GitLab Repository Management

Delve into managing your projects with GitLab, while tailoring it to fit your environment

Jonathan M. Hethey

[PACKT] open source*

PUBLISHING

GitLab Repository Management

Table of Contents

GitLab Repository Management

GitLab Repository Management

Credits

Author

Jonathan M. Hethey

Reviewers

Jeroen van Baarsen

Eric Pidoux

Acquisition Editor

Rubal Kaur

Commissioning Editors

Neha Nagwekar

Deepika Singh

Technical Editors

Nikhil Potdukhe

Sonali Vernekar

Copy Editors

Roshni Banerjee

Janbal Dharamraj

Dipti Kapadia

Kirti Pai

Project Coordinator

Joel Goveya

Proofreader

Clyde Jenkins

Indexer

Tejal Soni

Graphics

Rounak Dhruv

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

Jonathan M. Hethey has been writing code since the age of 14 and has actively participated in shaping and experimenting with the IT systems around him. After finishing his finals and an apprenticeship as an IT supporter, he studied Multimedia Design in Kolding, Denmark, followed by studies of Web Development in Odense, Denmark.

Besides his studies, he has been working on several web projects—both on a freelance basis and in his company which was founded in 2011.

Because programming and working in teams efficiently are keys to success in a rapidly changing industry, whether we are speaking of app development or creating beautiful web experiences, he quickly became a fan of Git, and started looking for great implementations of it on the server-side that would allow newcomers to pick it up quicker and extend it with additional functionality and usability.

I would like to thank the open source community and, of course, especially the creators of GitLab for their great work. Secondly, I would thank the team of and around Linus Torvalds, who created Git in the first place. Also, Git and GitLab are only my personal favorite flavors, there are other projects doing a fantastic job out there! Last, but not least, I have to thank everybody around me who have supported me while I was writing this book!

About the Reviewers

Jeroen van Baarsen works as a Ruby developer at Mobillion. He started working as a developer in the PHP world. After working for seven years as a PHP developer, he made the switch to become a Ruby developer.

He first used Version 4.0 of GitLab and has upgraded to every version ever since.

Mobillion is a company that specializes in helping charity organizations raise funds for their campaigns. They offer a social fund raising platform to do so.

Eric Pidoux has a degree in Computer Science from Miage Aix-Marseille and is currently working as the Lead Web Developer in Lausanne (Switzerland), especially on Symfony2 framework (six years into PHP with four years into Symfony). Passionate about IT, he developed a lot of websites, including mobile apps using Sencha Touch, and managed them with GitLab.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

In this book, we will take a tour of the version control system GitLab, which is based on Git. It's open source and has many great features that we will learn chapter wise as listed here. First, we will take a look at what GitLab can do for us, and later we will see how we can install it on a server and configure it to match our needs.

The next step is to take a close look at the web interface, and then we will learn how to handle permissions and teams, document our code, track issues, and show example workflows.

Lastly, we will learn how to perform maintenance for our installation, including the creation of backups and upgrading to the most recent version. We will also cover getting in touch with the community and developers on the respective channels.

What this book covers

[Chapter 1](), *Kickstarting with GitLab,* covers a short introduction of GitLab, its key features, and origin of the project.

[Chapter 2](), *Installation,* covers the preparation of the environment and installation of the dependencies and GitLab itself on a Linux box.

[Chapter 3](), *Configuring GitLab,* explains the setting up of GitLab's configuration and making it match our needs in terms of security; we will also learn what setting can be changed from where.

[Chapter 4](), *Roles and Permissions,* covers exploring of the web interface. We will also learn how to use roles inside GitLab to distribute power, access restrictions, and set up teams that work nicely.

[Chapter 5](), *Issues and Wikis,* covers the documentation of software and its development through built-in wikis. Use of issue tracking inside GitLab and assigning issues to developers will also be covered in this chapter.

[Chapter 6](), *Workflows,* explores the different styles of working with Git and GitLab. We will also learn how to find the right workflow and see how GitLab supports this.

[Chapter 7](), *Updating GitLab,* covers the creation of backups, stop services, and updates for the latest version. We will also run tests to see if everything is up to speed.

[Chapter 8](), *Help and Community,* covers the connection with the community and developers. We will also come to know the different channels and where we can submit feature requests or make contributions to the project.

What you need for this book

We will need either a computer or virtual machine that is running Debian or Debian derivative Linux, such as Ubuntu. Furthermore, we'll need an Internet connection for the download of the required software.

Who this book is for

Do you want to drive Git to its full potential? Do you want to involve your team members at different skill levels, take work off your day-to-day workflow with Git, and migrate to it with an approachable interface? That's what you can do with GitLab, and this book is for you then!

Also, if you want to gain experience with one of the best possibilities of creating a server's IDE infrastructure for code version control, this comprehensive guide will provide steps to follow and possibilities to explore.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Some examples of these styles are given here with an explanation of their meaning.

Code words in text are shown as follows: "The great thing about Git is that it will install all versions at once, and you can choose between them at any time by running the `checkout` command for the version compatible to GitLab 5.0."

A block of code is set as follows:

```
git remote add origin ssh://user@host:3101/git/example
```

Any command-line input or output is written as follows:

```
cd homegit/gitlab
sudo gem install charlock_holmes --version '0.6.9.4'
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "By navigating to the **Issues** tab of a repository in the web interface, you can easily create new issues."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files emailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Kickstarting with GitLab

Keeping your code close and private, when you want to, is surely as important as easily sharing it and letting the community contribute. GitLab (http://gitlab.org) understands this and lets you self-host Git repositories that fit your company or team. In this chapter, we will cover what is GitLab and the points that stand out the most when compared to its competitors.

Topics covered in this chapter are as follows:

- GitLab features
- Where GitLab excels
- Cloud-hosted GitLab
- Support for your own GitLab
- Competitors

In this chapter, we will take a look at what features are available, how you can use them, and which competitors are worth examining.

What is GitLab?

GitLab is a system for managing Git repositories. It's written in Ruby, and allows you to deploy meaningful version control for your code easily.

It was first published on GitHub in October 2011, and has grown into a powerful tool since then. GitLab is published under the MIT license, but it is mandatory to keep a mention of the author when redistributing the code.
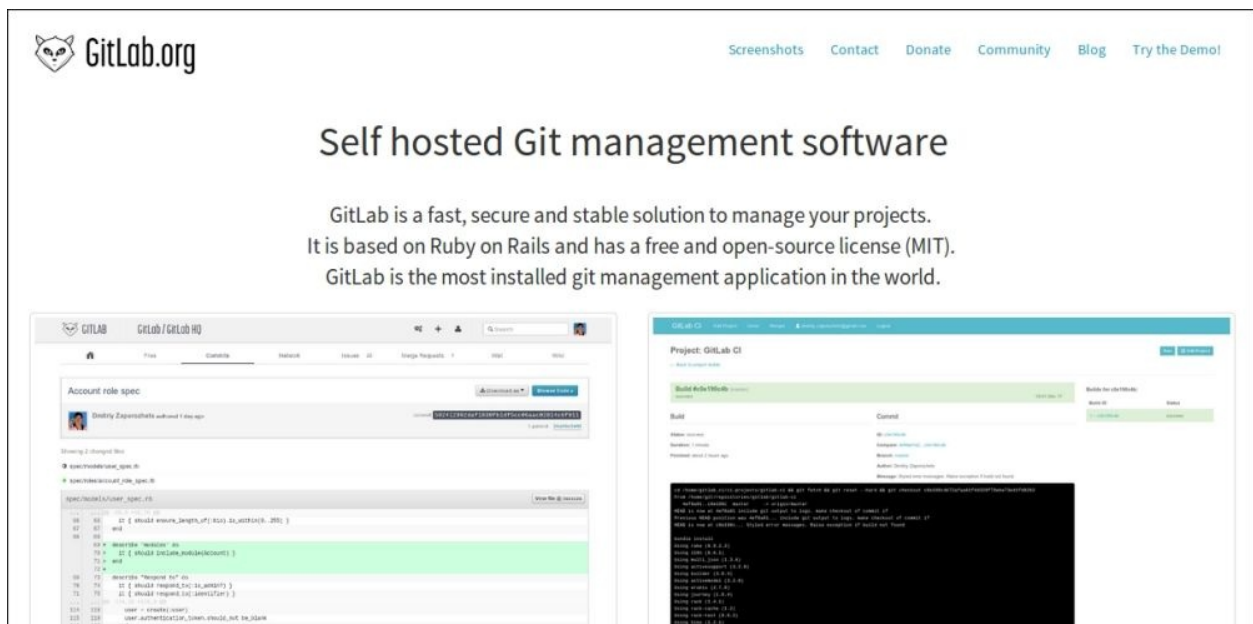
The lead developer is *Dmitriy Zaporozhets,* who also runs a hosted platform for GitLab installations at gitlab.com.

GitLab features

GitLab has many features. The most outstanding ones, surely, are the user-friendly web interface and the possibilities of managing permissions. The built-in features for documentation of projects or tracking needed changes with the issue tracker are very strong points for GitLab.

Web interfaces

The web interface resembles the one of GitHub in many cases, which has proven very functional to many developers.

It is very consistent and plain, in spite of the quantity of functions it contains. It's possible to open merge requests for branches, view diffs, or even edit files right in the web interface, and then stage the changes to a commit.

Managing permissions

Who has the permission to do what, and how easy is it to add and change team members? These are the key questions you should ask yourself when dealing with administering users on any platform.

GitLab has some very well-designed roles, and allows you to apply these. From guest to master, you have diverse possibilities to limit and grant access to functions such as pushing to repositories or which branches a developer can push to, so that you can control which code ends up in your product or project.

Documenting your project

Documenting projects can be a difficult thing to track, because it is often not described as an important task from the start; but, with GitLab you have great tools for making it more meaningful and faster to create. Through built-in issue tracking and wikis, you can cross reference files, commits, and of course other wiki pages. These are great tools that can enable your team to keep everybody on the same page and prevent projects, no matter how complex, from growing into something that is hard to understand.

Where GitLab excels

Anyone who develops software or manages software development can benefit from using GitLab. The core concept of Git is to get distributed to keep revisions of your code maintainable and make a team work together more efficiently.

Git is great at these things. The problem, which especially is an obstacle for beginners, is that it does not present this very clearly.

Everything it shows is in a terminal window, and without using additional software around it. It's not a very intuitive experience if you don't have years of experience with the command-line interface or similar tools.

GitLab is like a user-friendly layer on top of Git, which provides ease of use, easily managing permissions with clicks, instead of long shell commands. It has a much more visual workflow that increases your speed and ease of working with Git.

Also, it makes it much easier to help the newly hired developers to get started, or update the head of your department on the state of a project—all in the browser. Developers who join a project months after it has been kicked off can easily get an overview, check the progress, and understand it quickly.

GitLab allows you to use the full potential of Git, without demanding you to work only in the terminal on the server side.

Cloud-hosted GitLab

If you choose not to host GitLab yourself, but to let the people who know it best host it for you, you can subscribe to a dynamic subscription model at http://www.gitlab.com/cloud/. Hosting is free up to 10 users, which gives small development teams a chance to try it out before making a commitment. In case you need more contributors, you can sign up for a paid plan.

At the time of writing this, up to 15 users is as cheap as 9 USD a month; whereas, a maximum of 25 will cost you 49 USD; and the largest solution with support of a 1000 contributors is 1999 USD a month.

Support for your own GitLab

If you need support for your own installation of GitLab along the way, a support subscription right now (at the time of writing) is priced at 149 USD a month, according to http://www.gitlab.com/subscription/.

Consider a support plan if you have a system administrator who does not want to compromise other tasks for either troubleshooting or optimizing the new GitLab installation.

Competitors

GitLab is not the only solution of this kind. It's my favorite because of the ease of installation and maintenance, the easy-to-grasp interface, and it is quick in production. Another open source solution you can deploy yourself is Gitorious. You can take a look at their free web hosting solution for public repositories.

GitHub, of course, is an outstanding service in terms of usability, design, and much more; but, it only offers free hosting of public repositories. It is a platform of choice when you are aiming for gathering contributors around your project, because it has such a large community.

Summary

Now you've read what GitLab is all about and how it is available, both for you to install and as a hosted service. We've scratched the surface of features that grant administrators flexibility and control and the interfaces that provide usability for the users.

In the next chapter, we are going to cover the installation process and how to set up your own copy of GitLab on a server of your choice.

Chapter 2. Installation

Installing GitLab is fairly easy. It does not come with a one-click installer, but I will guide you through each step, one by one.

The time required for this chapter depends both on bandwidth and the performance of your hardware. It took me around 30 minutes to complete these steps on a virtual private server running **Debian**.

One of the most time-consuming parts is compiling Ruby from source; apart from that, various package downloads will take time to complete, depending on your network connection speed. These occasions can be used wisely to get another coffee to stay sharp and focused along the installation.

In this chapter, the following topics will be covered:

- Hardware requirements
- Supported operating systems
- Setting up dependencies
- Compiling a recent Ruby version
- Installing database servers
- Installing GitLab through Git
- Running the test suite

Hardware

For the hardware, you need to run GitLab on a quad-core processor with 1 gigabyte of RAM, as officially recommended.

However, you will be able to run GitLab with both fewer cores and less memory. We will take a closer look at that in the chapter on configuration.

With 1.5 GB of memory, you should be able to support 1,000 and more users. This is stated by the author, *Dmitriy Zaporozhets*, considering that he runs probably the biggest installation of GitLab at http://gitlab.com, I'll take his word for it.

Operating system – Linux

Like many services and applications, GitLab also runs best on Linux, which provides us with a stable and modular platform.

If you're on a Mac, and you want to test GitLab locally, feel free to try and install carefully; because it not officially supported, projects such as Homebrew may provide you with the needed packages.

On Windows, your only solution is to run a virtual machine with one of the supported operating systems and install GitLab within it.

Leave any fear of the command line behind, because we'll get to use that in a little while now!

Debian/Ubuntu

The installation instructions in this book are going to be tailored for Debian and Ubuntu, mainly due to the use of the APT package manager.

We need to download a few packages with APT before we can proceed with the installation. First of all, we make sure that we have our system up-to-date by running the following commands as the root user:

```
apt-get update
apt-get upgrade
apt-get install sudo
```

Tip

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Make sure the editor of your choice is installed too, because we will need to edit configuration files on the way; officially recommended is **Vim**, but if you feel more comfortable in **nano** or others, go with that.

```
sudo apt-get install -y vim
```

The preceding line will install Vim for you.

```
sudo apt-get install -y nano
```

This will install a little simpler, yet functional nano editor.

The required packages

Installing the required packages through APT is easy, and they are dependencies of GitLab. By running the command line below, you will install all of the listed packages, without getting prompted for confirmation. Make sure no server, such as Apache, is running on port 80 already, because Nginx will also try to use port 80 by default.

```
sudo apt-get install -y build-essential zlib1g-dev libyaml-dev
libssl-dev libgdbm-dev libreadline-dev libncurses5-dev libffi-dev
curl git-core openssh-server redis-server postfix checkinstall
```

```
libxml2-dev libxslt-dev libcurl4-openssl-dev libicu-dev nginx
```

The preceding command will install tools to build programs from source, the git-core package that is necessary to use Git, and also server applications, such as SSH, Postfix for sending e-mails, and Redis server. Lastly we install Nginx, which should only be installed if you choose to run GitLab with Nginx as the web server. If you choose to use Apache, either because of personal preference or because it is necessary on the host system, please do not install Nginx, but refer to some unofficial documentation in the GitLab recipes repository on GitHub at https://github.com/gitlabhq/gitlab-recipes.

Other distributions

GitLab certainly works with Linux distributions other than Debian or Ubuntu, but it's not officially supported by the author.

If you run another Linux distribution and wish to use it with GitLab, I recommend checking out the respective wiki pages of your distribution. Examples of that are the Arch Linux wiki and the Gentoo wiki.

One of the biggest differences between many distributions compared to Debian and Ubuntu is that they rely on systemd instead of init/sysvinit. This means that the init script described in the next chapter will not work on these distributions and you have to either find a script that supports systemd in the GitLab recipes repository on GitHub at https://github.com/gitlabhq/gitlab-recipes or write it yourself.

Python

Due to different naming conventions, installing Python might involve an extra step for you. Here are the steps we will go through to satisfy the Python dependency:

1. Install Python.
2. Test the installed version.
3. If necessary, install another version.
4. Ensure the right version is executable by Python 2.

```
sudo apt-get -y install python
```

Now, we test which version has been just installed:

```
python --version
```

This will show an output such as Python 2.7.1. If the first digit is 3, we need to install Python 2.7.x separately.

```
sudo apt-get –y install python2.7
```

Because some Linux distributions and their versions install Python 2 as `python`, because it still is their default, we test if Python 2 is working.

```
python2 --version
```

If we get the **command not found** error, we just need to link the name `python2` to `python`, as shown in the following command line:

```
sudo ln -s usrbin/python usrbin/python2
```

To confirm that this part of the installation was successful, just make sure that the command `python2 --version` is working properly.

Ruby

Even though you have Ruby available in your APT package manager, it is recommended that you build it from source. Ruby is the programming language in which GitLab is written; therefore, the author has decided to stay informed on the updates of the language as much as possible.

We will go through the following steps to ensure we satisfy the Ruby dependency:

1. Download the Ruby source code.
2. Prepare dependencies for the installation.
3. Install the recent Ruby version.
4. Test if the Ruby version is correct.

Download and compile

The up-to-date installation instructions will be linked at the GitLab repository at GitHub at all times, so be sure to look at the required Ruby version there. In the following example, it is `ruby-1.9.3-p392`:

```
mkdir tmpruby && cd tmpruby
curl --progress http://ftp.ruby-lang.org/pub/ruby/1.9/ruby-1.9.3-
p392.tar.gz | tar xz
cd ruby-1.9.3-p392
./configure
make
sudo make install
```

These commands create a temporary directory for the Ruby installation, download the specified source code version through `curl`, and continue to compile it with the `make` command.

Databases

You need multiple database systems to run GitLab, which may sound unusual, but they are very different databases.

MySQL is used for long-term storage of data, where Redis is responsible for queueing up the jobs, distributing them over several processes, and to ensure fast response time.

We will go through the following steps to install MySQL for GitLab:

1. Install the MySQL client and server.
2. Create the GitLab database user.
3. Generate a secure password.
4. Create the GitLab database.
5. Grant permissions to the GitLab database user.
6. Test the connection.

MySQL

Installing the MySQL server is easy and is followed by creating a dedicated user for GitLab.

Do not just use a common user or even root to do this, it's a potential security vulnerability.

```
sudo apt-get -y install mysql-server mysqlclient libmysqlclient-dev
```

Running the preceding code will install the MySQL server and client, with developer extensions on your machine, after which you can login (this time as root) to create a user for GitLab's storage with the following commands:

```
    ## Login to MySQL
mysql -u root -p
```

This logs you into your running MySQL server with your root password, which you defined in the dialogue while installing the MySQL packages.

```
    ## Create a user for GitLab. (change $password to a real
password)
    mysql> CREATE USER 'gitlab'@'localhost' IDENTIFIED BY
'$password';
```

Security notice: Common recommendations for strong passwords are at least 12 random characters, including lower-and uppercase letters, numbers, and special characters. This kind of password can be generated by the little utility pwgen, for example, by writing:

```
pwgen -yn 12
```

It will provide you with a long list of passwords that resembles the following:

```
    Ezei\m2eeph2 oo-Fe$M0Aili yo8gaxai&F4a xoo1aiTe>ifu Eeb-
aeTee4ro Shae[Ph5eeR2
    aiXi8Ap#aora iec[ae6Fah5a Ahc1vig;ohce Xaih#oh7aiG{
eic]oCoon7hi ze8Eiw4gahf_
    ohf5Ijo2mee/ ahgha8yieHa-Ee9eo3eer=ah iLu7Ohch6Fo/ Sas0ohj7Aij\
```

Pick a strong password generated by either pwgen or a similar online service and save it.

Now, all that is left is creating the database and giving the freshly created user

permissions over it.

```
  # Create the GitLab production database
    mysql> CREATE DATABASE IF NOT EXISTS `gitlabhq_production`
DEFAULT CHARACTER SET `utf8` COLLATE `utf8_unicode_ci`;

  # Grant the GitLab user necessary permissions on the table.
    mysql> GRANT SELECT, LOCK TABLES, INSERT, UPDATE, DELETE,
CREATE, DROP, INDEX, ALTER ON `gitlabhq_production`.* TO
'gitlab'@'localhost';
```

With the preceding code, in case it does not already exist, let's say from a previous installation, we create a database named `gitlabhq_production_`, succeeded by granting the `mysql` user `gitlab` the necessary privileges over the database.

After completing these operations successfully, you can quit the command-line connection to your MySQL server with the following command line:

```
mysql> \q
```

Testing the connection

To make sure nothing unexpected has occurred and check whether the new GitLab user has permissions we want him/her to have, we test the connection as following:

```
sudo -u git -H mysql -u gitlab -p -D gitlabhq_production
```

Now being prompted for the password we assigned previously, we should see if we succeed using the following command line:

```
    mysql>
```

Now you're logged in with the `mysql` client for the command line as the user `gitlab`.

If you don't see the output as mentioned earlier, please make sure you have assigned the correct user credentials, especially the password; additionally, double-check the database name. You can delete databases by running:

```
mysql> DROP DATABASE `gitlabhq_production
```

Use this with care, because it will delete all the data in that database.

Redis

Redis is probably the easiest setup. It's done already, since we installed the package `redis-server` in the beginning.

GitLab will handle the rest of it, unless you want Redis to run on a different server, which might be the case if you have to handle a lot of users at an instance.

Redis on Debian 6.0 Squeeze

To install a more up-to-date and compatible version of Redis on Debian Squeeze, which at the moment is widespread, especially through the VPS-hosting landscapes, we have to install `redis-server` from the backports repository in the following way:

```
echo "deb http://backports.debian.org/debian-backports squeeze-
backports main" >> etcapt/sources.list
apt-get update
apt-get -t squeeze-backports install redis-server
```

In the first line, we add the repository to our list of available package sources at *etc*`apt/sources.list`; after that, update the index used by the APT package manager, and finally install `redis-server`.

Omitting this step on **Debian Squeeze** will cause errors when trying to run the configured installation, especially the web frontend.

GitLab Shell

GitLab shell is an essential component, that replaced gitolite as part of the project in the Version 5.0. It handles the connection between the web interface and the command-line components of the overall functionality.

First, we have to make sure that we are logged in as the `git` user in his/her home directory with the following commands:

```
sudo su git
cd homegit
```

To install it, we simply have to clone the repository on which it is published using the following command:

```
git clone https://github.com/gitlabhq/gitlab-shell.git
```

Now, we change to the directory to which the repository was cloned, using the following line:

```
cd homegit/gitlab-shell
```

Choosing the right version

After successfully cloning the repository, we need to find the version that corresponds with the version of GitLab we're trying to run. To find out which version it is, look at the file *docs*install/installation.md in the repository of `git labhq`.

A list of compatible versions at the time of writing are:

- gitlabhq 5.0: gitlab-shell v1.1.0 (stable)
- gitlabhq 5.1: gitlab-shell v1.2.0
- gitlabhq 5.2: gitlab-shell v1.4.0 (assumed)

The great thing about Git is that it will install all versions at once, and you can choose among them at any time by running the `checkout` command for the version compatible to GitLab 5.0:

```
git checkout -b v1.1.0
```

Make sure to choose a version according to release of GitLab you want to run! However, we will check this later, using the provided built-in tests.

GitLab

Just as we install the GitLab Shell, installing the GitLab repository also requires cloning the repository as the `git` user.

Again, first we make sure we are in the home directory of the `git` user:

```
cd homegit
sudo -u git -H git clone https://github.com/gitlabhq/gitlabhq.git gitlab
```

Again, just like when we installed GitLab Shell, we have to make sure we are in the right directory and also, again, as the `git` user switch to the desired branch of the Git repository.

```
cd homegit/gitlab
sudo -u git -H git checkout 5-0-stable
```

This will switch the repository to the `5.0-stable` branch of the project.

Gem dependencies

Lastly, we have to make sure that the Ruby packages on which GitLab depends are installed on our host system. To do so, we install the `bundler` gem up front:

```
sudo gem install bundler
```

When the gem has been installed, we can proceed to install the dependencies listed in Gemfile at `homegit/gitlab/`. We do so by letting the Ruby package manager download and install the respective components:

```
 cd homegit/gitlab
sudo gem install charlock_holmes --version '0.6.9.4'
```

Depending on the database server you choose to use, either install the gem for MySQL or PostgreSQL connection:

```
sudo -u git -H bundle install --deployment --without development test mysql
sudo -u git -H bundle install --deployment --without development test postgres
```

Summary

Congratulations! You have now successfully created a powerful server environment, set up a database connection, and downloaded the required Git repositories that contain GitLab's source code and you're almost ready to run it.

What is ahead of us now is tying it all together and configuring everything the way you need it to work.

Chapter 3. Configuring GitLab

The configuration of GitLab happens mostly through the config files, if the settings do not concern repositories or user permissions. You can edit these through the web interface.

Setting and editing the options will mostly happen through a text editor; it's up to you if you want to edit the files right on the server or download them to your computer first to edit them with the editor with which you are most comfortable. We will cover components such as GitLab Shell, different database systems, and the web servers Puma and Nginx.

In this chapter, we will cover the following topics:

- Configuring the base components and creating the directories they require to operate
- Setting up databases and Ruby processes to handle our data and connections
- Looking at the connections handled by the Secure Shell host protocol and how to authenticate with it
- Lastly, setting up Nginx to display the web interface

Configuring the parts

We will now go through the configuration of the different components which are required to run GitLab. For the configuration, we will mostly be using the recommended settings, but we will look at the parts which need to be changed for each setups. If not stated otherwise, the directory we will be working on is *home*git/.

GitLab Shell

The `gitlab-shell` configuration is quite short compared to other components; it has some vital security settings available.

The storage path of the Git repositories is also controlled by `gitlab-shell`.

In the config file, `config.yml` under *home*git/gitlab-shell, you can define or change the Linux system user on which you want GitLab to operate. If you're migrating from another platform, you might want to keep your original Git user and can redefine it here, if you've done so in the previous steps of the installation.

The `gitlab_url` setting must be the same as the domain that is pointed at the server on which GitLab is running, because this address is necessary for API calls between the web interface and the developer's clients communicating through HTTP or SSH.

Since we have already downloaded the `gitlab-shell` repository, let's change the directory and copy the config file:

```
cd homegit/gitlab-shell/
cp config.yml.example config.yml
```

To complete the configuration of `gitlab-shell`, we run the built-in `install` command:

```
./bin/install
```

And the output would be:

```
mkdir -p homegit/repositories: true
mkdir -p homegit/.ssh: true
chmod 700 homegit/.ssh: true
touch homegit/.ssh/authorized_keys: true
chmod 600 homegit/.ssh/authorized_keys: true
chmod -R ug+rwX,o-rwx homegit/repositories: true
find homegit/repositories -type d -print0 | xargs -0 chmod g+s:
true
```

Permissions and directories

To ensure that GitLab has the necessary permissions over folders to store temporary data, process IDs, and sockets, we run the following command:

```
cd homegit/gitlab
sudo chown -R git log/
sudo chown -R git tmp/
sudo chmod -R u+rwX  log/
sudo chmod -R u+rwX  tmp/
```

Now the directories for logs and temporary data have been given the correct permissions for the Git user to store the data in them.

GitLab makes use of a folder named `gitlab-satellites`, which are copies of repositories used for handling branches and merges. The folder containing these mirrored copies will be stored in the `home` folder of the Git user.

We create the folder by running the following command:

```
sudo -u git -H mkdir homegit/gitlab-satellites
```

To manage PIDs of the processes and the sockets created when starting and running GitLab, we need to create directories and set correct privileges using the following commands:

```
sudo -u git -H mkdir tmp/pids/
sudo -u git -H mkdir tmp/sockets/
sudo chmod -R u+rwX  tmp/pids/
sudo chmod -R u+rwX  tmp/sockets/
```

To ensure the functionality of backups, we create the uploads folder under the public folder in the same manner:

```
sudo -u git -H mkdir public/uploads
sudo chmod -R u+rwX  public/uploads
```

To later create a backup, you can execute the `backup:create` function like so:

```
sudo -u git -H bundle exec rake gitlab:backup:create
RAILS_ENV=production
```

Databases

To configure your database differently than in the setup process, you need to choose your service (MySQL/MariaDB or PostgreSQL) and edit the corresponding file which is suffixed appropriately.

```
# Mysql
sudo -u git cp config/database.yml.mysql config/database.yml

# PostgreSQL
sudo -u git cp config/database.yml.postgresql
config/database.yml
```

To change the configuration for use with a MySQL database, open the file `database.yml.mysql` by navigating to `gitlab/config`. In order to configure the PostgreSQL connection, you open the `database.yml.postgresql` file under `gitlab/config`.

MySQL

Due to the spread of MySQL or MariaDB, we will use the configuration of those as a reference here. The database name must be the same as defined in the installation. The same goes for username and password.

If you're running your database on a server other than the GitLab application, you must remove the comment symbol # on the line for the host and fill in either an IP address or domain of the host on which your database resides. For a custom location of the socket, define it as an absolute path.

```
database: gitlabhq_production
pool: 10
username: gitlab
password: "secure password"
# host: localhost
# socket: tmpmysql.sock
```

Puma

Puma is the web server that delivers the web interface to both administrators and users through Nginx. Puma is built for concurrency, which means it can deal with a lot of users at once, compared to its competitive projects. The GitLab project switched to Puma in Version 5.1.

```
# Copy the example Puma config
sudo -u git -H cp config/puma.rb.example config/puma.rb
```

In the Puma configuration file, you will find multiple ways to alter the performance of GitLab. You can define the number of threads, even as a range. By default, this option is set from 0 to 16 threads. To have more operations running parallel within GitLab, you can set up multiple workers for Puma. These options should be adjusted carefully, and even for two workers, at least 1.5 GB of RAM is recommended.

GitLab itself

Configuring GitLab is achieved through editing a number of config files that we've partly been in touch with during the installation. The main config file is `gitlab.yml` under `gitlabhq/config`.

This file will not be touched by updates, because it is created from the example `gitlab.yml.example` during the installation.

To create the config file, you simply copy it from the example file and fill out the necessary blanks and placeholders. First, we create the new file from the template `gitlab.yml.example` by typing the following command:

```
cd home git/gitlab/config
cp gitlab.yml.example gitlab.yml
```

Now, we open the main configuration file with our favorite text editor. You can feel free to either download and later upload the file or use an available editor on the server. I will use Vim, because it is available within the terminal and also on the server side. If you have a desktop environment available, you can also use your favorite editor.

Now, let us open the config file using the command:

```
vim gitlab.yml
```

If you want to open the config file with another editor, let's say nano, use the following command:

```
nano gitlab.yml
```

Luckily, it has a very comprehensive configuration file that allows us to quickly make the necessary and also the nice-to-have adjustments. The necessary adjustments can be made by filling in the variables for **host** and **email_from**.

Because these are the most vital adjustments to make, we will return to this file at a later point to adjust any of the following points:

- Security and port configuration
- Setting up an issue tracker
- Authentication with LDAP or OAuth
- Placement of satellites and backups

```
    cd homegit/gitlab/
    sudo -u git -H bundle exec rake gitlab:setup
RAILS_ENV=production
```

During this process, you will be prompted to make sure you intend to create a new set of tables in the database:

This will create the necessary database tables and seed the database.

You will lose any previous data stored in the database.

Do you want to continue (yes/no)?

On a clean installation, proceed with **yes**. If you run this command at a later point, make sure to have a backup of your database, or answer **no**.

Secure Shell host protocol

As with other Git servers, you can manage the access of your users through SSH. SSH is a secure communication protocol used primarily to establish encrypted connections to servers in order to administrate them.

Because you don't want to manually administer user accounts on your server or have your users enter a password every time they connect, the connections and permissions will be granted through the shared Git user.

Default port

Because the private/public key authentication is prioritized, users will not be prompted for passwords if they have the other side of the key pair on their machine. We'll look at some of the basic options you can set through the global SSH server configuration file, `sshd_config`, commonly located at `etcssh`.

A common security practice is to change the port on which SSH runs, which can be done by changing the following line:

```
# What ports, IPs and protocols we listen for
Port 22
```

Important to mention is that your users will have to take a different port into account in either their Git remotes or SSH configurations.

For adding a remote repository from Git, the port of the SSH connection can be appended to the username, as in the following line:

```
git remote add origin ssh://user@host:3101/git/example
```

Where `3101` would be the new port.

Users can also define a port for a specific host globally in their `~/.ssh/config` file, which will be used for SSH connections.

```
Host example.com # top level domain
    Port 3101
```

Alternatively, you can define a port while connecting through SSH using the `-p` argument as follows:

```
ssh -p 3101 user@example.com
```

The port recognized by GitLab can be changed in the `config` file at
*home*git/gitlab/config/gitlab.yml.

Key storage

The location of the authorized keys for users can be changed, and this will also
decide where the keys of users registered through the web interface will be
stored. The path is relative to each user's directory, and can be altered through
uncommenting and editing the following line in the `sshd_config`:

```
#AuthorizedKeysFile     %h/.ssh/authorized_keys
```

Changing this after GitLab installation has been in production, which means that
you have to copy the SSH public keys of your users into the file at the new
location, because they will not be able to access repositories otherwise.

Nginx

Meeting another high point of software built for scale and concurrency, the last configuration before being able to start our freshly installed GitLab instance is Nginx.

The sample configuration file for the chosen version of GitLab is shipped within the repository for you. Once again, we don't have to come up with the configuration, but merely fill in the values that are important to us.

By copying the example config to your Nginx `sites-available` folder and linking it to the `sites-enabled` directory, you will start serving it once Nginx is restarted. So as the root user, we now copy the Nginx configuration and link it properly.

```
cp homegit/gitlab/lib/support/nginx/gitlab etcnginx/sites-
available
ln -s etcnginx/sites-available/gitlab etcnginx/sites-
enabled/gitlab
```

Now, let's edit the configuration and make sure we fill in all points that are fully written in capital letters.

```
vim etcnginx/sites-available/gitlab
```

YOUR_SERVER_IP and **YOUR_SERVER_FQDN** need to be altered, so they reflect the IP address and fully qualified domain name (FQDN) of your server.

Finding IP and FQDN

Typically, you would know the IP address of the server on which you intend to run GitLab and also the fully qualified domain name, but in case you have not yet decided on the domain name or are new to the server environment on which you are installing GitLab, there are simple ways to find both IP and FQDN.

Find your IP address by running the following command or by logging into your host provider's administration panel, finding the respective server:

```
ifconfig | grep inet
```

You will get a list of interfaces listed, but the line that starts with `inet` will reveal your server's IP address.

An example of the output is as shown:

```
inet addr:127.0.0.1  Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
inet addr:192.168.1.75  Bcast:192.168.1.255  Mask:255.255.255.0
inet6 addr: fe80::a617:31ff:fe53:a9ef/64 Scope:Link
```

The address that is not `127.0.0.1`, will be the IP address for which you're looking. If your server is a VPS, your network interface might be named **venet0:0** or similar.

Your FQDN is the domain that you are pointing at the server. If you are on a local network or are trying out GitLab on your local machine, the following command should tell you the hostname, which will work too:

```
cat etchostname
```

Finish by restarting the Nginx server, to make it recognize the new site:

```
sudo service nginx restart
```

Starting GitLab

Because GitLab, as you have noticed throughout the installation and configuration, consists of many components, we have to install an init script that starts these in the right order and flattens them to one command. Let's install the init script from the repository:

```
    sudo curl --output etcinit.d/gitlab
https://raw.github.com/gitlabhq/gitlabhq/5-2-
stable/lib/support/init.d/gitlab
    sudo chmod +x etcinit.d/gitlab
```

Remember to replace `5-2-stable` with the version you want to use.

Testing the configuration

We can test the system with the built-in functionality and configuration test scripts, both while installing and when in production.

The two commands, first the less, and second the more elaborate one, are as follows:

```
# less verbose
sudo -u git -H bundle exec rake gitlab:env:info
RAILS_ENV=production

# more verbose
sudo -u git -H bundle exec rake gitlab:check
RAILS_ENV=production
```

An example output of these test scripts can be found in the `configuration_test` folder under `assets` of the included files.

If any of the tests fail, I recommend you to go back and carefully check if the respective component was installed correctly.

Starting up GitLab

If every check from the testing phase is completed flawlessly, we can proceed to start it up. Depending on your system setup, you can either use the service command or run the init script with an argument directly.

```
# either
sudo service gitlab start
# or
sudo etcinit.d/gitlab restart
```

Automatically start GitLab on system start-up

To ensure the GitLab service starts when your system starts, you need to add it to the list of services that automatically start when the operating system starts. To enable this auto-load ability, run the following parameters with the `update-rc.d` utility:

```
sudo update-rc.d gitlab defaults 21
```

Visit your site

That's it. You can go to your domain and see the following login site:

The default login, set at the installation is the following:

Username or Email: `admin@local.host`

Password: `5iveL !fe`

Feel free to look around, and look at the initial steps in the next chapter!

Summary

You have accomplished the configurations of all the pieces required to run GitLab now, from the command-line tools and protocols for GitLab Shell and SSH to the web-related services like Puma and Nginx. Further, you've had a look at possibilities to alter configurations. In the next chapter, we will learn about various roles and permissions that are granted in GitLab.

Chapter 4. Roles and Permissions

Roles and the permissions they grant over repositories in GitLab are some of the most powerful features of the whole service. In this chapter, we will look closely at the roles that are available and what they are individually capable of. We are going to go through setting up of users and their details, as well as how to combine multiple users into teams.

We will go through the following topics in this chapter:

- Creating your SSH key
- Adding users
- Enabling users to sign up
- The different permissions
- Managing teams
- Managing groups

First steps

Since we have just installed and configured GitLab, we would take one of our local Git repositories and attempt to push it to the fresh GitLab installation.

Logging in

The default login information for the freshly installed system is
`admin@local.host`, and the password is `5ifeL!fe`.

The next step is heading to your own profile in order to change your password.
For this, navigate to **Top Menu** | **My Profile**.

Creating your key

On any system which has SSH tools installed, you can easily generate an SSH key by running the following command:

```
ssh-keygen -t rsa -C "you@yourdomain.com"
```

Depending on your system, you will be asked where to save the generated key pair. By default, your keys would be saved in `.ssh` as `id_rsa` in your home directory. On Linux, it is present in *home*username/.ssh/, and on Mac OS, it is present in *Users*username/.ssh/.

After the key has been generated, we need to open the file of the pair that ends with `.pub`. Depending on your personal preference, either open it with your favorite editor or output it to the terminal by using cat:

cat *home*username/.ssh/id_rsa.pub

The output will be the public key, which must be copied into the GitLab interface in order to make the server recognize the private key that forms the other part of the pair. Thereby, it grants you the permission to make changes to the Git repositories or read nonpublic repositories on the server:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCj+ewVwLCD6nHQQcN6ZZIIaZni6D0QlPvZt6w
SzWJxX1W2sVrk5KPPHo4l0D3X0B5t6EQCQ8O3bTmmPW+Z/
11uMY1gqEUvI7MjAzFnI9u84tHppF8e7EIobtJlfI3e0axA8RlUTSXLPfP9e+K+HVpf
uitgQnZiOrZ8SNCSVcp991FwSB/7/HoQCZlqo6QJ9QhlABHUJJjG1fszhuvKhx1hZTH
S5Y9h5XssiG/VfVC/
PzmL3lhVMvKvNGitBrapS7PN0xFiXmFZuJP0dAOf53LKZewJmlxOyMSImw0WLDUtigZ
UgljfR you@yourdomain.com
```

In our browser window, we navigate to `yourdomain.com/keys` and press **Add New**, pasting the output of `id_rsa.pub` inside, as shown in the following screenshot:

The management of SSH keys is a very valuable feature for administrators since GitLab will automatically add the saved key to *home*git/.ssh/authorized_keys for us. Without a solution like GitLab, admins would have to manually add and remove keys from this file to grant or revoke access to the Git servers.

Pushing for the first time

When you create a new project, you will be shown a list of commands to connect a repository on your local machine to the server, or to create a new one.

The tutorial like and copy & paste ready screen looks as the following figure:

```
Create Repository

 mkdir lib_amazing
 cd lib_amazing
 git init
 touch README
 git add README
 git commit -m 'first commit'
 git remote add origin git@transb.in:jonathan/lib_amazing.git
 git push -u origin master

Existing Git Repo?

 cd existing_git_repo
 git remote add origin git@transb.in:jonathan/lib_amazing.git
 git push -u origin master

                                                    Remove project
```

Remember to add the SSH key to your agent using the following command before attempting to push. Otherwise, the server will reject your login attempt or ask for a password for the `git` user, at which point the authentication attempt through a key file will have failed.

Add the SSH key with a command like the following on your local machine:

```
ssh-add ~/.ssh/keyname
```
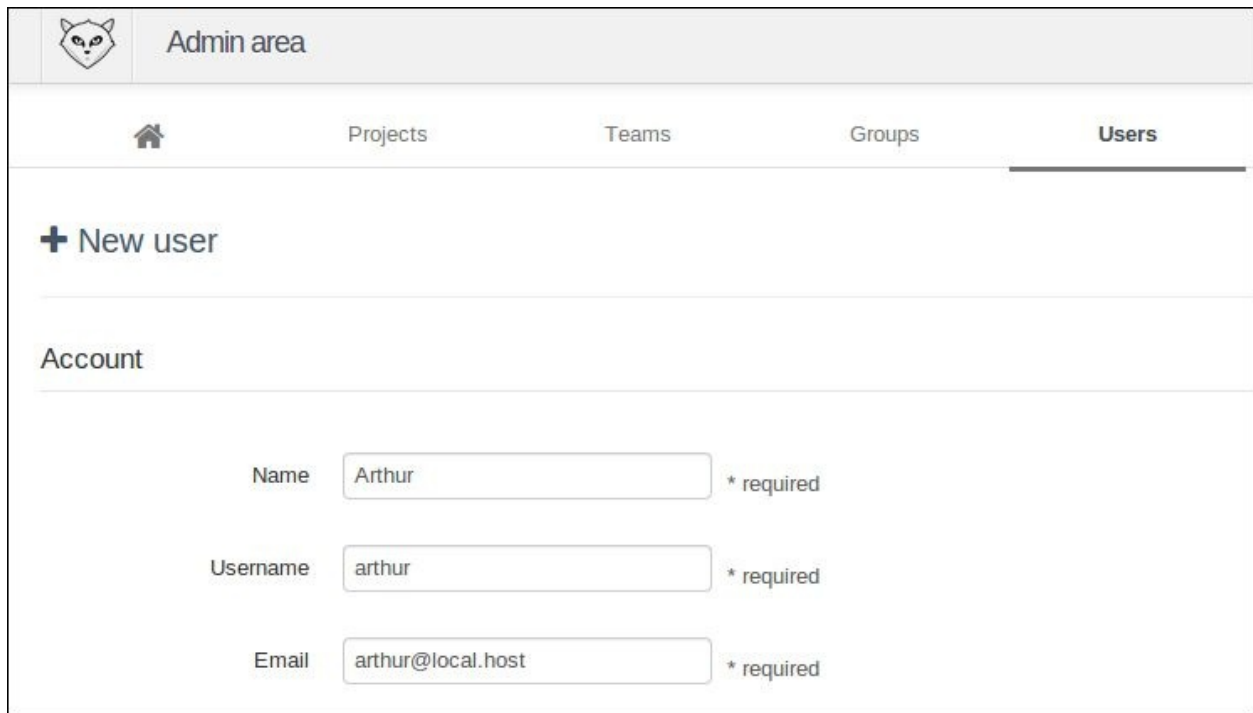
It will then ask you for your password for the previously generated key.

The second user

As in the Unix philosophy, I recommend you to work on the `Administrator` or `root` account unless necessary. For development, use another account so that you do not easily give up on administrative privileges over the server in case security is compromised.

Adding users manually

To create a second user, you can either add one in the role of the Administrator by clicking on **Admin area** in the top navigation bar, followed by clicking on the **New user** button. You will be prompted for the new users details in a form, as shown in the following screenshot:

On this page, you will get a detailed form that needs some information about the new user. You can set the name, username, and e-mail address, along with a desired or randomly generated password.

Secondly, you set some permissions such as whether the new user is able to create teams, groups, or even promote them to an Administrator. If you want them to have any number of maximum projects, this is defined in a `config` file at *home*`git/gitlab/config/gitlab.yml`. You can increase or decrease the number here.

Lastly, you can specify more contact information; at the time of writing, you can set their Skype, Twitter, and LinkedIn profile.

Enabling signup

To enable the signup feature for the site, you can edit the `config` file at *home*git/gitlab/config/gitlab.yml.

Removing the leading `#` from the `signup_enabled` line will enable users to sign up and start their own projects:

```
## Project settings
    default_projects_limit: 10
    # signup_enabled: true            # default: false - Account
passwords are not sent via the email if signup is enabled.
    # username_changing_enabled: false # default: true - User can
change her username/namespace
```

After making a change to the `config` file, remember to restart the `gitlab` service:

    sudo service gitlab restart

After that, when somebody accesses the domain of the server where GitLab is installed, they will be greeted with the default login page, with an additional line: **Don't have an account? Sign up**. **Sign up** is a link to the signup page, which resembles the page for adding a user manually.

Using and understanding different roles

In GitLab, four different roles exist, which can be applied to projects. The roles are split into:

- Guests
- Reporters
- Developer
- Master

They are designed to fulfill different parts of the development process and should be used meaningfully. It can be important to limit access to certain parts according to the competencies a team member represents.

This table shows an overview of the available roles:

Guest	Reporter	Developer	Master
Create new issue	Create new issue	Create new issue	Create new issue
Leave comments	Leave comments	Leave comments	Leave comments
Write on project wall	Write on project wall	Write on project wall	Write on project wall
	Pull project code	Pull project code	Pull project code
	Download project	Download project	Download project
	Create a code snippets	Create a code snippets	Create a code snippets

		Create new merge request	Create new merge request
		Create new branches	Create new branches
		Push to non-protected branches	Push to non-protected branches
		Remove non-protected branches	Remove non-protected branches
		Add tags	Add tags
		Write a Wiki	Write a Wiki
			Add new team members
			Push to protected branches
			Remove protected branches
			Push with force option
			Edit project
			Add deploy keys to project

			Configure project hooks

The following capabilities are taken for the help pages that ship with GitLab and can be accessed by visiting your own installation at `http://yourdomain.com/help/permissions`.

The Guest – a visitor with limited access

The **Guest** role is of the lowest rank and also the most restricted. It's limited to the following capabilities:

- Creating a new issue
- Leaving comments
- Writing on the project wall

The Guest has no influence on the source code directly, but can point out lines of code through comments, which will be visible to the rest of the team. A Guest can also create general issues and leave status updates, such as notices on the project wall. This makes the Guest a communicator that can supervise and inform, but not intervene. Guests cannot browse the file tree directly.

The Reporter – a communicative observer

As a **Reporter**, the user can do a number of things, such as look at the source code in the web interface or by pulling it to their machine. However, they are not allowed to change existing code or contribute additional code. Their capabilities are listed as follows:

- Creating a new issue
- Leaving comments
- Writing on the project wall
- Pulling the project code
- Downloading project
- Creating code snippets

The Reporter is similar to the Guest, but can download the full source code using either a `.zip` file or pulling through Git. This gives the Reporter more possibilities to actually audit the code, run tests on it, and report any issues that might be concluded from the tests.

Also, the creation of snippets is allowed for the Reporter if this is enabled in the project settings. Snippets are great for sharing parts of code, proposals, or possibly small notes with the team members of the same repository.

The Developer – the workforce

The **Developer** role is by far the most self-descriptive role. It allows developers to exchange and merge their code by pushing and pulling through Git. In case they are working on a new feature or something that requires effort to integrate into the project, developers can also push to a new branch on the same repository to keep the changes from having to go to the master branch directly.

But, the Developer role can only push to branches and remove the changes that are not flagged as protected. This can be done by going to the **Commits** | **Branches** panel of a project and needs **Master** or **Owner** privileges. You would typically want to protect branches that go into production and therefore require review or audit.

Adding tags can also be very useful for marking milestones or releases. So, Developers are allowed to create these too. Lastly, wikis can be created by this role to document usage, source code, or other information that should remain available to the team. The capabilities of Developer role are given as follows:

- Creating a new issue
- Leaving comments
- Writing on the project wall
- Pulling the project code
- Downloading the project
- Creating a new merge request
- Creating code snippets
- Creating new branches
- Pushing to non-protected branches
- Removing non-protected branches
- Adding tags
- Writing a Wiki

The Master – powerful and in control

The **Master** role is tailored for maintainers and they have great possibilities to make decisions over the course of the project. In many workflows this role is referred to as a lieutenant. Their capabilities are listed as follows:

- Creating a new issue
- Leaving comments
- Writing on the project wall
- Pulling the project code
- Downloading the project
- Creating new merge request
- Creating code snippets
- Creating new branches
- Pushing to non-protected branches
- Removing non-protected branches
- Adding tags
- Writing a Wiki
- Adding new team members
- Pushing to protected branches
- Removing protected branches
- Pushing with force option
- Editing a project
- Adding deploy keys to project
- Configuring the project hooks

The Owner – the creator of a project

The **Owner** is not really a ruler, but the one who initially created the project. The permissions are the same as the ones for the Master role, except two additional capabilities:

- Transferring a project to another namespace
- Removing a project

These permissions can obviously have a quite severe impact on the project by either removing it from the server and thereby denying everybody access to it and also changing the address it is available at for the other members of the team by changing the namespace.

This address counts for both Git access as the one of the web interface also. So, changing this without notifying the team would lead to errors trying to pull from or push to the repository. However, the link will be updated in the web interface.

Creating a team

In order to collaborate on a project, we need more developers, which we can manage through a team in GitLab. Teams, in essence, just means more than one individual having access to certain functionalities.

Other than adding more individuals to a project, you can predefine teams by name, description, and members or import an existing team from another project.

Within a team, you can deal out the roles described earlier in this chapter. Additionally, when adding a team or importing a team from another project, you can define a maximum user role for the imported teams to limit user ranks on the current project, but not the project they originated from.
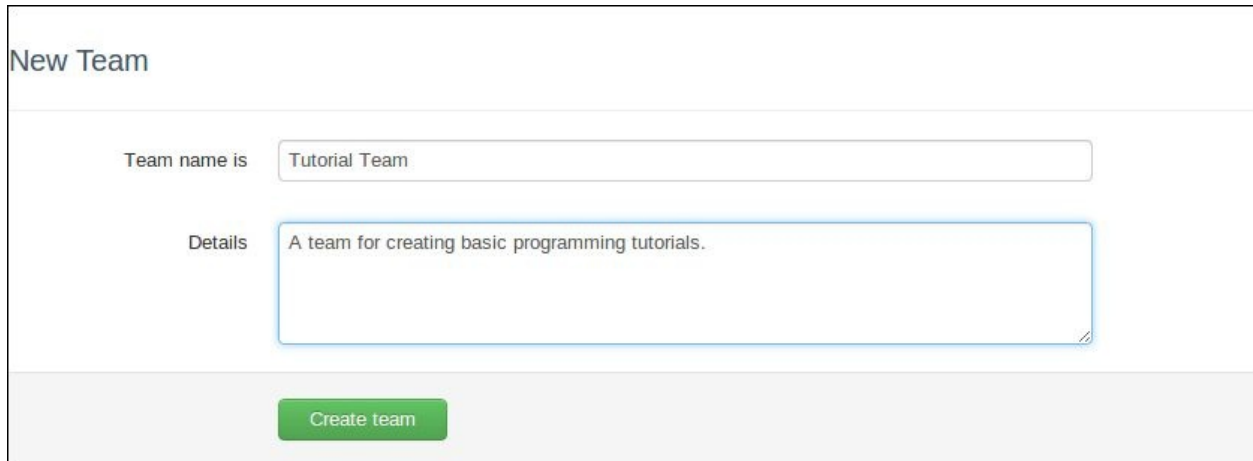
Note

Since GitLab 6, the team and group functionalities have been combined. So, permissions do not have to be granted individually, users added will automatically gain access to the repositories that are assigned to their team.

Adding a team

To add a team, you can either use the handy sidebar widget on the right-hand side, which is displayed on the dashboard on your home screen. If you press the add button, it tells you which projects, teams, and groups you are a part of, or you can check through the administrative interface.

The following screenshot shows the form for team creation:

It just requires a name and the description is optional. I would always recommend filling out both though, this avoids confusion between users and reassures them.

When it comes to adding users to a team, you will come across an automatically completing form, like the ones used on Twitter or GitHub:
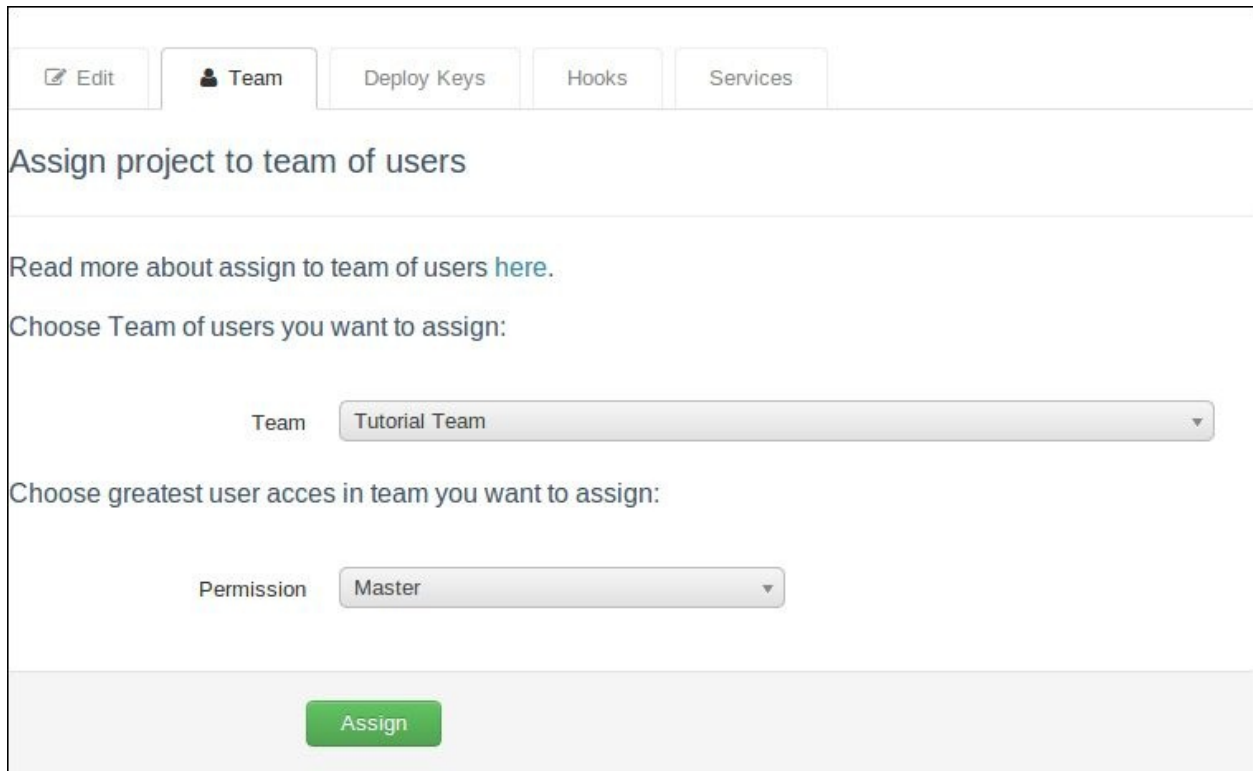
This allows you to quickly add the desired users to the respective team. In the second column, you can set the users role within the team. Lastly, we can make them give access to administrative actions on the team, such as changing other user's roles and adding them as a member.

Importing an existing team

Once a team is created, you can start assigning projects to the team. On any project you can now assign an existing team by going to **Settings** | **Team**.

Apart from assigning a team, you can also assign users from another project right away. In the following screenshot you can see the dialogue for doing so:

Within the project, you can easily change the permissions of individual users of a defined team. Knowing that organizational structures change, this can be very useful and does not lock any repository to a specific user. You can see the form for changing the roles in the following screenshot:

Masters (1)

Administrator
root

| Master ▼ |

`This is you!` `Owner`

Developers (2)

Arthur
arthur

| Developer ▼ | ⊖

Samantha
samantha

| Developer ▼ | ⊖

Guest
Reporter
Developer
Master

Changing teams

Any settings you apply while creating a team can be changed at a later point if you have administrative privileges on the team. The following screenshot shows the overview of team members in an existing team:

This includes the different roles of users, title, and description, as well as projects the team is assigned to.

Creating a group

Groups are different from teams and a little more powerful. The biggest advantage of a group is that they form a relationship between multiple repositories. This is achieved by holding the projects not as user-owned, but within the group's namespace.

Each user must be added to a project inside a group to see it, or gain access by being part of a team that gets assigned to a project within the group. This means that a user will not automatically gain access to projects inside a group that are not explicitly made available to him/her. Groups cannot be browsed by anybody else other than the Administrator. By default, they are private and do not appear in a list for users to pick.

If a project was started by an individual user, you can move it into a group afterwards. Note that this will lead to a change in namespace, and developers working on this particular project will need access to the group and have to change the remote URL of the project in their `.git/config` file.

Note

Since GitLab 6, the team and group functionalities have been combined. So that permissions do not have to be granted individually, users added will automatically gain access to the repositories their team is assigned to.

Managing SSH keys

You can't access other user's SSH public keys through the web interface, but you can do this by opening the file that lists all the added SSH keys.

Open the file on the server-side at `home`git/.ssh/authorized_keys.

It will show you a list of the keys. The users e-mail address is shown at the end of every line, which is how you can differentiate between them.

In a scenario such as when a user's private key is compromised, they can revoke their own keys by deleting them from the account. This can be done by pressing the **Remove** button from within their list of SSH keys inside their profile.

If the user can't do it because they don't have access to either a computer or the Internet, you can manually delete the key from this file to prevent any Git access with this key.

Summary

You have now learned about the different roles and the permissions they grant. GitLab offers diverse roles to ensure the management of projects will fit your needs. You now know how to assign roles to keep your code safe, but enable the opening of issues, for example, through interns or external testers with the Guest role. You also can distribute permissions for developers, decide to which branches they may push to and which branches are controlled by their project manager, and who decides which changes go into production.

In the next chapter, we are going to have a look at how we can use GitLab's built-in functionality to document and improve your projects through issues and wikis. We will also have a look at how to distribute issues or tasks to team members and the markup used to reference different entities within GitLab.

Chapter 5. Issues and Wikis

The built-in features for issue tracking and documentation will be very beneficial to you, especially if you're working on extensive software projects, the ones with many components, or those that need to be supported in multiple versions at once, for example, stable, testing, and unstable.

In this chapter, we will have a closer look at the formats that are supported for issues and wiki pages (in particular, Markdown); also the elements that can be referenced from within these and how issues can be organized. Furthermore, we will go through the process of assigning issues to team members, and keeping documentation in wiki pages, which can also be edited locally. Lastly, we will see how the RSS feeds generated by GitLab can keep your team in a closer loop around the projects they work on.

The metadata covered in this chapter may seem trivial, but many famous software projects have gained traction due to their extensive and well-written documentation, which initially was done by core developers. It enables your users to do the same with their projects, even if only internally; it opens up for a much more efficient collaboration.

GitLab-flavored Markdown

GitLab comes with a Markdown formatting parser that is fairly similar to GitHubs, which makes it very easy to adapt and migrate. Many standalone editors also support this format, such as Mou (http://mouapp.com/) for Mac or MarkdownPad (http://markdownpad.com/) for Windows. On Linux, editors with a split view, such as ReText (http://sourceforge.net/projects/retext/) or the more Zen-writing UberWriter (http://uberwriter.wolfvollprecht.de/) are available.

For the popular Vim editor, multiple Markdown plugins too are up for grabs on a number of GitHub repositories; one of them is Vim Markdown (https://github.com/tpope/vim-markdown) by *Tim Pope*.

Lastly, I'd like to mention that you don't need a dedicated editor for Markdown because they are plain text files. The mentioned editors simply enhance the view through syntax highlighting and preview modes.

About Markdown

Markdown was originally written by *John Gruber*, and has since evolved into various flavors. The intention of this very lightweight markup language is to have a source that is easy to edit and can be transformed into meaningful HTML to be displayed on the Web. Different variations of Markdown have made it to a majority of very successful software projects as the default language; readme files, documentation, and even blogging engines adopt it.

In Markdown, text styles can be applied, links placed, and images can be inserted. If ever Markdown, by default, does not support what you are currently trying to do, you can insert plain HTML, which will not be altered by the Markdown parser.

Referring to elements inside GitLab

When working with source code, it can be of importance to refer to a line of code, a file, or other things, when discussing something. Because many development teams are nowadays spread throughout the world, GitLab adapts to that and makes it easy to refer and reference many things directly from comments, wiki pages, or issues.

Some things like files or lines can be referenced via links, because GitLab has unique links to the branches of a repository; others are more directly accessible. The following items (basically, prefixed strings or IDs) can be referenced through shortcodes:

- commit messages
- comments
- wall posts
- issues
- merge requests
- milestones
- wiki pages

To reference items, use the following shortcodes inside any field that supports Markdown or RDoc on the web interface:

- `@foo` for team members
- `#123` for issues
- `!123` for merge requests
- `$123` for snippets
- `1234567` for commits

Issues, knowing what needs to be done

An issue is a text message of variable length, describing a bug in the code, an improvement to be made, or something else that should be done or discussed. By commenting on the issue, developers or project leaders can respond to this request or statement. The meta information attached to an issue can be very valuable to the team, because developers can be assigned to an issue, and it can be tagged or labeled with keywords that describe the content or area to which it belongs. Furthermore, you can also set a goal for the milestone to be included in this fix or feature. In the following screenshot, you can see the interface for issues:
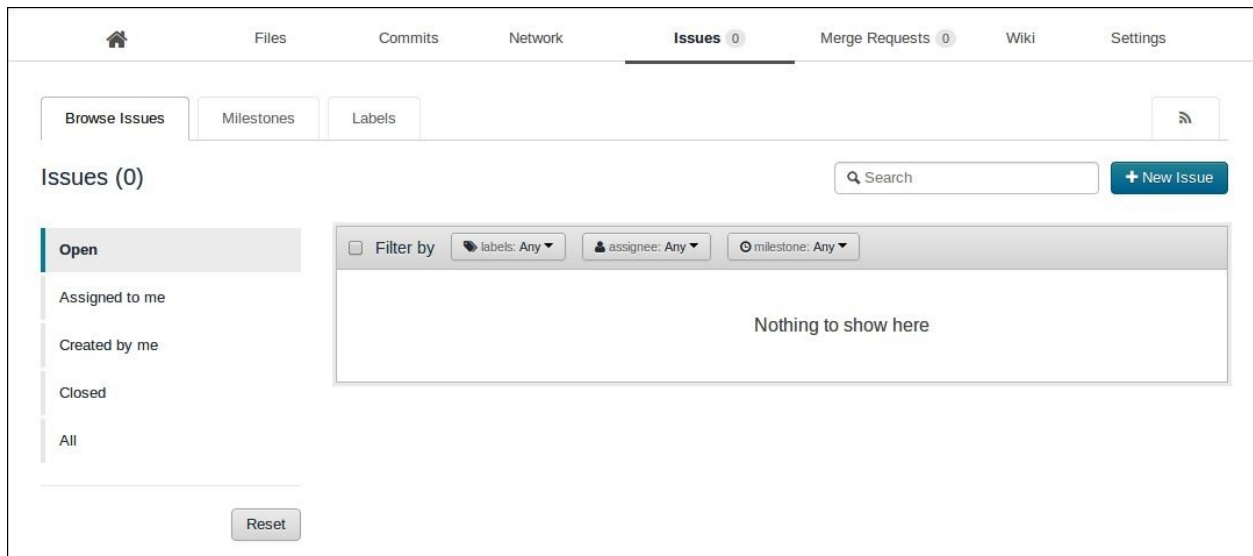
Creating issues

By navigating to the **Issues** tab of a repository in the web interface, you can easily create new issues. Their title should be brief and precise, because a more elaborate description area is available.

The description area supports the GitLab-flavored Markdown, as mentioned previously.

Upon creation, you can choose a milestone and a user to assign an issue to, but you can also leave these fields unset, possibly to let your developers themselves choose with what they want to work and at what time. Before they begin their work, they can assign the issues to themselves. In the following screenshot, you can see what the issue creation form looks like:

Working with labels or tags

Labels are tags used to organize issues by the topic and severity.

Creating labels is as easy as inserting them, separated by a comma, into the respective field while creating an issue.

Currently in Version 5.2, certain keywords trigger a certain background color on the label. Labels like **critical** or **bug** turn red, **feature** turns green, and other labels are blue by default.

The following screenshot shows what a list of labeled features looks like:
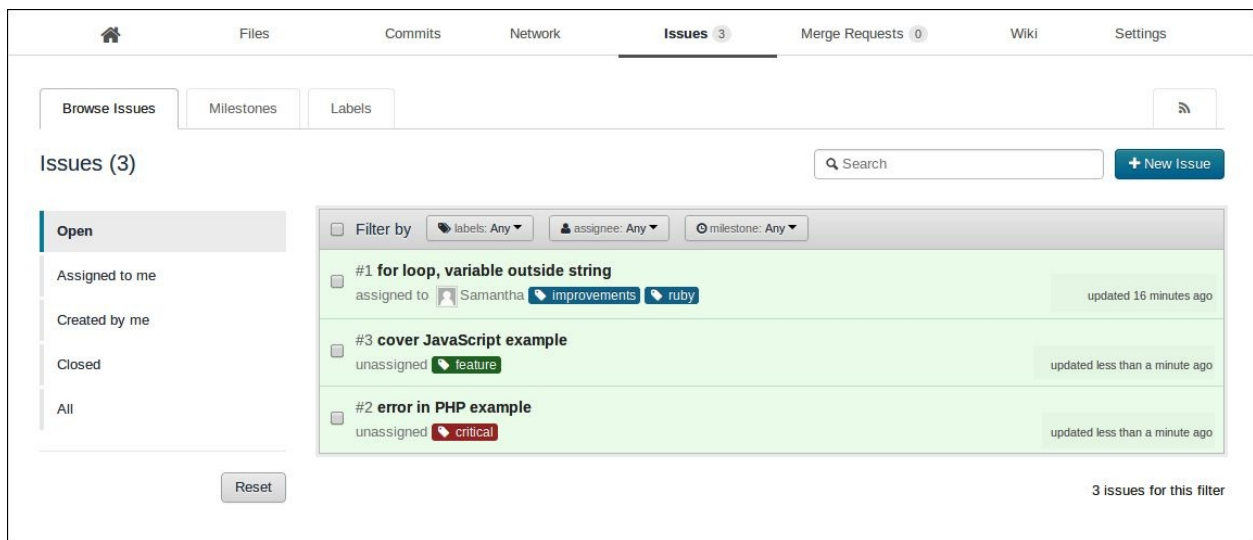
After the creation of a label, it will be listed under the **Labels** tab within the **Issues** page, with a link that lists all the issues that have been labeled the same.

Filtering by the label, assigned user, or milestone is also possible from the list of issues within each projects overview.

Assigning users

If you have not assigned a user to an issue upon creation of the issue, you can do so afterwards by pressing the **Edit** button on an issue on the list. Actions performed on issues will be logged under the issues, and will be visible to the users in the project.

To see the issues assigned to a certain user, you can filter the list of issues by the user as shown in the following screenshot:
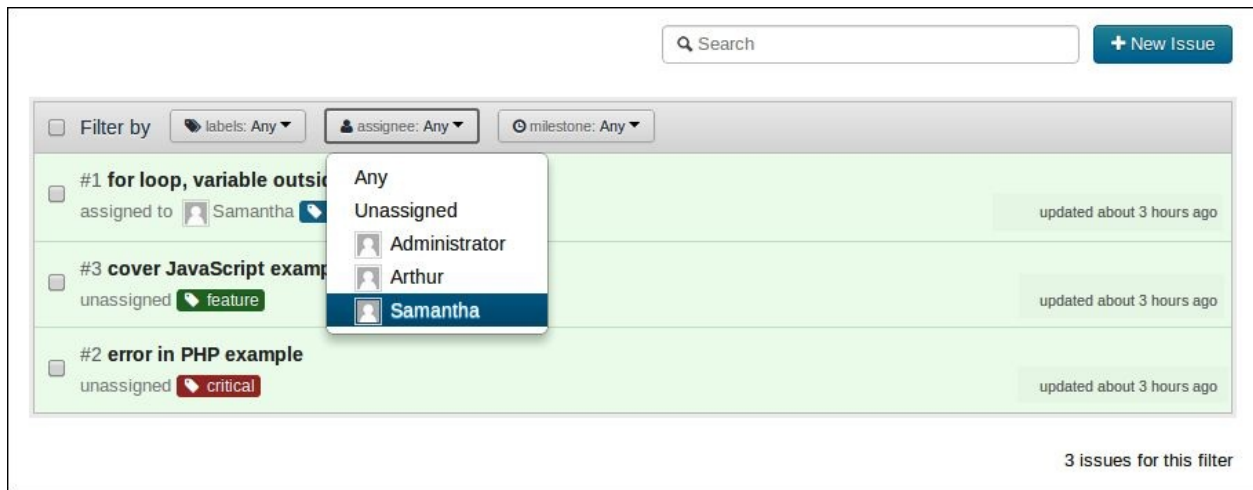
Fast documentation with wikis

To document your project on multiple subjects or purposes, wikis are a great tool. Just like the famous use case of Wikipedia, they are editable, and you can manage the permissions through GitLab's role and permission management. Wikis in GitLab behave very similar to the ones at GitHub. They consist of Markdown files that are named in the pattern *pagename.markdown*, and are stored in your `repository.wiki.git` path.

Editing online

To edit the wiki online, you simply browse to the **Wiki** tab on the project and you end up on the wiki's home page; any other created page is listed under **Pages**.

When editing a page, you can change the content by either using Markdown or RDoc as the preferred markup language, and add a commit message that represents the changes made.

Editing locally

For local editing, you can use Gollum, which is a Ruby-powered tool that starts a web server on your local machine or just any text editor of your liking. Some dedicated Markdown editors have been mentioned in the section *GitLab-flavored Markdown*.

The commands to clone the wiki locally are visible inside the **Wiki** tab under **Git Access**.

After making changes, you can commit and push back to the repository.

RSS feeds

Nobody likes to be tied to one interface only, which is why important information such as commits on a project can be syndicated to a number of devices through RSS.

This case is essential for staying up to date with progress from a mobile device or just through a feed reader of your choice. You can access the RSS feed through the little RSS beacon icon, usually on the far right of the web interface.

The link pointing at an RSS feed includes a user's private token, which can be changed in the **Account** settings.

An example of such a link is
`http://localhost:8080/root/simple_loops/commits/master.atom?private_token=Wy9wt1Fbxufbyjwx3fcx`.

The private token gives the feed reader access to the information stored inside GitLab without a dedicated login. I've used Liferea on Linux, but any feed reader, whether web-or desktop-based application, works that has network access to your server. Here is a screenshot for the display of feeds from GitLab:
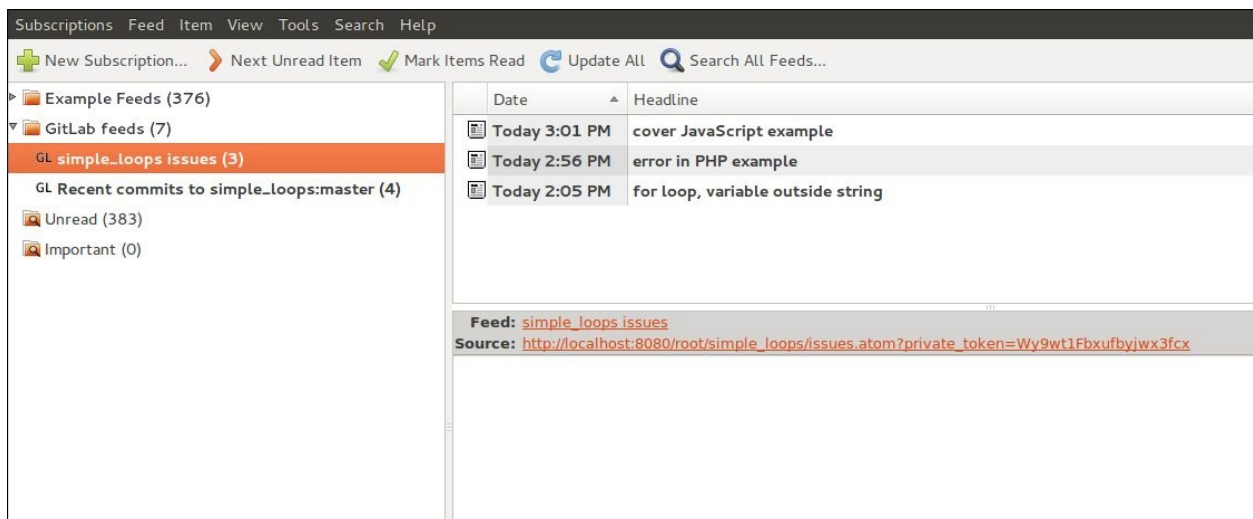
The title of the individual posts will be a link address pointing to your GitLab installation. In addition, the title of the items in the RSS feed will represent

either the title of the commit or the issue it represents.

Changing a private token

A user can generate a new private token that will invalidate the previously issued token. This can be necessary if the access to a user's machine or feed reader was granted to unauthorized third parties. To prevent future access of these third parties, users can simply visit their **Account** settings page. The dialog box for changing the token looks as follows:

This token also grants API access, which can be utilized in either custom solutions or compatible software.

Understanding the value of metadata

Documentation and other metadata that are not directly source code are important to projects and everybody who works with it. It builds a more future-proof and solid foundation of even the largest products. Make use of them, no matter you start your projects on GitLab or migrate existing ones to it!

In your installation, you have great possibilities to offer your users communication that goes beyond just code. A mistake often made within the programming community is to underestimate the value of documentation and the ability of non-programmers to approach a project.

Summary

In this chapter, we have had a look at the project management side of things. You can now make use of the built-in possibilities to distribute tasks across team members through issues, keep track of things that still have to do with the issues, or enable observers to point out bugs.

You have learned about wikis and how you can keep track of documentation, manuals, and much more, either through changing it online or editing it locally.

In the next chapter, we will look at the workflows that are possible with GitLab, and find out which one is right for you and your team!

Chapter 6. Workflows

Git is a tool that meets many developer needs, but there are a couple of different methods that have emerged on how to work together on this platform. Git is a very flexible tool and sometimes it can be hard to understand what the best practices are and which way to go, given a set of requirements. In this chapter we will look at the different possibilities that are offered and supported by GitLab.

We will also investigate the different ways of working with Git and GitLab in particular. We will be able to choose between a single branched and a feature branched workflow and also look at an example. In the example we will learn how to create and respond to merge requests.

The term **merge request** will be used a lot in this chapter, which is very similar to the **pull request** that is well known among GitHub users (http://github.com). A merge request is basically a request to merge code from another branch or fork.

Single branch

The single branch method may seem intuitive, but it limits developers to always having to create stable code and discourages experimentation. This way of collaborating allows multiple developers to pull from and push to the same branch. This way, the state of the project is pretty much the same across the different developer machines and everybody is in sync.

A big drawback of this approach is that there is no proposal of code changes, just direct implementation. Since you can always roll back into history and just pick the changes you actually want in your project, it's not a vital issue. But it can consume more time than other approaches.

The developer writes code, checks if any new code is on the repository, merges it if possible and pushes. Now his/her changes are in the project, which is usually a good thing, except when the changes they made change something that was not desired, for example, breaking a feature or introducing a possible security vulnerability.

Feature branch

This part will describe the workflow for creating multiple branches, one per feature. When considered done, these branches will be merged back into the master branch.

In the typical workflow for feature branches, a developer creates a new branch in the project. This is easily achieved by the following terminal commands:

```
git branch FeatureName
git checkout FeatureName
# make your changes
git push origin FeatureName
```

After that, a new branch will appear in the web view of GitLab and the development of the feature of the branch functionality will be tracked separately. When considered done, the respective developer will open a merge request, typically to the **master** branch. This will of course vary with the requirements and structure of your projects; it could as well be **stable** or **development**.

Creating a merge request

GitLab can make use of this approach by displaying merge requests when a feature or change is considered done and ready to be merged back into the main branch of the project.

To create a merge request, you just have to press the button named **New Merge Request**, and fill out the form shown in the following screenshot:

The form asks for a source branch and a destination branch, so it assumes that the contents of the two branches differ. These branches are also referenced as **head branch** and **base branch**.

Additionally, upon selection of a branch, it will automatically load the hash of the most recent commit with the short commit message attached. Lastly, you may specify a title for the merge request, ideally a summary of the commits that are being proposed as a change. A user can accept or deny the merge request, as also set **milestone** for which version of your software the changes are targeted can be selected. You can assign a user to do the desired merge.

Responding to a merge request

When a merge request is submitted, users having the Master or Owner role can decide if the changes should be accepted into the targeted branch.

The merge request inside GitLab shows the changed lines as displayed in the following screenshot:
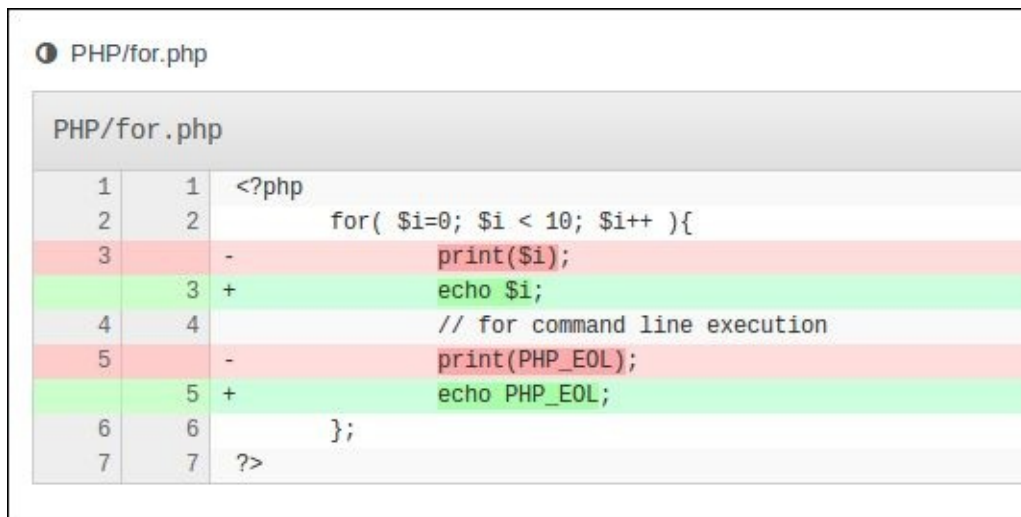
To give the Master of a project meaningful insights into the changes made in a branch, a difference is appended directly into the merge request opened by the developers.

Furthermore, you can also download the patch file in a plain Git output or something you can send per mail as a `.patch` file. Such a file pretty bluntly shows the lines that have been removed and added to the respective files.

The following is an example of the patch file corresponding to the example of the PHP echo fix:

```
From fc9d2913df34d41d329303e4e79a620d41fd9838 Mon Sep 17 00:00:00
2001
    From: "Samantha" <samantha@host.local>
    Date: Sat, 15 Jun 2013 17:41:12 +0200
    Subject: [PATCH] for.php now uses echo

    ---
      PHP/for.php | 4 ++--
```

```
  1 file changed, 2 insertions(+), 2 deletions(-)

diff --git a/PHP/for.php b/PHP/for.php
index 08270c3..00ff5a4 100644
--- a/PHP/for.php
+++ b/PHP/for.php
@@ -1,7 +1,7 @@
  <?php
    for( $i=0; $i < 10; $i++ ){
-       print($i);
+       echo $i;
       // for command line execution
-       print(PHP_EOL);
+       echo PHP_EOL;
   };
 ?>
--
1.8.1.2
```

Monitoring branches

Within GitLab, the process of projects with multiple branches can easily be viewed in **Project Network Graph**, which can be accessed through the **Network** tab on the projects.

The following screenshot shows how such a network graph looks after the branching and merging in the previous example:
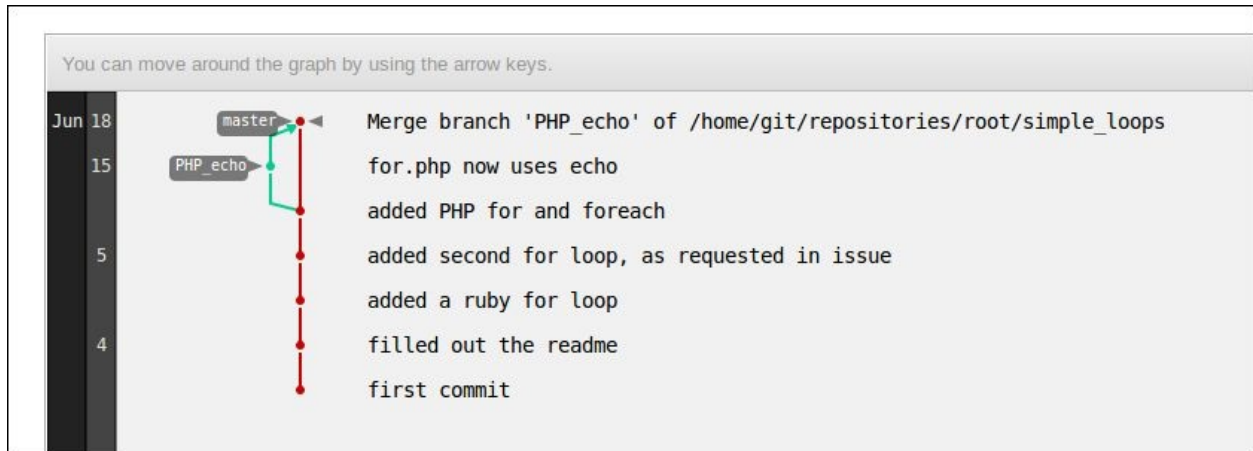
As you can see, the different branches receive their own colors and the vertically aligned dots represent different commits. The commit messages are displayed and if available, pictures of the committers will be filled in through Gravatar.

Forking repositories

In GitLab 5.2, the famous one-click fork feature was introduced. Developers with access to a repository can now copy a project to their own namespace and account to work on the code. The screenshot displays where to find the **Fork** button in GitLab on the front page of a project or repository:

After pressing the **Fork** button on a project, the user will be redirected to the project in their own namespace. The limitations on the capabilities of the protected branches will not apply to their copies, but they have to push back to a branch on the original project or make someone pull their repository to merge the changes.

A pull request feature across projects by different owners is in the making at the time of writing this book.

Hooks

Hooks in Git are programs that are run when Git takes a specific action, for example, attempting to add a commit to the history. We will have a look at how they can be created and what they can do. Hooks can be written in any programming language that can be interpreted by the server; for example, bash, Ruby, or PHP.

Hook examples

Git in its default configuration creates some example hooks that illustrate the capabilities and the events they are run in. We can take a look at any of the repository examples in the folder `.git`hooks.

In this folder we can find a file named `commit-msg.sample`, which has the following content:

```
#!/bin/sh

#

# An example hook script to check the commit log message.

# Called by "git commit" with one argument, the name of the file

# that has the commit message.  The hook should exit with non-zero

# status after issuing an appropriate message if it wants to stop the

# commit.  The hook is allowed to edit the commit message file.

#

# To enable this hook, rename this file to "commit-msg".


# Uncomment the below to add a Signed-off-by line to the message.

# Doing this in a hook is a bad idea in general, but the prepare-commit-msg

# hook is more suited to it.

#
```

```
# SOB=$(git var GIT_AUTHOR_IDENT | sed -n 's/^\(.*>\).*$/Signed-
off-by: \1/p')

# grep -qs "^$SOB" "$1" || echo "$SOB" >> "$1"


# This example catches duplicate Signed-off-by lines.

test "" = "$(grep '^Signed-off-by: ' "$1" |

  sort | uniq -c | sed -e '/^[  ]*1[    ]/d')" || {

  echo >&2 Duplicate Signed-off-by lines.

  exit 1

}
```

As we can see, it's thoroughly documented and at the beginning of the file #!/bin/sh shows that it is a simple shell script. It does some testing for a duplicate text in a commit message.

Hooks with the GitLab API

You can connect to the GitLab API using your API key to perform actions inside GitLab when performing, for example, a commit. This you can do from the client side, but of course, you can also use the server-side hooks, such as post-receive to run programs and scripts after they have reached the remote.

A great example can be found in the GitLab Wiki pages at https://github.com/gitlabhq/gitlab-public-wiki/wiki/Hooks. I've modified it for easier understanding here:

```
#!/bin/sh

PRIVATE_TOKEN="API_KEY"
GITLAB_URL="https://your_gitlab_domain.com"

URL=`git config --get remote.origin.url`
PROJECT=`basename ${URL} .git | cut -d':' -f2`

for issue_id in `grep -o -e "\(closes\|fixes\) #[0-9]\+" $1 | cut -
d'#' -f2`; do
    curl -X PUT -d "closed=1" \
```

```
${GITLAB_URL}api/v3/projects/${PROJECT}/issues/${issue_id}/?
private_token=${PRIVATE_TOKEN}
done
```

This script will automatically look at commit messages and if they include something like `fixes #2`, where `2` refers to the issue number inside the project on GitLab, it will close the issue for you. Saving this script as `.git/hooks/commit-msg` and making it executable through `chmod +x` will enable this functionality. You can do much more automation with this and use it to streamline and improve your workflows.

Summary

In this chapter you got an overview of the three ways of using Git to collaborate and split the different stages of an overall project. Choosing the one you prefer and the requirements of your tasks is up to you at any point. There is no universal solution for these things since it all depends on how your company or organization is built.

It might be easier running a single branch if you don't get many commits per week; however, if you get commits by the hour, you should probably have someone to look at maintaining and merging the possible merge conflicts, to give the developers feedback and decide if their changes can be permitted into the master branch.

Only trying it out will tell what works best for you!

In the next chapter we will go through how we can keep our GitLab installation up to date and prior to that, how to create backups of all the existing projects and users. We will look closely at stopping services, downloading a new version, migrating configurations if necessary, running tests, and starting a backup.

Chapter 7. Updating GitLab

In this chapter we will go through the steps that are required for updating GitLab. First and most importantly, we will go through how to keep your repositories and database entries created with GitLab safe and make backups, after which we will proceed to the actual update process.

Lastly, to make sure everything runs as it is supposed to, we will run the included test suite and start our server backup. We are covering the following topics in this chapter:

- Stopping the GitLab service
- Creating backups
- Upgrading to a new version
- Upgrading configuration files
- Restarting the GitLab service

Preparing for an update

As every book, guide, and tutorial warns you, back up your data. Copy the directory and database, just in case something crashes. It will take a lot longer to restore everything in case of a disaster than to copy and download what you have in your installation right now.

Stopping GitLab

Before we start downloading the new files, we need to stop our GitLab server. From this point onwards, your GitLab site will be unreachable, so consider scheduling updates with your users.

It is important to stop the service so that no files are locked during the update and stop the program to avoid crashes that could occur if you update while running.

On Debian-or Ubuntu-based systems, you can stop GitLab by using the following command:

```
    service gitlab stop in case this fails, you can also access the
init script manually:    etcinit.d/gitlab stop
```

Backup

To create backups of your GitLab installation, you need to take multiple things into account. First of all, of course the repositories created that your users depend on. Let's start by creating a backup of these.

The repositories created by GitLab are located at the following path, unless it was configured differently:

*home*git/repositories/

Knowing the directory, you can either transfer the directory somewhere else or create an archive with a utility such as TAR:

```
cd homegit/repositories/
sudo -u git -H tar zcvf ~/backup_repositories.tar.gz .
```

This command will create an archive named `backup_repositories.tar.gz`, which contains all the repositories.

The next important thing we need to back up are our configuration files. Usually they should just be ignored during an update, but it's always better to be safe than sorry. They are located at the following locations:

*home*git/gitlab/config/gitlab.yml

*home*git/gitlab/config/*

*home*git/gitlab-shell/config.yml

*etc*init.d/gitlab

*etc*nginx/sites-available/gitlab

Database

GitLab comes with a rake task to create a backup of the database, which is given as follows:

```
cd homegit/gitlab
sudo -u git -H RAILS_ENV=production bundle exec rake
gitlab:backup:create
```

However, you can also use the plain old `mysqldump` in case you're running GitLab with a MySQL database:

```
mysqldump gitlabhq_production -u gitlab -p > dump.sql
```

While running this command, you will be prompted to enter the password that you've set for the `gitlab` user during the installation.

Update

Updating GitLab is fairly easy. It requires just a few terminal commands at the time of writing, but you should always watch out for the current update guides available at the official repository.

Also, the author includes update guidelines with every release on the official blog at http://blog.gitlab.org/.

Getting the new version (6.1)

Generally, GitLab can be upgraded across versions, which means you can skip versions and directly go to the most recent one. However, it's sometimes recommended to upgrade to a major release before upgrading further. In this case, it would be optimal to upgrade to 6.0 before upgrading further to 6.1.

Getting the new version of GitLab usually happens through Git as follows:

```
cd homegit/gitlab
sudo -u git -H git fetch
sudo -u git -H git checkout 6-1-stable
```

With these commands, you can go to your installation path at *home*git/gitlab and pull the current version of the code directly from GitHub. Notice that we do this as the `git` user to maintain the proper privileges.

As the output of the last-mentioned command, you should see something as follows:

```
Branch 6-1-stable set up to track remote branch 6-1-stable from
origin.
Switched to a new branch '1-1-stable'
```

By typing `git branch`, you will receive a list of the currently available branches in the GitLab source. The output will resemble the following:

```
  6-0-stable
* 6-1-stable
  master
```

Dependencies and databases

To make sure that we match the dependencies for the new version, we again run the `bundle install` command that matches our database setup.

For MySQL the command is:

```
sudo -u git -H bundle install --without development test
postgres --deployment
```

For PostgreSQL:

```
sudo -u git -H bundle install --without development test mysql
--deployment
```

When this process is done, we have to run the rake migrate task that makes sure any changes done to the database structure are incorporated:

```
sudo -u git -H bundle exec rake db:migrate RAILS_ENV=production
```

If you want to switch to another branch, let's say master (careful, may be unstable), you will tell Git to checkout master using the following command:

```
git checkout master
```

As a result, the changes will be applied to the files in $home$git/gitlab.

Reconfiguring after update

It is recommended to manually migrate the configuration files that contain the desired settings. From experience, I can say that this is not always necessary, but the new versions of the sample configuration files should be looked through to see if there are new options that need to be supported in your `config` file.

The `config` files that need to be changed depend on the version of GitLab you're updating. In general, the settings in the `config` files can be set to be updated with the features of GitLab and dependencies.

If required by the new version, we need to manually migrate our changes in the `config` files, `gitlab.yml` and `puma.rb` in *home*`git/gitlab/config/`.

To move to the new `config` files, you can back up your own and then create new `config` files from the up-to-date version using the following commands:

```
    sudo -u git -H homegit/gitlab/config/gitlab.yml.example
homegit/gitlab/config/gitlab.yml
    sudo -u git -H homegit/gitlab/config/puma.rb.example
homegit/gitlab/config/puma.rb
```

The init script

The `init` script can change during updates, but does not necessarily. Please check the release notes of the respective version to be sure.

To install the new `init` script, we first remove the old one and copy the new one into the correct directory, after which we give it the right permissions to be executed again.

```
sudo rm etcinit.d/gitlab
cp homegit/gitlab/lib/support/init.d/gitlab etcinit.d/gitlab
chmod +x etcinit.d/gitlab
```

If you have made changes to your `init` script during the installation, please reapply the changes after updating it.

Updating GitLab Shell

Updating the GitLab Shell works almost the same way, except that you don't have to install dependencies in the shape of Ruby gems, with the `bundle install` command.

Firstly, the upgrade to Version 5.3 does not require an update of `gitlab-shell`, but future versions might.

To update `gitlab-shell`, we will have to navigate to the directory it is installed in and execute the following command:

```
cd home git/gitlab-shell
sudo -u git -H git fetch
sudo -u git -H git checkout $specified_version
```

Testing the update

After all the steps have been completed, you can start GitLab again and restart your `nginx` server. If you run multiple sites on this box, all of them will reset their open connections and be available again when `nginx` has successfully restarted.

```
sudo service gitlab start
sudo service nginx restart
```

To test the just performed update, we will run two familiar commands that will inform us if anything goes wrong during the initial installation:

```
sudo -u git -H bundle exec rake gitlab:env:info
RAILS_ENV=production
sudo -u git -H bundle exec rake gitlab:check
RAILS_ENV=production
```

That's it! Seriously, it's that easy to update a fairly complex software project through simple tools. This is due to the fact that the authors, of course, have put a lot of thought into this and keep the project maintainable for their users.

Summary

Now, we have successfully gone through the process of updating GitLab, ways to secure the data it manages and the `config` files that make it adjust to our environment, and ways to stop the service before the update. Also, we have downloaded the new version through Git, started its backup, and we know how to run the test suite to make sure it operates as intended.

In the next chapter, we are going to cover ways to get in touch with the fantastic GitLab community that works out solutions for each other every day, for exotic setups or very specific needs.

Chapter 8. Help and Community

The GitLab community is growing, and it is active on a couple of different channels. In this chapter, we will look at the different ways to get in touch with other users and developers. We will cover the official website, blog, and presence on GitHub and Google Groups. We will go over the right place to ask questions, give feedback, request features, and even contribute code.

Official channels

There are two official channels that are hosted by the developers of the project, which is why I mention them first. These are the primary sources for reliable information.

The GitLab blog

The official blog is very useful for releases, which always includes installation and upgrade instructions for the most recently released version. It's located at [blog.gitlab.org](). Also on the official website is a community section that gives you an overview of the main developers and contributors, and guides you on how to start contributing to GitLab and affiliated projects at `gitlab.org/community`.

Feedback and feature requests

At feedback.gitlab.com, you can submit your feature requests and browse other's. Also, you can give out upvotes for features that are very important for you.
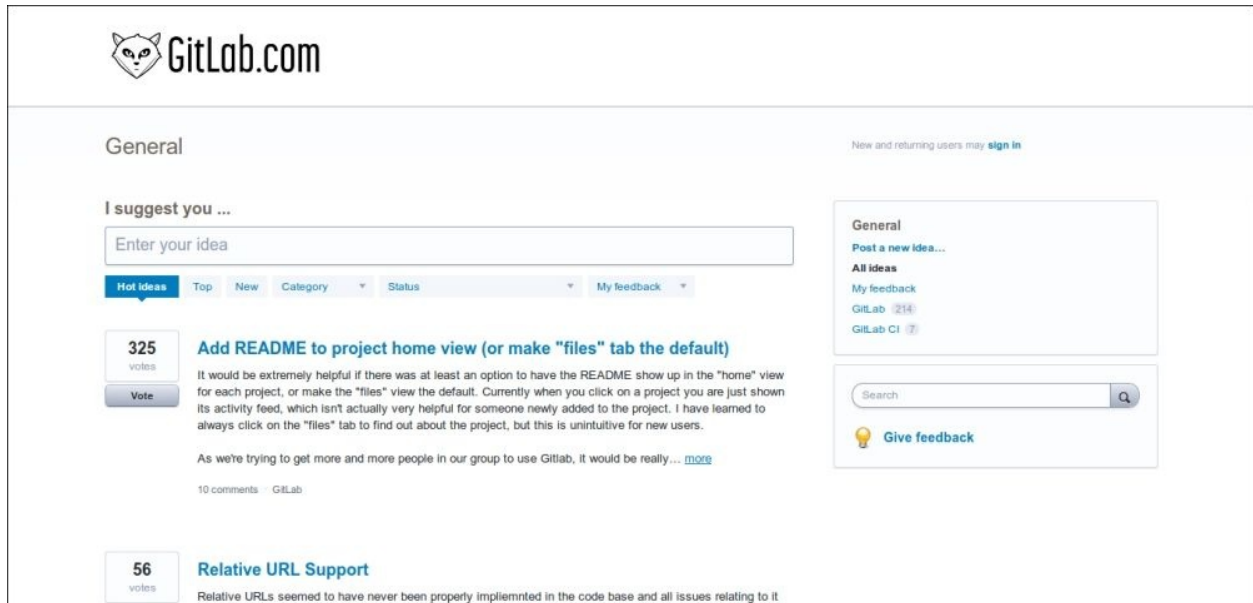
All threads are searchable, and the developers express their opinions and announce the release that will contain a certain feature if it's on the roadmap already.

Other places

The official channels will often link to GitHub and, thanks to the Web, everything can be cross-referenced anywhere else. Here is a collection of the most common channels that are not at gitlab.org or gitlab.com.

GitHub

GitHub is the publishing channel for GitLab. You may ask why they aren't using GitLab to publish it? The answer is because GitHub's focus is much more on open source projects and a globally connected community, instead of having a more isolated pool of developers. In this way, possible bugs can be fixed faster, and more people can contribute to the project without signing up for another site. It's directly available to the huge open source community already on GitHub. No further signup is needed, and the development is public.

Repositories

You can find the official account at github.com/gitlabhq, which most importantly includes the repositories for:

- gitlabhq
- gitlab-shell
- gitlab-recepies
- gitlab-ci

Of these, gitlabhq also contains documentation and installation instructions. The gitlab-recepies repository contains scripts and configurations that should be used with care. Alternative configurations to run GitLab with lighttpd and Apache are also available.

If you want to change something during the installation and configuration to make GitLab run under different circumstances than the ones in the requirements, let's say on Gentoo, Arch Linux, or RHEL, you can submit your changed configs or init scripts there.

Issues

Many users turn to the issue tracker on GitHub to find a solution to their GitLab-related problem, and a search query there will mostly yield precise results. You should always look for issues in the right repository of the part with which you're

having difficulties. If you're not sure of which one that is, check them both.
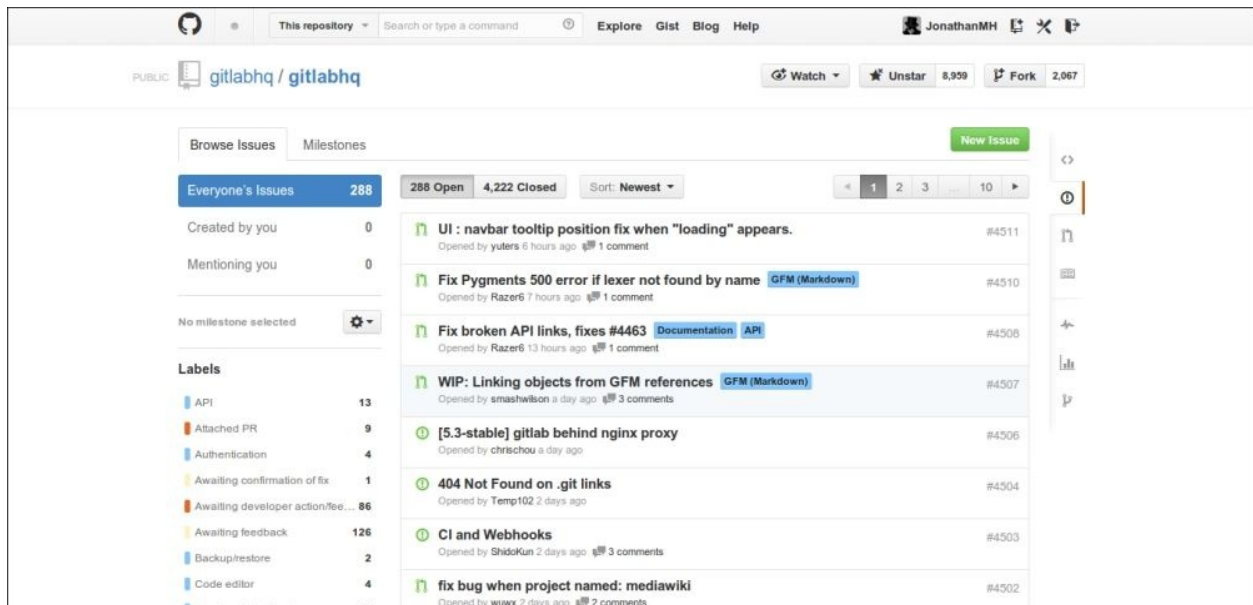
The main projects issue tracker can be found at
github.com/gitlabhq/gitlabhq/issues.

Stackoverflow

There's a tag on stackoverflow for GitLab; you can view all questions related to it at http://stackoverflow.com/questions/tagged/gitlab. I would especially recommend this place if you have very specific questions, or if you're experimenting with things. If you suspect that a bug inside GitLab is causing you trouble, take it to GitHub instead.

Google Groups

The Google Groups forum can be reached at
https://groups.google.com/forum/#!forum/gitlabhq and is populated with
questions around GitLab, where the developers engage actively in
troubleshooting.

Not only are issues reported here, but also questions on how the components work or what purpose they fulfill within the project are discussed.

Troubleshooting

Let's hope you'll never have to read this, but here are a couple of the most common issues faced when installing and running GitLab. I've run into most of them myself, usually because I did something wrong.

The official troubleshooting guide can be found at https://github.com/gitlabhq/gitlab-public-wiki/wiki/TroubleShooting-Guide, which covers many common pitfalls.

Read your logs

When something isn't working, it's probably somewhere in your `.log` files already. A good idea is to read the last couple of lines of a `.log` file when you encounter an issue. Linux has a handy command for that named `tail`.

So to read the last lines of the `application.log` file, you would run:

```
tail home git/gitlab/log/application.log
```

We receive an output similar to the following:

```
June 01, 2013 20:56: User "Arthur" (arthur@local.host) was
created
June 01, 2013 23:51: User "Samantha" (samantha@local.host) was
created
June 01, 2013 23:51: Administrator created a new project
"Administrator / simple_loops"
```

These were the lovely example users that accompanied us along the way.

Redis

The most important thing is to make sure that Redis is running. If you get error messages such as **Can't save project. Please try again later**, but you see the web interface just fine, that's a typical symptom. You can see if the process `redis-server` is running through a tool, such as top or htop, or by simply running the following code:

```
ps aux | grep redis
```

This lists processes and will always list your search, because it contains the word `redis`. If it shows no more processes with that name, you need to start your

`redis-server` by running:

```
sudo service redis-server start
```

Repository permissions

If you receive error messages while trying to pull from / push to a repository, make sure that you have added your private `ssh` key or your GUI client is able to find it.

A typical sign that the key authentication failed is a login prompt in the terminal. Try to add your key manually if in doubt, for example, using:

```
ssh-add home user/.ssh/keyname
```

After which, you will have to unlock your key with your password.

Summary

We have now looked at the different places to reach out for help, get together with the community, or even suggest features for upcoming releases. Also, we know where the source code of GitLab is developed and kept and how you, as part of the community, can influence this process.

Index

B

C

D

E

F

G

H

I

K

- key storage
  - about /

L

M

N

O

P

R

S

T

U

V

W

- web interface
  - about / [Web interfaces](#)
- wikis
  - about / [Fast documentation with wikis](#)
  - editing, online / [Editing online](#)
  - editing, locally / [Editing locally](#)