

Capacitación de Git y GitLab
Orientado a la Administración de
Proyectos

INDICE

1. Git
 - 1.1. ¿Qué es Git?
 - 1.2. Características
 - 1.3. Primeros pasos
 - 1.4. Cheat Sheet
2. GitLab
 - 2.1. Dashboard
 - 2.1.1. Activity
 - 2.1.2. Projects List
 - 2.1.3. Issues List
 - 2.1.4. Merge Requests List
 - 2.1.5. Help
 - 2.2. Explore
 - 2.3. Snippets
 - 2.4. Admin Area
 - 2.4.1. Overview
 - 2.4.2. Admin Projects
 - 2.4.3. Admin Users
 - 2.4.4. Admin Groups
 - 2.4.5. View Logs
 - 2.4.6. View Messages
 - 2.4.7. Hooks
 - 2.4.8. Background Jobs
 - 2.5. Profile Settings
 - 2.5.1. Profile
 - 2.5.2. Account
 - 2.5.3. Emails
 - 2.5.4. Password
 - 2.5.5. Notifications
 - 2.5.6. SSH Keys
 - 2.5.7. Design
 - 2.5.8. Groups
 - 2.5.9. History
 - 2.6. Project Details
 - 2.6.1. Project
 - 2.6.2. Files
 - 2.6.3. Commits
 - 2.6.3.1. Commits
 - 2.6.3.2. Compare
 - 2.6.3.3. Branches
 - 2.6.3.4. Tags
 - 2.6.4. Network
 - 2.6.5. Graphs
 - 2.6.6. Issues

- 2.6.6.1. Issues
- 2.6.6.2. Merge Requests
- 2.6.6.3. Milestones
- 2.6.6.4. Labels

2.6.7. Wiki

- 2.6.8. Project Settings
 - 2.6.8.1. Project
 - 2.6.8.2. Members
 - 2.6.8.3. Deploy keys
 - 2.6.8.4. Web Hooks
 - 2.6.8.5. Services
 - 2.6.8.6. Protected branches

3. Git

- 3.1. Sistema de versionado
- 3.2. Branching model

4. Conclusión

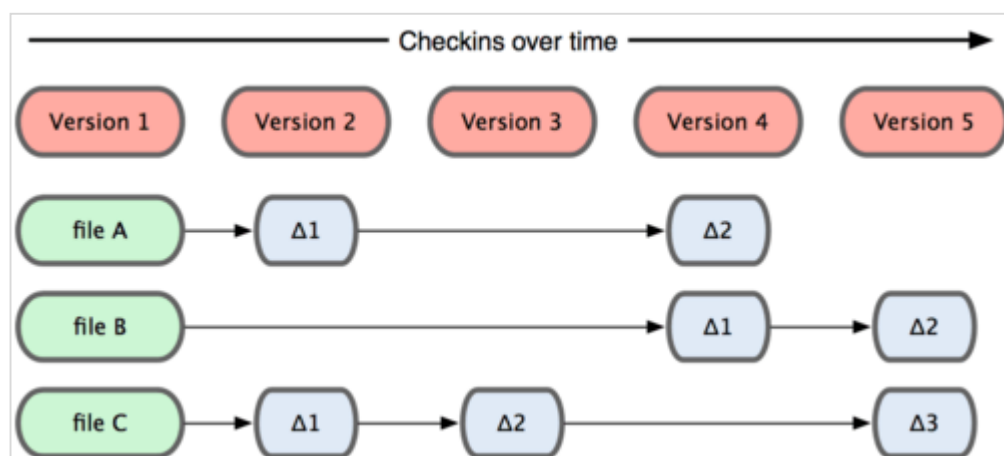
1. Git

1.1. ¿Qué es Git?

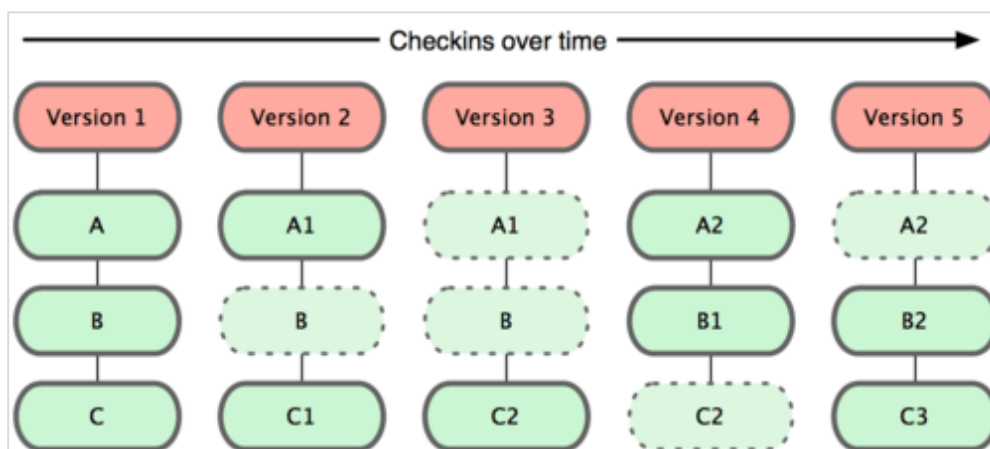
Git (pronunciado "guit") es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

1.2. Características

La principal diferencia entre Git y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, como ilustra la siguiente imagen.



Git no modela ni almacena sus datos de este modo. Lo hace más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado. Git modela sus datos más como en la siguiente figura.



Esta es una distinción importante entre Git y prácticamente todos los demás VCSs. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas

copiaron de la generación anterior. Esto hace que Git se parezca más a un mini sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, que a un VCS. Exploraremos algunos de los beneficios que obtienes al modelar tus datos de esta manera cuando veamos ramificaciones (branching) en Git.

Casi cualquier operación es local

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de tu red. Si estás acostumbrado a un VCS donde la mayoría de las operaciones tienen esa sobrecarga del retardo de la red, este aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Como tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita salir al servidor para obtener la historia y mostrártela, simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi al instante. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedas hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es imposible o muy doloroso. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor; y en Subversion y CVS, puedes editar archivos, pero no puedes confirmar los cambios a tu base de datos (porque tu base de datos no tiene conexión). Esto puede no parecer gran cosa, pero te sorprendería la diferencia que puede suponer.

Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-

1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura de directorios. Un hash SHA-1 tiene esta pinta:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Verás estos valores hash por todos lados en Git, ya que los usa con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

Git generalmente sólo añade información

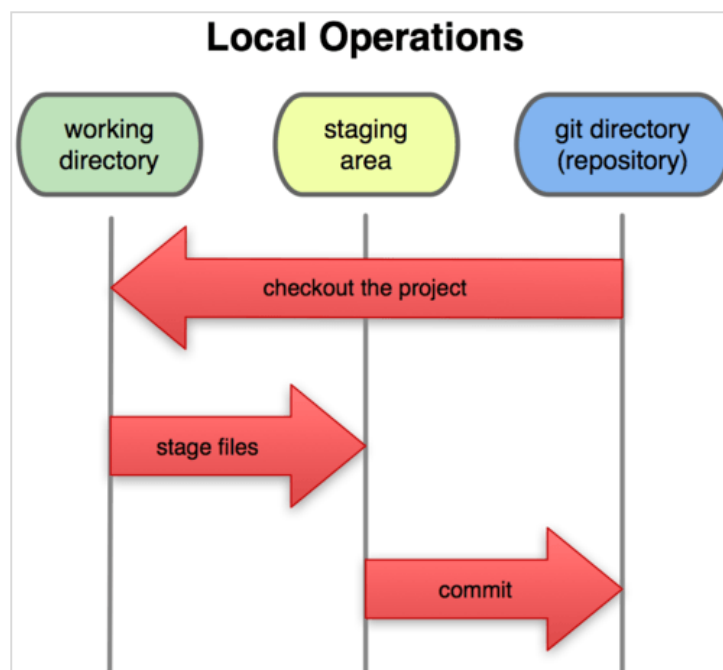
Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy difícil de perder, especialmente si envías (push) tu base de datos a otro repositorio con regularidad.

Esto hace que usar Git sea un placer, porque sabemos que podemos experimentar sin peligro de fastidiar gravemente las cosas.

Los tres estados

Ahora presta atención. Esto es lo más importante a recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). *Confirmado* significa que los datos están almacenados de manera segura en tu base de datos local. *Modificado* significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. *Preparado* significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).



El *directorio de Git* es donde Git almacena los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otro ordenador.

El *directorio de trabajo* es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El *área de preparación* es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice, pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.

3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

1.3. Primeros Pasos

El entrenamiento básico puede verse en la página oficial de Git:

<https://try.github.io/levels/1/challenges/1>

Debes realizar este entrenamiento antes de seguir con la capacitación. De otra manera les será difícil entender el funcionamiento de este sistema de versionado.

1.4. Cheat Sheet

Los siguientes comandos representan las operaciones principales que pueden realizarse en un repositorio Git.

- `git fetch`
Descarga los cambios realizados en el repositorio remoto. No modifica tus archivos, sólo actualiza tu base de datos local de cambios.
- `git merge <nombre_rama>:`
Impacta en la rama en la que te encuentras parado, los cambios realizados en la rama “nombre_rama”.
- `git pull:`
Unifica los comandos *fetch* y *merge* en un único comando.
- `git commit -m "<mensaje>":`
Confirma los cambios realizados. El “mensaje” generalmente se usa para asociar al *commit* una breve descripción de los cambios realizados. Por defecto es obligatorio agregar un mensaje a los commits.
- `git push origin <nombre_rama>:`
Sube la rama “nombre_rama” al servidor remoto.
- `git status:`
Muestra el estado actual de la rama, como los cambios que hay sin commitear.
- `git add <nombre_archivo>:`
Comienza a trackear el archivo “nombre_archivo”.
- `git checkout -b <nombre_rama_nueva>:`
Crea una rama a partir de la que te encuentres parado con el nombre “nombre_rama_nueva”, y luego salta sobre la rama nueva, por lo que quedas parado en ésta última.

- `git checkout -t origin/<nombre_rama>`:
Si existe una rama remota de nombre “nombre_rama”, al ejecutar este comando se crea una rama local con el nombre “nombre_rama” para hacer un seguimiento de la rama remota con el mismo nombre.
- `git branch`:
Lista todas las ramas locales.
- `git branch -a`:
Lista todas las ramas locales y remotas.
- `git branch -d <nombre_rama>`:
Elimina la rama local con el nombre “nombre_rama”.
- `git push origin :<nombre_rama>`:
Elimina la rama remote con el nombre “nombre_rama”.
- `git remote prune origin`:
Actualiza tu repositorio remoto en caso que algún otro desarrollador haya eliminado alguna rama remota.
- `git reset --hard HEAD`:
Elimina los cambios realizados que aún no se hayan hecho *commit*.
- `git revert <hash_commit>`:
Revierte el *commit* realizado, identificado por el “hash_commit”. El “hash_commit” es el SHA-1 que vimos anteriormente.

2. GitLab

Es un software basado en tecnología web utilizado para gestionar repositorios Git. Además tiene integradas otras características como Wiki y Snippets, entre otras, que veremos más adelante.

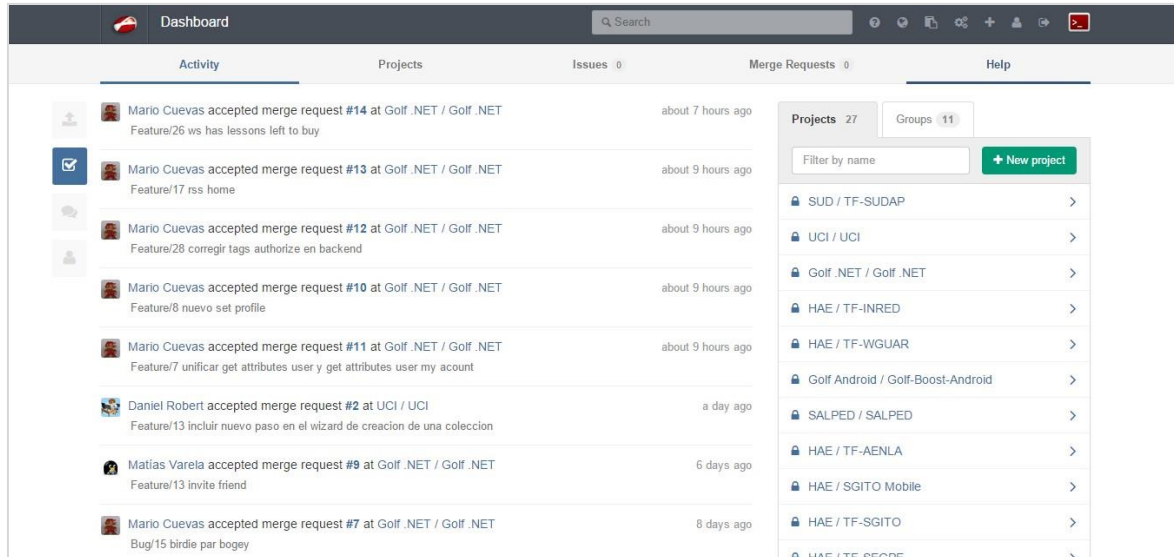
Está escrito en Ruby y tiene dos versiones:

- a) GitLab CE: Community Edition (gratis)
- b) GitLab EE: Enterprise Edition (de pago)

La diferencia entre ambas versiones es que la EE tiene algunas funcionalidades extras como tablero de scrum, entre otras.

2.1. Dashboard

Aquí puedes ver un resumen de las actividades, proyectos, issues y merge requests, cuyos significados veremos más adelante. Además, puedes encontrar otros links de interés.



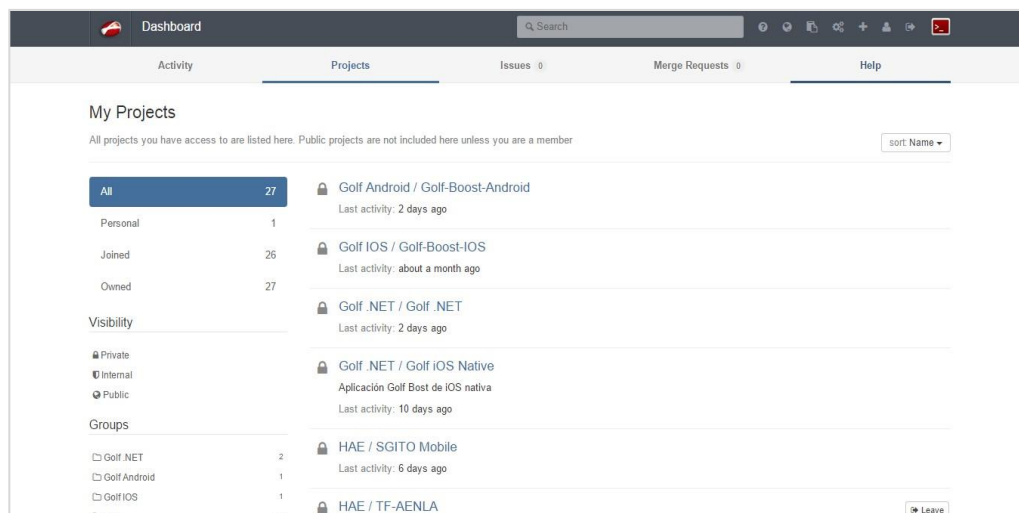
2.1.1. Activity

Listado de incidencias en todos los proyectos en los que tienes participación. Puedes filtrar las actividades para observar diferentes cambios.

En la sección derecha tienes un detalle de los proyectos y los grupos a los que perteneces. Además tienes links que podrán ayudarte a comprender el funcionamiento de Git.

2.1.2. Projects List

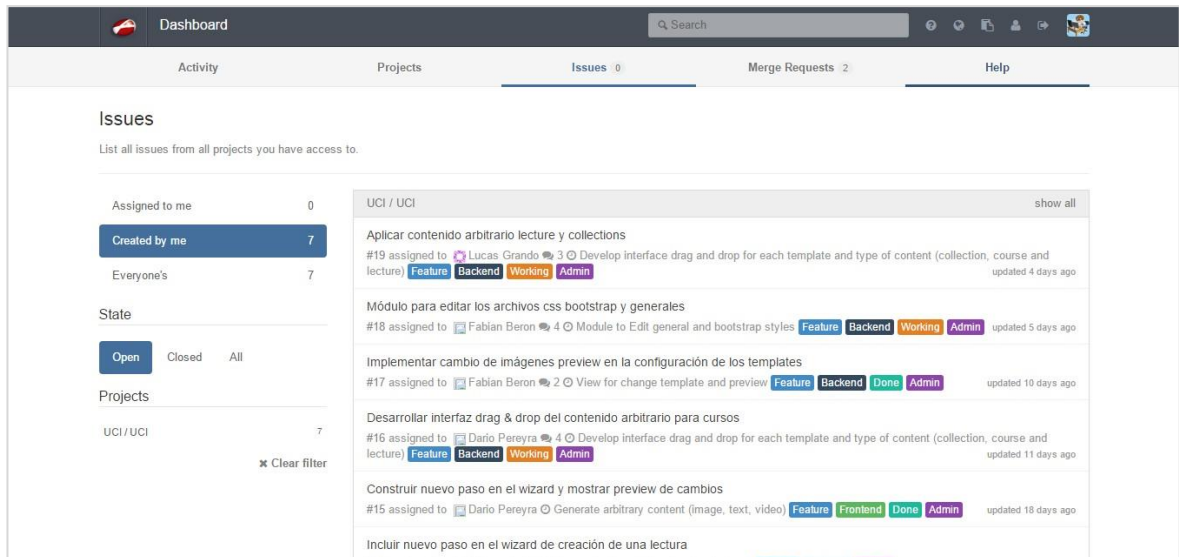
En esta vista podrás encontrar un listado de todos los proyectos en los que estas participando.



El sidebar derecho te permitirá filtrar el listado dependiendo del tipo de participación y de la visibilidad de los mismos. Además, podrás filtrar por los grupos de cada proyecto y ordenarlos por fecha de creación o edición.

2.1.3. Issues List

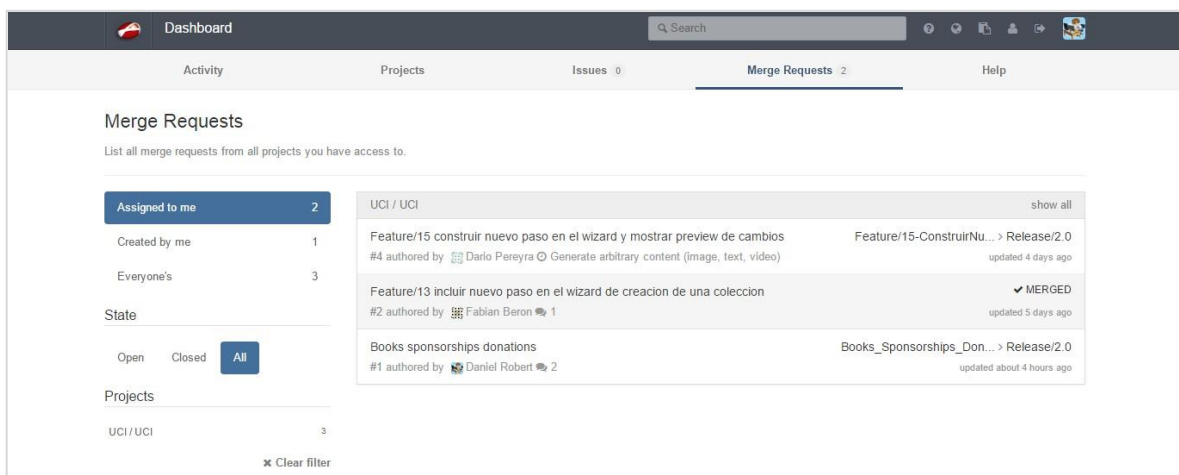
Aquí podrás ver un resumen de los issues de todos tus proyectos.



El listado te permitirá acceder al detalle de un issue, cerrarlo o editarlo. Además podrás filtrar por el nivel de usuario como así también por el estado y por el proyecto.

2.1.4. Merge Requests List

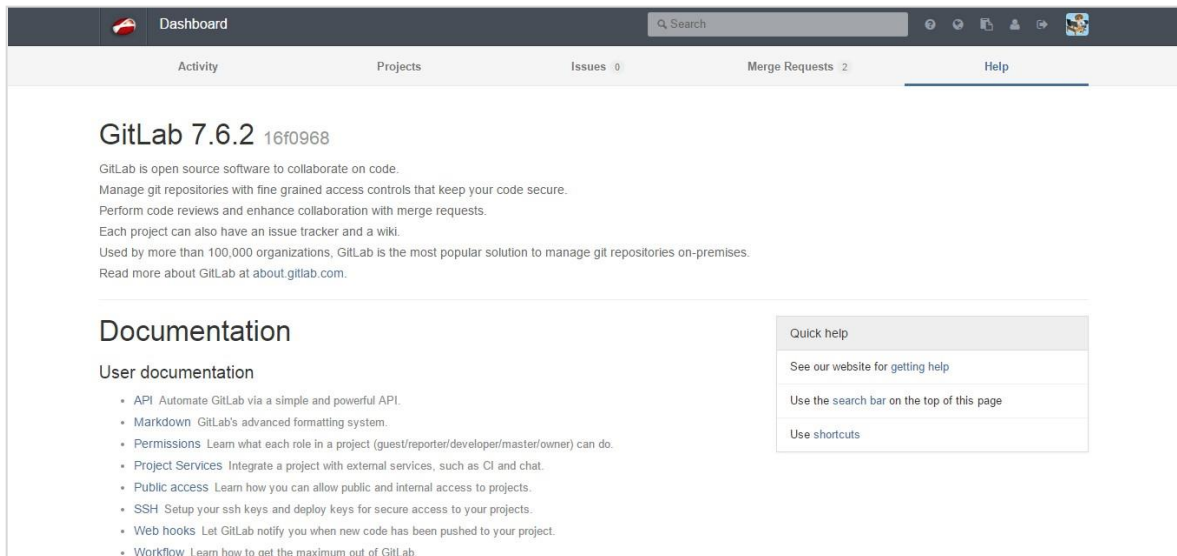
A continuación podrás visualizar los merge requests de todos tus proyectos.



Puedes filtrar por nivel de usuario, por estado y por proyecto. Además, a través del nombre del merge request ingresarás al detalle.

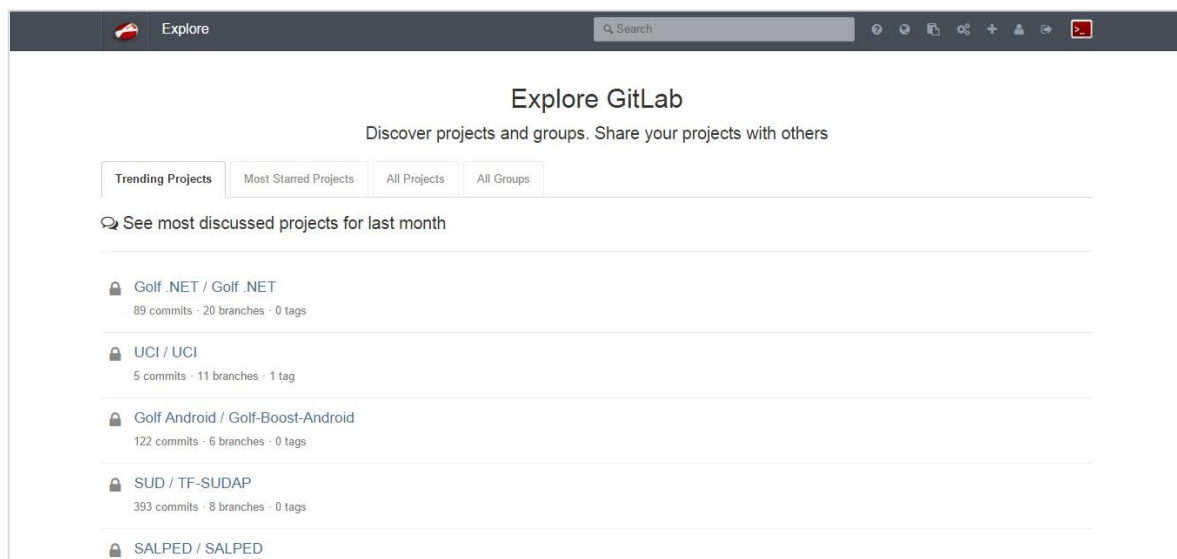
2.1.5. Help

En esta sección encontrarás una gran cantidad de links que te ayudarán a entender el funcionamiento de Git y GitLab.



2.2. Explore

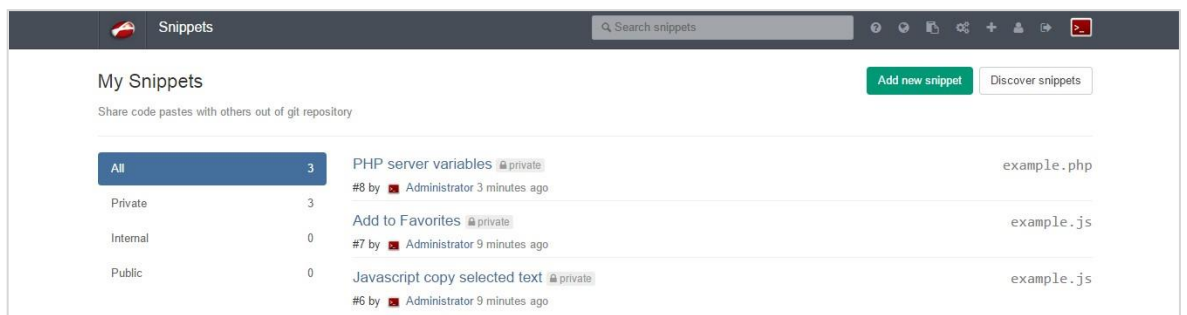
La vista de exploración te permitirá acceder a tus proyectos y a los grupos a los que perteneces.



2.3. Snippets

Un snippet es una pequeña parte reusable de código fuente. Pueden ser funciones, métodos de clase o simplemente una línea de código. Son usados para agilizar la resolución de problemas comunes y para disminuir la redundancia de código.

En la siguiente pantalla podrás visualizar el listado de tus Snippets:

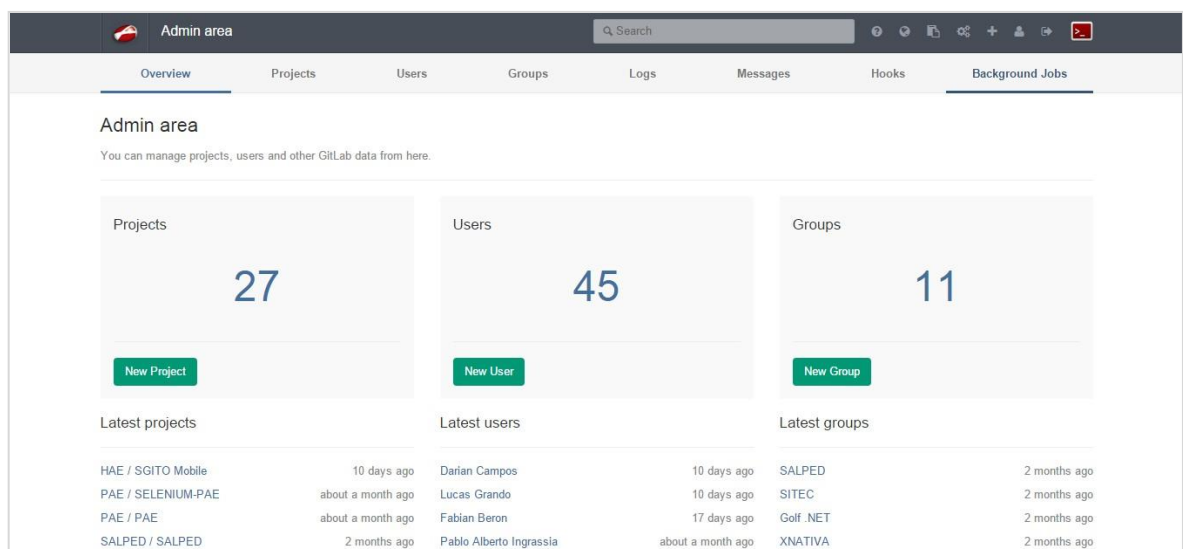


El sidebar izquierdo te permite filtrar por el nivel de visibilidad. Puedes agregar un nuevo Snippet y ver los Snippets públicos de todos los usuarios a través del botón “Discover snippets”.

2.4. Admin Area

2.4.1. Overview

Aquí podrás ver un resumen de los proyectos, usuarios y grupos, como así también algunas estadísticas.



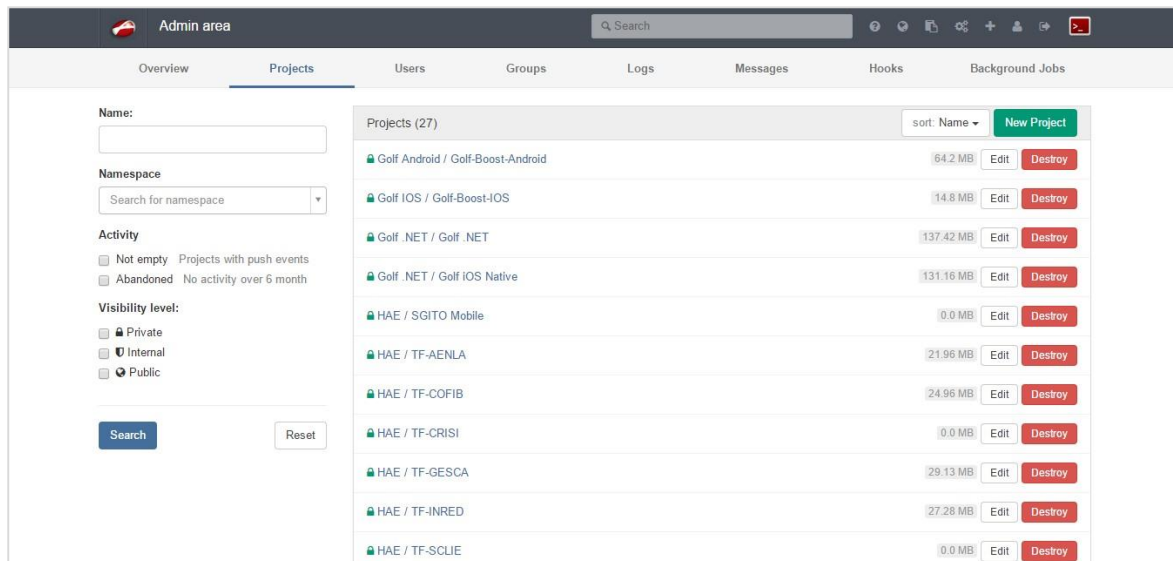
| Stats | Features | Components |
|---------------------------|----------|------------|
| Forks | 0 | Sign up |
| Issues | 70 | LDAP |
| Merge Requests | 31 | Gravatar |
| Notes | 201 | OmniAuth |
| Snippets | 6 | |
| SSH Keys | 5 | |
| Milestones | 39 | |
| Active users last 30 days | 27 | |

| | |
|--------------|-----------|
| GitLab | 7.6.2 |
| GitLab Shell | 2.4.0 |
| GitLab API | v3 |
| Ruby | 2.1.4p265 |
| Rails | 4.1.1 |

Puedes acceder al detalle de cada proyecto, usuario o grupo haciendo clic en el nombre o crear uno nuevo.

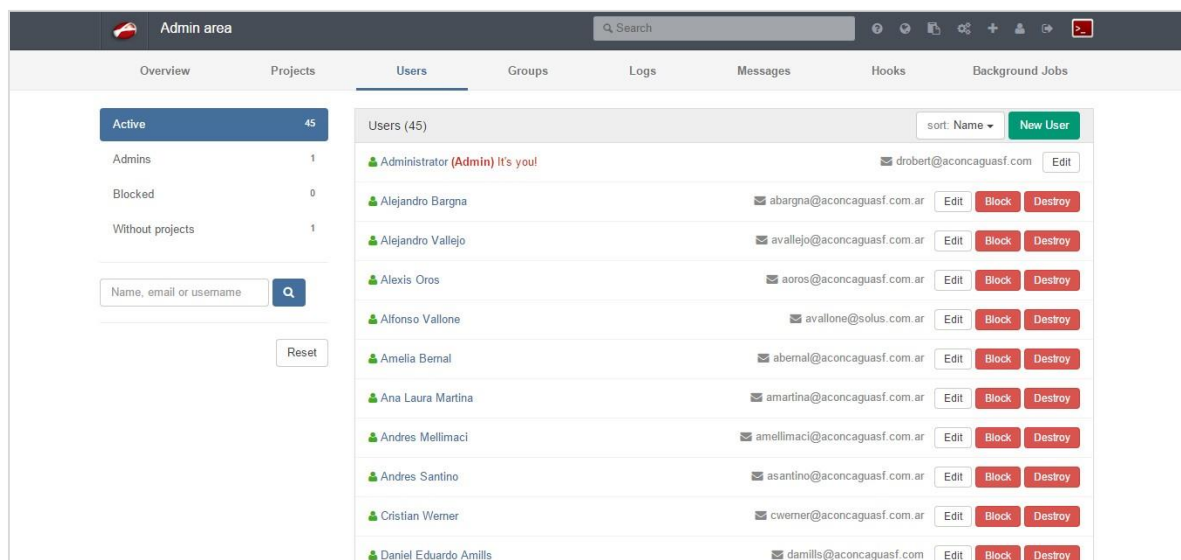
2.4.2. Admin Projects

Esta pantalla te permite acceder a la edición un proyecto o eliminarlo. También puedes filtrar el listado de proyectos por nombre, grupo, nivel de visibilidad, etc.



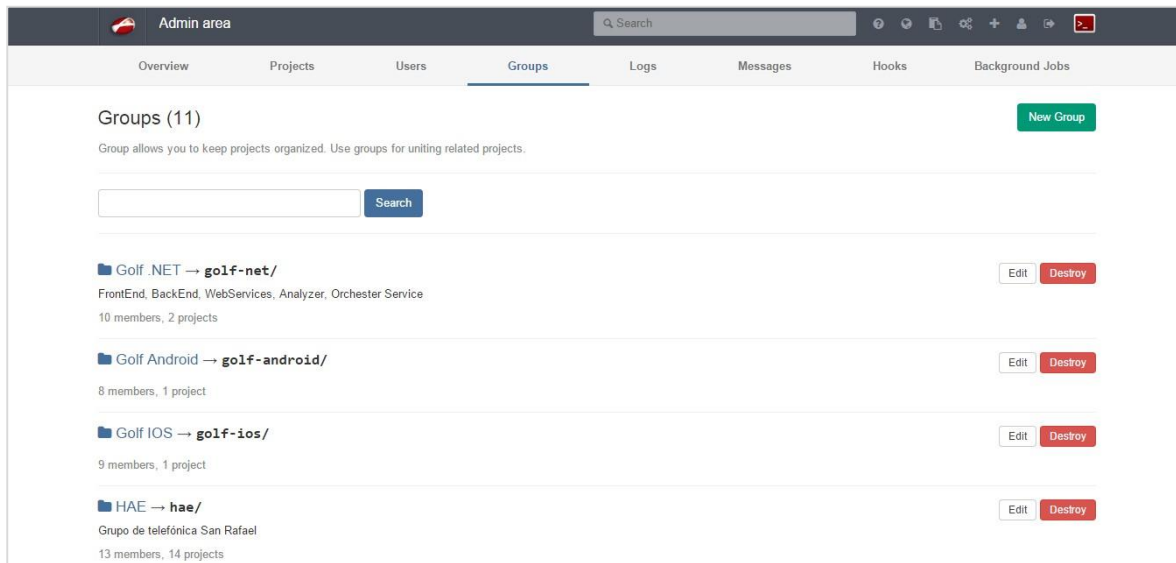
2.4.3. Admin Users

En esta sección visualizarás el listado de usuarios. Tienes botones para acceder a la edición de un usuario o eliminarlo. Además el sidebar izquierdo te permite filtrar el listado por nombre, grupo o nivel de visibilidad.



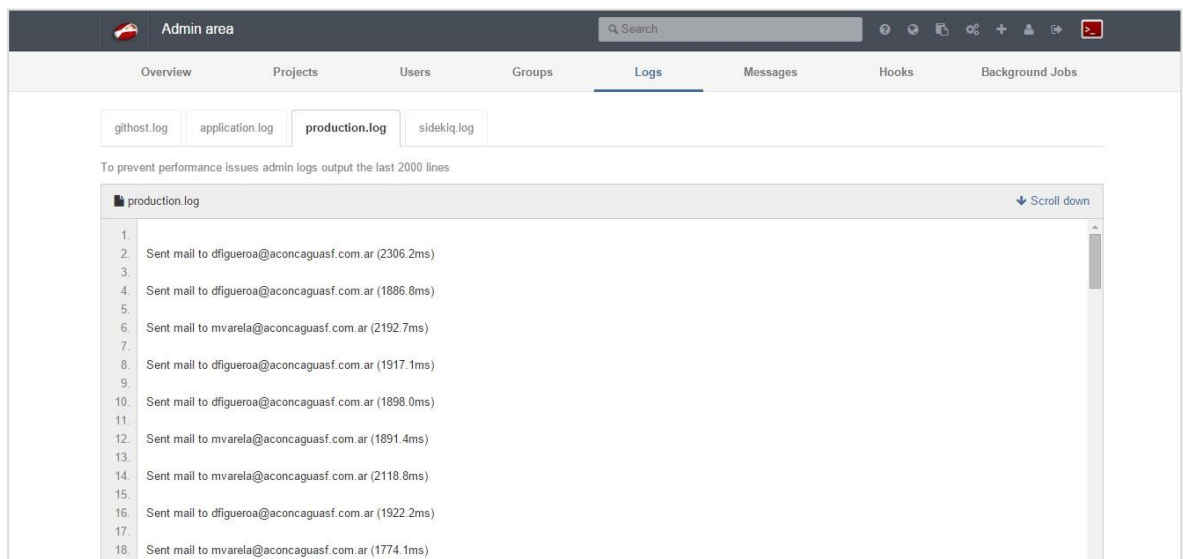
2.4.4. Admin Groups

Al igual que con los usuarios, el listado de grupos te permite crear, editar, eliminar un grupo como así también filtrar.



2.4.5. View Logs

En cada tab que ves a continuación puedes visualizar los logs de GitLab.



2.4.6. View Messages

En la siguiente sección puedes configurar mensajes que les aparecerá a los usuarios al loguearse. Esa funcionalidad es útil por ejemplo cuando quieres realizar actualizaciones, mantenimientos programados, etc.

The screenshot shows the 'Admin area' of GitLab with the 'Messages' tab selected. The page title is 'Broadcast Messages'. Below the title, a description states: 'Broadcast messages are displayed for every user and can be used to notify users about scheduled maintenance, recent upgrades and more.' There is a placeholder for a message: 'Your message here'. Below this, there are input fields for 'Message', 'Background Color' (with a hint: '6 character hex values starting with a # sign.'), and 'Font Color' (with the same hint). There are also 'Starts at' and 'Ends at' date and time pickers, both set to '2015', 'June', '20', '21', and '09'. At the bottom, there is a green button labeled 'Add broadcast message'.

2.4.7. Hooks

Los hooks pueden ser utilizados para notificaciones, cambios en el servidor LDAP, etc.

The screenshot shows the 'Admin area' of GitLab with the 'Hooks' tab selected. The page title is 'System hooks'. Below the title, a description states: 'System hooks can be used for binding events when GitLab creates a User or Project.' There is a 'URL:' label followed by an input field. At the bottom, there is a green button labeled 'Add System Hook'.

Para más información de los hooks visita:

http://asf-git-server.aconcaguasf.com.ar/help/system_hooks/system_hooks

2.4.8. Background Jobs

En esta sección se puede visualizar las tareas que realiza Git en background. Pueden filtrarse y cuenta con un gráfico donde se comparan las tareas procesadas con las fallidas.

Admin area

Q Search

Overview

Projects

Users

Groups

Logs

Messages

Hooks

Background Jobs

Background Jobs


GitLab uses sidekiq library for async job processing

Sidekiq running processes

| USER | PID | CPU | MEM | STATE | START | COMMAND |
|------|-------|-----|------|-------|-------|---|
| git | 14679 | 0.1 | 17.1 | Ssl | Apr | 22 sidekiq 2.17.8 gitlab-rails [0 of 25 busy] |

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

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

Sidekiq  inactivo

Panel de Control

Trabajadores

Filas

Reintentos

Programadas

11,530

Procesadas

10,287

Fallidas

0

Ocupado

0

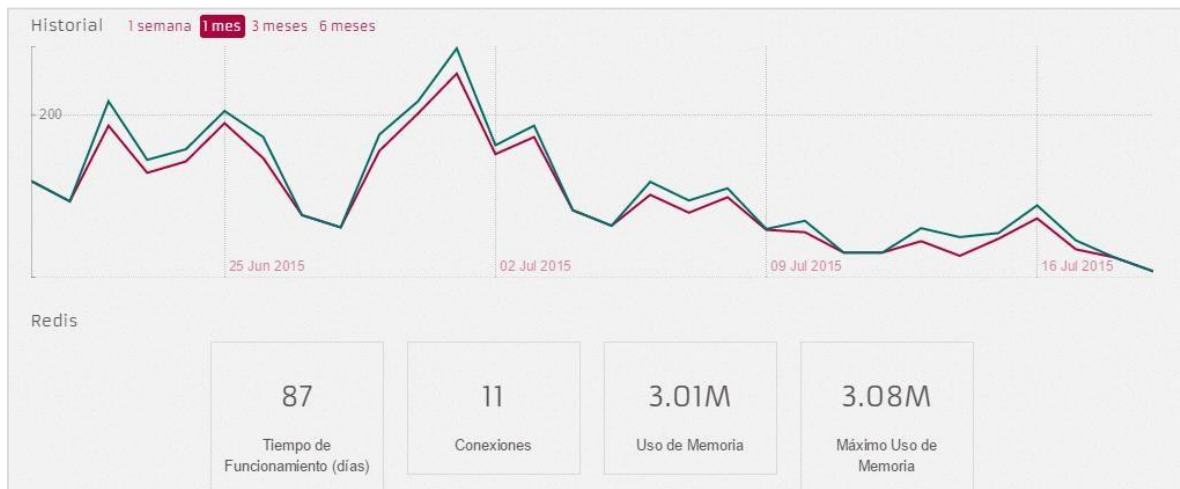
En Fila

58

Reintentos

0

Programadas

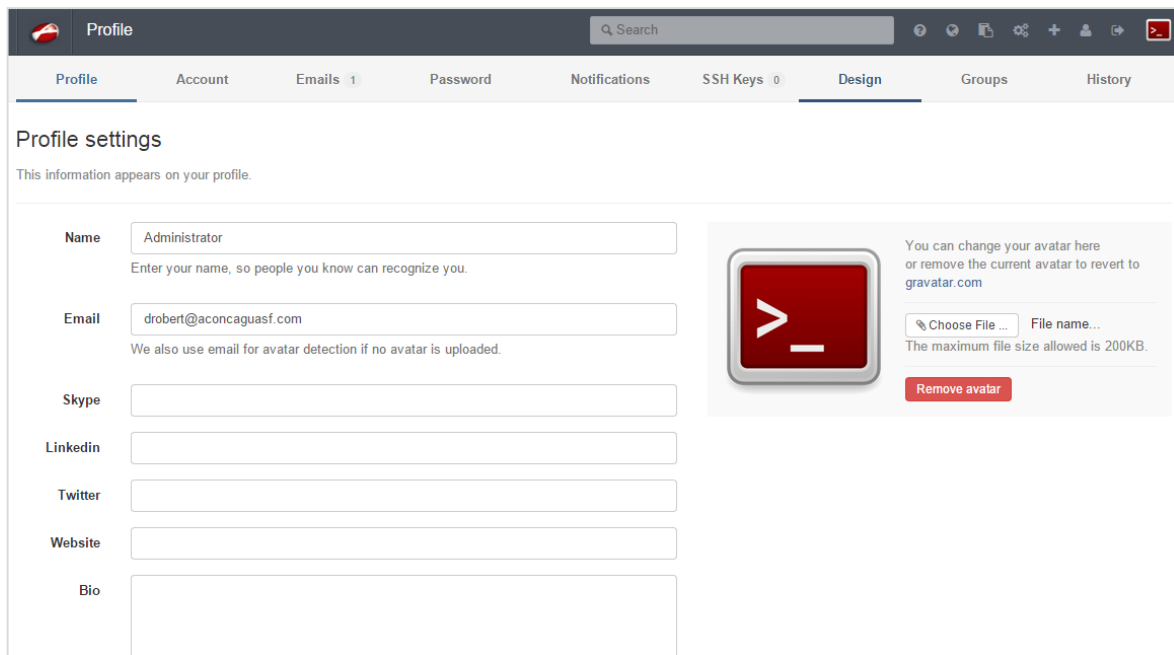


2.5. Profile Settings

2.5.1. Profile

Esta pantalla permite cargar la información relacionada con tu perfil.

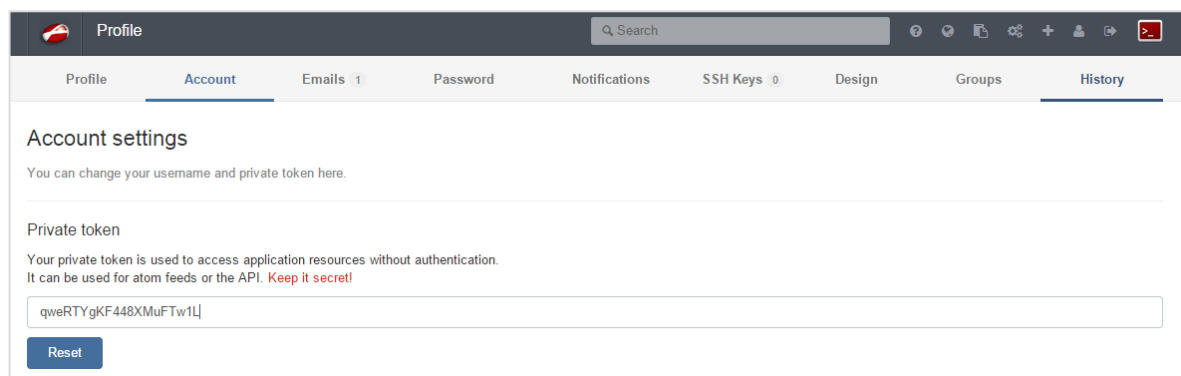
La imagen de avatar puedes cargarla de forma manual, o aún mejor que lo haga automáticamente GitLab a través de [Gravatar](#).



The screenshot shows the GitLab web interface with the 'Profile' tab selected. The page title is 'Profile settings'. Below the title, it says 'This information appears on your profile.' The form contains several input fields: 'Name' (filled with 'Administrator'), 'Email' (filled with 'drobert@aconcaguasf.com'), 'Skype', 'Linkedin', 'Twitter', 'Website', and 'Bio'. To the right of the form, there is an avatar section. It shows a red square avatar with a white terminal symbol. Text next to it says: 'You can change your avatar here or remove the current avatar to revert to gravatar.com'. Below this text are two buttons: 'Choose File ...' and 'File name...'. A note below these buttons states: 'The maximum file size allowed is 200KB.' At the bottom of the avatar section is a red button labeled 'Remove avatar'.

2.5.2. Account

El token que visualizas a continuación es utilizado por aplicaciones de terceros que interactúan con GitLab y requieren autenticación. Puedes resetearlo si lo deseas.

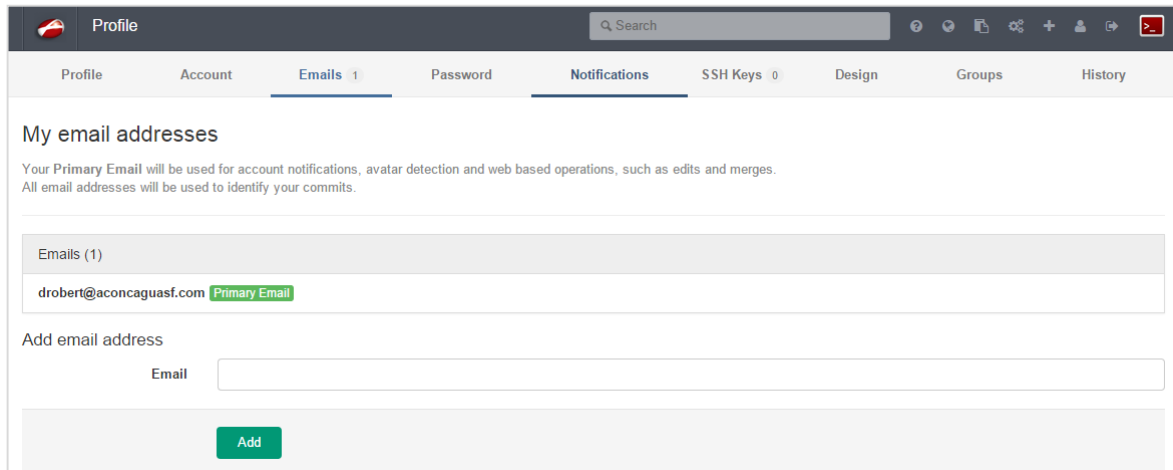


The screenshot shows the GitLab web interface with the 'Account' tab selected. The page title is 'Account settings'. Below the title, it says 'You can change your username and private token here.' The form contains a section for 'Private token'. It says: 'Your private token is used to access application resources without authentication. It can be used for atom feeds or the API. Keep it secret!'. Below this text is a text input field containing the token 'qweRTYgKF448XMuFTw1L'. At the bottom of the section is a blue button labeled 'Reset'.

2.5.3. Emails

Tu email primario se utilizará para las notificaciones de la cuenta, detección de avatar y operaciones basadas en web, tales como ediciones y merges.

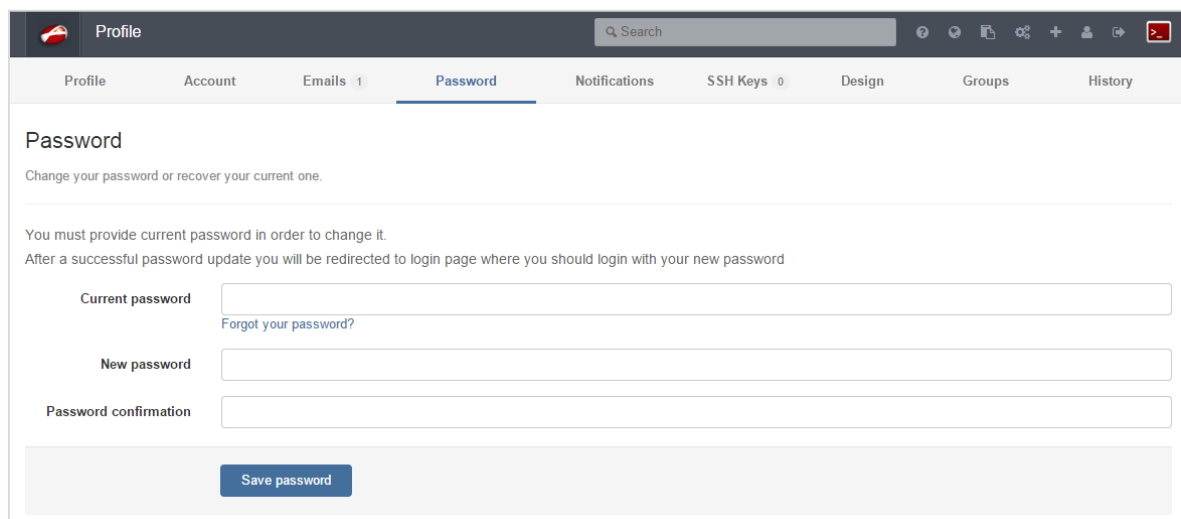
Todas las direcciones de correo electrónico se utilizarán para identificar sus commits.



The screenshot shows the GitLab Profile page with the 'Emails' tab selected. The page title is 'My email addresses'. Below the title, there is a note: 'Your Primary Email will be used for account notifications, avatar detection and web based operations, such as edits and merges. All email addresses will be used to identify your commits.' A table lists the email addresses, showing 'drobert@aconcaguasf.com' as the 'Primary Email'. Below the table, there is a form to 'Add email address' with an 'Email' input field and an 'Add' button.

2.5.4. Password

La siguiente sección te permitirá cambiar tu password. Un punto importante es que si en tu organización están utilizando LDAP como sistema de autenticación, tu contraseña de dominio es tu contraseña en GitLab. Esto ocasiona que si cambias el password de GitLab no se verá reflejado en el servidor de LDAP, perdiendo la sincronidad entre ambas herramientas. Si deseas cambiar la contraseña, el proceso correcto es **indicarle al Administrador para que él pueda ayudarte**.



The screenshot shows the GitLab Profile page with the 'Password' tab selected. The page title is 'Password'. Below the title, there is a note: 'Change your password or recover your current one.' A sub-note states: 'You must provide current password in order to change it. After a successful password update you will be redirected to login page where you should login with your new password'. The form contains three input fields: 'Current password', 'New password', and 'Password confirmation'. There is a link 'Forgot your password?' next to the 'Current password' field. At the bottom, there is a 'Save password' button.

2.5.5. Notifications

La configuración de notificaciones podrás utilizarla para elegir el nivel de notificaciones de forma general, por grupo o por proyecto.

The screenshot shows the 'Notifications settings' page in GitLab. At the top, there's a navigation bar with tabs: Profile, Account, Emails (1), Password, Notifications (selected), SSH Keys (0), Design, Groups, and History. Below the navigation bar, the page title is 'Notifications settings'. A sub-header states: 'GitLab uses the email specified in your profile for notifications'. The main content area has a section for 'Notification level' with three radio buttons: 'Disabled' (selected), 'Participating', and 'Watch'. Each option has a description: 'Disabled' means 'You will not get any notifications via email'; 'Participating' means 'You will only receive notifications from related resources (e.g. from your commits or assigned issues)'; 'Watch' means 'You will receive all notifications from projects in which you participate'. Below this, there are two sections: 'Groups' and 'Projects'. Each section has a list of items with a 'global' dropdown menu. The 'Groups' list includes PAE, SUD, HAE, Golf Android, and Golf IOS. The 'Projects' list includes Administrator / testing, HAE / TF-SGITX, HAE / TF-SGITO, HAE / TF-WGUAR, and HAE / TF-AENLA. A note at the bottom left says: 'You can also specify notification level per group or per project. By default all projects and groups uses notification level set above.'

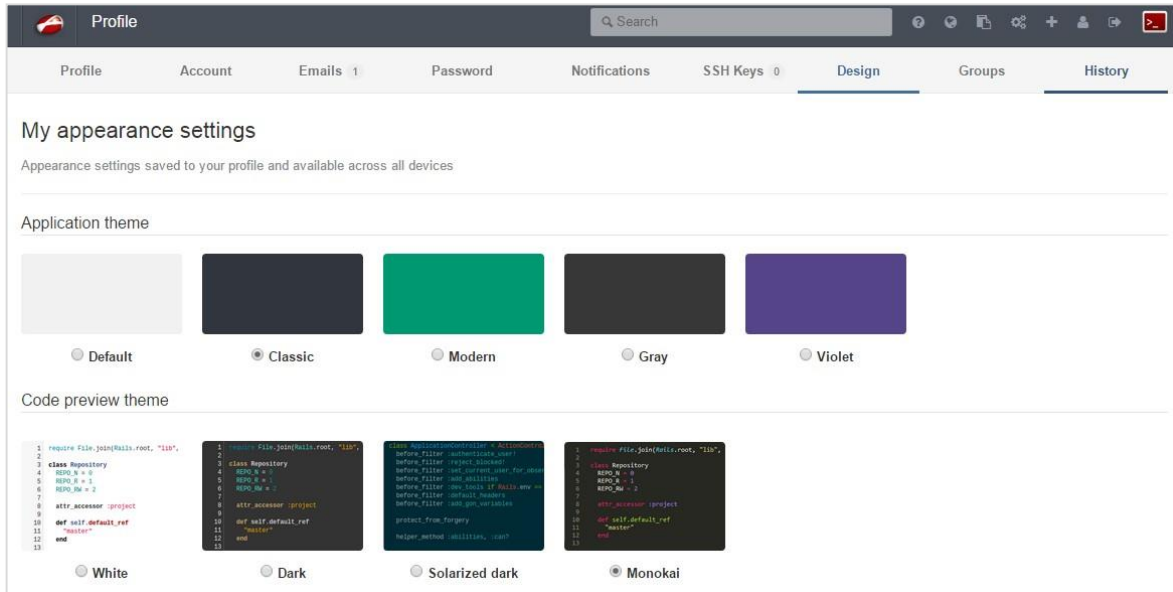
2.5.6. SSH Keys

Las claves SSH te permiten establecer una conexión segura entre tu computadora y GitLab. De esta forma no es necesario que utilices tu nombre de usuario y contraseña.

The screenshot shows the 'SSH Keys' page in GitLab. At the top, there's a navigation bar with tabs: Profile, Account, Emails (1), Password, Notifications, SSH Keys (0) (selected), Design, Groups, and History. Below the navigation bar, the page title is 'My SSH keys'. There's a green button labeled 'Add SSH Key'. A sub-header states: 'SSH keys allow you to establish a secure connection between your computer and GitLab. Before you can add an SSH key you need to generate it'. Below this, there's a section titled 'SSH Keys (0)' with a message: 'There are no SSH keys with access to your account.'

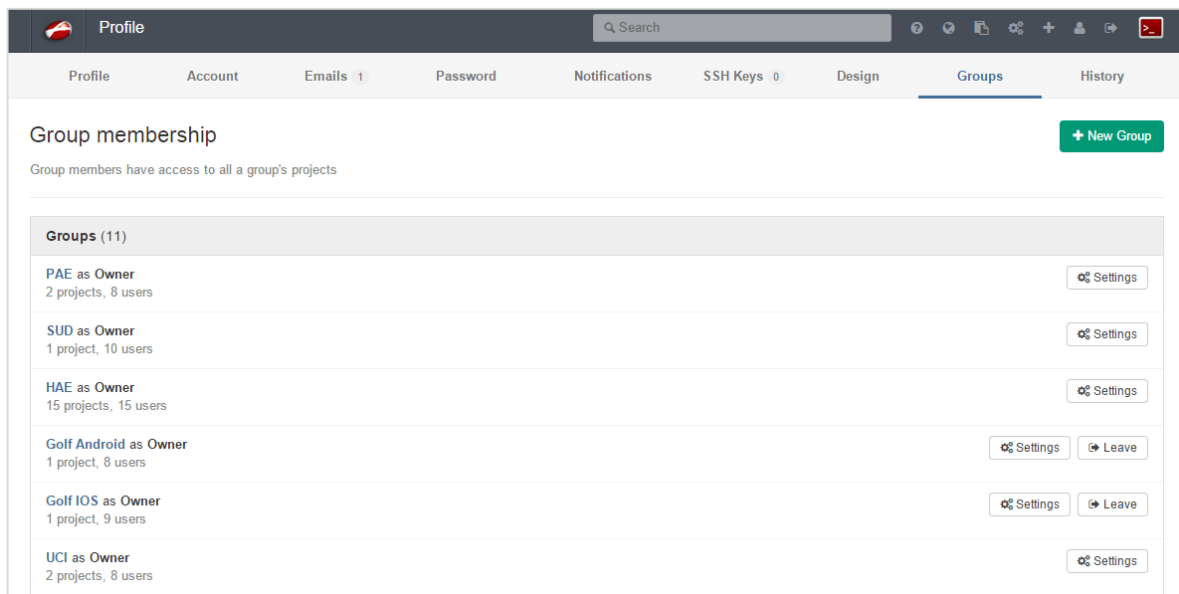
2.5.7. Design

La configuración de la apariencia tiene dos partes: Interfaz de GitLab y Vista previa de código fuente.



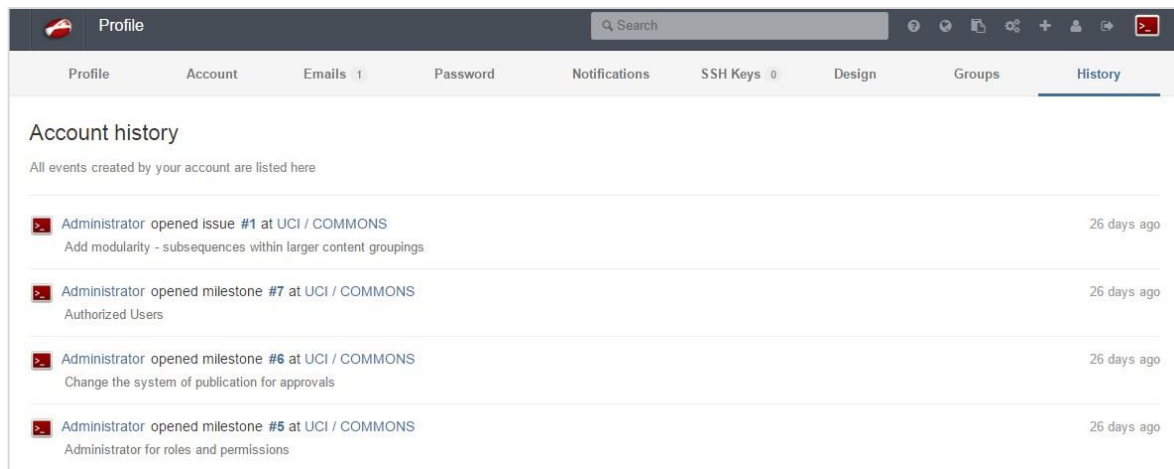
2.5.8. Groups

Desde esta sección podrás ver el listado de grupos a los que perteneces con su correspondiente rol. También podrás crear un nuevo grupo o editarlos en caso de tener los privilegios suficientes.



2.5.9. History

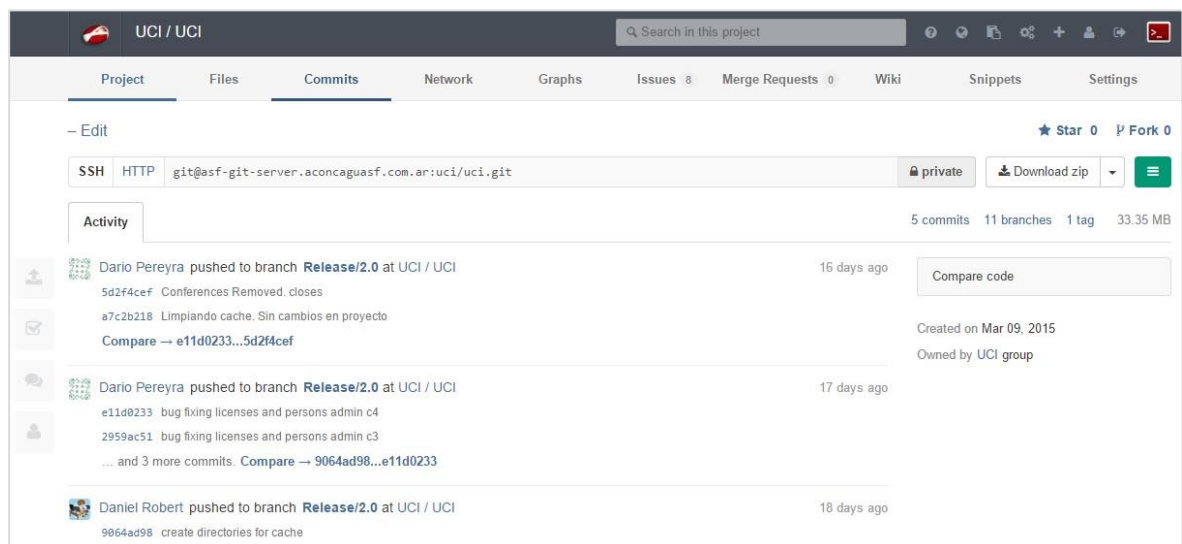
Muestra el detalle de toda tu actividad dentro de GitLab.



2.6. Project Details

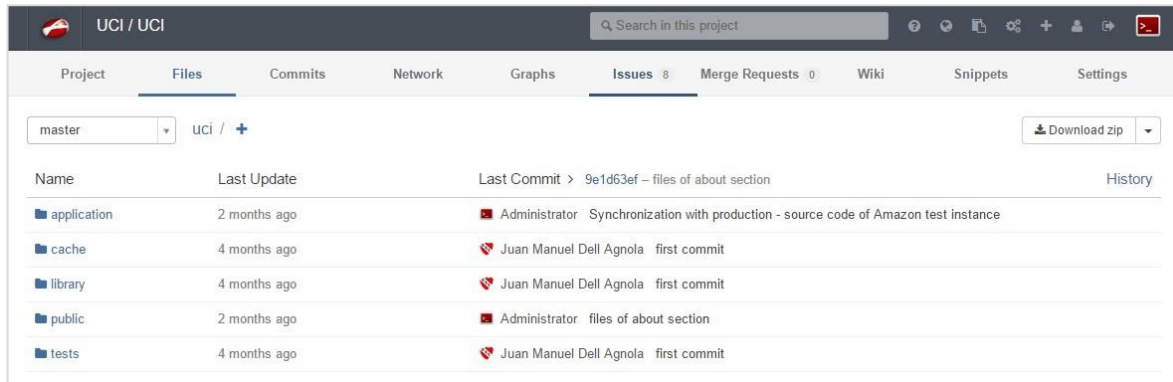
2.6.1. Project

Esta sección muestra el detalle de un proyecto. Desde aquí podrás ver la actividad (y filtrarla), la cantidad de commits, de branches y tags. Además tienes un link para editar el proyecto y las urls para clonarlo, entre otras cosas.



2.6.2. Files

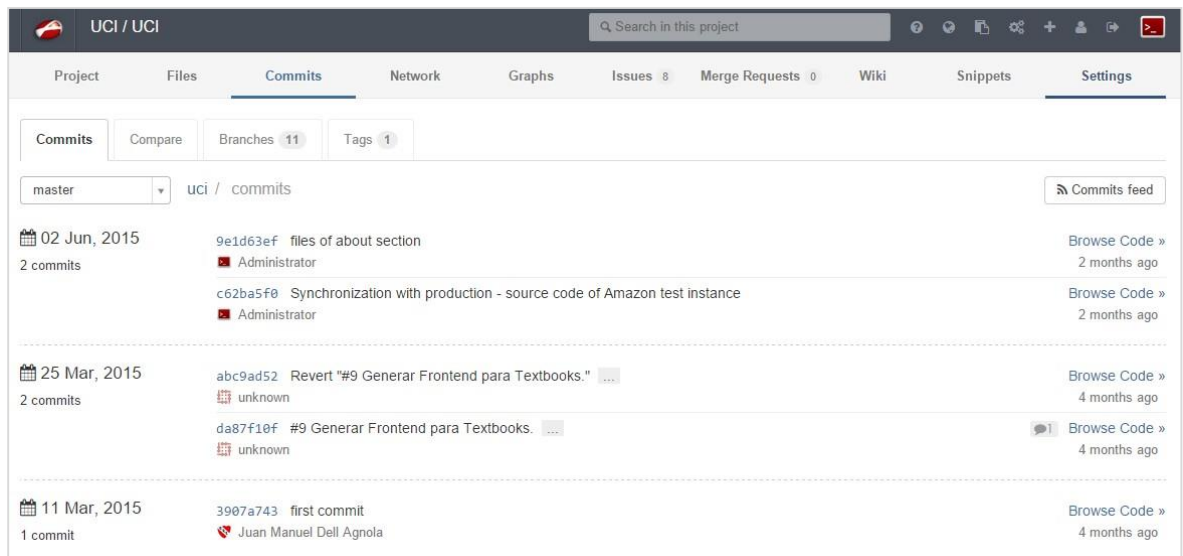
Aquí podrás visualizar los archivos del proyecto, filtrarlos por branch y ver el último commit relacionado a cada archivo. Además podrás descargar el branch completo en formato zip.



2.6.3. Commits

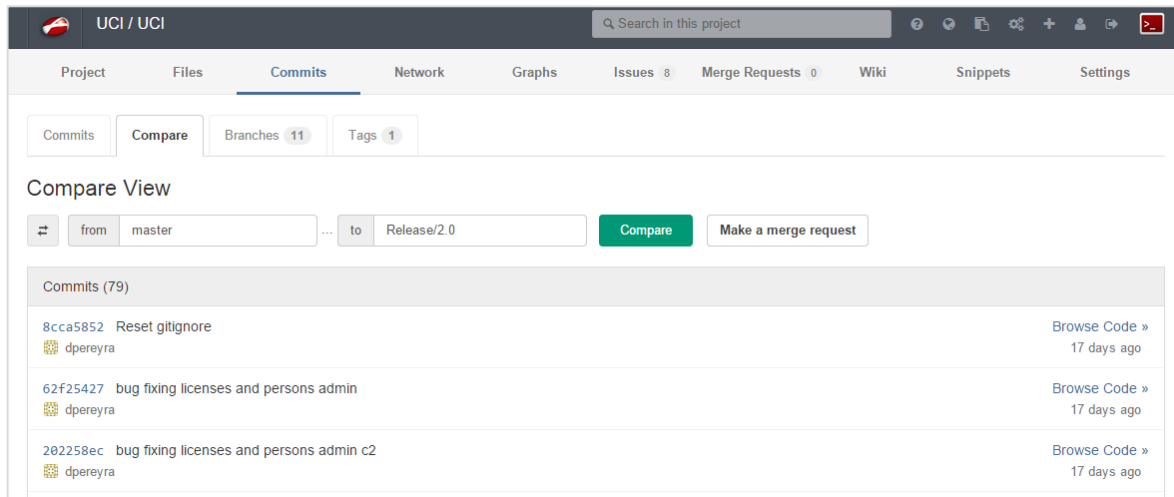
2.6.3.1. Commits

Esta sección es muy importante porque desde aquí podrás visualizar y administrar los commits, acceder al código de ese commit y filtrar por branch.



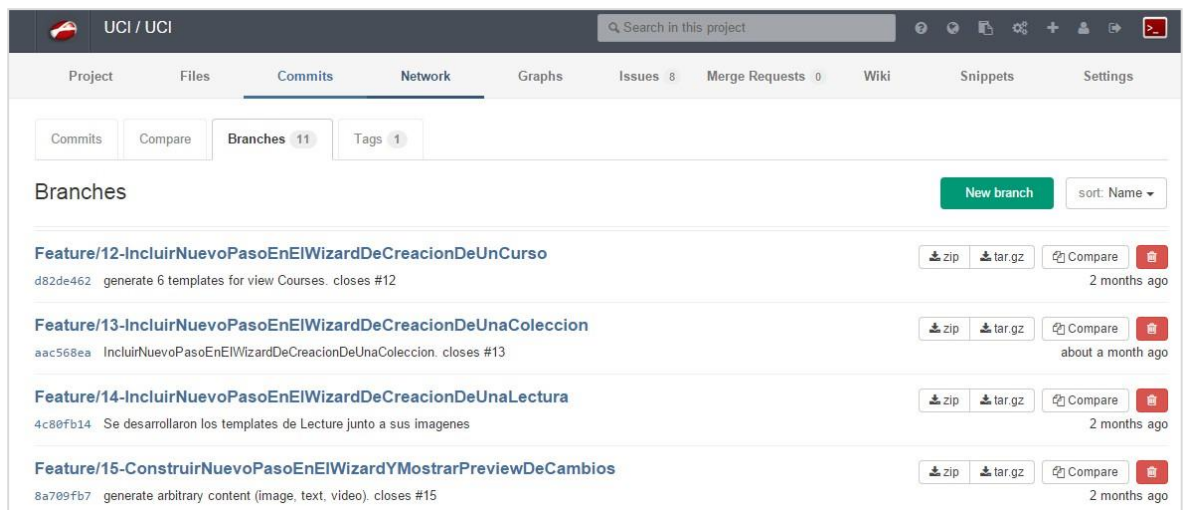
2.6.3.2. Compare

La funcionalidad de comparación de branches es muy útil al momento de hacer un deploy. Además desde aquí puedes hacer un Merge Request, cuyo significado veremos más adelante.



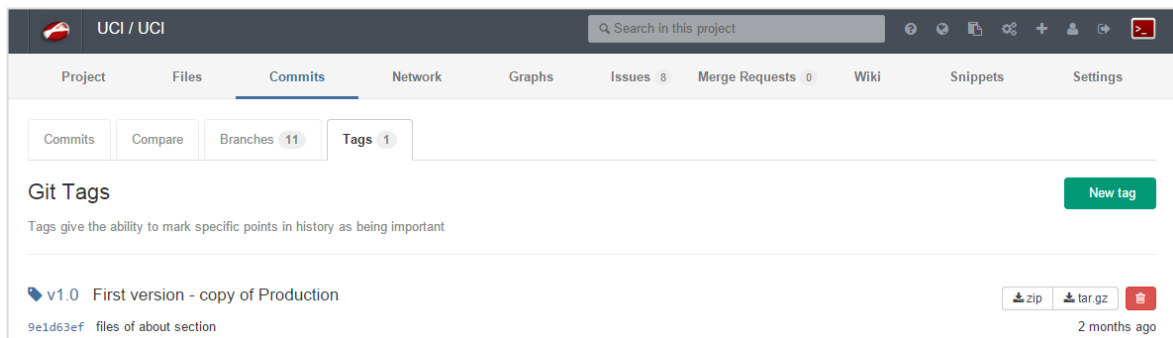
2.6.3.3. Branches

Esta pantalla te permite visualizar el listado de branches del proyecto. Además puedes descargar los archivos de cada branch en diferentes formatos como así también crear uno nuevo o eliminar uno existente.



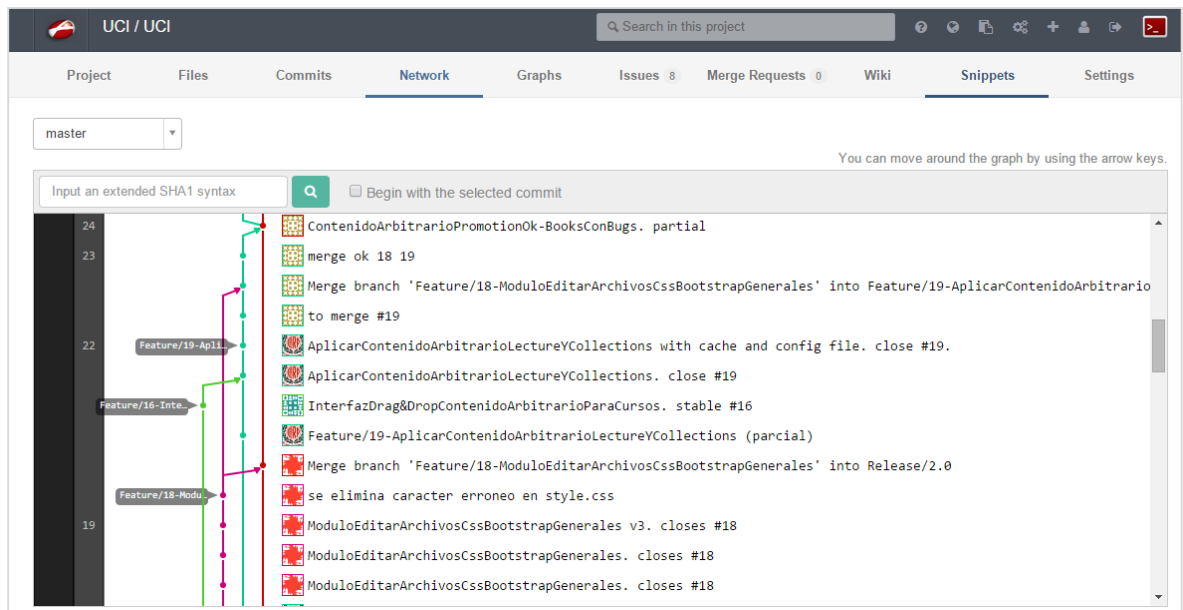
2.6.3.4. Tags

El listado de tags corresponde a las diferentes versiones del sistema, cada tag es un deploy.



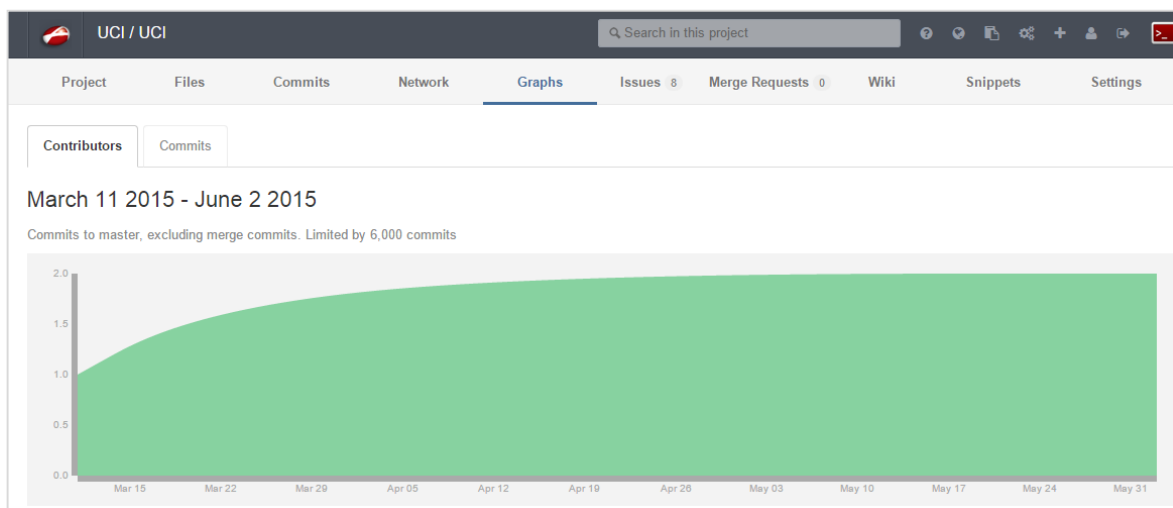
2.6.4. Network

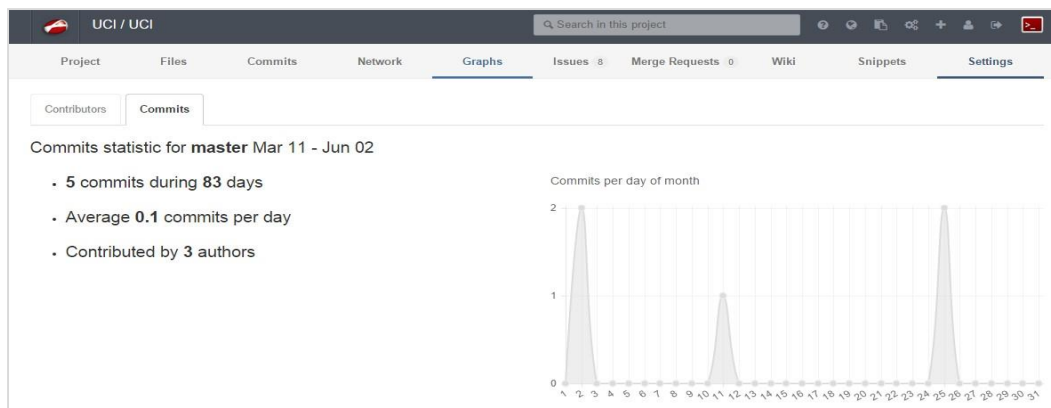
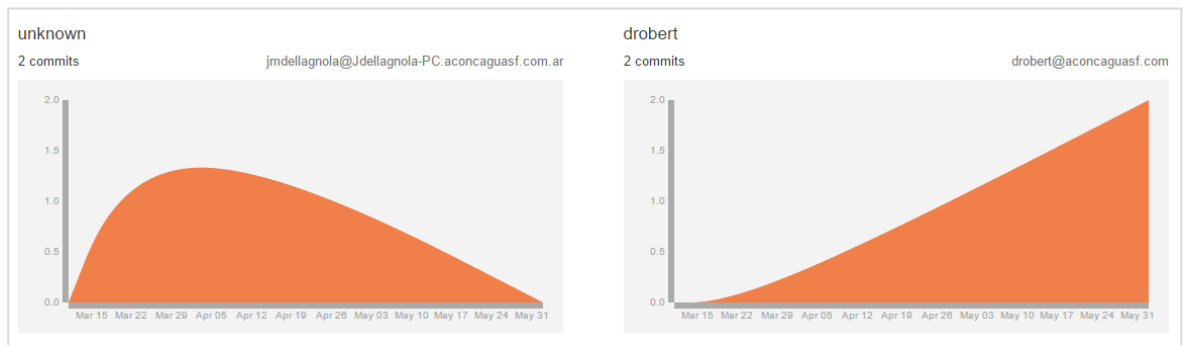
El árbol de branches permite visualizar la interacción entre los branches.



2.6.5. Graphs

GitLab arroja gráficos de forma que el administrador pueda sacar estadísticas de la participación de cada integrante del proyecto. A continuación se muestran algunos de los gráficos mencionados:





2.6.6. Issues

2.6.6.1. Issues

Esta sección muestra el listado de issues, la persona asignada para realizarlo, el autor y el milestone relacionado. Puede filtrarse por diferentes parámetros como por ejemplo por la asignación, el estado, la descripción y los labels, entre otros. Además desde aquí podrás crear un nuevo issue.

UCI / UCI

Search in this project

Project Files Commits Network Graphs **Issues 8** Merge Requests 0 Wiki Snippets Settings

Issues Merge Requests Milestones Labels

Filter by title or description + New Issue

Everyone's 8

Assigned to me 0

Created by me 0

State

Open Closed All

Labels

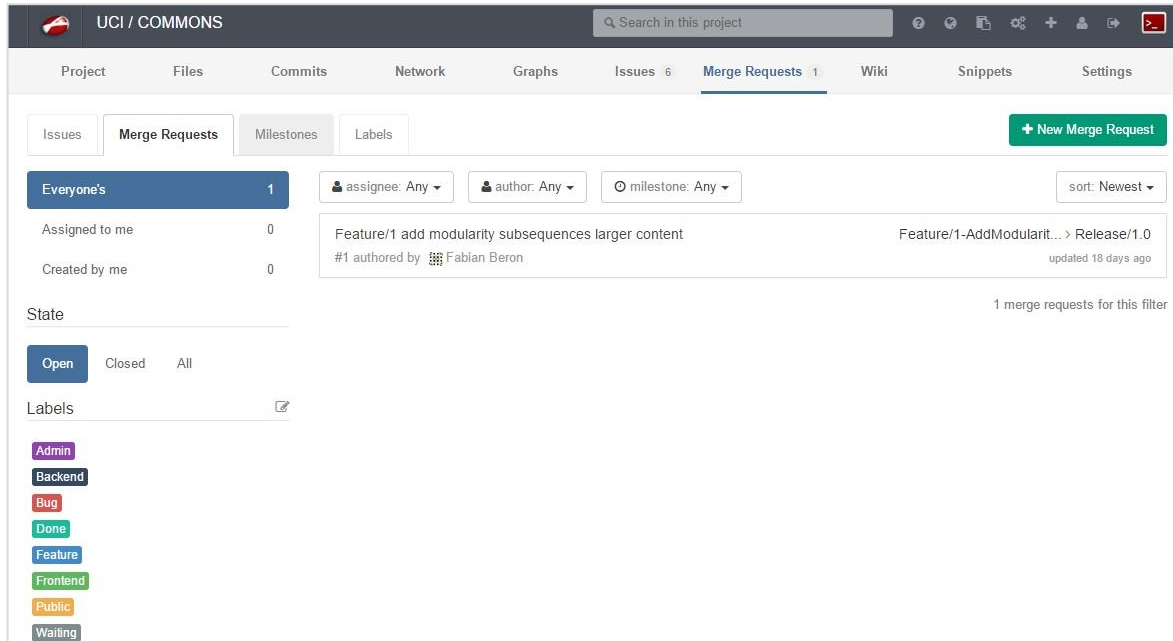
Admin Backend Bug Done Feature Frontend Public Waiting

assignee: Any author: Any milestone: Any sort: Newest

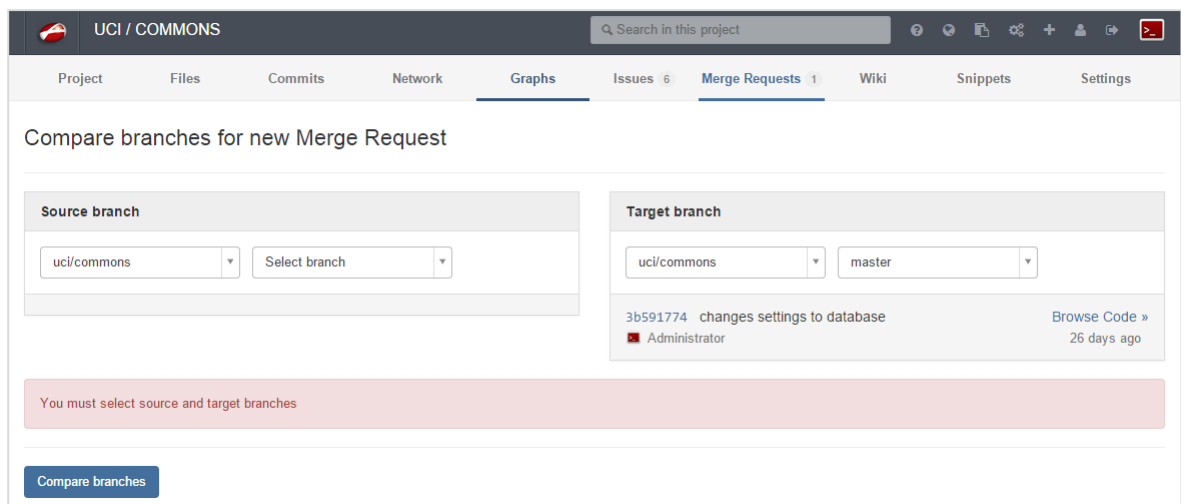
- Resolver bugs en la edición de licencias y en la edición de personas
#20 assigned to Dario Pereyra Bug Backend Done Admin updated 18 days ago
- Aplicar contenido arbitrario lecture y collections
#19 assigned to Lucas Grando 9 Develop interface drag and drop for each template and type of content (collection, course and lecture) Feature Backend Working Done Admin updated about a month ago
- Módulo para editar los archivos css bootstrap y generales
#18 assigned to Fabian Beron 5 Module to Edit general and bootstrap styles Feature Backend Done Admin updated about a month ago
- Implementar cambio de imágenes preview en la configuración de los templates
#17 assigned to Fabian Beron 2 View for change template and preview Feature Backend Done Admin updated about a month ago
- Desarrollar interfaz drag & drop del contenido arbitrario para cursos
#16 assigned to Dario Pereyra 7 Develop interface drag and drop for each template and type of content (collection, course and lecture) Feature Backend Done Admin updated 2 months ago
- Construir nuevo paso en el wizard y mostrar preview de cambios

2.6.6.2. Merge Request

Un Merge Request se define como una “Petición de fusión” entre dos branches. Al igual que con los issues puedes ver el listado como así también filtrarlo.

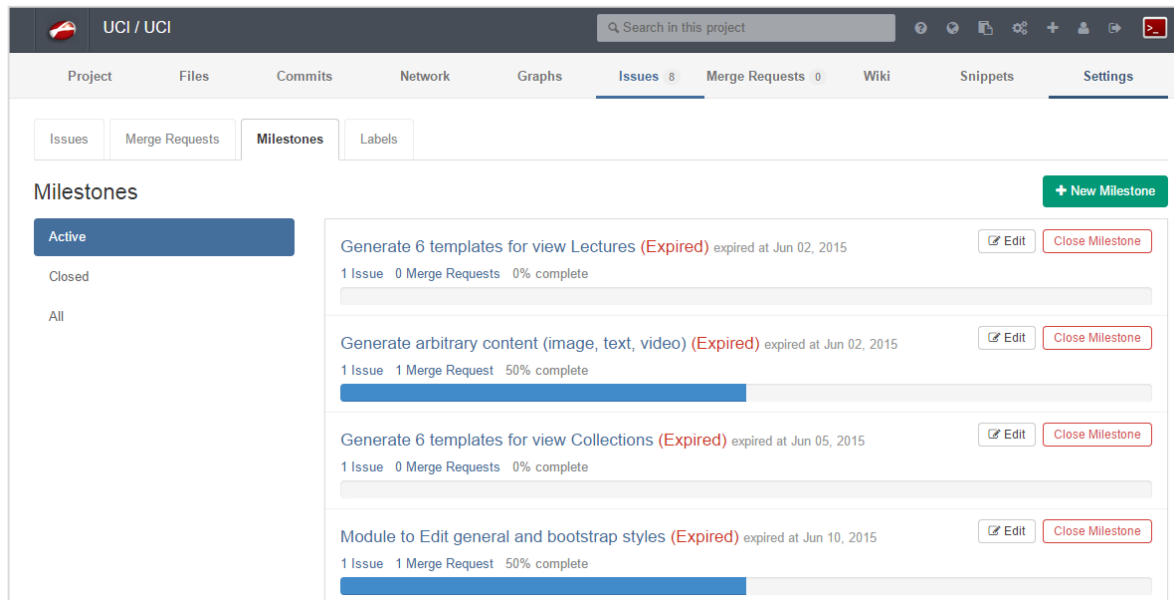


Además puedes crear un nuevo Merge Request a partir de la comparación de dos branches, seleccionando el branch fuente (por lo general el de tu issue) y el branch target (por lo general el de desarrollo o release)



2.6.6.3. Milestones

Un milestone se traduce al español como “hito”, un evento significativo que ocurre durante el proyecto, que generalmente coincide con la terminación de un entregable principal del proyecto. En el listado de milestones se puede filtrar por el estado.



Además se puede crear un nuevo milestones cargando los siguientes datos:

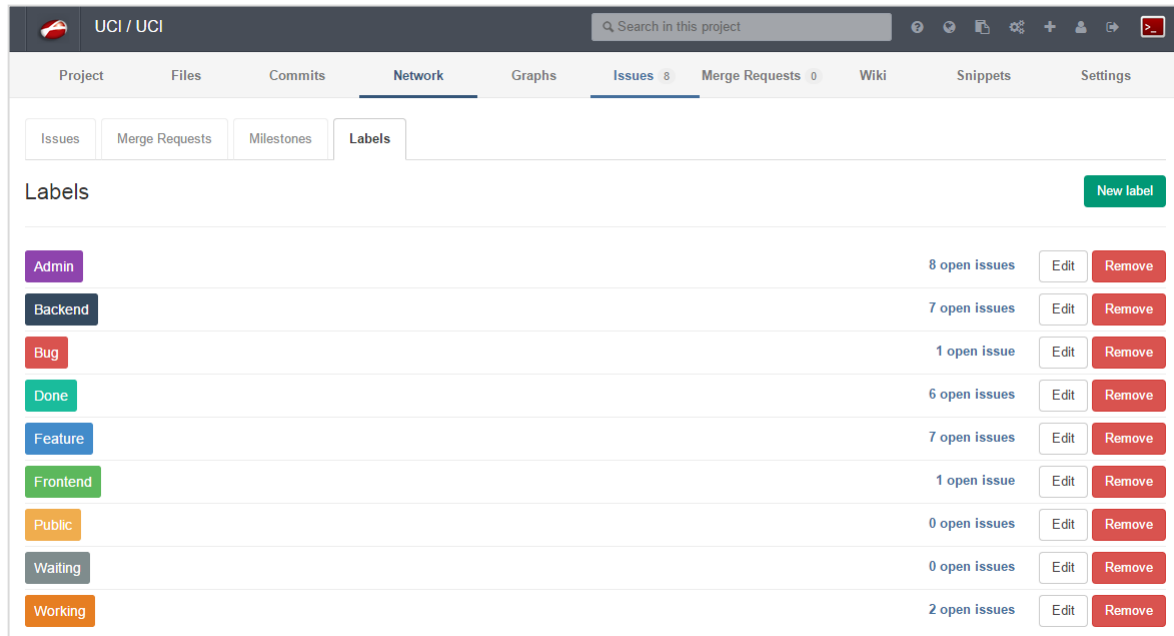
The screenshot shows the 'New Milestone' form. It includes the following fields and components:

- Title:** A text input field with a 'Required' label.
- Due Date:** A date picker showing a calendar for July 2015. The date 19 is selected.
- Description:** A text area with 'Write' and 'Preview' tabs. There is an 'EDIT IN FULLSCREEN' link.
- Footer:** A 'Create milestone' button and a 'Cancel' button.

Below the description field, there is a note: 'Milestones are parsed with GitLab Flavored Markdown. Attach images (JPG, PNG, GIF) by dragging & dropping or selecting them.'

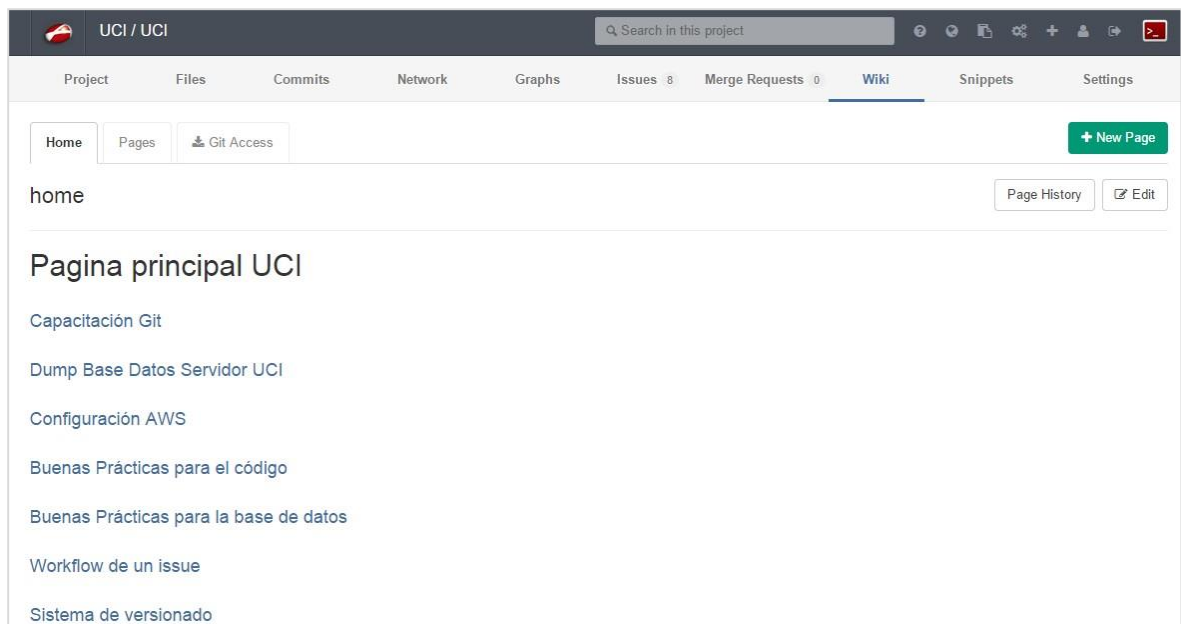
2.6.6.4. Labels

Los labels sirven para llevar una clasificación de los issues. Se pueden crear y asociar a los issues todos los labels que necesites. Además puedes asociarle a cada label el color que desees.



2.6.7. Wiki

La wiki del proyecto puede ser utilizada para documentar procesos, metodologías, accesos, y todo lo que necesites compartir con el resto del equipo.



La creación de páginas utiliza Markdown. Para más información visita el sitio web de [ayuda de markdown](#).

2.6.8. Project Settings

2.6.8.1. Project

Desde esta sección puedes cambiar el nombre del proyecto, el branch por default, el nivel de visibilidad, las características, entre otras cosas.

Project settings:

Some settings, such as "Transfer Project", are hidden inside the danger area below.

Project name: UCI

Project description (optional): Awesome project

Default Branch: master

Visibility Level (?):

- ☒ Private: Project access must be granted explicitly for each user.
- ☐ Internal: The project can be cloned by any logged in user.
- ☐ Public: The project can be cloned without any authentication.

2.6.8.2. Members

Desde esta pantalla podrás administrar los miembros del proyecto.

Users with access to this project

Read more about project permissions [here](#)

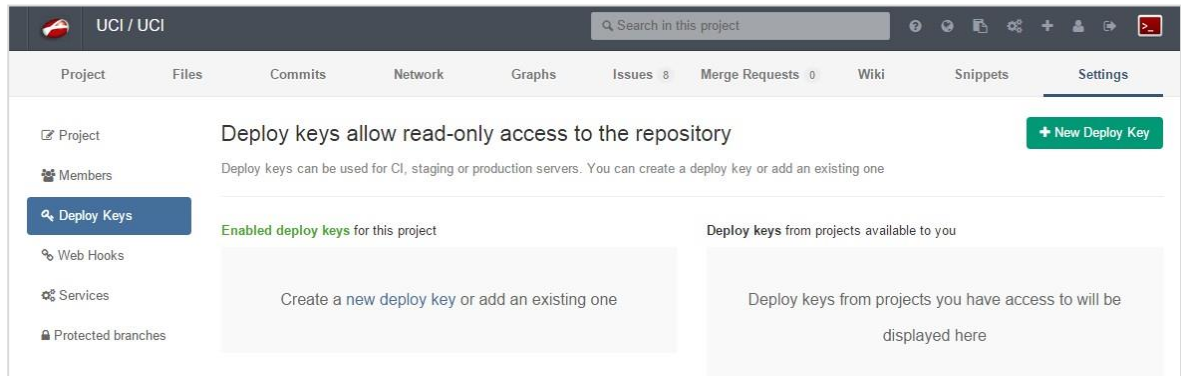
UCI project members (0)

UCI group members (8)

| | |
|--|-----------|
| Administrator root It's you | Owner |
| Daniel Robert drobert | Master |
| Ana Laura Martina amartina | Master |
| Dario Pereyra dpereyra | Developer |
| Fabian Beron fberon | Developer |
| Lucas Grando lgrando | Developer |
| Juan Manuel Dell Agnola jmdellagnola | Developer |
| Nelson Arabel narabel | Developer |

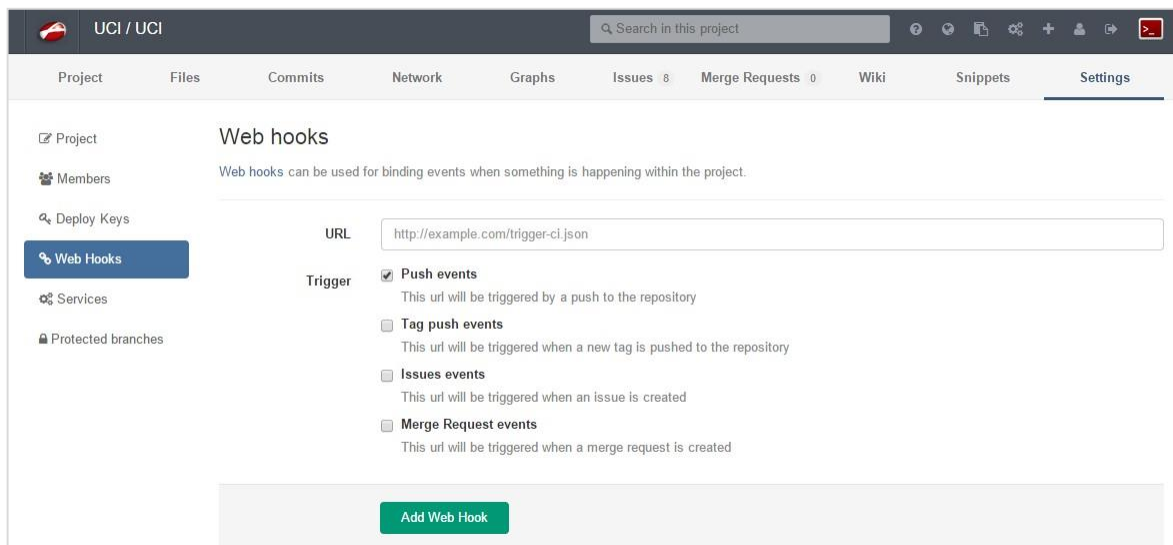
2.6.8.3. Deploy Keys

Las deploy keys son utilizadas para Integración Continua (CI). Si deseas saber más sobre CI, puede visitar el [sitio de referencia](#).



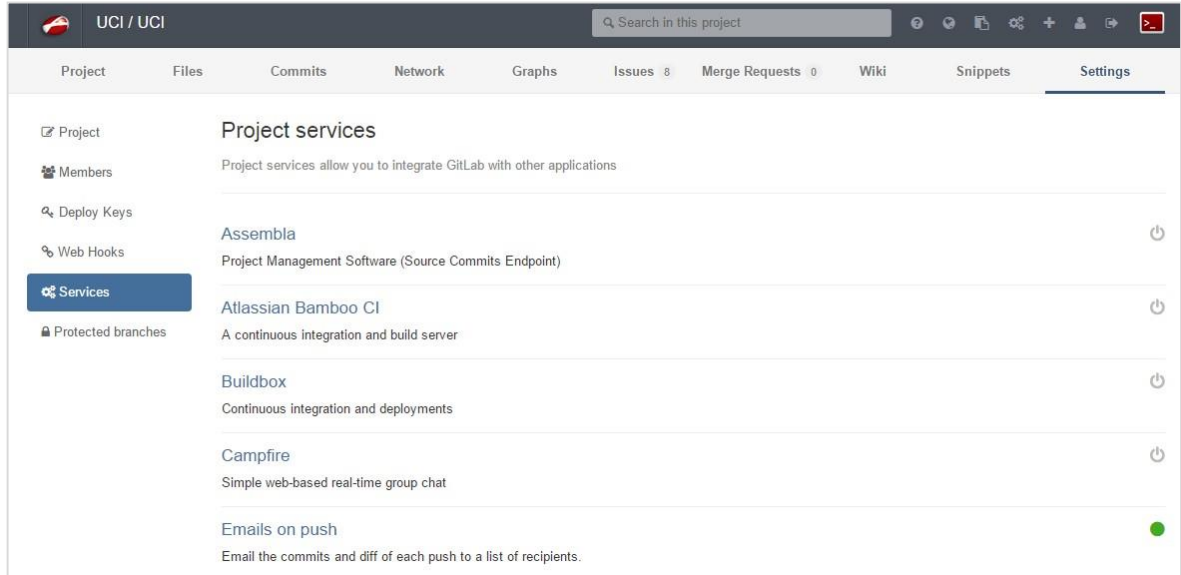
2.6.8.4. Web Hooks

Los Web Hooks son utilizados para disparar eventos en ciertos momentos que ocurren dentro del proyecto. Simplemente debes completar la url a la que se debe llamar y el evento asociado.



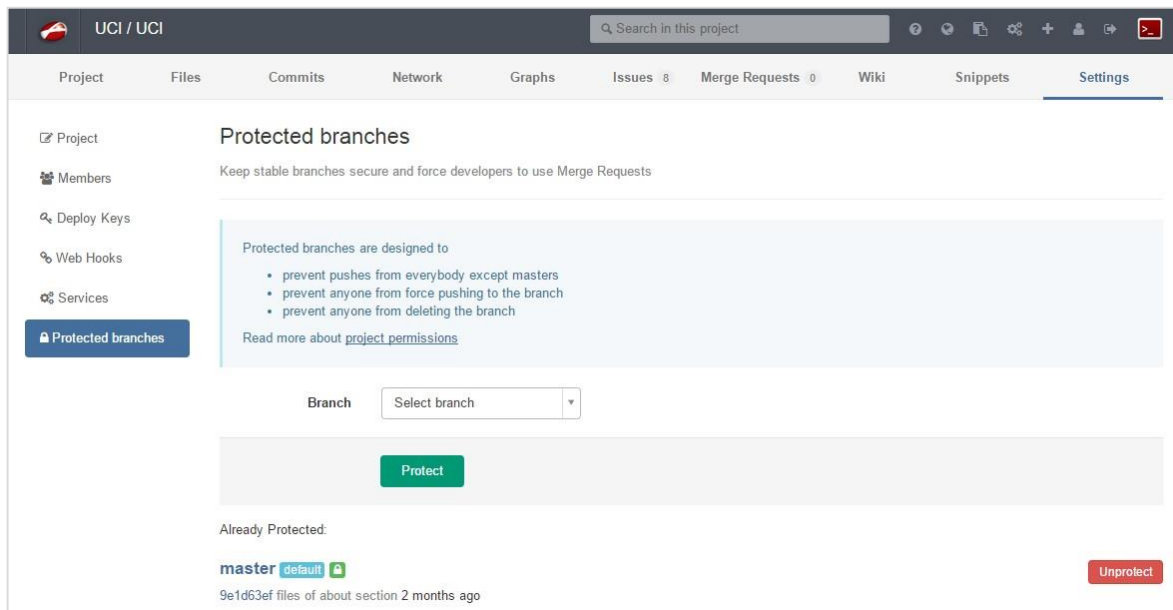
2.6.8.5. Services

Los servicios son herramientas que pueden integrarse a GitLab. Cada herramienta tiene una forma diferente de integrarse, algunas simplemente solicitan una url y otras tokens o SSH Keys. Algunas de las herramientas que pueden integrarse son las siguientes:



2.6.8.6. Protected branches

Desde la siguiente pantalla puedes “proteger” los branches para que nadie pueda modificarlos a través de pushes. Esta funcionalidad es útil para los branches de deploys ya que un branch en producción nunca debe ser modificado.



3. Git

3.1. Sistema de versionado

En el mundo de la gestión de software existe el temor de caer en algún momento en el llamado “infierno de las dependencias”. Mientras más grande crece tu sistema y mientras más paquetes integras en tu software, más probable es que te encuentres, un día, en este pozo de la desesperación.

En sistemas con muchas dependencias, publicar nuevas versiones de un paquete puede transformarse rápidamente en una pesadilla. Si las especificaciones de dependencia son muy estrictas está el peligro del bloqueo de versiones (la imposibilidad de actualizar un paquete sin tener que actualizar todos los que dependen de este). Si las dependencias se especifican de manera muy amplia, inevitablemente caerás en promiscuidad de versiones (asumir más compatibilidad con versiones futuras de lo razonable). El infierno de las dependencias es donde te encuentras cuando el bloqueo de versiones o la promiscuidad de versiones te impiden avanzar en tu proyecto de manera fácil y segura.

Como solución a este problema, propongo un set simple de reglas y requerimientos que dictan cómo asignar y cómo aumentar los números de versión. Para que este sistema funcione, tienes que declarar primero un API pública. Esto puede consistir en documentación o ser explicitado en el código mismo. De cualquier forma, es importante que esta API sea clara y precisa. Una vez que identificaste tu API pública, comunicas cambios a ella con aumentos específicos al número de versión. Considera un formato de versión del tipo X.Y.Z (Major.Minor.Patch). Los arreglos de bugs que no cambian el API incrementan el patch, los cambios y adiciones que no rompen la compatibilidad de las dependencias anteriores incrementan el menor, y los cambios que rompen la compatibilidad incrementan el mayor.

Yo llamo a este sistema “Versionamiento Semántico”. Bajo este esquema, los números de versión y la forma en que cambian entregan significado del código que está detrás y lo que fue modificado de una versión a otra.

Especificación de Versionamiento Semántico (en inglés SemVer)

En el documento original se usaba el RFC 2119 para el uso de las palabras MUST, MUST NOT, SHOULD, SHOULD NOT y MAY. Para que la traducción sea lo más fiel posible, he traducido siempre MUST por el verbo deber en presente (DEBE, DEBEN), SHOULD como el verbo deber en condicional (DEBERÍA, DEBERÍAN) y el verbo MAY como el verbo PODER.

- a) El software que use Versionamiento Semántico DEBE declarar una API pública. Esta API puede ser declarada en el código mismo o existir en documentación estricta. De cualquier manera, debería ser precisa y completa.
- b) Un número normal de versión DEBE tomar la forma X.Y.Z donde X, Y, y Z son enteros no negativos. X es la versión “mayor”, Y es la versión “minor”, y Z es la versión “patch”. Cada elemento DEBE incrementarse numéricamente en incrementos de 1. Por ejemplo: 1.9.0 -> 1.10.0 -> 1.11.0.
- c) Una vez que un paquete versionado ha sido liberado (release), los contenidos de esa versión NO DEBEN ser modificadas. Cualquier modificación DEBE ser liberada como una nueva versión.
- d) La versión mayor en cero (0.y.z) es para desarrollo inicial.
- e) La versión 1.0.0 define el API pública. La forma en que el número de versión

esbincrementado después de este release depende de esta API pública y de cómo esta cambia.

- f) La versión patch Z ($x.y.Z \mid x > 0$) DEBE incrementarse cuando se introducen solo arreglos compatibles con la versión anterior. Un arreglo de bug se define como un cambio interno que corrige un comportamiento erróneo.
- g) La versión minor Y ($x.Y.z \mid x > 0$) DEBE ser incrementada si se introduce nueva funcionalidad compatible con la versión anterior. Se DEBE incrementar si cualquier funcionalidad de la API es marcada como deprecada. PUEDE ser incrementada si se agrega funcionalidad o arreglos considerables al código privado. Puede incluir cambios de nivel patch. La versión patch DEBE ser reseteada a 0 cuando la versión minor es incrementada.
- h) La versión major X ($X.y.z \mid X > 0$) DEBE ser incrementada si cualquier cambio no compatible con la versión anterior es introducida a la API pública. PUEDE incluir cambios de nivel minor y/o patch. Las versiones patch y minor DEBEN ser reseteadas a 0 cuando se incrementa la versión major.
- i) Una versión pre-release PUEDE ser representada por adjuntar un guión y una serie de identificadores separados por puntos inmediatamente después de la versión patch. Los identificadores DEBEN consistir solo de caracteres ASCII alfanuméricos y el guión [0-9A-Za-z-]. Las versiones pre-release tienen una menor precedencia que la versión normal asociada. Ejemplos: 1.0.0-alpha, 1.0.0-beta.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.
- j) La metadata de build PUEDE ser representada adjuntando un signo más y una serie de identificadores separados por puntos inmediatamente después de la versión patch o la pre-release. Los identificadores DEBEN consistir sólo de caracteres ASCII alfanuméricos y el guión [0-9A-Za-z-]. Los metadatos de build DEBIERAN ser ignorados cuando se intenta determinar precedencia de versiones. Luego, dos paquetes con la misma versión, pero distinta metadata de build se consideran la misma versión. Ejemplos: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85.
- k) La precedencia DEBE ser calculada separando la versión en major, minor, patch e identificadores pre-release en ese orden (La metadata de build no figuran en la precedencia). Las versiones major, minor, y patch son siempre comparadas numéricamente. La precedencia de pre-release DEBE ser determinada comparando cada identificador separado por puntos de la siguiente manera: los identificadores que solo consisten de números son comparados numéricamente y los identificadores con letras o guiones son comparados de acuerdo al orden establecido por ASCII. Los identificadores numéricos siempre tienen una precedencia menor que los no-numéricos. Ejemplo: 1.0.0-alpha

¿Por qué usar Versionamiento Semántico?

Esta no es una idea nueva o revolucionaria. De hecho, probablemente ya haces algo cercano hoy en día. El problema es que “cercano” no es suficientemente bueno. Sin un acuerdo en algún tipo de especificación formal, los números de versiones son esencialmente inútiles para el manejo de dependencias. Al darle un nombre y una definición clara a las ideas expresadas arriba, se hace fácil comunicar tus intenciones a los usuarios de tu software. Una vez que estas intenciones son claras y flexibles (pero no demasiado flexibles) finalmente se pueden hacer especificaciones de dependencias.

Un ejemplo simple puede demostrar como el Versionamiento Semántico puede ayudar a que el infierno de dependencias quede en el pasado. Considera una librería llamada "TransaccionesBancarias" y requiere de un paquete de tercero Semanticamente Versionado llamado "Seguridad". En el momento en que se crea TransaccionesBancarias, Seguridad está en su versión 3.1.0. Como TransaccionesBancarias usa algunas de las funcionalidades que recién se estrenaron en la versión 3.1.0, puedes tranquilamente definir la dependencia de Seguridad como mayor o igual a 3.1.0, pero menor a 4.0.0. Ahora, cuando la versión 3.1.1 y

3.2.0 de Seguridad sean liberadas, puedes usarlas en tu sistema de versionamiento de paquetes sabiendo que serán compatibles con el software dependiente.

Como desarrollador responsable que eres, claro, querrás verificar que cualquier actualización de paquete funcione como se anunció. El mundo real es complejo; no hay nada que podamos hacer salvo estar atentos. Lo que puedes hacer es dejar que el Versionamiento Semántico te provea de una manera sana de liberar y actualizar paquetes sin tener que entregar nuevas versiones de tus paquetes dependientes, ahorrándote tiempo y molestias.

Si todo esto suena deseable, todo lo que tienes que hacer para comenzar usando Versionamiento Semántico es declarar que lo estás haciendo y seguir las reglas.

FAQ

¿Cómo tengo que hacer con las revisiones en la etapa inicial de desarrollo 0.y.z ?

Lo más fácil es empezar tu desarrollo inicial en 0.1.0 e incrementar la versión minor en cada release.

¿Cómo sé cuándo liberar la versión 1?0.0?

Si tu software está siendo usado en producción, probablemente ya deberías estar en 1.0.0. Si tienes una API estable de la cual tus usuarios ya están dependiendo, deberías estar en 1.0.0. Si te preocupa mucho la compatibilidad con versiones anteriores, ya deberías estar en 1.0.0.

¿Esto no desincentiva el desarrollo rápido y las iteraciones cortas?

La versión mayor en cero se trata de desarrollo rápido. Si estás cambiando la API todos los días debieras o bien estar todavía en la versión 0.y.z o estar trabajando en un branch separado para la próxima versión mayor.

Si incluso el cambio incompatible con la versión anterior más pequeño requiere un aumento de la versión mayor ¿No voy a terminar en la versión 42.0.0 demasiado rápido?

Este es un tema de desarrollo responsable y visión anticipada. Los cambios incompatibles no debieran ser introducidos con ligereza al software del cual depende mucho código. El costo de actualizar puede ser significativo. Tener que aumentar la versión mayor para publicar cambios incompatibles significa que vas a pensar bien el efecto de tus cambios, y evaluar el costo/beneficio asociado.

Documentar la API pública entera es demasiado trabajo!

Es tu responsabilidad como desarrollador profesional documentar adecuadamente el software pensado para que lo usen otros. Gestionar la complejidad del software es una parte tremendamente importante de mantener un proyecto eficiente, y es muy difícil de lograr si nadie sabe cómo usar tu software o qué métodos es seguro llamar. A largo plazo, el Versionamiento Semántico, y la insistencia en una API pública bien definida pueden asegurar que todos y todo corra de manera ordenada.

¿Qué hago si accidentalmente publico un cambio incompatible con la versión anterior como un cambio de versión minor?

Apenas te des cuenta de que rompiste con la especificación de Versionamiento Semántico, repara el problema y publica una nueva versión minor que corrige el problema y recupera la

compatibilidad. Incluso en esta circunstancia, es inaceptable modificar releases versionados. Si corresponde, documenta la versión que rompe la especificación e informa a tus usuarios que estén atentos a ella.

¿Qué debería hacer si actualizo mis propias dependencias sin cambiar la API pública?

Eso sería considerado compatible ya que no afecta al API pública. El software que explícitamente depende de las mismas dependencias que tu paquete debiera tener sus propias especificaciones de dependencia y el autor va a notar cualquier conflicto. La determinación de si el cambio es a nivel de patch o minor depende de si actualizaste tus dependencias para arreglar un bug o para agregar funcionalidad nueva. Yo esperaría que haya código adicional si se trata de lo segundo, en cuyo caso el incremento del minor es la opción obvia.

¿Qué debería hacer si el bug que estoy arreglando justamente hace que vuelva a estar de acuerdo con la especificación del API pública? (es decir, el código estaba incorrectamente desincronizado con la documentación del API)

Usa tu sentido común. Si tienes una gran audiencia que se va a ver drásticamente afectada al cambiar el comportamiento a lo que la API pública debía hacer, entonces lo mejor puede ser hacer un release mayor, incluso si el arreglo es estrictamente un release de tipo patch. Recuerda, el Versionamiento Semántico se trata de incorporar significado a la forma en que el número de versión cambia. Si estos cambios son importantes para tus usuarios, usa el número de versión para informarlos.

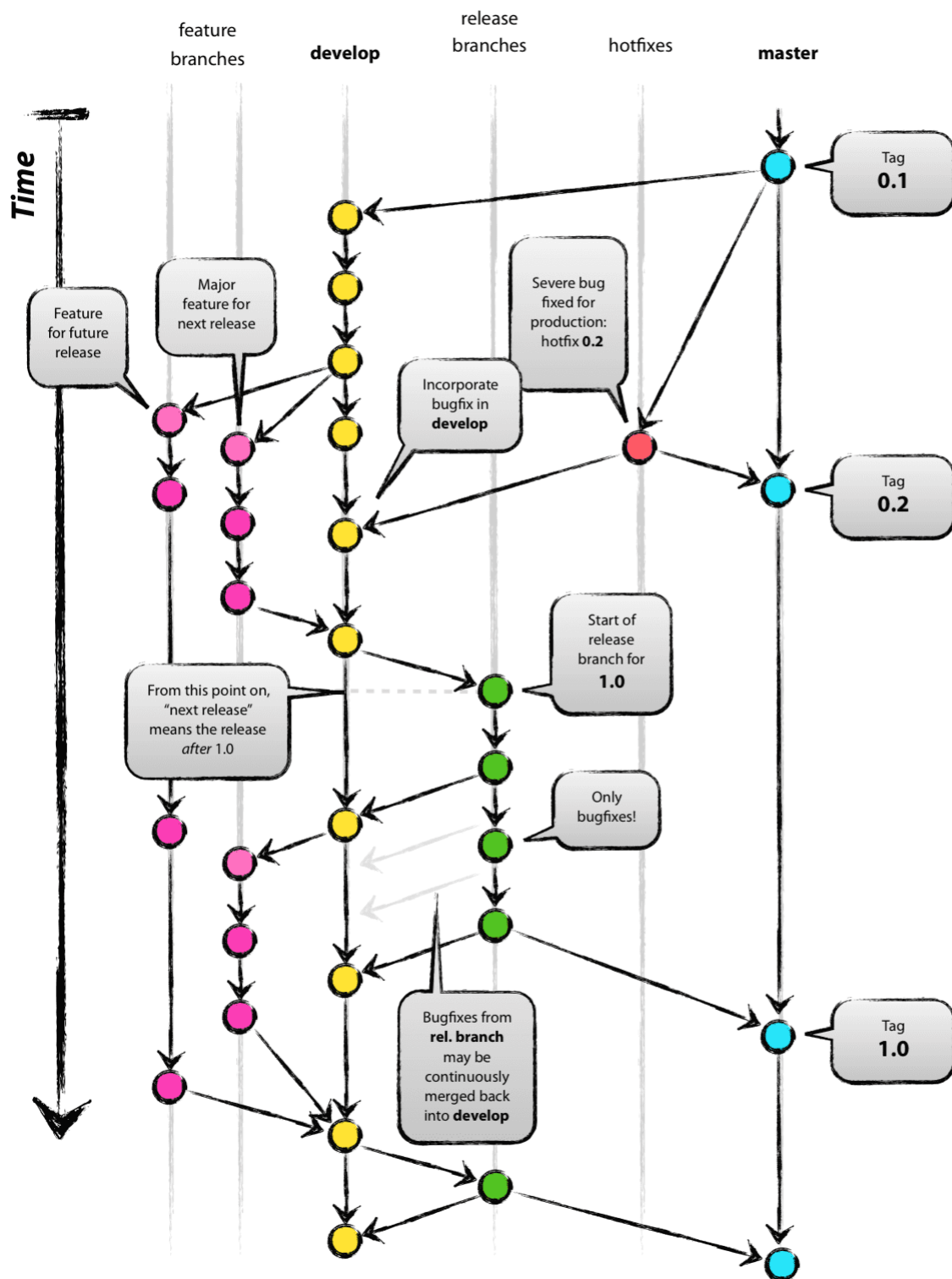
¿Cómo debiera controlar la funcionalidad deprecada?

Deprecar funcionalidad existente es una parte normal del desarrollo de software y es típicamente requerida para avanzar. Cuando desapruebas parte de tu API pública, deberías hacer dos cosas: (1) actualizar tu documentación para permitir que los usuarios sepan del cambio, (2) emitir un nuevo release minor con la desaprobación. Antes de sacar la funcionalidad por completo en un nuevo release mayor, debería haber al menos un release minor que contenga la desaprobación de manera tal que los usuarios puedan traspasarse a la nueva API de manera ordenada.

3.2. Branching Model

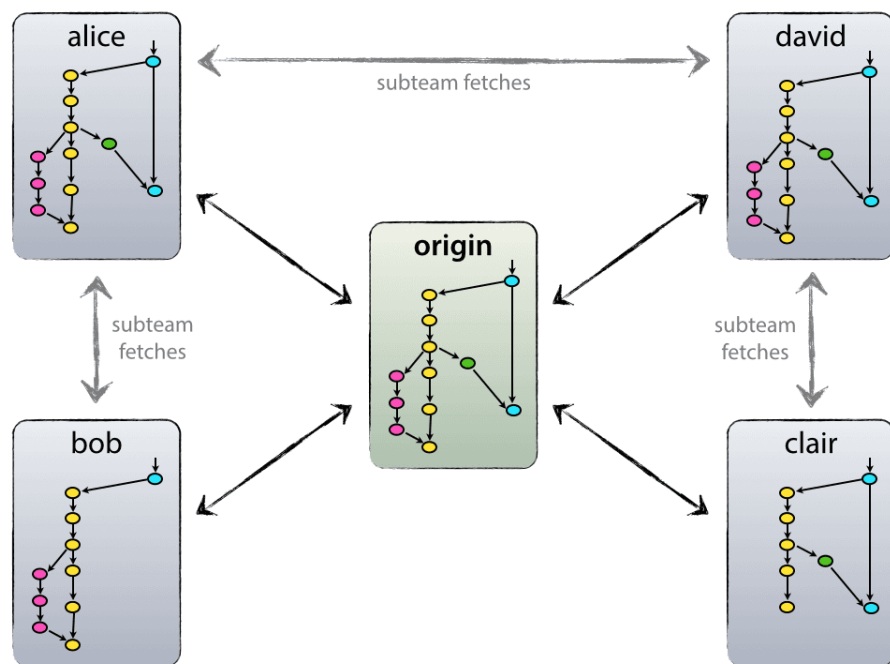
A continuación les presento una estrategia de ramificación y gestión de releases.

El modelo que voy a presentar aquí es esencialmente un conjunto de procedimientos que cada miembro del equipo tiene que seguir con el fin de llegar a un proceso de desarrollo de software administrado.



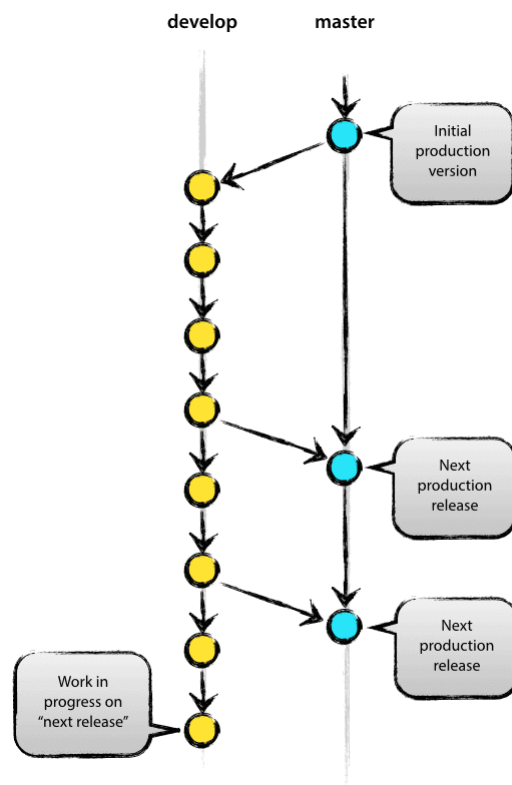
Descentralizado pero centralizado

La configuración de repositorio que usamos y que funciona bien con este modelo de ramificación, es con un "origin" como repositorio central. Debes tener en cuenta que este repo sólo se *considera* para ser el central (ya que Git es un DVCS, no hay tal cosa como un repo central a nivel técnico). Nos referimos a este repo como origin, ya que este nombre es familiar para todos los usuarios de Git.



Cada desarrollador *pullea* desde el origen y *pushea* al origen. Pero además de las relaciones pull-push, cada desarrollador también puede interactuar con otros compañeros para formar equipos sub. Por ejemplo, esto podría ser útil para trabajar en conjunto con dos o más desarrolladores en una gran novedad, antes de *pushear* los trabajos en curso al origen antes de tiempo. En la figura anterior, hay subequipos de Alice y Bob, Alice y David, y Clair y David.

Las ramas principales



En la figura anterior, el modelo de desarrollo se inspira en gran medida por los modelos existentes que hay. El repo central tiene dos ramas principales con una vida infinita:

- **master**
- **develop**

La rama *master* es un espejo del código fuente que se encuentra en producción.

La rama *develop* es la rama de desarrollo. Algunos llamarían a esta la "rama de integración". Aquí es donde se integran los desarrollos de todos los programadores.

Cuando el código fuente en el *develop* llega a un estado estable para hacer un nuevo reléase y un posterior deploy en producción, todos los cambios deberían fusionarse de nuevo en *master* y luego etiquetados con un número de versión como un tag. Cómo se hace esto en detalle se discutirá más adelante.

Al lado de las principales ramas *master* y *develop*, nuestro modelo de desarrollo utiliza una variedad de ramas de apoyo para ayudar al desarrollo paralelo entre los miembros del equipo, la facilidad de seguimiento de características, prepararse para los releases en producción y para ayudar en la resolución rápidamente a los bugs en producción. A diferencia de las ramas principales, estas ramas siempre tienen un tiempo de vida limitado, ya que serán eliminados eventualmente.

Los diferentes tipos de ramas que pueden utilizarse son:

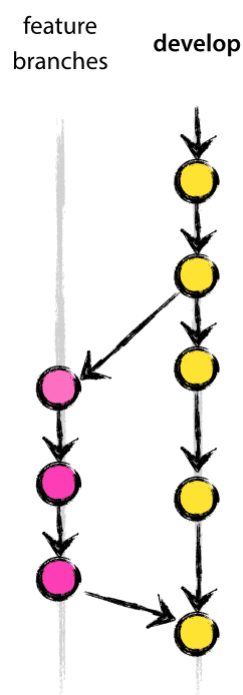
- **Feature branches**
- **Release branches**
- **Hotfix branches**

Cada una de estas ramas tiene un propósito específico y ciertas reglas estrictas de combinación.

Estas ramas no son "especiales" desde una perspectiva técnica. Los tipos de ramas se clasifican por la forma en que las usamos.

Característica ramas

a) **Feature Branches**



- ❑ Puede ramificarse a partir de: develop
- ❑ Debe combinar de nuevo en: develop
- ❑ Convención de nombres: feature/nroIssue-descripcionDelIssue

Estas ramas son las utilizadas para las funcionalidades a desarrollar para el nuevo release.

Creación de una rama feature

Al comenzar a trabajar en una nueva característica, se ramifica desde el develop la rama. git

```
git checkout -b feature/nroIssue-descripcionDelIssue develop
```

El uso de “/” es un pequeño truco de Git. Hace que las herramientas visuales como SourceTree o TortoiseGit creen un árbol de issues dejando la palabra clave *feature* como padre del árbol. El número del issue nos lo provee GitLab y para la descripción del issue podemos usar [UpperCamelCase](#). Los nombres de las ramas no pueden contener espacios en blanco.

La incorporación de una funcionalidad terminada de desarrollar

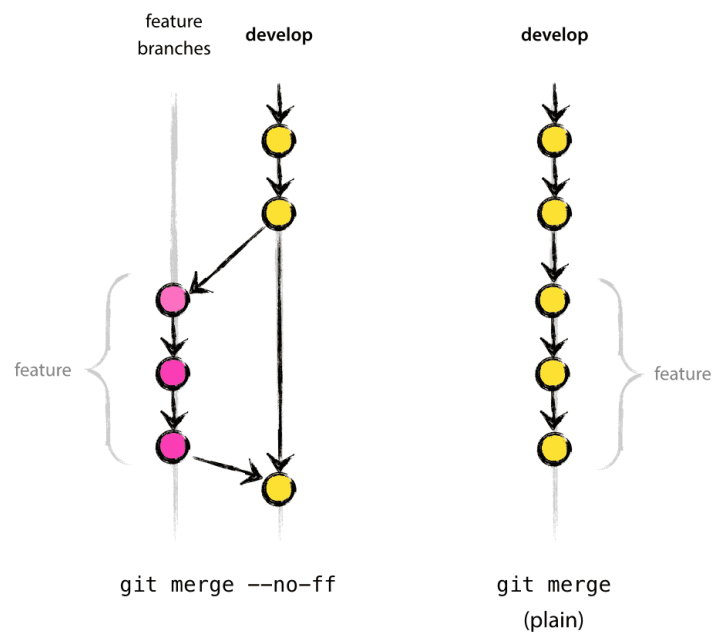
Las funcionalidades terminadas pueden fusionarse en develop para añadirlos a la próxima versión:

```
git checkout develop
```

Con el comando anterior nos cambiamos a la rama develop git

```
git merge --no-ff myfeature
```

Hace un merge entre develop y nuestro branch de desarrollo.



```
git push origin develop
```

Sube los cambios al servidor de GitLab

La bandera --no-ff agrupa todas las confirmaciones del branch myfeature en uno sólo.

En la imagen anterior se puede ver la diferencia de resultados en la rama develop si utilizamos o no la bandera --no-ff. Este es una funcionalidad extremadamente útil. Por ejemplo, ¿a qué desarrollador no le ha sucedido que programamos una funcionalidad para incluir en el próximo release y al cliente se arrepiente de incluirlo? Imagina que la funcionalidad a desestimar es un desarrollo de 80 hs que incluye 10 commits. Eliminar la funcionalidad en otros VCS es un verdadero dolor de cabeza, donde por lo general debemos hacerlo manualmente. En git es un sólo 2 comandos, donde descartamos el commit del merge: Primero hacemos un checkout a la rama donde queremos eliminar el commit. Supongamos que es en la rama develop:

```
git checkout develop
```

Y después eliminamos el commit con su sha-1:

```
git reset --hard <sha1-commit-id>
```

b) Release branches

- ☐ Puede ramifican a partir de: develop
- ☐ Debe combinar de nuevo en: develop y master
- ☐ Convención de nombres: release/nroDeRelease

Las ramas de release son utilizadas para los deploy en producción. Esta metodología es muy útil cuando en el equipo hay desarrolladores y testers. De esta manera los testers pueden probar el release en un entorno estable y sin cambios (en el branch release), mientras que los desarrolladores pueden seguir programando las funcionalidades del próximo release, entre muchas otras ventajas.

La creación de una rama de liberación

La creación de un branch de release debe hacerse, por lo general, a partir del branch develop.

```
git checkout develop
```

```
git checkout -b release/nroDeRelease
```

Luego de que los testers ejecuten los casos de pruebas y todos ellos han pasado, entonces estamos listos para el deploy en producción. Para ello debemos subir los cambios del release a la rama master, recuerden que esta rama debe ser un espejo de producción.

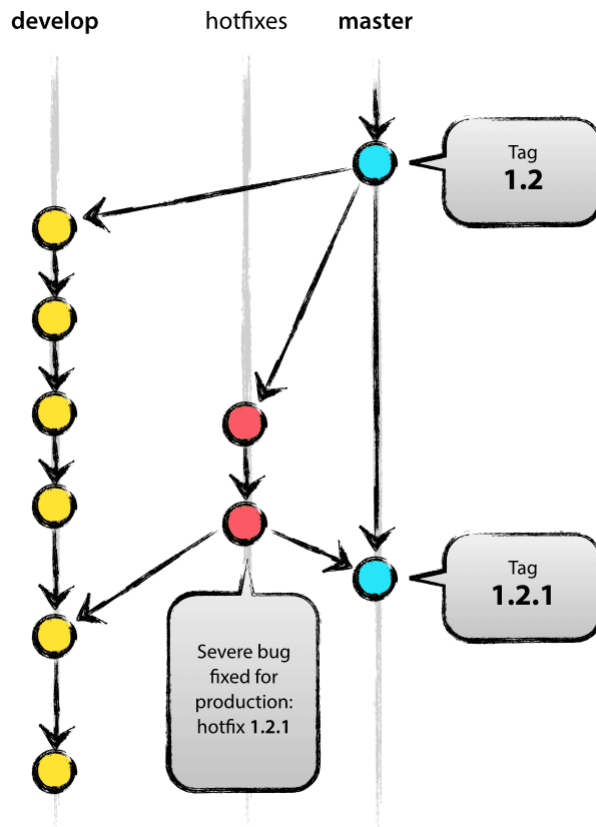
```
git checkout master
```

```
git merge --no-ff release/nroDeRelease git
```

```
tag release/nroDeRelease
```

Los tags son utilizados como paquetes para subir a producción. Nunca debe modificarse un tag.

c) Hotfix Branchs



- Puede ramifican a partir de:
master
- Debe combinar de nuevo en:
develop y master
- Convención de nombres:
hotfix/nroDeBug-Descripcion

Las ramas de Hotfix o de Bugs son como las ramas de release en el que también están destinadas a prepararse para una nueva versión de producción, aunque no planificado. Surgen de la necesidad de actuar inmediatamente después de un estado no deseado de una versión de producción. Cuando ocurre un error crítico en una versión de producción se debe resolver de inmediato, una rama hotfix puede ser ramificado fuera de la etiqueta correspondiente en la rama principal que marca la versión de producción.

La esencia de esta metodología es la organización de trabajo de los miembros del equipo donde en la rama develop puede continuar el desarrollo, mientras que otra persona se prepara para la resolución del bug.

Creación de la rama de hotfix

Las ramas Hotfix se crean desde la rama master. Por ejemplo, digamos que la versión 1.2 es la versión de producción actual y hay un bug grave. Pero los cambios en develop son todavía inestables. Podemos entonces ramificar en una rama de bug y empezar a solucionar el problema:

```
git checkout -b hotfix/nroDeBug-Descripcion
```

Luego arreglaríamos el bug y subiríamos la rama con el arreglo al server de GitLab para que los testers den su visto bueno. Después de esto, ya estamos lista para el deploy en producción.

```
git checkout master
```

```
git merge --no-ff hotfix/nroDeBug-Descripcion git
```

```
tag nuevoNroRelease
```

4. Conclusión

En el comienzo de la capacitación aprendimos las características principales de Git y sus diferencias con el resto de los sistemas de versionado. Luego vimos los comandos básicos para el manejo de código y su integración con GitLab. Esta herramienta nos permite administrar proyectos de Git, los issues, los merge requests, los branches entre muchas cosas más.

Después expusimos un modelo de branches que soporte los cambios en un proyecto y que pueda ser aplicado a la mayoría de ellos. Y en el final vimos el workflow de issues en distintos tipos de proyectos, dando más enfoque en los proyectos de desarrollo.

Todas las convenciones, estrategias y metodologías no son reglas absolutas. Y aunque funcionan en un gran número de equipos, puede que en tu caso necesites adaptarlas a tu forma de trabajar.

Además, no hemos visto la totalidad de funcionalidades que tiene Git. Por lo tanto, los invito a que incursionen más en el mundo del versionado y su utilización en la administración de proyectos.