      enticon",
    "web_url": "http://gitlab.example.com/root"
  }
]
```

As we can see only “root” user has access to the project.

Project members control documentation is available at
<https://docs.gitlab.com/ee/api/members.html>

Let's add "demouser1" as a developer to the project "demo1":

```
$ curl -s -X POST -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr" -H  
"Content-Type: application/json" -d '{"user_id": "3", "access_level":  
"30"}' http://gitlab.example.com/api/v4/projects/root%2Fdemo1/members | jq  
{  
  "id": 3,  
  "name": "Demo User",  
  "username": "demouser1",  
  "state": "active",  
  "avatar_url":  
  "https://www.gravatar.com/avatar/8c39b968dbf4e595e3c8bc8c2bfbe103?s=80&d=id  
enticon",  
  "web_url": "http://gitlab.example.com/demouser1",  
  "access_level": 30,  
  "expires_at": null  
}
```

Well done! Now the user "demouser1" has Developer access to the project "demo1".

We can verify that using GitLab Web UI:

- Login as "root"
- Go to "Projects" => "Your projects"
- Click on "demo1"

The screenshot shows the 'Your projects' section of the GitLab interface. At the top, there are tabs for 'Your projects', 'Starred projects', and 'Explore projects'. Below that, there are filters for 'All' and 'Personal'. Two projects are listed:

- D Administrator / demo1 (Maintainer)**
Demo project 1 after API update
- F Administrator / first_repo (Maintainer)**
The first GitLab Repo

- Click on "Settings" => "Members" on the left panel:

- Make sure that “Demo User @demouser1” exists and has proper access rights (“Developer”).

Managing branches

By default, GitLab creates the “**master**” branch for a GitLab repository. You may list all repository branches by GET at “/api/v4/projects/<PROJECT ID>/repository/branches”.

First, we need to initiate the repository using git CLI. In the next example, we are going to create a README.md file and push it to the repo.

We need to know project URL:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/projects/root%2Fdemo1 | jq
".http_url_to_repo"
"http://gitlab.example.com/root/demo1.git"
```

Clone the repository locally from any vagrant VM:

```
$ git clone http://gitlab.example.com/root/demo1.git
Cloning into 'demo1'...
Username for 'http://gitlab.example.com': root
Password for 'http://root@gitlab.example.com': DevOps123
warning: You appear to have cloned an empty repository.
```

Create README.md and push changes to the master branch:

```
$ cd demo1/
$ echo "Simple README.md" > README.md
$ git add README.md
$ git commit -m "created README.md"
$ git push origin master
Username for 'http://gitlab.example.com': root
Password for 'http://root@gitlab.example.com': DevOps123
Counting objects: 3, done.
Writing objects: 100% (3/3), 240 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To http://gitlab.example.com/root/demo1.git
 * [new branch]      master -> master
```

The following example lists all branches associated with project “demo1” which we created recently:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"
http://gitlab.example.com/api/v4/projects/root%2Fdemo1/repository/branches
| jq
[
  {
    "name": "master",
    "commit": {
      ...
      <OUTPUT OMITTED>
      ...
    },
    "merged": false,
    "protected": true,
    "developers_can_push": false,
    "developers_can_merge": false,
    "can_push": true
  }
]
```

We need to specify the source branch using “**ref**” parameter to be able to create a new branch using master as a source branch. We also need to specify new branch name using the “**branch**” parameter.

```
$ curl -s -X POST -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr" -H "Content-Type: application/json" --data '{"ref": "master", "branch": "demobranch"}' http://gitlab.example.com/api/v4/projects/root%2Fdemo1/repository/branches | jq
{
  "name": "demobranch",
  <OMITTED>
```

Try to list of project branches.

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr" http://gitlab.example.com/api/v4/projects/root%2Fdemo1/repository/branches | jq
"[.[]].name"
[
  "demobranch",
  "master"
]
```

Note! We filtered information using jq query.

As we can see, the new branch has been successfully created via API.

Managing merge requests

Merge request API documentation is available on official GitLab API page at https://docs.gitlab.com/ee/api/merge_requests.html.

All merge requests can be listed via API calls “/api/v4/merge_requests”. Project specific merge requests are available using “/projects/<PROJECT ID>/merge_requests”.

We didn't create any merge requests and need to create one to move forward. Once merge request is created we will show you how to display merge request information and look for created/opened/closed merge requests.

We need to specify the project, source, and target branches to create a merge request. We also need to give a name to merge request and it is a good idea to provide some additional information in the description property.

Let's create a merge request for “**demo1**” project from “**demobranch**” to the “**master**” branch.

```
$ curl -s -X POST -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr" -H "Content-Type: application/json" --data '{"source_branch": "demobranch", "target_branch": "master", "title": "The first merge request", "description": "This merge request do not implement anything"}' http://gitlab.example.com/api/v4/projects/root%2Fdemo1/merge_requests | jq
```

```
{
  "id": 1,
  "iid": 1,
  "project_id": 3,
  "title": "The first merge request",
  "description": "This merge request do not implement anything",
  "state": "opened",
  ...
  <OUTPUT OMITTED>
  ...
}
```

Well, we created a merge request. Let's verify that it was created using GitLab UI:

- Login using root account
- Click on “demo1” project
- Click on “Merge Requests” on the left panel

The screenshot shows the GitLab interface for the 'demo1' project. On the left, there's a sidebar with links for Project, Repository, Issues, Merge Requests (which has 1 item), CI / CD, and Operations. The main area is titled 'Merge Requests' and shows a single entry: 'The first merge request' opened 4 hours ago by Administrator. There are buttons for 'Edit merge requests' and 'New merge request'. Below the main area, there's a summary: 'Open' (1 item), 'Merged' (0), 'Closed' (0), and 'All' (1). A search bar and a 'Created date' filter are also present.

- Click on the merge request named “The first merge request”

Open Opened 4 hours ago by **Administrator** **Edit** **Close merge request**

The first merge request

This merge request do not implement anything

This screenshot shows the details of the 'The first merge request'. It displays the title 'Request to merge demobranch into master'. Below the title are several buttons: 'Open in Web IDE', 'Check out branch', and a dropdown menu represented by a gear icon. The overall layout is clean and modern, typical of the GitLab user interface.

As we can see, description and title fields have been set.

Now, we can try to look for merge requests using API:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5wFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects/root%2Fdemo1/merge_requests | jq  
".[] | {id, title, description, source_branch, target_branch}"  
{  
    "id": 1,  
    "title": "The first merge request",  
    "description": "This merge request do not implement anything",  
    "source_branch": "demobranch",  
    "target_branch": "master"  
}
```

API allows to look for merge requests by their state. For example, we can show merge requests which are in “opened” state:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5wFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects/root%2Fdemo1/merge_requests?state  
=opened | jq ".[] | {id, title, description, source_branch, target_branch}"  
{  
    "id": 1,  
    "title": "The first merge request",  
    "description": "This merge request do not implement anything",  
    "source_branch": "demobranch",  
    "target_branch": "master"  
}
```

Both our branches are equal and there are no changes between them. We need to push an update to the “**demobranch**” branch to show you how to handle merge requests approvals using API.

Clone demo1 project again and switch to **demobranch** branch using GIT CLI:

```
$ cd .. ; rm -rf demo1  
$ git clone -b demobranch http://gitlab.example.com/root/demo1.git  
Cloning into 'demo1'...  
Username for 'http://gitlab.example.com': root  
Password for 'http://root@gitlab.example.com': DevOps123  
remote: Counting objects: 3, done.  
remote: Total 3 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.
```

Create a new file named “example.txt” and push changes to GitLab:

```
$ cd demo1
```

```

$ echo "Example text file" > example.txt
$ git add example.txt
$ git commit -m "Added example.txt"
$ git push origin demobranch
Username for 'http://gitlab.example.com': root
Password for 'http://root@gitlab.example.com': DevOps123
...
<OUTPUT OMITTED>
...
To http://gitlab.example.com/root/demo1.git
  be593c2..14bdd42  demobranch -> demobranch

```

Now the merge requests will show changes using WebUI:

The screenshot shows the GitLab web interface for a project named 'demo1'. The left sidebar lists project details: demo1 (Project), Repository (Repository), Issues (0), Merge Requests (1), CI / CD, Operations, Wiki, Snippets, and Settings. The main content area displays the 'The first merge request' page. It shows a 'Request to merge demobranch into master' button with options to 'Open in Web IDE', 'Check out branch', and a dropdown menu. Below this is a pipeline status card for 'Pipeline #8 pending for 2001c4b4 on demobranch'. A dropdown menu indicates 'Merge when pipeline succeeds' with options to 'Remove source branch' and 'Modify commit message'. A note says 'You can merge this merge request manually using the command line'. At the bottom, there are discussion, commits, pipelines, and changes tabs, with the changes tab selected. The changes view shows a single file 'example.txt' with 1 addition and 0 deletions. There are also buttons for 'Filter files', 'Changes between', 'Hide whitespace changes', and 'View file @ 2001c4b4'.

We have merge request. It has a number of changes (added example.txt). Now it is a good time to merge it using API. Merge request approval is done using PUT at /api/v4/projects/<PROJECT ID>/merge_requests/<MERGE REQUEST ID>/merge First check merge request ID we want to accept:

```
$ curl -s -X GET -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects/root%2Fdemo1/merge_requests | jq  
".[].iid"  
1
```

Use this ID to accept merge request via API:

```
$ curl -s -X PUT -H "Private-Token: 1X-TyHMnB5WFGdDxpwmr"  
http://gitlab.example.com/api/v4/projects/root%2Fdemo1/merge_requests/1/mer  
ge | jq  
{  
  "id": 1,  
  "iid": 1,  
  "project_id": 3,  
  "title": "The first merge request",  
  "description": "This merge request do not implement anything",  
  "state": "merged",  
<OUTPUT OMITTED>
```

Well done! We merged our request via API. Check it out at http://gitlab.example.com/root/demo1/merge_requests/1.

The screenshot shows a GitLab project named 'demo1'. On the left sidebar, under 'Merge Requests', there is one item: 'Request to merge demobranch into master' (Merged, opened 5 hours ago by Administrator). Below this, a message states: 'This merge request do not implement anything'. The main content area displays the merge request details, including a summary of the merge, a failed pipeline status, and a successful merge history. At the bottom, there are activity metrics (0 likes, 0 dislikes, 0 comments), a changes tab (1 changed file, 1 addition, 0 deletions), and a file viewer for 'example.txt'.

Note: that pipeline currently in the failed state. It's going to change once we reach the integration part with Jenkins.

In the example above, we have shown you how to work with GitLab via its API. You created a project, a new branch from the master branch, a user. You gave **user** access to the project, and finally, you created and merged a merge request. This is something that you can use to automate your code development, test, and even deployment at work.

Managing other GitLab entities

GitLab API allows managing any GitLab entities, like users or projects. We have shown you basic examples of how to work with GitLab API and you can easily extend your knowledge by reading official GitLab API documentation.

You can access any GitLab entity “`/api/v4/<ENTITY TYPE>`” to be able to create/delete and update GitLab objects like projects, users, merge requests, etc. Refer GitLab API documentation available at:

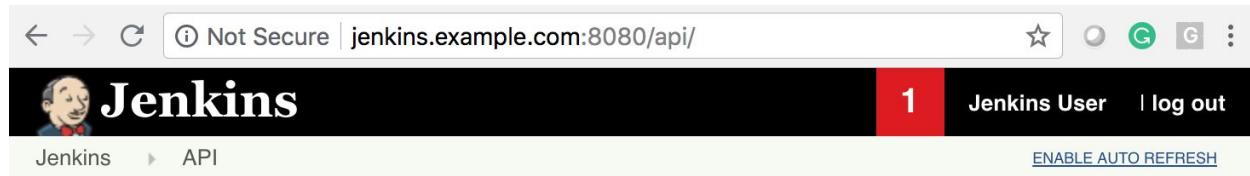
- <https://docs.gitlab.com/ee/api/>
- <https://docs.gitlab.com/ce/api/>

Using Jenkins API

Jenkins API

As we mentioned before Jenkins provides API which allows performing basic Jenkins management like create and execute a job.

You may want to use additional API documentation which is available on Jenkins homepage: <https://wiki.jenkins.io/display/JENKINS/Remote+access+API>. Also, if you have Jenkins installed, you may access documentation at “`<JENKINS_URL>/api`” like <http://jenkins.example.com:8080/api/>. The example page is shown below:



REST API

Many objects of Jenkins provide the remote access API. They are available at `/.../api/` where “`...`” portion is the object for which you'd like to access.

XML API

Access data exposed in [HTML](#) as XML for machine consumption. [Schema](#) is also available.

You can also specify optional XPath to control the fragment you'd like to obtain (but see [below](#)). For example, `.../api/xml?xpath=/*[0]`.

For XPath that matches multiple nodes, you need to also specify the “wrapper” query parameter to specify the name of the root XML element to be created so that the resulting XML becomes well-formed.

Jenkins comes with XML-based API by default. JSON-based API is also available.

You may try to access XML-based API via the following link in our environment: <http://jenkins.example.com:8080/api/xml>. If you try to open this URL in your browser, you should see similar to the following:

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<hudson _class="hudson.model.Hudson">
  ▼<assignedLabel>
    <name>master</name>
  </assignedLabel>
  <mode>NORMAL</mode>
  <nodeDescription>the master Jenkins node</nodeDescription>
  <nodeName/>
  <numExecutors>2</numExecutors>
  ▼<job _class="org.jenkinsci.plugins.workflow.job.WorkflowJob">
    <name>pipeline1</name>
    <url>http://jenkins.example.com:8080/job/pipeline1/</url>
    <color>blue</color>
  </job>
  <overallLoad/>
  ▼<primaryView _class="hudson.model.AllView">
    <name>all</name>
    <url>http://jenkins.example.com:8080/</url>
  </primaryView>
  <quietingDown>false</quietingDown>
  <slaveAgentPort>-1</slaveAgentPort>
  <unlabeledLoad _class="jenkins.model.UnlabeledLoadStatistics"/>
  <useCrumbs>true</useCrumbs>
  <useSecurity>true</useSecurity>
  ▼<view _class="hudson.model.AllView">
    <name>all</name>
    <url>http://jenkins.example.com:8080/</url>
  </view>
</hudson>
```

JSON-based API is accessible via <http://jenkins.example.com:8080/api/json?pretty=true>. Here is an example of output:

```
← → C ⓘ Not Secure | jenkins.example.com:8080/api/json?pretty=true

{
  "_class" : "hudson.model.Hudson",
  "assignedLabels" : [
    {
      "name" : "master"
    }
  ],
  "mode" : "NORMAL",
  "nodeDescription" : "the master Jenkins node",
  "nodeName" : "",
  "numExecutors" : 2,
  "description" : null,
  "jobs" : [
    {
      "_class" : "org.jenkinsci.plugins.workflow.job.WorkflowJob",
      "name" : "pipeline1",
      "url" : "http://jenkins.example.com:8080/job/pipeline1/",
      "color" : "blue"
    }
  ],
  "overallLoad" : {

  },
  "primaryView" : {
    "_class" : "hudson.model.AllView",
    "name" : "all",
    "url" : "http://jenkins.example.com:8080/"
  },
  "quietingDown" : false,
  "slaveAgentPort" : -1,
  "unlabeledLoad" : {
    "_class" : "jenkins.model.UnlabeledLoadStatistics"
  },
  "useCrumbs" : true,
  "useSecurity" : true,
  "views" : [
    {
      "_class" : "hudson.model.AllView",
      "name" : "all",
      "url" : "http://jenkins.example.com:8080/"
    }
  ]
}
```

Next labs will use only JSON-based API.

Authentication

Most of the API functions require authentication. If you will try to access API without authentication Jenkins API will return an error:

```
$ curl -s http://jenkins.example.com:8080/api/json
...
<OUTPUT OMITTED>
...
Authentication required
...
<OUTPUT OMITTED>
...
```

Jenkins API support 2 types of authentication - username/password and username/token. Both types allow accessing Jenkins API entities.

Username/password authentication

The pair username/password it is not recommended because the risk of revealing the password, and the human tendency to reuse the same password in different places. However, you may still want to use it to quickly check something in a lab environment.

Recently we created an admin user with name “jenkins” and password “DevOps”. Let’s try to access root json api object:

```
$ curl -s -u jenkins:DevOps123 http://jenkins.example.com:8080/api/json | jq
{
  "_class": "hudson.model.Hudson",
  "assignedLabels": [
    {
      "name": "master"
    }
  ],
  "mode": "NORMAL",
  "nodeDescription": "the master Jenkins node",
  ...
<OUTPUT OMITTED>
...
```

Note: We do not recommend using this method as you are exposing your username credentials, and it does not work very well if you have 2 factor authentication or one time passwords.

Username/token authentication

To make scripted clients (such as wget) invoke operations that require authorization (such as scheduling a build), use HTTP BASIC authentication to specify the username and the API token. This is often more convenient than emulating the form-based authentication. You need to an API token to use this feature.

The API token is available in your personal configuration page. You may generate a new API token as follows:

- Login as “jenkins” at <http://jenkins.example.com:8080>
- Click your name on the top right corner on every page, then click “Configure”:

The screenshot shows the Jenkins User interface. At the top, there is a navigation bar with tabs for 'Learning', 'jenkins.example.com', 'Authentic...', 'Jenkins', and 'Artemii'. Below the navigation bar, the main header says 'Jenkins' with a user icon. The left sidebar contains links: 'People', 'Status', 'Builds', 'Configure', 'My Views', and 'Credentials'. A red notification badge with the number '1' is visible above the 'Configure' link. A dropdown menu is open over the 'Configure' link, listing 'Builds', 'Configure' (which is highlighted in blue), 'My Views', and 'Credentials'. The main content area below the sidebar has a title 'Jenkins User' with a user icon.

- On the “API token” Section click locate the “Add new Token” button:

The screenshot shows the 'API Token' section. The heading is 'API Token'. Below it, a sub-section titled 'Current token(s)' displays the message 'There is no registered token for this user'. At the bottom of this section is a grey button labeled 'Add new Token'.

- Click on “Add new Token” and specify the token name. It can be any string. Click on the “Generate” button:

The screenshot shows the 'API Token' section again. The 'my new token' input field is highlighted with a blue border. To its right are two buttons: 'Generate' (grey) and 'Cancel' (red). Below these buttons is another 'Add new Token' button.

The screenshot shows the 'API Token' section. The 'my new token' input field contains the value 'my new token'. To its right, a generated token '11f51e6fcf746d94b3b2a4f7db3760df51' is displayed in a text area. Below this is a yellow warning message: '⚠ Copy this token now, because it cannot be recovered in the future.' At the bottom is another 'Add new Token' button.

- Copy the token. Jenkins will not show it again

In our case token is “11f51e6fcf746d94b3b2a4f7db3760df51”. We are going to use this token in the next labs.

We can verify that username/token authentication works fine by using token instead of password as shown below:

```
$ curl -s -u jenkins:11f51e6fcf746d94b3b2a4f7db3760df51
http://jenkins.example.com:8080/api/json | jq
{
  "_class": "hudson.model.Hudson",
  "assignedLabels": [
    {
      "name": "master"
    }
  ],
  "mode": "NORMAL",
  "nodeDescription": "the master Jenkins node",
  ...
<OUTPUT OMITTED>
...
```

Note! We used token instead of password.

Tokens are the most preferred method for Jenkins API authentication because you can generate tokens per your application and automation tool and then easily revoke when needed. So try to avoid using password authentication.

List jobs

Jenkins return job list as a part of /api/json object. The job list can be obtained as shown below:

```
$ curl -s -u jenkins:11f51e6fcf746d94b3b2a4f7db3760df51
http://jenkins.example.com:8080/api/json | jq .jobs
[
  {
    "_class": "org.jenkinsci.plugins.workflow.job.WorkflowJob",
    "name": "pipeline1",
    "url": "http://jenkins.example.com:8080/job/pipeline1/",
    "color": "red"
  }
]
```

The output may contain a lot of lines which will be difficult to read and parse. API allows to limit json objects returned using filters.

For example, if there is a need to return just all objects in jobs tree, you may want to use the following:

```
$ curl -s -g -u jenkins:11f51e6fcf746d94b3b2a4f7db3760df51
"http://jenkins.example.com:8080/api/json?tree=jobs[name,url]" | jq
{
  "_class": "hudson.model.Hudson",
  "jobs": [
    {
      "_class": "org.jenkinsci.plugins.workflow.job.WorkflowJob",
      "name": "pipeline1",
      "url": "http://jenkins.example.com:8080/job/pipeline1/"
    }
  ]
}
```

Note! Curl “-g” option switches off the “URL globbing parser”. When you set this option, you can specify URLs that contain the letters {} without having them being interpreted by curl itself. These letters are not normal legal URL contents but they should be encoded according to the URI standard.

The API request above limited tree returned by api to “jobs” object which includes a list of jobs. We also requested to limit fields to be displayed to name and URL. This means that _class, name and URL fields will be returned.

Create a Job

Job creation process required to pass XML job definition to API. It may be a bit difficult for those who is unfamiliar how jobs are defined in their configuration files. But worry not, there is a simple way to gather XML job definition, we can get configuration from another job and then use it as a template.

The example above returned job URL <http://jenkins.example.com:8080/job/pipeline1/>. We are going to use this URL to access job configuration object.

```
$ curl -s -u jenkins:11f51e6fcf746d94b3b2a4f7db3760df51
http://jenkins.example.com:8080/job/pipeline1/config.xml
<?xml version='1.1' encoding='UTF-8'?>
<flow-definition plugin="workflow-job@2.24">
  <actions/>
  <description></description>
  <keepDependencies>false</keepDependencies>
  <properties/>
  <definition class="org.jenkinsci.plugins.workflow.cps.CpsFlowDefinition"
  plugin="workflow-cps@2.54">
```

```

<script>node {
    stage("Stage 1") {
        echo "Step 1"
    }
    stage("Stage 2") {
        echo "Step 2"
    }
    stage("Stage 3") {
        echo "Step 3"
    }
}</script>
<sandbox>true</sandbox>
</definition>
<triggers/>
<disabled>false</disabled>
</flow-definition>

```

Note! This returns a XML object (not JSON)!

You may want to save this output to a separate file since we need to pass it to API during job creation process. This can be easily achieved via:

```

$ curl -s -v -u jenkins:11f51e6fcf746d94b3b2a4f7db3760df51 -o job.xml
http://jenkins.example.com:8080/job/pipeline1/config.xml
...
<OUTPUT OMITTED>
...
< HTTP/1.1 200 OK
...
<OUTPUT OMITTED>
...

```

Make sure that you have return code **200 OK**.

Let's create a new job using job.xml as a job configuration:

```

$ curl -s -v -X POST -u jenkins:11f51e6fcf746d94b3b2a4f7db3760df51 -H
"Content-Type: application/xml" -d @job.xml
http://jenkins.example.com:8080/createItem?name=newjob1
...
<OUTPUT OMITTED>
...
< HTTP/1.1 200 OK

```

```
...  
<OUTPUT OMITTED>  
...
```

Make sure that you have return code **200 OK**.

Check jobs:

```
$ curl -s -g -u jenkins:11f51e6fcf746d94b3b2a4f7db3760df51  
"http://jenkins.example.com:8080/api/json?tree=jobs[name]" | jq  
{  
    "_class": "hudson.model.Hudson",  
    "jobs": [  
        {  
            "_class": "org.jenkinsci.plugins.workflow.job.WorkflowJob",  
            "name": "newjob1"  
        },  
        {  
            "_class": "org.jenkinsci.plugins.workflow.job.WorkflowJob",  
            "name": "pipeline1"  
        }  
    ]  
}
```

You can see that now we have two jobs

You may also want to verify that job exists using web interface:

The screenshot shows the Jenkins dashboard with a table of jobs. The table has columns: S (Status), W (Last build status), Name (sorted by name), Last Success, Last Failure, and Last Duration. There are two jobs listed: 'newjob1' and 'pipeline1'. Both jobs are marked as 'green' (success). The 'newjob1' row shows 'N/A' for last success and failure, and 'N/A' for duration. The 'pipeline1' row shows '17 min - #5' for last success, '3 days 12 hr - #2' for last failure, and '1.1 sec' for duration. Below the table, there is a legend with icons for 'S' (Status), 'M' (Last build status), and 'L' (Last duration).

S	W	Name ↓	Last Success	Last Failure	Last Duration
		newjob1	N/A	N/A	
		pipeline1	17 min - #5	3 days 12 hr - #2	1.1 sec

Icon: [S](#) [M](#) [L](#)

[Legend](#) [RSS for all](#) [RSS for failures](#) [RSS for just latest builds](#)

Looks good and both jobs are available. The job “**newjob1**” we just created is a copy of the job “**pipeline1**”.

Trigger a build

Previously we create a new job named “**newjob1**” which is a copy of **pipeline1**. Jenkins API allows executing a job remotely. This can be achieved by POST request to **“/job/<JOBNAME>/build”**

Let's start a build for newjob1:

```
$ curl -X POST -s -u jenkins:11f51e6fcf746d94b3b2a4f7db3760df51  
http://jenkins.example.com:8080/job/newjob1/build
```

Once you run the command, Jenkins should show that the job last build is successful.

S	W	Name ↓	Last Success	Last Failure	Last Duration
		newjob1	3 min 45 sec - #2	5 min 24 sec - #1	0.34 sec
		pipeline1	12 hr - #5	4 days 0 hr - #2	1.1 sec

Icon: [S](#) [M](#) [L](#)

[Legend](#) [RSS for all](#) [RSS for failures](#) [RSS for just latest builds](#)

This simple method allows you triggering Jenkins jobs remotely from other systems or as a part of another pipeline. We will work with this kind of tasks in the following Chapters.

Making GitLab and Jenkins to work together

Why we need it

In the previous chapters, we worked with Jenkins and GitLab separately. Now it is time to move all the pieces together. It's quite important to understand why we need to do the integration between Jenkins and GitLab if we want to make a step further towards real DevOps. This is where you start feeling the real power of both GitLab and Jenkins when they are working together as one thing in your DevOps environment.

In this Chapter, we need both GitLab and Jenkins servers up and running, so Chapters [Getting Started with GitLab](#) and [Getting Started with Jenkins](#) are prerequisites for this Chapter.

What we want to achieve

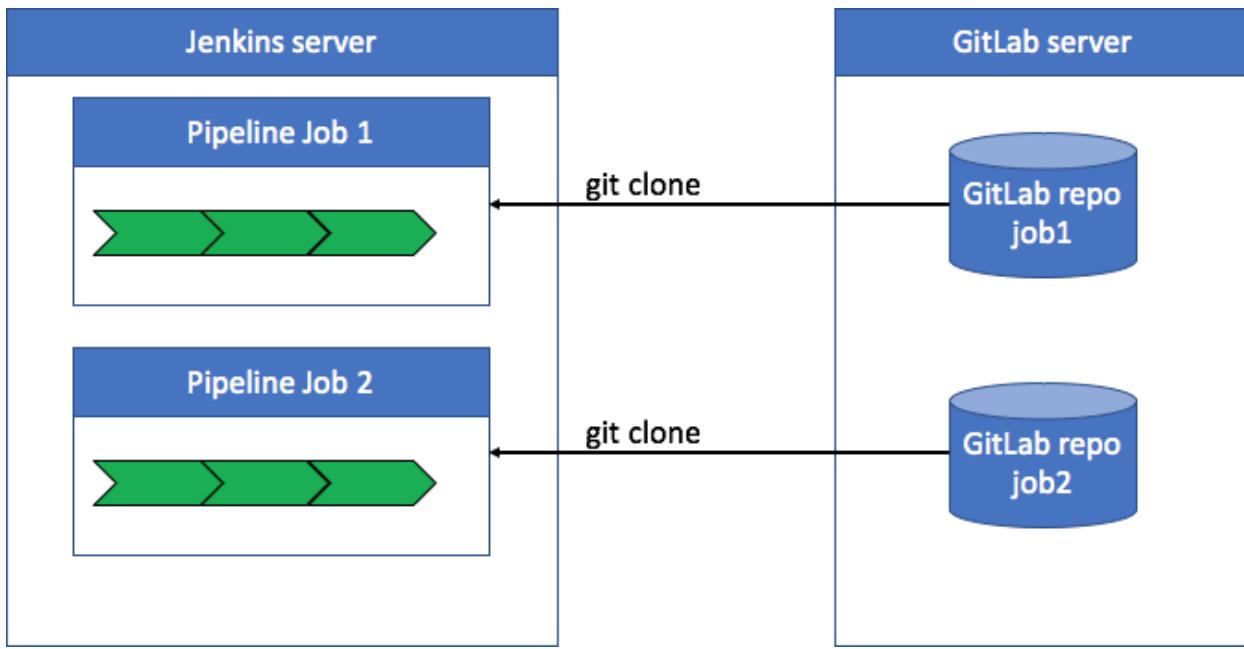
This chapter gives a step by step guide how to configure Jenkins and GitLab integration. At the end of the chapter, we will have a fully integrated solution which triggers a Jenkins build on merge request event in GitLab. All pipeline stage results will report their statuses to GitLab. We will also configure GitLab project settings to allow the merge to be done only if pipeline is successful.

Our plan contains a number of steps to achieve the goal:

- Move Jenkins pipeline script under version control
- Configure GitLab to trigger Jenkins pipeline
- Configure Jenkins to report Pipeline status back to GitLab

Moving Jenkins pipelines to GitLab

We already know all the advantages of version control systems like Git and its repository management solution (GitLab, GitHub and others). In the previous chapters, we configured pipelines and stored the script in Jenkins job itself. In this Chapter, we want to slightly modify Jenkins job definition and configure Jenkins to fetch Pipeline file from a GitLab repo. Basically, Jenkins goes to Gitlab, clones the repo and looks for Jenkinsfile. This approach is shown in the diagram below:



In this configuration, Jenkins gets Pipeline script (called `Jenkinsfile`) from GitLab each time pipeline starts. This approach has a number of advantages:

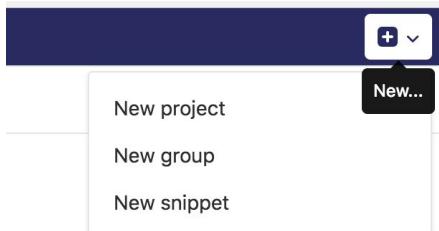
- Your pipeline is under version control system
- You don't have to update your Jenkins job each time when you want to modify it - it is enough to push new changes to git

Note! This approach is also applicable to any GIT repository management systems like GitLab, GitHub, BitBucket, Gogs. It doesn't require any GitLab special configuration.

Create a GitLab repo with pipeline script

First, we need to create a new GitLab repo that stores pipeline details. Let's call it “`pipeline3`”:

- Log in at GitLab using user account (**user/DevOps123**)
- Click on “+” button on the top and click on “New Project” menu item:



- Configure new repository parameters:

Parameter	Value
Name	<code>pipeline3</code>
Visibility level	Private

Initialize repository with a README Checked

Blank project Create from template Import project

Project path: http://gitlab.example.com/user/ Project name: pipeline3

Want to house several dependent projects under the same namespace? [Create a group](#)

Project description (optional): Example pipeline stored in GitLab

Visibility Level: Private
Project access must be granted explicitly to each user.

Internal
The project can be accessed by any logged in user.

Public
The project can be accessed without any authentication.

Initialize repository with a README
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

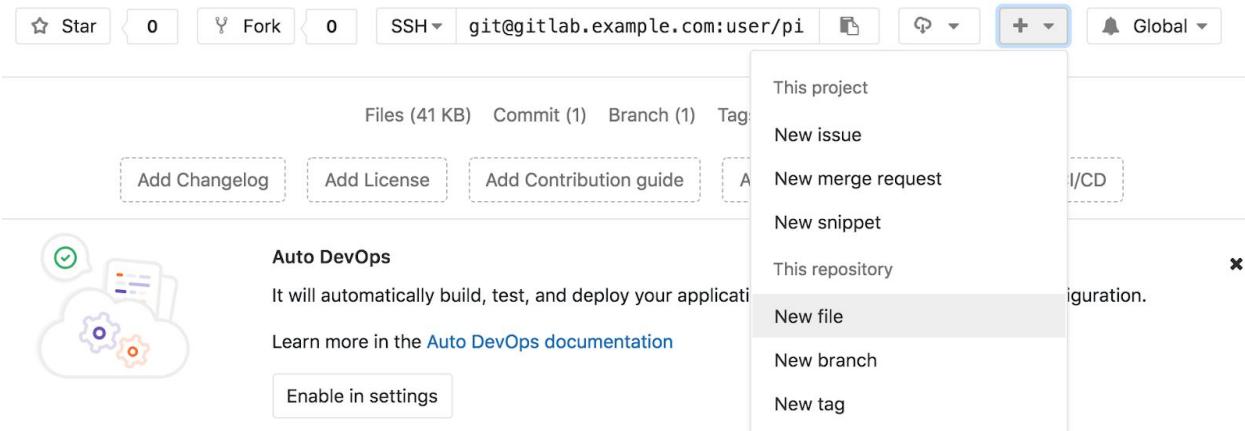
[Create project](#) [Cancel](#)

- Click on “Create project” button

When project is created it will be initialized with empty “README.md” file.

We need to put a pipeline file under the root of the repository. By default, Jenkins uses the file named “**Jenkinsfile**” as pipeline script. We want to keep the default pipeline file name. Let’s create pipeline script named “Jenkinsfile”. Please make sure that “J” is capitalized. We may want to use git CLI to create the file and push changes to git. GitLab also supports Web UI to do a commit.

- Click on the “+” button
- Choose “New file” as shown below



- Type “Jenkinsfile” in the filename

- Type the following simple content:

```
node {
    stage("Checkout") {
        checkout scm
    }
    stage("Verify") {
        sh 'true'
    }
}
```

Note! This is a demo pipeline which we are going to modify. Just for now it is enough to check the integration. We are also going to explain all stages in details.

- Click on the “Commit changes” button

Commit message: Add new file

Target Branch: master

Commit changes Cancel

- Make sure that both README.md and Jenkinsfile exist in the repository

Name	Last commit	Last update
Jenkinsfile	Add new file	44 seconds ago
README.md	Initial commit	1 hour ago

Well, we prepared our GitLab repository.

Install git on Jenkins

Git software must be installed on Jenkins servers which will handle pipelines. If **git** yum package is not installed install it as shown below:

```
$ sudo yum install -y git
```

Note! If git package is not installed pipelines will not work

Create Jenkins job

We already created a number of Jenkins jobs in previous chapters. Let's create a new one named "**pipeline3**".

- Log in Jenkins
- Click on "New Item" in the left panel



New Item



People



Build History



Manage Jenkins



My Views



Credentials



New View

- Type "pipeline3" under "Enter an item name"
- Choose "Pipeline"
- Click OK
- Leave "General" and "Build Triggers" and "Advanced project options" sections as is

General Build Triggers Advanced Project Options Pipeline

Description

[Plain text] [Preview](#)

Discard old builds [?](#)

Do not allow concurrent builds [?](#)

Do not allow the pipeline to resume if the master restarts [?](#)

GitHub project [?](#)

Pipeline speed/durability override [?](#)

Preserve stashes from completed builds [?](#)

This project is parameterized [?](#)

Throttle builds [?](#)

Build Triggers

Build after other projects are built [?](#)

Build periodically [?](#)

GitHub hook trigger for GITScm polling [?](#)

Poll SCM [?](#)

Disable this project [?](#)

Quiet period [?](#)

Trigger builds remotely (e.g., from scripts) [?](#)

Advanced Project Options

[Advanced...](#)

- Choose “Pipeline script from SCM” at “Pipeline” section:

Pipeline

Definition Pipeline script from SCM

SCM None

Script Path Jenkinsfile

Lightweight checkout

[Pipeline Syntax](#)

[Save](#) [Apply](#)

- Choose “GIT” under SCM
- Type GitLab project URL: <http://gitlab.example.com/user/pipeline3.git> (this may be very depending on your job and gitlab hostname)

Pipeline

Definition Pipeline script from SCM

SCM Git

Repositories

Repository URL <http://gitlab.example.com/user/pipeline3.git>

```
Failed to connect to repository : Command
"git ls-remote -h
http://gitlab.example.com/user/pipeline3.git"
HEAD" returned status code 128:
stdout:
stderr: fatal: Authentication failed for
'http://gitlab.example.com/user/pipeline3.git/'
```

Credentials [Add](#)

[Advanced...](#) [Add Repository](#)

- Choose proper Credentials to access the repository. If credentials have not been configured click on “Add” button and choose “Jenkins”

Pipeline

Definition Pipeline script from SCM

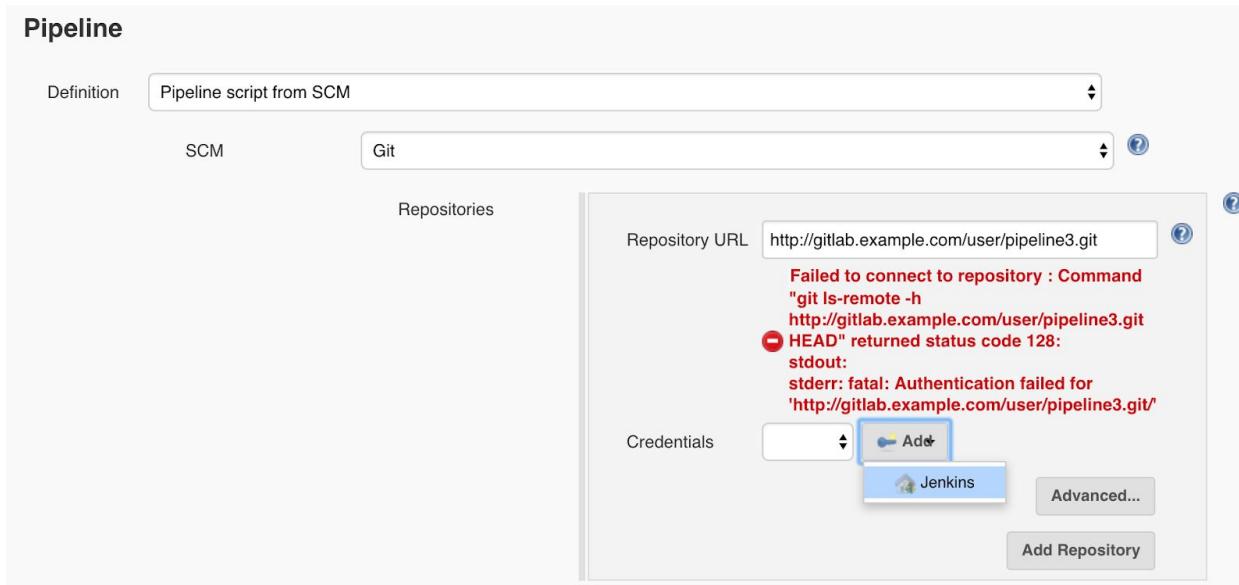
SCM Git

Repositories

Repository URL http://gitlab.example.com/user/pipeline3.git

Failed to connect to repository : Command
"git ls-remote -h
http://gitlab.example.com/user/pipeline3.git
HEAD" returned status code 128:
stdout:
stderr: fatal: Authentication failed for
'http://gitlab.example.com/user/pipeline3.git/'

Credentials Add Jenkins Advanced... Add Repository



- Type username and password to access the GitLab repo and press Add

Jenkins Credentials Provider: Jenkins

Add Credentials

Domain Global credentials (unrestricted)

Kind Username with password

Scope Global (Jenkins, nodes, items, all child items, etc)

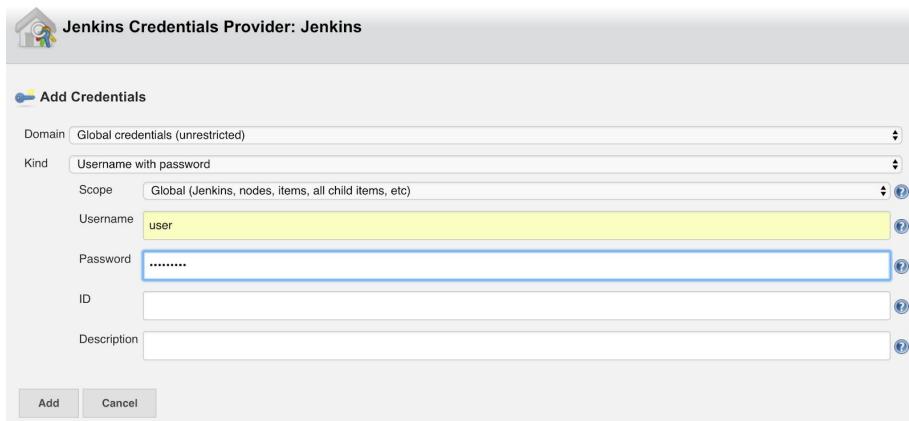
Username user

Password

ID

Description

Add Cancel



- Choose proper credentials and press "Save" under job configuration

General Build Triggers Advanced Project Options Pipeline

Pipeline

Definition Pipeline script from SCM

SCM Git

Repositories

Repository URL http://gitlab.example.com/l
Credentials user/******** Add Advanced... Add Repository

Branches to build

Branch Specifier (blank for 'any') */master Add Branch

Repository browser (Auto)

Additional Behaviours Add

Script Path Jenkinsfile

Lightweight checkout

Save **Apply**

Once the job is created, you may want to start the job.

Jenkins > pipeline3 > Pipeline pipeline3

ENABLE AUTO REFRESH

- Back to Dashboard
- Status
- Changes
- Build Now
- Delete Pipeline
- Configure
- Full Stage View
- Rename
- Pipeline Syntax

Pipeline pipeline3

add description **Disable Project**

Stage View
No data available. This Pipeline has not yet run.

Build History trend =
find
[RSS for all](#) [RSS for failures](#)

Permalinks

You may start the job by pressing on “Build Now”.

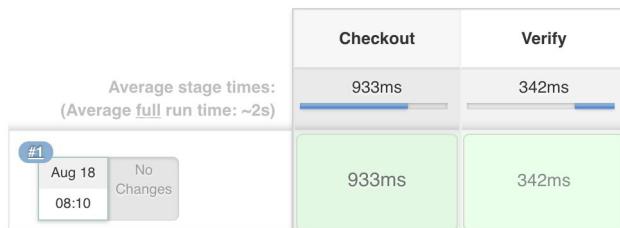
Pipeline pipeline3

Average stage times:
(Average full run time: ~2s)

Checkout	Verify
933ms	342ms
933ms	342ms

Recent Changes

Stage View



Well! It works.

Let's check that the pipeline has been cloned from GIT. Check the build results. Click on the link named "#1" (or any other build number).

Back to Project
Status
Changes
Console Output
Edit Build Information
Delete Build
Git Build Data
No Tags
Replay
Pipeline Steps

Build #1 (Aug 18, 2018 12:10:08 PM)

Started by user Jenkins User
Revision: 8a8c902b19d39c1c4b2faee3de199d3861dab10a
refs/remotes/origin/master

In the job details click on "Console Output".

Back to Project
Status
Changes
Console Output
View as plain text
Edit Build Information
Delete Build
Git Build Data
No Tags
Replay
Pipeline Steps

Console Output

```
Started by user Jenkins User
Obtained Jenkinsfile from git http://gitlab.example.com/user/pipeline3.git
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipeline3
[Pipeline]
[Pipeline] stage
[Pipeline] { (Checkout)
[Pipeline] checkout
Cloning the remote Git repository
Cloning repository http://gitlab.example.com/user/pipeline3.git
> git init /var/lib/jenkins/workspace/pipeline3 # timeout=10
Fetching upstream changes from http://gitlab.example.com/user/pipeline3.git
> git --version # timeout=10
using GIT_ASKPASS to set credentials
> git fetch --tags --progress http://gitlab.example.com/user/pipeline3.git
+refs/heads/*:refs/remotes/origin/*
> git config remote.origin.url http://gitlab.example.com/user/pipeline3.git # timeout=10
> git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
> git config remote.origin.url http://gitlab.example.com/user/pipeline3.git # timeout=10
Fetching upstream changes from http://gitlab.example.com/user/pipeline3.git
using GIT_ASKPASS to set credentials
> git fetch --tags --progress http://gitlab.example.com/user/pipeline3.git
+refs/heads/*:refs/remotes/origin/*
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision 8a8c902b19d39c1c4b2faee3de199d3861dab10a (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 8a8c902b19d39c1c4b2faee3de199d3861dab10a
Commit message: "Add new file"
First time build. Skipping changelog.
[Pipeline]
```

You should see a number of git commands and Git repository URL.

Sometimes it is required to change the job. In case of using pipelines stored directly in GIT, all changes are done by updating GIT. In our example, we want to add an additional simple stage.

Let's update “**Jenkinsfile**” on our repository URL according to new requirements. Here is the example of new Jenkinsfile (we highlighted the difference):

```
node {  
    stage("Checkout") {  
        checkout scm  
    }  
    stage("Verify") {  
        sh 'true'  
    }  
    stage("Cleanup") {  
        sh 'true'  
    }  
}
```

Note! You can use the command “git” or GitLab UI. The following example is based on CLI “git” command on Jenkins VM:

- Clone the repository

```
$ git clone http://gitlab.example.com/user/pipeline3.git  
Cloning into 'pipeline3'...  
Username for 'http://gitlab.example.com': user  
Password for 'http://user@gitlab.example.com': DevOps123  
remote: Counting objects: 6, done.  
remote: Compressing objects: 100% (4/4), done.  
remote: Total 6 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (6/6), done.
```

- Update Jenkinsfile

```
$ cd pipeline3/  
$ cat <<EOF >Jenkinsfile  
node {  
    stage("Checkout") {  
        checkout scm  
    }  
    stage("Verify") {  
        sh 'true'
```

```

}
stage("Cleanup") {
    sh 'true'
}
}

EOF
$ git add Jenkinsfile
$ git commit -m "updated Jenkinsfile"
$ git push origin master
Username for 'http://gitlab.example.com': user
Password for 'http://user@gitlab.example.com': DevOps123
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 331 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To http://gitlab.example.com/user/pipeline3.git
  8a8c902..c2d9d16  master -> master

```

Well, changes are in GIT now!

Let's re-run Jenkins pipeline by pressing on "Build Now" button on the left panel of the job.

- Go to <http://jenkins.example.com:8080/job/pipeline3/>
- Press "Build Now"

	Checkout	Verify	Cleanup
Average stage times:	725ms	409ms	292ms
(Average full run time: ~2s)			

You may see that task has been updated successfully (Cleanup stage is shown).

What have we just done? We configured Jenkins to keep our pipeline script in GIT. This approach is used in the most production-like installation.

Triggering Jenkins job automatically

At this moment we have Jenkins server to be able to download Jenkinsfile along with all the code from GitLab server and execute it when we run Jenkins jobs manually. Can we do better? Hell yes, we can configure Jenkins to automatically start the job when we push the changes to GitLab and this is what we do next. For example, if a developer pushed a new code, we want to test the code against a number of rules, and also try to build artifacts and try to deploy on a test system. There are multiple ways to achieve that behavior. We will be describing all of the methods below.

Configuring Jenkins build triggers

Jenkins allows configuring job in the way to poll SCM (GitLab in our case) for new changes. If new changes are available, then job will be started. This method doesn't require any special configuration of the repository management system. That's why it is applicable for any GIT/SVN-based repositories.

Change the “**pipeline3**” job configuration to check for updates.

- Login to Jenkins
- Open pipeline3 job
- Click “Configure” on the left panel
- Go to “Build Triggers” tab

The screenshot shows the Jenkins job configuration interface. At the top, there are tabs: General, Build Triggers (which is selected and highlighted in blue), Advanced Project Options, and Pipeline. Below the tabs, the 'Build Triggers' section is titled 'Build Triggers'. It contains a list of checkboxes for various triggering options:

- Build after other projects are built
- Build periodically
- GitHub hook trigger for GITScm polling
- Poll SCM
- Disable this project
- Quiet period
- Trigger builds remotely (e.g., from scripts)

- Check “Poll SCM” and configure Jenkins to check for update every minute

- Click on “Save” to save changes and close the configuration window.

These changes are enough to enable Jenkins polling. Jenkins should now start the job if we push new changes to the repository. Let’s push a new commit to the repository.

Note! You can use the command “git” or GitLab UI. The following example is based on CLI “git” command. Go back to Jenkins VM and do the following:

- Update Jenkinsfile

```
$ cd ~/pipeline3/
$ cat <<EOF > Jenkinsfile
node {
    stage("Checkout") {
        checkout scm
    }
    stage("Verify") {
        sh 'date'
    }
    stage("Cleanup") {
        sh 'true'
    }
}
EOF
```

```
$ git commit -a -m "updated Jenkinsfile"
```

```
$ git push origin master
Username for 'http://gitlab.example.com': user
Password for 'http://user@gitlab.example.com': DevOps123
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 331 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To http://gitlab.example.com/user/pipeline3.git
  8a8c902..c2d9d16  master -> master
```

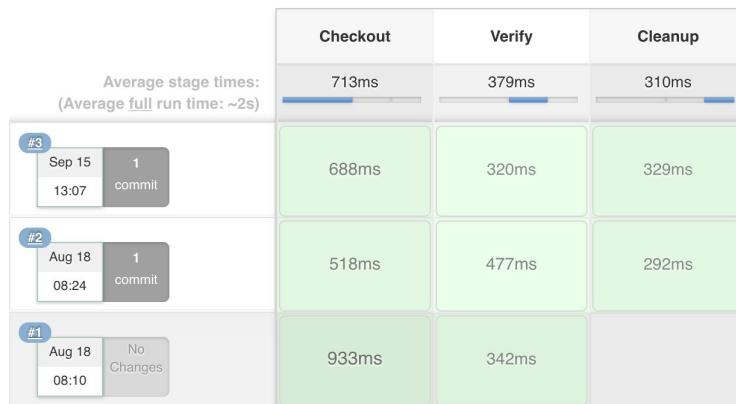
- Login to Jenkins and wait for a new pipeline has to be executed:

The screenshot shows the Jenkins Build History page. At the top, there is a search bar with the text "find". Below it, three build entries are listed:

- #3 (pending—???)
- #2 Aug 18, 2018 12:24 PM
- #1 Aug 18, 2018 12:10 PM

At the bottom of the page, there are two RSS feed links: "RSS for all" and "RSS for failures".

Stage View

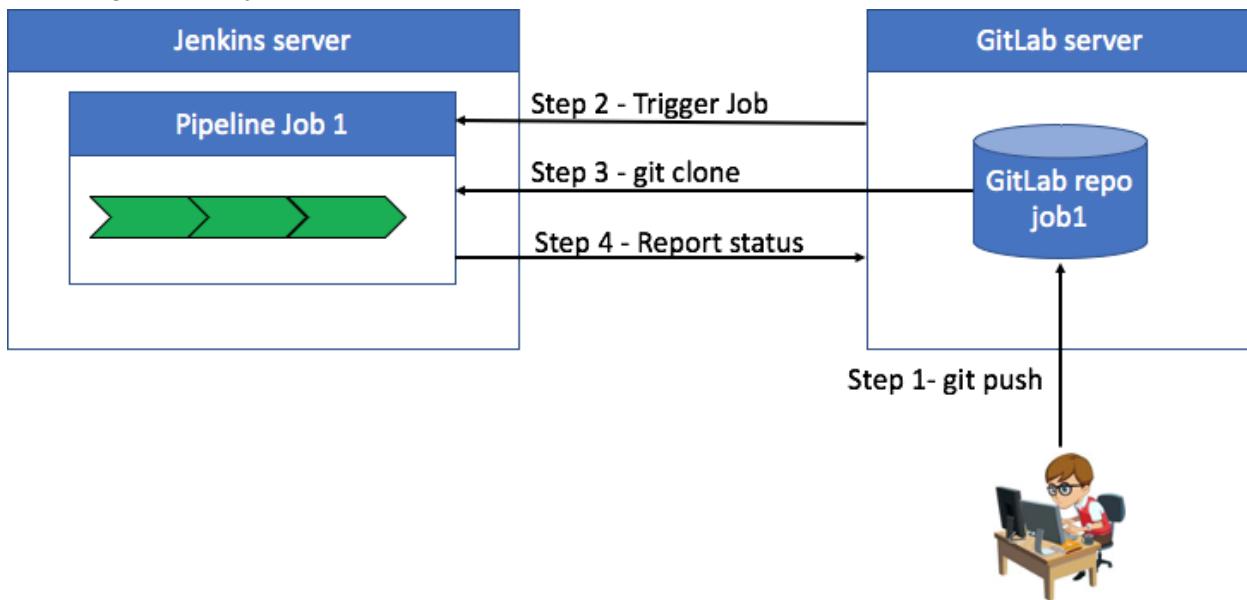


Note! Jenkins doesn't execute the job immediately. It polls for updates once a minute. You may be needed to wait for one more minute.

Using GitLab Jenkins plugin

This chapter describes a full integration between GitLab and Jenkins via “GitLab Jenkins plugin”. This method allows to trigger a Jenkins job on push or merge request event from GitLab. In that case, Jenkins reports back pipeline status to GitLab.

The integration way is shown below:



Install GitLab Jenkins plugin

GitLab Jenkins integration is achieved by an external plugin named “GitLab plugin”. The plugin needs to be installed in advance. The installation process is shown below:

- Log in Jenkins
- Click on “Manage Jenkins” on the left panel
- Click on Manage Plugins



Manage Plugins

Add, remove, disable or enable plugins that can extend the functionality of Jenkins.

⚠ There are updates available

- Click on the “Available” tab and type GitLab

Filter:

Updates	Available	Installed	Advanced
Install ↓			
Name			
Version			
Gitlab Authentication <input type="checkbox"/> This is the an authentication plugin using gitlab OAuth. 1.4			
GitLab <input type="checkbox"/> This plugin integrates GitLab to Jenkins by faking a GitLab CI Server. 1.5.9			
GitLab Logo <input type="checkbox"/> Display GitLab Repository Icon on dashboard 1.0.3			
Gitlab Merge Request Builder <input type="checkbox"/> Integrates Jenkins with Gitlab to build Merge Requests 2.0.0			
Violation Comments to GitLab <input type="checkbox"/> Finds violations reported by code analyzers and comments GitLab merge requests with them. 2.4			
Gitlab Hook <input type="checkbox"/> Enables Gitlab web hooks to be used to trigger SMC polling on Gitlab projects <input type="checkbox"/> Warning: This plugin version may not be safe to use. Please review the following security notices: <ul style="list-style-type: none"> • Gitlab API token stored and displayed in plain text 1.4.2			

Install without restart

Download now and install after restart

Update information obtained: 4 hr 58 min ago

Check now

- Choose all GiLab related plugins

Filter:

Updates	Available	Installed	Advanced
Install ↓			
Name			
Version			
Gitlab Authentication <input checked="" type="checkbox"/> This is the an authentication plugin using gitlab OAuth. 1.4			
GitLab <input checked="" type="checkbox"/> This plugin integrates GitLab to Jenkins by faking a GitLab CI Server. 1.5.9			
GitLab Logo <input checked="" type="checkbox"/> Display GitLab Repository Icon on dashboard 1.0.3			
Gitlab Merge Request Builder <input checked="" type="checkbox"/> Integrates Jenkins with Gitlab to build Merge Requests 2.0.0			
Violation Comments to GitLab <input checked="" type="checkbox"/> Finds violations reported by code analyzers and comments GitLab merge requests with them. 2.4			
Gitlab Hook <input checked="" type="checkbox"/> Enables Gitlab web hooks to be used to trigger SMC polling on Gitlab projects <input checked="" type="checkbox"/> Warning: This plugin version may not be safe to use. Please review the following security notices: <ul style="list-style-type: none"> • Gitlab API token stored and displayed in plain text 1.4.2			

Install without restart

Download now and install after restart

Update information obtained: 4 hr 58 min ago

Check now

- Install the plugin by pressing “Download now and install after restart”. Click on “Restart Jenkins when installation is complete and no jobs are running”.

Installing Plugins/Upgrades

Preparation

- Checking internet connectivity
- Checking update center connectivity
- Success

GitLab

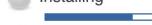
 Downloaded Successfully. Will be activated during the next boot

Gitlab Authentication

 Downloaded Successfully. Will be activated during the next boot

GitLab Logo

 Installing



Violation Comments to GitLab

 Pending

Gitlab Merge Request Builder

 Pending

Windows Slaves

 Pending

ruby-runtime

 Pending

Gitlab Hook

 Pending

Restarting Jenkins

 Pending

 Go back to the top page

(you can start using the installed plugins right away)

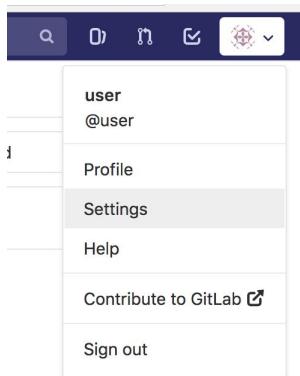
 Restart Jenkins when installation is complete and no jobs are running

- Wait until installation is done and Jenkins is restarted

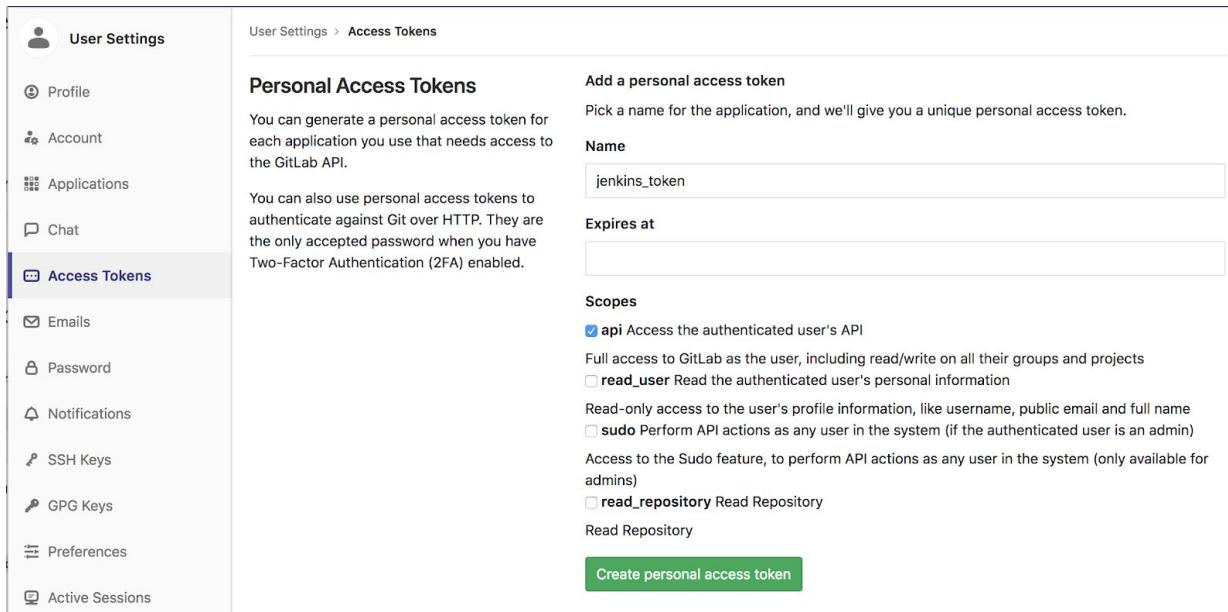
Configuring Jenkins to GitLab authentication

Next, we need to create a **GitLab API** token so that we can use it in Jenkins. We already gave you some examples of how to use personal access token in “Using GitLab API” chapter. Let’s create a token following simple steps.

- Open your GitLab home page at <http://gitlab.example.com/> and navigate to user settings at the top-right corner. Use username **user** and password **DevOps123**.



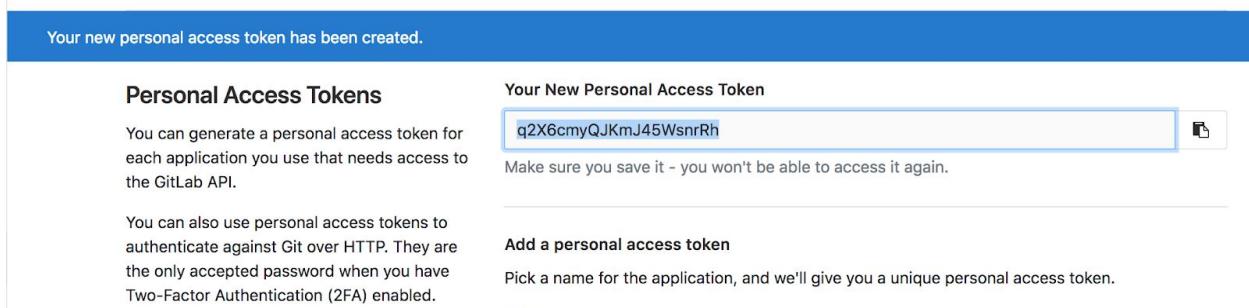
On the left sidebar go to **Access Tokens** and **create personal access token** with API access.



The screenshot shows the 'User Settings' page in GitLab, specifically the 'Access Tokens' section. On the left, there's a sidebar with links like Profile, Account, Applications, Chat, Access Tokens (which is selected and highlighted in blue), Emails, Password, Notifications, SSH Keys, GPG Keys, Preferences, and Active Sessions. The main content area has a header 'User Settings > Access Tokens'. Below it, there's a section titled 'Personal Access Tokens' with a brief description: 'You can generate a personal access token for each application you use that needs access to the GitLab API.' Another section below it says: 'You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.' To the right, there's a form for adding a new token. It includes fields for 'Name' (containing 'jenkins_token'), 'Expires at' (empty), and a 'Scopes' section with several checkboxes. One checkbox is checked: 'api Access the authenticated user's API'. Other options include 'read_user', 'sudo', 'read_repository', and 'Read Repository'. A green 'Create personal access token' button is at the bottom of the form.

Once the token is generated you need to copy it to a temporary location or leave this tab open because once you close this page, you won't be able to see this token anymore.

The following step is to configure Jenkins to connect to GitLab using API token we just created.



The screenshot shows the 'Personal Access Tokens' page again, but this time it displays a message: 'Your new personal access token has been created.' Below this message, there's a section for 'Your New Personal Access Token' which contains the token value 'q2X6cmYQJKmJ45WsnrRh' in a text input field. Next to the input field is a small icon of a clipboard with a pencil. Below the token value, there's a note: 'Make sure you save it - you won't be able to access it again.' At the bottom of the page, there's a link to 'Add a personal access token' and a note: 'Pick a name for the application, and we'll give you a unique personal access token.'

Configuring GitLab to Jenkins authentication

In the previous Chapter, we already discussed two types of credentials. One of them is Jenkins user Database and another one is Jenkins credentials database to store data such as certificates, usernames, passwords, SSH Keys, and API tokens. This is where we store GitLab API token we just generated.

From the Jenkins Homepage at <http://jenkins.example.com:8080/> click **Credentials**, in the page that appears, click on (global).

The screenshot shows the Jenkins interface. On the left, there's a sidebar with various icons and links: New Item, People, Build History, Manage Jenkins, My Views, Open Blue Ocean, Credentials (which is currently selected), System, and New View. The main content area is titled "Credentials". It has a header with "T", "P", and "Store ↓" and an icon selection dropdown labeled "Icon: S M L". Below this, it says "Stores scoped to Jenkins". A table lists credentials, with one row highlighted for "Jenkins".

And then **Add Credentials**

- Back to credential domains
- [Add Credentials](#)

In the credential **Kind** drop-down menu choose **GitLab API token** and fill in all the fields and press save.

- API token: <paste GitLab token we have just created>
- ID: **gitlab.example.com**
- Description: **GitLab Server**

The dialog box has a "Kind" dropdown set to "GitLab API token". Under "Scope", it says "Global (Jenkins, nodes, items, all child items, etc)". The "API token" field contains a redacted value. The "ID" field is set to "gitlab.example.com" and the "Description" field is set to "Gitlab server". At the bottom right is a blue "OK" button.

From the Jenkins Homepage at <http://jenkins.example.com:8080/> click on **Manage Jenkins** and from there press **Configure System**. That will get you to Jenkins system configuration page

Manage Jenkins

This is a screenshot of the "Manage Jenkins" configuration page. It features a large "Configure System" button with a gear icon. Below the button, there is descriptive text: "Configure global settings and paths.".

Scroll down a little bit till **GitLab** section.

The screenshot shows the Jenkins configuration interface for the GitLab plugin. It includes fields for 'Connection name' (empty), 'Gitlab host URL' (empty), and 'Credentials' (set to 'none'). Error messages indicate that 'Gitlab connection name required.', 'Gitlab host URL required.', and 'API Token for Gitlab access required.'.

We got this new section enabled once we installed the **GitLab plugin** in the last chapter.

Note: If it happens that you do not have this section in Jenkins, just navigate to Jenkins plugin manager, install GitLab plugin, and restart Jenkins server.

Fill in all the information for GitLab connectivity and press **Test Connection**:

- Enable authentication for '/project' end-point: **unchecked**
- Connection name: **gitlab_server**
- GitLab host URL: **http://gitlab.example.com/**
- Credentials: **GitLab API token (GitLab Server)**

Press “**Test Connection**” to verify connectivity to Gitlab server from Jenkins.

The screenshot shows the Jenkins configuration interface for the GitLab plugin. The 'Connection name' is set to 'gitlab_server', 'Gitlab host URL' is 'http://gitlab.example.com/', and 'Credentials' is 'GitLab API token (GitLab server)'. The 'Test Connection' button is highlighted in blue, and the status message 'Success' is displayed below it.

If you see a **Success** message, then you are on the right path and we have established connectivity with Gitlab Server.

Automatic job triggering via GitLab

At this moment we have Jenkins server to be able to download Jenkinsfile along with all the code from GitLab server and execute it when we run Jenkins jobs manually or execute pipeline if the new code is available.

Can we do better? Sure, we can configure Jenkins to automatically start the job when we push the changes to GitLab and this is what we do next. It looks very similar as we've done before. But there is a big difference. This method allows to trigger job and report job status back to GitLab. This also allows configuring integration in a better way. For example, verify code on merge request event.

Allowing remote API calls to Jenkins

First, we need to update Jenkins job to accept remote calls from GitLab.

- Go to Jenkins pipeline3 configuration settings at <http://jenkins.example.com:8080/job/pipeline3/configure>.
- Go down to **Build Triggers** section and select **Build when a change is pushed to GitLab** checkbox.

Build Triggers

Build after other projects are built

Build periodically

Build when a change is pushed to GitLab. GitLab webhook URL: <http://jenkins.example.com:8080/project/pipeline3>

When the new set of option appears, leave only **Push Events** selected.

Build when a change is pushed to GitLab. GitLab webhook URL: <http://jenkins.example.com:8080/project/pipeline1>

Enabled GitLab triggers	
Push Events	<input checked="" type="checkbox"/>
Opened Merge Request Events	<input type="checkbox"/>
Accepted Merge Request Events	<input type="checkbox"/>
Closed Merge Request Events	<input type="checkbox"/>
Rebuild open Merge Requests	Never
Approved Merge Requests (EE-only)	<input type="checkbox"/>
Comments	<input type="checkbox"/>
Comment (regex) for triggering a build	Jenkins please retry a build

Click on Advanced and Generate a new Token by pressing on the “Generate” button.

The screenshot shows the Jenkins job configuration page. Under the 'Build Triggers' section, there is a 'Secret token' field containing a long string of characters: 'b9cee608c88019f36b5c72867e0a0323'. To the right of this field are two buttons: 'Generate' (highlighted with a blue border) and 'Clear'.

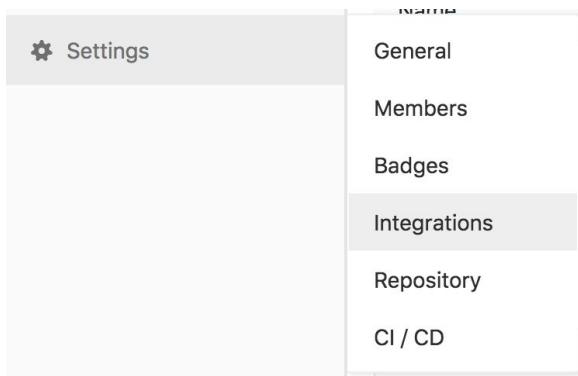
Note! You need to copy this secret token since it will be required in GitLab project configuration

Save the changes.

Note! We are going to show several ways of integration, the build trigger configuration will be changed several times.

Configuring GitLab Webhooks

Webhook is a POST request triggered by a specific event. In our case, GitLab triggers a webhook to start Jenkins job, once we push new code changes to GitLab. From GitLab **first_repo** at <http://gitlab.example.com/user/pipeline3>, navigate to **Settings -> Integrations**.



In the integration section add a new webhook with the following parameters:

- URL: <http://jenkins.example.com:8080/project/pipeline3>
- Token: Jenkins secret token we created during job configuration
- Trigger - leave only **push events** selected
- SSL Verification - **unchecked**.

user > pipeline3 > Integrations Settings

Integrations

Webhooks can be used for binding events when something is happening within the project.

URL: http://jenkins.example.com:8080/project/pipeline3

Secret Token:

Use this token to validate received payloads. It will be sent with the request in the X-Gitlab-Token HTTP header.

Trigger

- Push events**: This URL will be triggered by a push to the repository
- Tag push events**: This URL will be triggered when a new tag is pushed to the repository
- Comments**: This URL will be triggered when someone adds a comment
- Confidential Comments**: This URL will be triggered when someone adds a comment on a confidential issue
- Issues events**: This URL will be triggered when an issue is created/updated/merged
- Confidential Issues events**: This URL will be triggered when a confidential issue is created/updated/merged
- Merge request events**: This URL will be triggered when a merge request is created/updated/merged
- Job events**: This URL will be triggered when the job status changes
- Pipeline events**: This URL will be triggered when the pipeline status changes
- Wiki Page events**: This URL will be triggered when a wiki page is created/updated

SSL verification: Enable SSL verification

Add webhook

Note! GitLab highlights that URL is blocked:

Url is blocked: Requests to the local network are not allowed

By default, GitLab is not allowed to send webhooks over LAN subnets. That's strange... but we can fix it.

Login as **root** user and go to the **admin area**.



On the sidebar press **Settings => Network => Outbound requests**.

Settings

User and IP Rate Limits: Configure limits for web and API requests. **Expand**

Outbound requests: Allow requests to the local network from hooks and services. **Expand**

Click on **Expand** button. From there, select **Allow requests to the local network from hooks and services** and save the changes. That was not so obvious, right?

Login back as **user** and go to **pipeline3** integrating settings at <http://gitlab.example.com/user/pipeline3/settings/integrations> and create a webhook one more time.

In the integration section add a new webhook with the following parameters:

- URL: <http://jenkins.example.com:8080/project/pipeline3>
- Token: Jenkins secret token we created during job configuration (b9cee608c88019f36b5c72867e0a0323)
- Trigger - leave only **push events** selected
- SSL Verification - **unchecked**.

Press “Add WebHook” once you’ve done with parameters. The webhook should be added and shown if you scroll down:

The screenshot shows the Jenkins pipeline3 settings page under the integrations tab. It displays a single webhook configuration. The URL is <http://jenkins.example.com:8080/project/pipeline3>, the trigger is set to "Push Events", and "SSL Verification" is disabled. There are "Test" and "Edit" buttons, along with a delete icon.

You may emulate an event by pressing on “Test”. Click on “Test/Push Events”:

The screenshot shows the Jenkins pipeline3 settings page with the "Test" button highlighted. A dropdown menu is open, listing various event types: Push events, Tag push events, Issues events, Confidential issues events, Note events, Merge requests events, Job events, Pipeline events, and Wiki page events. The "Push events" option is currently selected.

And you should see the following message:

Hook executed successfully: HTTP 200

If you go back to the Jenkins **pipeline3** activity at <http://jenkins.example.com:8080/job/pipeline3/> you should see that the new job was started and executed properly and job description says “Started by GitLab push by Administrator”.

The screenshot shows the Jenkins pipeline3 build history. It lists a single build, #5, which was triggered on Sep 15, 2018 at 11:14 PM. The status of the build is "Started by GitLab push by user".

Now we are ready to do the change in our project scripts and push the changes back to GitLab. In this change, we want to add a simple python script and pipeline to check script syntax. This is enough for this stage.

Update repository files

Go back to the repository and add required files as show below:

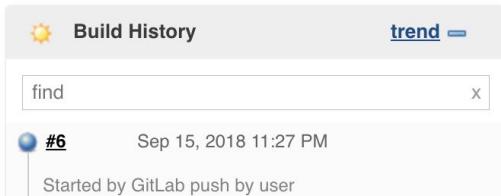
```
$ cd pipeline3
$ mkdir src
$ cat << EOF > src/simple_script.py
#!/use/bin/python
print("Hello word")
EOF

$ cat <<EOF > Jenkinsfile
node {
    stage("Checkout") {
        checkout scm
    }
    stage("Verify") {
        sh 'python -m py_compile src/*.py'
    }
    stage("Cleanup") {
        sh 'true'
    }
}
EOF

$ git add Jenkinsfile src/*
$ git commit -m "added python script"
$ git push origin master
```

Note! We highlighted as red a command to verify python syntax

Done! You may want to make sure that Jenkins pipeline has been executed successfully. Open the line <http://jenkins.example.com:8080/job/pipeline3/> and make sure that the latest version of the pipeline has been executed. You can see that it was triggered by GitLab:



The screenshot shows the Jenkins Build History page. At the top, there's a search bar with 'find' and a clear button 'X'. Below the search bar, a blue circular icon with '#6' indicates the latest build. To the right of the icon, the text 'Sep 15, 2018 11:27 PM' is displayed. Below this, the text 'Started by GitLab push by user' is shown. The overall interface is light gray with blue highlights for active links and buttons.

Here is a snippet of text job output:

```

> git rev-list --no-walk 5b4492321e3c9313fb63437b399e0551bc864862 # timeout=10
[Pipeline]
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Verify)
[Pipeline] sh
[Pipeline] [pipeline3] Running shell script
+ python -m py_compile src/simple_script.py
[Pipeline]
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Cleanup)
[Pipeline] sh
[Pipeline] [pipeline3] Running shell script
+ true
[Pipeline]
[Pipeline] // stage
[Pipeline]
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

We just configured Jenkins and GitLab to work together. Jenkins pipeline now checks that Python code can be compiled. The job is started automatically triggered by GitLab push event. You just became one step closer towards your automation of everything. There are few more section to be done before we finish up in this chapter.

Updating GitLab progress from Jenkins

Now we have Jenkins jobs triggered by GitLab webhooks automatically and everything looks good. The only problem is that Jenkins does not provide any pipeline progress and status feedback back to GitLab. In order to fix this problem, we use two new steps called **gitlabBuilds** and **gitlabCommitStatus**.

gitlabBuilds

gitlabBuilds notifies GitLab about Jenkins stage in progress. **GitLabBuilds** Basically reports to GitLab that Jenkins job has been started and notifies about the stages it started executing. **gitlabBuilds DOES NOT** notify GitLab about the status of every stage. Pipeline reports the status via **gitlabCommitStatus**. The syntax of **gitlabBuilds** is as following:

```

gitlabBuilds(builds: ["stage_name"]) {
    stage("stage_name") {
        gitlabCommitStatus(name: "stage_name") {
            <YOUR CODE GOES HERE>
        }
    }
}

```

Note: This is not the final pipeline.

gitlabCommitStatus

gitlabCommitStatus step notifies updates stage status that was created by **gitlabBuilds** step. Whether it passed, failed or needs an approval, **gitlabCommitStatus** is going to report back to GitLab. The syntax for **gitlabCommitStatus** is as following:

```
stage("stage_name") {  
    gitlabCommitStatus(name: "stage_name") {  
        <YOUR CODE GOES HERE>  
    }  
}
```

Note: This is not the final pipeline.

Using gitlabBuilds and gitlabCommitStatus

Update Jenkinsfile in **pipeline3** repo to show you how **gitlabBuilds** works alone. Update your Jenkinsfile with the following content:

```
node {  
    stage("Checkout") {  
        checkout scm  
    }  
    gitlabBuilds(builds: ["Verify"]) {  
        stage("Verify") {  
            gitlabCommitStatus(name: "Verify") {  
                sh 'python -m py_compile src/*.py'  
            }  
        }  
        gitlabBuilds(builds: ["Cleanup"]) {  
            stage("Cleanup") {  
                gitlabCommitStatus(name: "Cleanup") {  
                    sh 'sleep 60'  
                }  
            }  
        }  
    }  
}
```

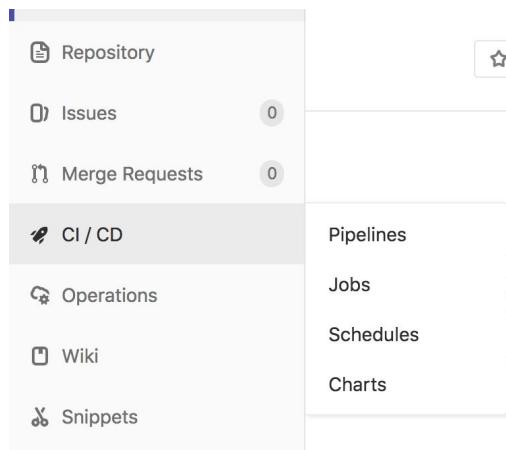
```
}
```

Note! make sure you put your `gitlabCommitStatus` or other similar steps after the SCM step that clones your project's source. Otherwise, you may get HTTP 400 errors, or you may find build status being sent to the wrong repo.

Add, commit and push the changes to GitLab.

```
$ git commit -a -m 'Introducing gitlabBuilds' && git push origin master
...
<OUTPUT OMITTED>
...
To gitlab.example.com:root/first_repo.git
 034f9a7..a249a9b  master -> master
```

This should trigger the Jenkins job. Jenkins itself should be notifying GitLab about the progress of this job. Navigate back to GitLab first_repo at <http://gitlab.example.com/user/pipeline3>, at sidebar click on **CI/CD -> Pipelines**.



In GitLab pipelines page, we should be able to see our Jenkins job status in **running** state.

user > pipeline3 > Pipelines				
All 2	Pending 0	Running 1	Finished 1	Branches Tags
Status	Pipeline	Commit	Stages	
running	#2 by latest	master -> 73153b48 Introducing gitlabBuilds		21 minutes ago X
failed	#1 by	master -> db68a1c7 Introducing gitlabBuilds		22 minutes ago

Click on **running** to go to pipeline details.

running Pipeline #2 triggered 21 minutes ago by user

Introducing gitlabBuilds

2 jobs from master in 0 seconds

-o 73153b48 ...

Pipeline Jobs 2

External

Cleanup Verify

This screenshot shows the GitLab CI interface for a pipeline named 'Pipeline #2'. At the top, it indicates the pipeline is 'running' and was triggered 21 minutes ago by a user. The main title is 'Introducing gitlabBuilds'. Below this, it shows '2 jobs from master in 0 seconds' and a specific commit '73153b48'. A navigation bar at the bottom says 'Pipeline Jobs 2'. Under the 'External' section, there are two stages: 'Cleanup' (which is currently running) and 'Verify'.

These are our 2 stages. You may note that **Cleanup** stage is still in progress (marked as green). After some time pipeline will be in **passed** state:

Status	Pipeline	Commit	Stages
passed	#2 by latest	master -o 73153b48 Introducing gitlabBuilds	✓

This screenshot shows the same pipeline interface as before, but the 'passed' status is now displayed in a green box. The 'Cleanup' stage is marked with a green checkmark, indicating it has completed successfully. The 'Verify' stage is also marked with a green checkmark. The commit details remain the same: 'master -o 73153b48' and 'Introducing gitlabBuilds'.

If you click on **passed**, you may see details list of stages.

Introducing gitlabBuilds

2 jobs from master in 0 seconds

-o 73153b48 ...

Pipeline Jobs 2

External

Cleanup Verify

This screenshot shows the same pipeline interface as the previous one, but the 'passed' status is now displayed in a green box. The 'Cleanup' stage is marked with a green checkmark, indicating it has completed successfully. The 'Verify' stage is also marked with a green checkmark. The commit details remain the same: 'master -o 73153b48' and 'Introducing gitlabBuilds'.

This method doesn't require to go to Jenkins each every time you need to know build status.

Integrating Jenkins pipelines with merge requests

We did a great job by configuring automatic pipelines to check our application against syntax checks. The pipeline allows us to verify application code on every commit. This is great but still, it is not enough to build a great automation workflow.

We need to understand how developers use GIT in the production environment. In the real world, none of the team commit everything to **master** branch and that's the key. Usually, every feature is developed in a feature branch. Once feature code is ready it is merged via Merge Requests. Developers create merge request and leave build and test activities for CI/CD. This is why CI/CD pipelines should be integrated with Merge Request functionality of GitLab. This is something we are going to focus on.

This chapter explains Jenkins and GitLab integration approach which is applicable for most of production CI/CD implementations. This assumes that every code will be delivered as a separate GIT branch in a GitLab repository. Developers can commit to this branch many times a day but it will not trigger automatic pipelines. The only one way to merge changes is to create a merge request. We will configure GitLab to start pipelines on any merge request events. Merging will not be allowed if tests are not passed.

Application

For education purposes, we will still use the same python script. The script will emulate a real application. We need to check that script syntax is OK.

Configure GitLab WebHook

We need to modify the WebHook we recently created for the **pipeline3** repository.

- Go to <http://gitlab.example.com/user/pipeline3/settings/integrations>
- Scroll down to see existing webhooks. You should see a webhook like that:

The screenshot shows the 'Webhooks' section of a GitLab repository settings page. It displays a single webhook entry for 'http://jenkins.example.com:8080/project/pipeline3'. The webhook is triggered by 'Push Events'. The 'SSL Verification' dropdown is set to 'disabled'. There are 'Edit', 'Test', and delete ('trash') buttons next to the webhook URL.

- Click Edit to modify web hooks settings
- Check "Merge request events" and "Pipeline events", uncheck "Push events":

Trigger

Push events
This URL will be triggered by a push to the repository

Tag push events
This URL will be triggered when a new tag is pushed to the repository

Comments
This URL will be triggered when someone adds a comment

Confidential Comments
This URL will be triggered when someone adds a comment on a confidential issue

Issues events
This URL will be triggered when an issue is created/updated/merged

Confidential Issues events
This URL will be triggered when a confidential issue is created/updated/merged

Merge request events
This URL will be triggered when a merge request is created/updated/merged

Job events
This URL will be triggered when the job status changes

Pipeline events
This URL will be triggered when the pipeline status changes

Wiki Page events
This URL will be triggered when a wiki page is created/updated

- Click “Save changes”

Well, now we modified events which GitLab will handle and execute the Webhook.

Configure GitLab merge settings

Now we need to allow merging only if the pipeline is completed successfully. GitLab allows to disable Merge Request **Merge** button and wait until the pipeline is successfully complete. In case if pipelines failed, The “Merge” button will be disabled. This allows protecting GitLab repository against accepting nonworking code.

Let's configure the **pipeline3** repository in that way.

- Open repository link <http://gitlab.example.com/user/pipeline3>
- Click “Settings” => “General”
- Click the “Expand” button near “Merge request”

Merge request

[Expand](#)

Customize your merge request restrictions.

- Check “**Only allow merge requests to be merged if the pipeline succeeds**” and press “Save changes”

Merge request

[Collapse](#)

Customize your merge request restrictions.

Merge method

Merge commit

A merge commit is created for every merge, and merging is allowed as long as there are no conflicts.

Merge commit with semi-linear history

A merge commit is created for every merge, but merging is only allowed if fast-forward merge is possible. This way you could make sure that if this merge request would build, after merging to target branch it would also build.

When fast-forward merge is not possible, the user is given the option to rebase.

Fast-forward merge

No merge commits are created and all merges are fast-forwarded, which means that merging is only allowed if the branch could be fast-forwarded.

When fast-forward merge is not possible, the user is given the option to rebase.

Only allow merge requests to be merged if the pipeline succeeds

Pipelines need to be configured to enable this feature. [?](#)

Only allow merge requests to be merged if all discussions are resolved

Automatically resolve merge request diff discussions when they become outdated

Show link to create/view merge request when pushing from the command line

[Save changes](#)

*Note! We do not need other features related to **merge requests**.*

Configure Jenkins job parameters

Jenkins job needs to be configured to accept merge request data. This requires to change the number of job parameters. For example, the job should be aware of the files stored in merge request source branch. Also the job should accept triggering by a merge request event.

First, we need to change job triggers:

- Open Job configuration <http://jenkins.example.com:8080/job/pipeline3/>
- Click Configure on the left panel to modify job parameters
- Modify build triggers as shown below:

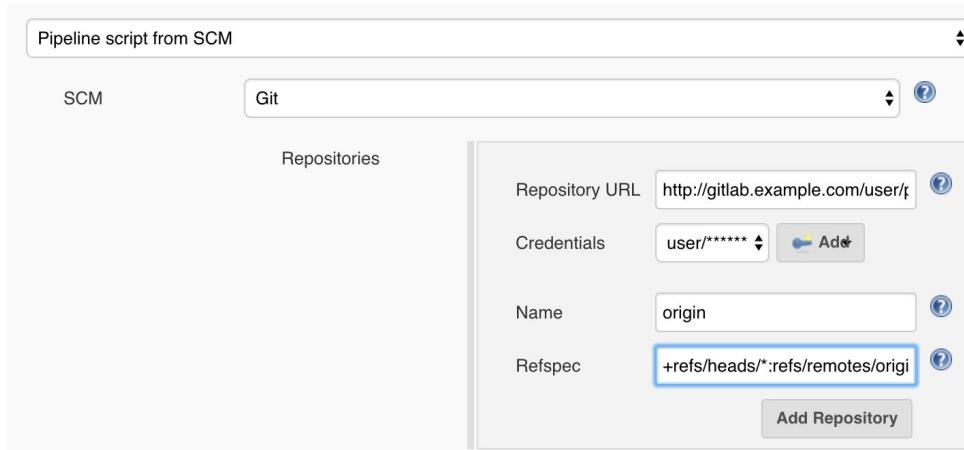
The screenshot shows the Jenkins 'Build Triggers' configuration page. It lists three main trigger types: 'Build after other projects are built', 'Build periodically', and 'Build when a change is pushed to GitLab'. The third option is selected and includes a 'GitLab webhook URL' input field containing 'http://jenkins.example.com:8080/project/pipeline3'. Below this, under 'Enabled GitLab triggers', several event types are checked: 'Push Events', 'Opened Merge Request Events', 'Accepted Merge Request Events', 'Closed Merge Request Events', 'Rebuild open Merge Requests' (with a dropdown menu set to 'On push to source branch'), 'Approved Merge Requests (EE-only)', and 'Comments'. At the bottom, there is a 'Comment (regex) for triggering a build' input field with the placeholder 'Jenkins please retry a build' and an 'Advanced...' button.

Note! “Rebuild open Merge Requests” is set to “On Push to source branch”. This feature allows to rerun opened pipeline if the new code is available in source branch.

Jenkins job should understand from where to clone code for the pipeline. With **merge requests** code is checkout from special repositories. You may find a details description at GitLab Plugin home page <https://github.com/jenkinsci/gitlab-plugin#git-configuration>. Right now we need to change the number of Job Git-related settings:

- Scroll down to Pipeline settings
- Click “Advanced” in “Repositories” section:
 - Name: **origin**
 - Refspec: “`+refs/heads/*:refs/remotes/origin/*`
`+refs/merge-requests/*:head:refs/remotes/origin/merge-requests/*`”

Note: Refspec value is a single line inside double quotes “”



The screenshot shows the Jenkins Pipeline configuration for an SCM repository. Under the 'Repositories' section, there is one entry for 'origin'. The 'Repository URL' is set to `http://gitlab.example.com/user/`. The 'Credentials' dropdown is set to 'user/*********' with an 'Add' button. The 'Name' is set to 'origin'. The 'Refspec' field contains the value `+refs/heads/*:refs/remotes/origin/* +refs/merge-requests/*:head:refs/remotes/origin/merge-requests/*`. A blue box highlights this 'Refspec' field. At the bottom right of the repository card is a 'Add Repository' button.

- Modify “Branch to build” to “`origin/${gitlabSourceBranch}`”



The screenshot shows the Jenkins Pipeline configuration for 'Branches to build'. There is one entry with a red 'X' icon. The 'Branch Specifier (blank for 'any')' field contains the value `origin/${gitlabS}`. Below it is an 'Add Branch' button.

- Click “Save” to apply settings we have just done

Listed above settings are mandatory to handle merge requests data properly. GitLab defines a number of variables and passes them to Jenkins. For example, GitLab defines `gitlabBranch`, `gitlabSourceBranch`, `gitlabTargetBranch`, and many others.

Creating a merge request

We are going to slightly update our pipeline to perform python lint and syntax testing. Additional software is required on **Jenkins VM**:

```
$ sudo yum install pylint -y
```

```
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
...
output omitted for brevity
```

...
Complete!

Note! pylint allows checking the code against python best practices

Once pylint software is installed we can update our pipeline.

Since we want to use merge request functionality we need to create a new branch which contains all new features.

First, we need to create a new branch named “feature1” from Jenkins VM:

```
$ cd ~/pipeline3
$ git checkout -b feature1
Switched to a new branch 'feature1'
```

Now update our Jenkinsfile. Please note that we added pylint shell command.

```
$ cat Jenkinsfile
node {
    stage("Checkout") {
        checkout scm
    }
    gitlabBuilds(builds: ["Verify"]) {
        stage("Verify") {
            gitlabCommitStatus(name: "Verify") {
                sh 'python -m py_compile src/*.py'
                sh 'pylint src/*.py'
            }
        }
        gitlabBuilds(builds: ["Cleanup"]) {
            stage("Cleanup") {
                gitlabCommitStatus(name: "Cleanup") {
                    sh 'sleep 60'
                }
            }
        }
    }
}
```

Commit and push changes to GitLab:

```
$ git add Jenkinsfile
$ git commit -m "added pylint"
$ git push origin feature1
Username for 'http://gitlab.example.com': user
Password for 'http://user@gitlab.example.comoutput omitted for brevity
...
\* \[new branch\]    feature1 -> feature1
```

We can create now a merge request to merge changes from our “feature1” branch to the master branch. This may be achieved by GitLab Web UI:

- Open your browser at http://gitlab.example.com/user/pipeline3/merge_requests

The screenshot shows the GitLab Merge Requests page. At the top, there is a message: "You pushed to **feature1** 10 hours ago". To the right of this message is a blue "Create merge request" button. Below this, there are filter buttons for "Open 0", "Merged 0", "Closed 4", and "All 4". To the right of these buttons are "Edit merge requests" and a green "New merge request" button. At the bottom of the page, there is a search bar with the placeholder "Search or filter results..." and a dropdown menu for "Created date".

- Press “Create merge request” blue button
- Give a title and merge request description:

New Merge Request

From **feature1** into **master**

[Change branches](#)

Title

Enabled pylint

Start the title with **WIP:** to prevent a **Work In Progress** merge request from being merged before it's ready.

Add [description templates](#) to help your contributors communicate effectively!

Description

[Write](#) [Preview](#)

B I 11 <> 11 11 11 11 11

This merge request adds python lint testing|



Markdown and quick actions are supported

[Attach a file](#)

- Make sure that Source branch is “feature1” and destination branch is “master”

Source branch

feature1

Target branch

master

[Change branches](#)

Remove source branch when merge request is accepted.

Squash commits when merge request is accepted. [About this feature](#)

[Submit merge request](#)

[Cancel](#)

- Create a merge request by “Submit merge request”

GitLab triggers Jenkins pipeline automatically. You may see progress for the pipeline in Jenkins:

The screenshot shows two Jenkins build entries in the "Build History" interface. Both builds were triggered by GitLab Merge Request #5, specifically from user/feature1 to master. Build #7 was triggered on Sep 16, 2018 at 4:28 AM, and build #6 was triggered on Sep 15, 2018 at 11:27 PM. Build #7 is currently in progress, indicated by a blue progress bar. The Jenkins UI includes a search bar, a "trend" dropdown, and standard navigation controls like arrows and a refresh button.

Build ID	Triggered By	Timestamp
#7	GitLab Merge Request #5: user/feature1 => master	Sep 16, 2018 4:28 AM
#6	GitLab push by user	Sep 15, 2018 11:27 PM

Our pipeline fails:



Note! This is expected behavior because of additional testing performed by pylint

Click on **Verify** stage and “Logs” and see the output snipped. Similar information can be produced by “pylint src/*.py”

```
[pipeline3] Running shell script
+ pylint src/simple_script.py
No config file found, using default configuration
*****
Module simple_script
C: 2, 0: Unnecessary parens after 'print' keyword (superfluous-parens)
C: 1, 0: Missing module docstring (missing-docstring)
```

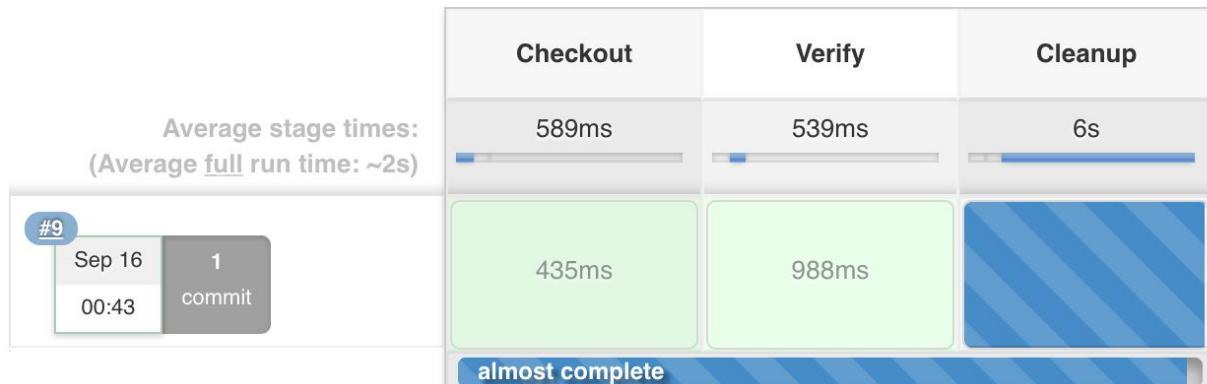
It looks like “pylint” dislikes our code. We are using python2 in our examples. Let’s modify the script to be aligned with pylint requirements:

```
$ cat src/simple_script.py
#!/use/bin/python
"""Example script to learn CI/CD processes """
print "Hello world"
```

Commit and push changes to **feature1** branch.

```
$ git add src/*.py
$ git commit -m "fixed pylint issues"
$ git push origin feature1
```

Jenkins starts pipeline automatically.



You may also see progress from GitLab merge request view:

The screenshot shows a GitLab merge request interface. At the top, it says "Request to merge feature1 into master". Below that, a pipeline status card says "Pipeline #7 running for 40a47d3e on feature1". In the middle section, there's a "Merge" button with a checkmark, followed by dropdown menus for "Merge when pipeline succeeds" (set to "Merge when pipeline succeeds"), "Remove source branch" (unchecked), "Squash commits" (unchecked), and "Modify commit message". A note at the bottom says "You can merge this merge request manually using the command line".

Once pipeline succeeds you may merge the code by pressing “Merge”:

The screenshot shows a GitLab merge request interface for a merge request opened 10 hours ago by a user. It has "Open" and "Close merge request" buttons. Below the header, the merge request title is "Enabled pylint". A note says "This merge request adds python lint testing".

The screenshot shows a GitLab merge request interface for a merge request opened 10 hours ago by a user. It has "Open" and "Close merge request" buttons. Below the header, the merge request title is "Enabled pylint". A note says "This merge request adds python lint testing".

Let's merge the code. You can check “remove source branch” box if you want, this will remove **feature1** branch after the merge is complete and our changes are pushed to master branch.

More options of integration

You may always visit the home page of GitLab plugin for advanced options:
<https://github.com/jenkinsci/gitlab-plugin>

Conclusions

The configuration we have covered in this book is very similar to what a lot of companies are using in production. Integration with merge requests allows every developer to trigger pipeline on merge request event. Pipeline highlights all problems and GitLab is configured to not allow merging code which is not ready for production. If an error occurs, GitLab prevents pressing “Merge” button by disabling that. This gives an additional layer of readiness testing.