Mastering GitLab 12

Implement DevOps culture and repository management solutions

Packt
www.packt.com

Joost Evertse

Mastering GitLab 12

Implement DevOps culture and repository management solutions

Joost Evertse

Packt

BIRMINGHAM - MUMBAI

Mastering GitLab 12

Commissioning Editor: Pavan Ramchandani
Acquisition Editor: Rohit Rajkumar
Content Development Editor: Drashti Panchal
Senior Editor: Rahul Dsouza
Technical Editor: Komal Karne
Copy Editor: Safis Editing
Project Coordinator: Jagdish Prabhu
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Production Designer: Aparna Bhagat

First published: August 2019

Production reference: 2201219

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78953-128-2

`www.packt.com`

For my family, who supported me throughout the entire effort of writing this book.

Contributors

About the author

Joost Evertse is an all-round professional with over 20 years of experience in IT in the financial and telecom sectors. He has worked for big and small organizations and has lived in different worlds, including Unix, Oracle, Java, and Windows. Creating order from chaos has been a big focus during his system-engineering years. After 10 years of system administration, he moved into software development and started using CI/CD tools, including GitLab.

At the end of 2016, he started at a significant financial company in the GitLab team, shifting his focus more toward the entire CI/CD pipeline, with the mission of making the CI/CD platform more stable and highly available. His team eventually migrated GitLab to a private cloud and improved release cycles.

About the reviewer

Orlando Monreal is a software engineer with over 12 years of experience, currently working at HCL Technologies Mexico, as part of the Source Code Management team in his project account. He has worked with GitLab applications as an administrator and contact for application support queries, handling upgrade processes and troubleshooting performance and configuration-related issues with the application.

Table of Contents

Preface

GitLab is a tool to enhance the workflow of teams and enable parts of the DevOps life cycle. It started out as a tool only for source code management, but today, GitLab can offer help ranging from managing an initial idea to building and testing source code, all the way from development to production.

You'll learn ways to use all of the features available in GitLab to enhance your business via the integration of all phases of the development process. You'll benefit from lower friction by creating one platform on-premises or in the cloud, increase collaboration, and drive competitive advantage with more efficient operations.

Who this book is for

This book is for developers and DevOps professionals who want to master the software development workflow in GitLab and boost their productivity by putting their teams to work on GitLab via an on-premise installation or cloud-based infrastructure.

What this book covers

`Chapter 1`, *Introducing the GitLab Architecture*, provides a short introduction to the company and the people that created the product, along with a high-level overview of GitLab and its components.

`Chapter 2`, *Installing GitLab*, shows you how to install and configure GitLab via several different methods. This can be done from scratch, or via the Omnibus installer. Special attention is given to Docker and Kubernetes when outlining containerized solutions. Finally, a cloud installation via the DigitalOcean infrastructure is taken as an example.

`Chapter 3`, *Configuring GitLab Using the UI*, explains the options in the GitLab web UI that can be configured after installation. This chapter also covers the administration pages where these instance-level options are situated.

`Chapter 4`, *Configuring GitLab from the Terminal*, looks at the different ways of configuring GitLab. The first approach is by using the Omnibus package installer provided by GitLab, which automates most of the installation. The chapter continues with configuring a source installation. Configuring Docker containers and managing a Kubernetes installation are also covered.

`Chapter 5`, *Importing Your Project from GitHub to GitLab*, outlines the process of migration from GitHub via a hands-on lab. It starts by exploring settings that should be altered in your GitHub project. After this, the settings necessary in GitLab to prepare an import are shown, and finally, the procedure for running the import is addressed.

`Chapter 6`, *Migrating from CVS*, begins with a comparison of the fundamentally different systems of CVS and Git. It then provides directions on preparing for migration. Actual conversion is addressed, as is the cleaning up of artifacts not needed anymore.

`Chapter 7`, *Switching from SVN*, begins by explaining the subtle and not-so-subtle differences between SVN and Git. The reader is shown how to migrate using two different methods: mirroring with SubGit and using the svn2git tool.

`Chapter 8`, *Moving Repositories from TFS*, first deals with the differences between TFS and Git. Subsequently, the act of migrating information from a TFS project to Git is shown via the use of the git-tfs tool.

`Chapter 9`, *GitLab Vision - the Whole Toolchain in One Application*, explains GitLab's vision of providing the whole DevOps toolchain to the developer, looking at the origins of XP and the Agile manifesto. The emergence of the DevOps paradigm is also explored, and the toolchain that GitLab provides is summarized.

`Chapter 10`, *Create Your Product, Verify It, and Package It*, shows how the product vision for GitLab and its workflow is centered around the idea of providing a complete toolchain to create a product. This chapter focuses on the different phases and explains the relevant concepts with examples.

`Chapter 11`, *The Release and Configure Phase*, discusses one of the big features of GitLab: the ability to offer the complete journey to production with different, easy-to-design stages. This way, you can create different environments and, ultimately, automate the whole pipeline for a product.

`Chapter 12`, *Monitoring with Prometheus*, handles ways of monitoring your GitLab environment by using the built-in Prometheus feature and default scripting languages. The second part of this chapter explains the different security tests that are available.

`Chapter 13`, *Integrating GitLab with CI/CD Tools*, explains how, although GitLab aims to provide a complete toolchain in the real world, there will always be a need for integration. This chapter explains some of the bigger possible integrations that are configurable out of the box. It closes with a section on how webhooks provide a general way to consume information from GitLab.

`Chapter 14`, *Setting Up Your Project for GitLab Continuous Integration*, describes GitLab CI concepts that are present on the application server and can be fine-tuned and customized per project. The second part of the chapter mainly focuses on how to get your project ready to use these CI concepts and set up a runner for it to use.

`Chapter 15`, *Installing and Configuring GitLab Runners*, explains the way GitLab runners work, by installing them. The next step is creating an example project and building it with a shell executor.

`Chapter 16`, *Using GitLab Runners with Docker or Kubernetes,* examines the architecture of Docker-based runners and runners using the Kubernetes API, using the same examples as in earlier chapters.

`Chapter 17`, *Autoscaling GitLab CI Runners*, demonstrates the architecture of runners using autoscaling. The number of runners required will decrease and increase based on demand. The example shown uses VirtualBox and **Amazon Web Services** (**AWS**) to deploy instances.

`Chapter 18`, *Monitoring CI Metrics,* deals with monitoring specific GitLab runners. Using a lab, we demonstrate how to enable monitoring inside the runner. After this introduction, the specific functional and system metrics are explained.

`Chapter 19`, *Creating a Basic HA Architecture by Using Horizontal Scaling,* visualizes the way in which different components interact. Secondly, the preparation of databases is shown, as well as several all-in-one application servers. Finally, the shared filesystem for repositories and Redis caching in this **high availability** (**HA**) setup is explained. We will use Terraform and Ansible to create the demonstration environment.

`Chapter 20`, *Managing a Hybrid HA Environment*, builds on the earlier architecture of horizontal HA, but continues to grow in complexity. The main difference is that the application servers combined several components that are now split into new tiers.

`Chapter 21`, *Making Your Environment Fully Distributed,* builds on earlier chapters. A fully distributed architecture aims to create more fault tolerance by again splitting components into new tiers. There is now an SSH node and several sidekiq tiers.

`Chapter 22`, *Using Geo to Create Distributed Read-Only Copies of GitLab,* starts with an explanation of the GEO product, which is part of the Enterprise Edition license. Using the same tools as in earlier chapters from Section 5 of this book 'Scale the Server Infrastructure (High Availability Setup)', we will explain how to set up GEO to create replication between two different geographical locations.

To get the most out of this book

To get the most out of this book, you should have access to a Linux or macOS machine, have an internet connection, and have Amazon AWS, Google, and Microsoft Azure accounts. These are all necessary to run the examples.

Some basic IT knowledge is necessary to read this book. The subjects you need experience in are as follows:

- Linux
- Shell scripting
- Basic programming skills in Ruby and JavaScript
- A basic understanding of Docker containers
- A basic understanding of using Terraform to create infrastructure as code
- A basic understanding of Ansible

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-GitLab-12`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Code in Action

Visit the following link to see the code being executed:

`http://bit.ly/2KirIoO`

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781789531282_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Let's continue with installing web documents in `/usr/local/www`."

A block of code is set as follows:

```
server {
listen 8080;
server_name localhost;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
server {
listen 8080;
server_name localhost;
```

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

```
$mkdir /usr/local/www
$chmod 755 /usr/local/www
$cd /usr/local/www
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "You can do this by clicking the **Choose File** button near the **Logo** section."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Section 1: Install and Set Up GitLab On-Premises or in the Cloud

This section will give you a solid understanding of GitLab deployment options and GitLab component architecture, leaving you able to install and configure GitLab on-premises and in the cloud.

This section comprises the following chapters:

1
Introducing the GitLab Architecture

Understanding the context of the GitLab project will help us to appreciate the choices that were made with regard to the design of the GitLab workflow. The GitLab project started out as a small, open source project, and has grown to be an organization of 400 people and thousands of volunteers. It is currently available in two versions, a free **Community Edition** (**CE**) and an **Enterprise Edition** (**EE**) with a proprietary license. There are several tiers of support for the enterprise version. Although it is proprietary licensed, the source code for that version is publicly available from GitLab.

To master GitLab, it is necessary to have a solid understanding of its individual components. In this chapter, we will look at the basic components of a GitLab installation, paying special attention to GitLab **Continuous Integration** (**CI**) and the accompanying runners. As the different components can be distributed across servers or even cloud providers, we will also provide an overview of those providers and how GitLab views them.

In this chapter, we will be covering the following topics:

- The origins of GitLab
- GitLab CE or EE
- The core components of GitLab
- GitLab CI
- GitLab Runners
- Cloud native

Technical requirements

To follow along with the instructions in this chapter, please download the Git repository with examples, commands and instructions, available at GitHub: `https://github.com/PacktPublishing/Mastering-GitLab-12/tree/master/Chapter01`. Look in the `Readme.md` file for a general explanation of the content of the directory.

To run or install software used in this chapter you need one of the following platforms:

- Debian 10 Linux codename 'Buster'
- CentOS 7.x or RHEL (Red Hat Enterprise Linux) 7.x
- macOS Sierra or later

The origins of GitLab

The story began in 2011, when Dimitri Zaporozhets, a web programmer from Ukraine, was faced with a common problem. He wanted to switch to Git for version management and GitHub to collaborate, but that was not allowed in his company. He needed a tool that did not hinder him in developing code and was easy to use. Like many developers, he had issues with the collaboration tool that he was obliged to use. To get around those issues, he created his side project in Ruby on Rails: GitLab. Together with his colleague, Valery Sizov, he developed this project alongside his regular work.

After this initiative, the project grew enormously:

Date	**Fact**
2011	Sytze Sybrandij, the future CEO of GitLab, is impressed by the GitLab project and code, and offers Zaporozhets the opportunity to try to commercialize it via `https://about.gitlab.com/`.
2012	GitLab was announced to a broader audience via Hacker News (`https://news.ycombinator.com/item?id=4428278`).
2013	Dimitri Zaporozhets decides to work full-time on GitLab and joins the company.
2015	GitLab becomes part of the Y Combinator class and received VC funding that year.
2018	GitLab receives another $100 million of VC funding and is valued at $1 billion.
2019	The GitLab company employs over 600 employees.

The initial idea of GitLab was to earn money from open source technology by offering support services. However, what happened was that companies started to bring in consultants only to upgrade GitLab, and then they would stop the service contract. It became clear that going for a 100% open source was not going to be competitive. Instead of this, therefore, they chose **open core**. Under open core, a company releases a core software system under an open source license. A different version of the software is sold under a commercial license and contains more features.

So, GitLab was split up into two editions: an open source version, and an enterprise version.

Exploring GitLab editions – CE and EE

The core of the GitLab software is called the **CE**. It is distributed under the MIT license, which is a permissive free software license created at the Massachusetts Institute of Technology. You are allowed to modify the software and use it in your creations.

No feature that ever made it to CE will ever be removed, or moved to a closed source version. When GitLab EE was created in 2013, it was, at its core, GitLab CE, but it had additional enterprise features, such as **Lightweight Directory Access Protocol** (**LDAP**) groups. Those features are not open source, per se, but can be added to the core version if they are perceived by the company as a core feature. The idea was that companies should also contribute as much as possible to solving problems and creating new features.

In 2016, the GitLab EE product was divided into three tiers: Starter, Premium, and Ultimate. Each tier is about five times more expensive than the previous one and contains more features and support options, as mentioned in the following table:

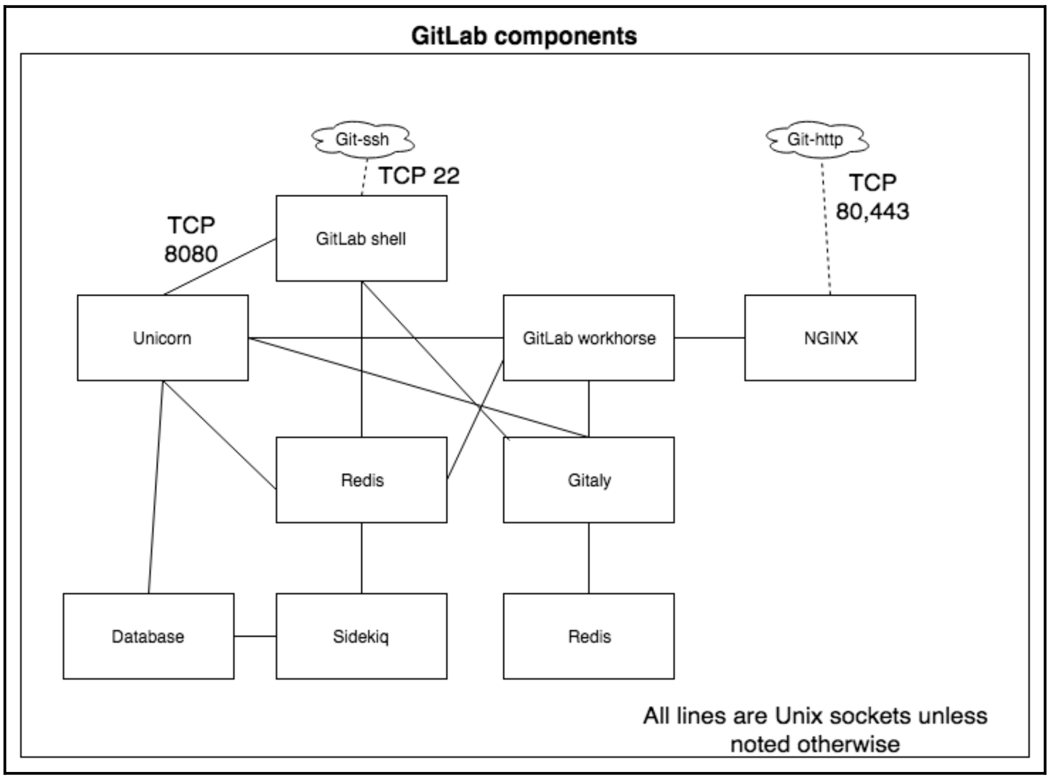
Version	**Features (short list)**
Starter	Everything on core GitLab CE: • CI/CD • Project Issue Board • Mattermost integrations • Time tracking • GitLab pages
Premium	More enterprise features such as the following: • Maven and NPM repository functionality • Protected environments • Burndown charts • Multiple LDAP servers and Active Directory support

Ultimate	All options, including the following: • All security scanning tools • Epics • Free guest users • Web terminal for the web IDE

GitLab has a lot of features, but let's concentrate first on the basic building blocks.

The core system components of GitLab

GitLab is not a monolithic application. It tries to follow the Unix philosophy, which means that a software module should do only one particular thing, and do it well. The components that GitLab is made of are not as small and elegant as Unix's `awk` and `sed`, but each component has a single purpose. You can find a high-level overview of these components in the following diagram:



Gitlab started as a pure Ruby on Rails application, but some components were later redesigned using Go. Ruby on Rails is a development framework built on top of the Ruby programming language. It implements a model-view-controller pattern and offers methods to connect to different databases (for example, ActiveRecord). It values convention over configuration and **don't-repeat-yourself** (**DRY**) programming. It is very well suited to rapid development, and at the same time, it is highly performant and has many features.

Let's dive a little deeper into those components in order to understand their roles.

NGINX

The Unicorn web component cannot be used directly as it does not offer all the features for handling clients. The reverse proxy that is bundled by default is NGINX. It is also possible to use Apache as a frontend for GitLab, but it is preferable to use NGINX. There are many web servers available that could be installed in front of Unicorn, but in the end, there are basically two types, which are as follows:

- Process-based (forking or threaded)
- Asynchronous

NGINX and lighttpd are probably the two most-well known asynchronous servers. Apache is without a doubt the de facto standard process-based server. The biggest difference between the two types is how they handle scalability. For a process-based server, any new connections require a thread, while an event-driven, asynchronous server such as NGINX only needs a few threads (or, theoretically, only one). For lighter workloads, this does not matter much, but you will see a big difference when the number of connections grows, especially in terms of RAM. When serving tens of thousands of simultaneous connections, the amount of RAM used by NGINX would still hover around a couple of megabytes. Apache would either use hundreds, or it would not work at all. This is why NGINX is the better choice.

Debugging NGINX

The first thing you will want to look at are the log files which by default are called `error.log` and `access.log`. In a GitLab environment installed from source these log files will typically reside in `/var/log/nginx/` and in a GitLab omnibus install in `/var/log/gitlab/nginx`.

Following is an example of the error log:

```
2019/09/08 20:45:14 [crit] 2387#2387: *95 connect() to
unix:/var/www/gitlab-app/tmp/sockets/unicorn.sock failed (2: No such file
or directory) while connecting to upstream, client: 127.0.0.1, server:
localhost, request: "GET /-/metrics HTTP/1.1", upstream:
"http://unix:/var/www/gitlab-app/tmp/sockets/unicorn.sock:/-/metrics",
host: "127.0.0.1:8080"
```

Unicorn

Unicorn is an HTTP server for applications that deal with well-performing clients on connections that show low latency and have enough bandwidth. It takes advantage of features that are present in the core of Linux-like systems. It is called a **Rack HTTP server** because it implements HTTP for Rack applications. Rack, in turn, is actually a Ruby implementation of a minimal interface to deal with web requests, which you can use in your code.

You can find the project at `https://rack.github.io`.

Unicorn runs as a daemon server in Unix and is programmed in Ruby and the C programming language. Using Ruby means that it can also run a Ruby on Rails application such as GitLab.

Debugging Unicorn

Maybe installing Unicorn produced errors, or you are experiencing bad performance that you suspect is caused by Unicorn not working properly.

There are several ways to find the cause. The log files can point you in the right direction.

Timeouts in Unicorn logs

The following output is what a Unicorn worker timeout looks like in `unicorn_stderr.log`. This is not necessarily bad; it just means that a new worker is spawned:

```
 [2015-06-05T10:58:08.660325 #56227] ERROR -- : worker=10 PID:53009 timeout
(61s > 60s), killing
```

```
 [2015-06-05T10:58:08.699360 #56227] ERROR -- : reaped #<Process::Status:
pid 53009 SIGKILL (signal 9)> worker=10
 [2015-06-05T10:58:08.708141 #62538] INFO -- : worker=10 spawned pid=62538
 [2015-06-05T10:58:08.708824 #62538] INFO -- : worker=10 ready
```

It could be that there are just not enough Unicorn workers available to respond to the requests at hand. NGINX buffers a lot of requests so we must check on the handover socket whether Unicorn can keep up. To do this, a little nifty script is available here: `https://github.com/jahio/unicorn-status`.

It can be called with the following command:

```
$ ruby unicorn_status.rb /var/opt/gitlab/gitlab-rails/sockets/gitlab.socket
10
Running infinite loop. Use CTRL+C to exit.
------------------------------------------
Active Requests Queued Requests
20 11
```

The first argument here is the `unicorn_status.rb` script, the second is the socket to connect to `../.socket`, and the last argument is the poll interval (`10`).

Unicorn processes disappear

On Linux, there is a mechanism called **Out-of-Memory** (**OOM**) **Killer** that will free up memory if the system is running low on memory, and you don't have any swap memory left. It might kill Unicorn if it is using too much memory.

Use `dmesg | egrep -i 'killed process'` to search for OOM events:

```
[102335.3134488] Killed process 5567 (ruby) total-vm:13423004kB, anon-
rss:554088kB
```

Other kinds of errors or 100% CPU load

The ultimate way to debug Unicorn processes is to run `strace` on them:

1. Run `sudo gdb -p (PID)` to attach to the Unicorn process.
2. Run `call (void) rb_backtrace()` in the GDB console and find the generated Ruby backtrace in `/var/log/gitlab/unicorn/unicorn_stderr.log`:

```
 from
/opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/bundler-1.16.2/lib/bu
ndler/cli/exec.rb:28:in `run'
 from
```

```
/opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/bundler-1.16.2/lib/bu
ndler/cli/exec.rb:74:in `kernel_load'
 from
/opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/bundler-1.16.2/lib/bu
ndler/cli/exec.rb:74:in `load'
 from /opt/gitlab/embedded/bin/unicorn:23:in `<top
(required)>'<br/> from /opt/gitlab/embedded/bin/unicorn:23:in `load
 from
/opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/unicorn-5.1.0/bin/uni
corn:126:in `<top (required)>'
 from
/opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/unicorn-5.1.0/lib/uni
corn/http_server.rb:132:in `start'
 from
/opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/unicorn-5.1.0/lib/uni
corn/http_server.rb:508:in `spawn_missing_workers'
 from
/opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/unicorn-5.1.0/lib/uni
corn/http_server.rb:678:in `worker_loop'
 from
/opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/unicorn-5.1.0/lib/uni
corn/http_server.rb:678:in `select'
```

3. When you are done, leave GDB with `detach` and `q`.

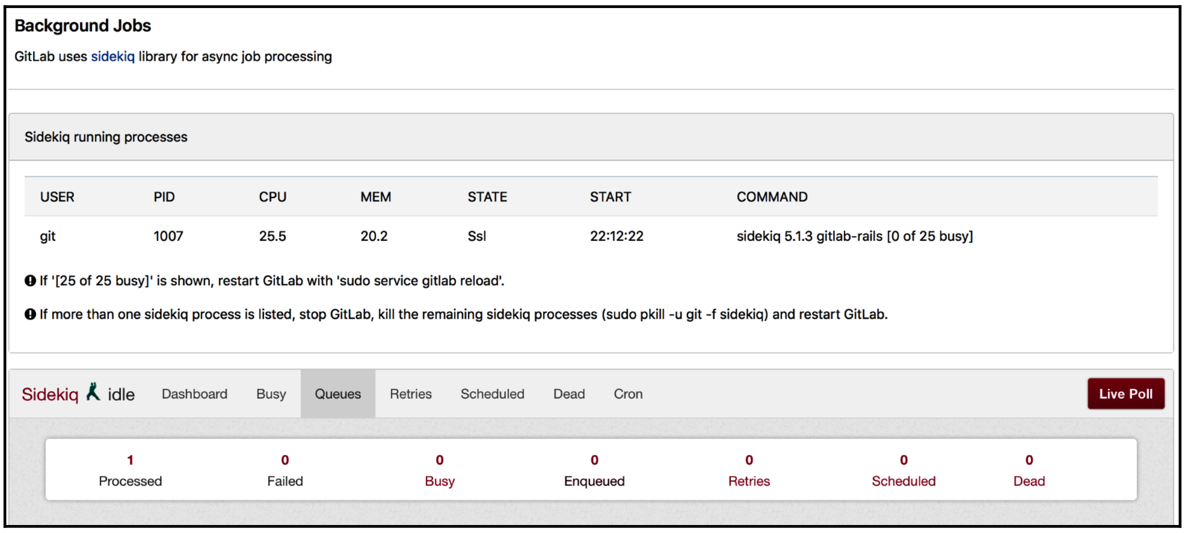
Sidekiq

Sidekiq is a framework for background job processing. It allows you to scale your application by performing work in the background. For more information on Sidekiq, consult the following website: `https://github.com/mperham/sidekiq/wiki`.

Each Sidekiq server process pulls jobs from the queue in Redis and processes them. Like your web processes, Sidekiq boots Rails so that your jobs and workers have the full Rails API available for use, including ActiveRecord. The server will instantiate the worker and call perform with the given arguments. Everything else is up to your code.

Debugging Sidekiq

As with Unicorn, there are several ways to debug Sidekiq processing. The easiest way is to log in to GitLab as an administrator and view the logs from there, and especially view the queues and jobs on the **Background Jobs** page, as shown in the following screenshot:



Background Jobs

GitLab uses sidekiq library for async job processing

Sidekiq running processes

USER	PID	CPU	MEM	STATE	START	COMMAND
git	1007	25.5	20.2	Ssl	22:12:22	sidekiq 5.1.3 gitlab-rails [0 of 25 busy]

If '[25 of 25 busy]' is shown, restart GitLab with 'sudo service gitlab reload'.

If more than one sidekiq process is listed, stop GitLab, kill the remaining sidekiq processes (sudo pkill -u git -f sidekiq) and restart GitLab.

1	0	0	0	0	0	0
Processed	Failed	Busy	Enqueued	Retries	Scheduled	Dead

Sometimes, you experience troubles and find situations on your Linux server.

Sidekiq processes disappear

As mentioned before, in the Unicorn section, the OOM Killer might kill Sidekiq if it is using too much memory.

Use `dmesg | egrep -i 'killed process'` to search for OOM events:

```
[102335.3134488] Killed process 8887 (ruby) total-vm:13523004kB, anon-
rss:5540458kB
```

A Sidekiq process is seemingly doing nothing

If Sidekiq isn't doing any work and it seems stuck most of the time, this means that the program is waiting for something. A common wait situation is when you are doing remote network calls. If you think this could be the case, you could make Sidekiq processes dump a backtrace to the log by sending it a TTIN signal.

This is what a Sidekiq worker looks like in the log file in `/var/log/gitlab/sidekiq/current`:

```
 {"severity":"INFO","time":"2019-06
23T19:00:14.493Z","class":"RemoteMirrorNotificationWorker","retry":3,"queue
":"remote_mirror_notification","jid":"69eb806bfb66b82315bcb249","created_at
":"2019-06-23T19:00:14.461Z","correlation_id":"toX0HnYW0s9","enqueued_at":"
2019-06-23T19:00:14.461Z","pid":471,"message":"RemoteMirrorNotificationWork
er JID-69eb806bfb66b82315bcb249: done: 0.03
sec","job_status":"done","duration":0.03,"completed_at":"2019-06-23T19:00:1
4.493Z"}
```

Since GitLab 12.0, the default output log format for Sidekiq is JSON, this makes it easier to read the log files into a tool like logstash because it is more structured.

Other kind of errors or 100% CPU load

The ultimate way to debug Sidekiq processes is to make it dump a backtrace via GDB:

1. Run `sudo gdb -p (PID)` to attach to the Sidekiq worker process.
2. Run `call (void) rb_backtrace()` in the GDB console and find the generated Ruby backtrace in `/var/log/gitlab/sidekiq/current`:

    ```
    2018-09-21_19:55:03.48430 from
    /opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/redis-3.3.5/lib/redis
    /connection/ruby.rb:83:in `_read_from_socket'
    2018-09-21_19:55:03.48431 from
    /opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/redis-3.3.5/lib/redis
    /connection/ruby.rb:87:in `rescue in _read_from_socket'
    2018-09-21_19:55:03.48432 from
    /opt/gitlab/embedded/lib/ruby/gems/2.4.0/gems/redis-3.3.5/lib/redis
    /connection/ruby.rb:87:in `select'
    ```

3. It is very hard to read backtraces, but this process was doing network operations while being traced, we can see a (`_read_from _socket`). You can read the source code to check what it is doing (there are line numbers mentioned).
4. When you are done, leave GDB with `detach` and quit.

You can also use other tracing tools to examine the behavior of the looping process. On Linux, for instance, `strace -p <pid>` allows you to view the system calls that are being made by the process.

GitLab Shell

This component is used to provide access to Git repositories through SSH. In fact, for pushes via the `git-http` protocol, it is also called instead of the Rails app. It's essentially a small Ruby wrapper around the Git client. Git, through SSH, uses predefined commands that can be executed on the GitLab server. For authorization, it makes calls to the GitLab API. Before GitLab 5.0, this functionality was delivered by Gitolite and powered by the Perl programming language.

The source code of this project can be found here: `https://gitlab.com/gitlab-org/gitlab-shell`. You can see the following page:

GitLab.org › gitlab-shell › Details
gitlab-shell
SSH access and repository management app for GitLab
Project ID: 14022
coverage 96.99%
pipeline passed
maintainability A
Star 92
Fork 143
SSH
git@gitlab.com:gitlab-org/gitl
Global
Files (6.3 MB) Commits (1,111) Branches (30) Tags (132) Readme Changelog LICENSE Contribution guide CI/CD configuration
master
gitlab-shell /
History
Find file

You can install it locally, but it's really only useful when deployed together with other GitLab components. When you have that installed (see `Chapter 2`, *Installing GitLab*, for instructions on how), the next section describes a way to debug when you have problems.

Debugging GitLab Shell

In an omnibus installation, the log file for GitLab Shell can be found in the following location:

```
/var/log/gitlab/gitlab-shell/gitlab-shell.log
```

Alternatively, it may be found in the following location, for installations from source:

```
/home/git/gitlab-shell/gitlab-shell.log
```

What you will generally find are log lines that concern the basic operations of GitLab Shell:

- Git commands (such as `git push` and `git pull`).
- Authorization calls to the GitLab Rails API to check whether you are allowed to connect
- Execution of pre-receive hooks
- Actions requested
- Post-receive actions
- Any custom post-receive actions

Here, we have listed some lines from the log file:

```
bash-4.1$ tail gitlab-shell.log
time="2018-09-26T08:59:53+02:00" level=info msg="executing git command"
command="gitaly-upload-pack unix:/var/opt/gitlab/gitaly/gitaly.socket
{\"repository\":{\"storage_name\":\"default\",\"relative_path\":\"xxx/xxx.g
it\",\"git_object_directory\":\"\",\"git_alternate_object_directories\":[],
\"gl_repository\":\"xxx\"},\"gl_repository\":\"project-
xx\",\"gl_id\":\"key-xx\",\"gl_username\":\"xxxxxx\"}" pid=18855 user="user
with key key-xx"

time="2018-09-26T08:59:53+02:00" level=info msg="finished HTTP request"
duration=0.228132057
method=POST pid=18890 url="http://127.0.0.1:8080/api/v4/internal/allowed"

time="2018-09-26T08:59:54+02:00" level=info msg="finished HTTP request"
duration=0.030036933 method=POST pid=18890
url="http://127.0.0.1:8080/api/v4/internal/pre_receive"

time="2018-09-26T08:59:54+02:00" level=info msg="finished HTTP request"
duration=0.094035804 method=POST pid=18979
url="http://127.0.0.1:8080/api/v4/internal/post_receive"
```

One way to find errors is to look for certain patterns, such as `failed`, as follows. This particular error points to a 500 error from Unicorn while checking whether a user has the right authorization to make a call to the GitLab API.

This error should show up in the Unicorn logs (`production.log`) if you search for an HTTP 500 error:

```
bash-4.1$ grep -i failed gitlab-shell.log
time="2018-09-26T08:05:52+02:00" level=error msg="API call failed"
body="{\"message\":\"500 Internal Server Error\"}" code=500 method=POST
pid=1587 url="http://127.0.0.1:8080/api/v4/internal/allowed"
time="2018-09-26T08:45:13+02:00" level=error msg="API call failed"
body="{\"message\":\"500 Internal Server Error\"}" code=500 method=POST
pid=24813 url="http://127.0.0.1:8080/api/v4/internal/allowed"
```

Redis

Redis is a caching tool and HTTP session store that allows you to save cached data and session information from your website to an external location. This means that your website doesn't have to calculate everything every time; instead, it can retrieve the data from the cache and load the website much faster. The user sessions are in memory even if the application goes down. Redis is a fast caching tool because it uses memory first. It has several useful advantages:

- Everything is stored in one place, so you only have to flush one cache.
- It is faster than Memcache. This is noticeable when using the websites of large shops.
- Sessions are stored in memory and not in the database.
- The backend becomes faster.

Redis is not merely a cache, but is also a data structure store. It is basically a database and should be viewed conceptually as such. With regard to its operation and how it handles data, it has more in common with a NoSQL database.

Basic data operations in Redis

We can discover some of the basics of Redis by playing with the data structures. You can install Redis using instructions found at `https://github.com/PacktPublishing/Mastering-GitLab-12/tree/master/Chapter01/InstallingRedis.md`.

Start the `redis-cli` command-line utility, and it will connect to the local Redis server:

```
$redis-cli
127.0.0.1:6379>
```

It is not fair to view Redis as a simple hash database with key values. But still, the five data structures that are provided do actually consist of a key and a value. Let's sum up the five data structures:

- **String**: You can use the `set` command to write a value to Redis. In the case of a simple string, you can simply save the value in the datastore shown as follows. After setting the string value, you can retrieve the value again by issuing the `get` command:

```
$ redis-cli
127.0.0.1:6379> set mykind "Human"
OK
127.0.0.1:6379> get mykind
"Human"
127.0.0.1:6379>
```

- **Hash**: In the same way as the string, you can `set` an arbitrary number of values to a key. Generally speaking, Redis treats values as a byte array and doesn't care what they are. This make Redis very handy for representing objects. Again, with the `get` command, you can retrieve the values. GitLab uses this type to store web session information from users:

```
$ redis-cli
127.0.0.1:6379> set programs:tron '{"name": "tron","kind":
"program"}'
OK
127.0.0.1:6379> get programs:tron
"{\"name\": \"tron\",\"kind\": \"program\"}"
```

- **List**: The list type in Redis is implemented as a linked list. You can add items to the list quite quickly with `rpush` (right push, to the tail of the list) or `lpush` (left push, to the head of the list). On the other hand, accessing an item by index is not that fast because it has to search the linked list. Still, for a queue mechanism, this is a good solution.

```
$ redis-cli
127.0.0.1:6379> rpush specieslist human computer cyborg
(integer) 3
127.0.0.1:6379> rpop specieslist
"cyborg"
127.0.0.1:6379> rpop specieslist
"computer"
127.0.0.1:6379> rpop specieslist
"human"
127.0.0.1:6379> rpop specieslist
(nil)
```

- **Sets**: Another datatype is the set. You add members with the `sadd` command. Don't forget that these sets are unordered, so if you ask for the members with `smembers`, the order will mostly be different to how you entered it:

```
$ redis-cli
127.0.0.1:6379> sadd speciesset human computer cyborg
(integer) 3
127.0.0.1:6379> smembers speciesset
1) "computer"
2) "human"
3) "cyborg"
```

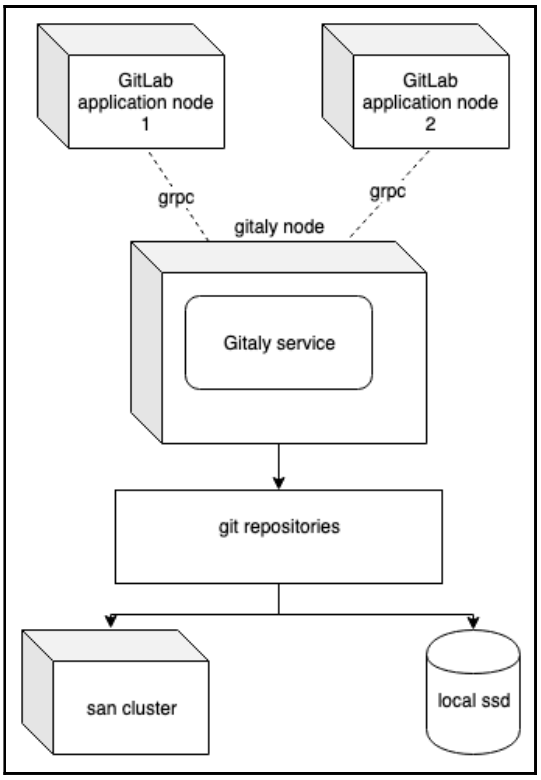
- **Sorted sets**: Fortunately, there is an ordered set as well. It is almost the same, but one difference is that you add a score to the entry, and that will automatically score the sort order, as you can see from the following:

```
127.0.0.1:6379> zadd speciessortedset 1 human
(integer) 1
127.0.0.1:6379> zadd speciessortedset 2 computer
(integer) 1
127.0.0.1:6379> zadd speciessortedset 3 cyborg
(integer) 1
127.0.0.1:6379> zrange speciessortedset 0 -1
1) "human"
2) "computer"
3) "cyborg"
```

Gitaly

In the first versions of GitLab, all Git operations relied on using a local disk or network share. Gitaly is a project that tries to eliminate reliance on the **Network File System** (**NFS**). Instead of calls to a filesystem service, Gitaly provides GitLab with a system based on **Remote Procedure Calls** (**RPCs**) to access Git repositories. It is written in Go and uses **gRPC Remote Procedure Call** (**gRPC**), a cross-platform RPC framework from Google. It has been steadily developing since the beginning of 2017, and since GitLab 11.4, it can replace the need for a shared NFS filesystem.

You can find an overview of Gitaly and its place in the GitLab architecture in the following screenshot:



On a small installation, it runs in the same servers as all other components. In big clustered environments, you can set up dedicated Gitaly servers, which can be used by Gitaly clients such as the following:

- Unicorn
- Sidekiq
- `gitlab-workhorse`
- `gitlab-shell`
- Elasticsearch indexer
- Gitaly as a client

The source code of this project can be found here: `https://gitlab.com/gitlab-org/gitaly`.