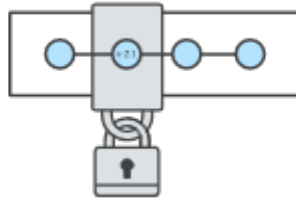


# Version control system | GIT



## 1. What is version control

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

For almost all software projects, the source code is like the crown jewels - a precious asset whose value must be protected. For most software teams, the source code is a repository of the invaluable knowledge and understanding about the problem domain that the developers have collected and refined through careful effort. Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent concurrent work from conflicting. Changes made in one part of the software can be incompatible with those made by another developer working at the same time. This problem should be discovered and solved in an orderly manner without blocking the work of the rest of the team. Further, in all software development, any change can introduce new bugs on its own and new software can't be trusted until it's tested. So testing and development proceed together until a new version is ready.

Good version control software supports a developer's preferred workflow without imposing one particular way of working. Ideally it also works on any platform, rather than dictate what operating system or tool chain developers must use. Great version control systems facilitate a smooth and continuous flow of changes to the code rather than the frustrating and clumsy mechanism of file locking - giving the green light to one developer at the expense of blocking the progress of others.

Software teams that do not use any form of version control often run into problems like not knowing which changes that have been made are available to users or the creation of incompatible

changes between two unrelated pieces of work that must then be painstakingly untangled and reworked. If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version. Perhaps you've commented out code blocks because you want to disable certain functionality without deleting the code, fearing that there may be a use for it later. Version control is a way out of these problems.

Version control software is an essential part of the every-day of the modern software team's professional practices. Individual software developers who are accustomed to working with a capable version control system in their teams typically recognize the incredible value version control also gives them even on small solo projects. Once accustomed to the powerful benefits of version control systems, many developers wouldn't consider working without it even for non-software projects.

## Benefits of version control

Developing software without using version control is risky, like not having backups. Version control can also enable developers to move faster and it allows software teams to preserve efficiency and agility as the team scales to include more developers.

Version Control Systems (VCS) have seen great improvements over the past few decades and some are better than others. VCS are sometimes known as SCM (Source Code Management) tools or RCS (Revision Control System). One of the most popular VCS tools in use today is called Git. Git is a Distributed VCS, a category known as DVCS, more on that later. Like many of the most popular VCS systems available today, Git is free and open source. Regardless of what they are called, or which system is used, the primary benefits you should expect from version control are as follows.

1. A complete long-term change history of every file. This means every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents. Different VCS tools differ on how well they handle renaming and moving of files. This history should also include the author, date and written notes on the purpose of each change. Having the complete history enables going back to previous versions to help in root cause analysis for bugs and it is crucial when needing to fix problems in older versions of software. If the software is being actively worked on, almost everything can be considered an "older version" of the software.
2. Branching and merging. Having team members work concurrently is a no-brainer, but even individuals working on their own can benefit from the ability to work on independent streams of changes. Creating a "branch" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict. Many software teams adopt a practice of branching for each feature or perhaps branching for each release, or both. There are many different workflows that teams can choose from when they decide how to make use of branching and merging facilities in VCS.
3. Traceability. Being able to trace each change made to the software and connect it to project management and bug tracking software such as JIRA, and being able to annotate each change with a message describing the purpose and intent of the change can help not only with root cause analysis and other forensics. Having the annotated history of the code at your fingertips when you are reading the code, trying to understand what it is doing and why it is so designed can enable developers to make correct and harmonious changes that are in accord with the intended long-term

design of the system. This can be especially important for working effectively with legacy code and is crucial in enabling developers to estimate future work with any accuracy.

While it is possible to develop software without using any version control, doing so subjects the project to a huge risk that no professional team would be advised to accept. So the question is not whether to use version control but which version control system to use.

There are many choices, but here we are going to focus on just one, Git.

## **2. What is Git**

By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel. A staggering number of software projects rely on Git for version control, including commercial projects as well as open source. Developers

who have worked with Git are well represented in the pool of available software development talent and it works well on a wide range of operating systems and IDEs (Integrated Development Environments).

Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

In addition to being distributed, Git has been designed with performance, security and flexibility in mind.

## Performance

The raw performance characteristics of Git are very strong when compared to many alternatives. Committing new changes, branching, merging and comparing past versions are all optimized for performance. The algorithms implemented inside Git take advantage of deep knowledge about common attributes of real source code file trees, how they are usually modified over time and what the access patterns are.

Unlike some version control software, Git is not fooled by the names of the files when determining what the storage and version history of the file tree should be, instead, Git focuses on the file content itself. After all, source code files are frequently renamed, split, and rearranged. The object format of Git's repository files uses a combination of delta encoding (storing content differences), compression and explicitly stores directory contents and version metadata objects.

Being distributed enables significant performance benefits as well.

For example, say a developer, Alice, makes changes to source code, adding a feature for the upcoming 2.0 release, then commits those changes with descriptive messages. She then works on a second feature and commits those changes too. Naturally these are stored as separate pieces of work in the version history. Alice then switches to the version 1.3 branch of the same software to fix a bug that affects only that older version. The purpose of this is to enable Alice's team to ship a bug fix release, version 1.3.1, before version 2.0 is ready. Alice can then return to the 2.0 branch to continue working on new features for 2.0 and all of this can occur without any network access and is therefore fast and reliable. She could even do it on an airplane. When she is ready to send all of the individually committed changes to the remote repository, Alice can "push" them in one command.

## Security

Git has been designed with the integrity of managed source code as a top priority. The content of the files as well as the true relationships between files and directories, versions, tags and commits, all of these objects in the Git repository are secured with a cryptographically secure hashing algorithm called SHA1. This protects the code and the change history against both accidental and malicious change and ensures that the history is fully traceable.

With Git, you can be sure you have an authen

tic content history of your source code.

Some other version control systems have no protections against secret alteration at a later date. This can be a serious information security vulnerability for any organization that relies on software development.

## **Flexibility**

One of Git's key design objectives is flexibility. Git is flexible in several respects: in support for various kinds of nonlinear development workflows, in its efficiency in both small and large projects and in its compatibility with many existing systems and protocols.

Git has been designed to support branching and tagging as first-class citizens (unlike SVN) and operations that affect branches and tags (such as merging or reverting) are also stored as part of the change history. Not all version control systems feature this level of tracking.

## **Version control with Git**

Git is the best choice for most software teams today. While every team is different and should do their own analysis, here are the main reasons why version control with Git is preferred over alternatives:

## **Git is good**

Git has the functionality, performance, security and flexibility that most teams and individual developers need. These attributes of Git are detailed above. In side-by-side comparisons with most other alternatives, many teams find that Git is very favorable.

## **Git is a de facto standard**

Git is the most broadly adopted tool of its kind. This makes Git attractive for the following reasons. At Atlassian, nearly all of our project source code is managed in Git.

Vast numbers of developers already have Git experience and a significant proportion of college graduates may have experience with only Git. While some organizations may need to climb the learning curve when migrating to Git from another version control system, many of their existing and future developers do not need to be trained on Git.

In addition to the benefits of a large talent pool, the predominance of Git also means that many third party software tools and services are already integrated with Git including IDEs, and our own tools like DVCS desktop client SourceTree, issue and project tracking software, JIRA, and code hosting service, Bitbucket.

If you are an inexperienced developer wanting to build up valuable skills in software development tools, when it comes to version control, Git should be on your list.

## Git is a quality open source project

Git is a very well supported open source project with over a decade of solid stewardship. The project maintainers have shown balanced judgment and a mature approach to meeting the long term needs of its users with regular releases that improve usability and functionality. The quality of the open source software is easily scrutinized and countless businesses rely heavily on that quality.

Git enjoys great community support and a vast user base. Documentation is excellent and plentiful, including books, tutorials and dedicated web sites. There are also podcasts and video tutorials.

Being open source lowers the cost for hobbyist developers as they can use Git without paying a fee. For use in open-source projects, Git is undoubtedly the successor to the previous generations of successful open source version control systems, SVN and CVS.

## Criticism of Git

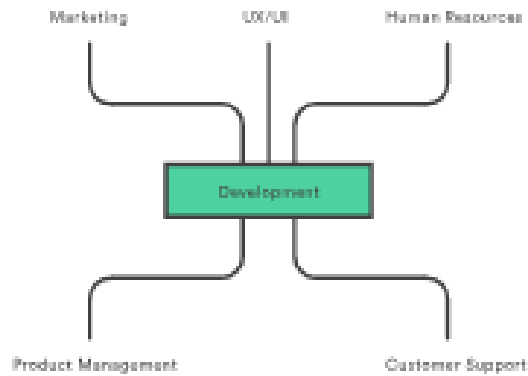
One common criticism of Git is that it can be difficult to learn. Some of the terminology in Git will be novel to newcomers and for users of other systems, the Git terminology may be different, for example, `revert` in Git has a different meaning than in SVN or CVS. Nevertheless, Git is very capable and provides a lot of power to its users. Learning to use that power can take some time, however once it has been learned, that power can be used by the team to increase their development speed.

For those teams coming from a non-distributed VCS, having a central repository may seem like a good thing that they don't want to lose. However, while Git has been designed as a distributed version control system (DVCS), with Git, you can still have an official, canonical repository where all changes to the software must be stored. With Git, because each developer's repository is complete, their work doesn't need to be constrained by the availability and performance of the "central" server. During outages or while offline, developers can still consult the full project history. Because Git is flexible as well as being distributed, you can work the way you are accustomed to but gain the additional benefits of Git, some of which you may not even realise you're missing.

Now that you understand what version control is, what Git is and why software teams should use it, read on to discover the benefits Git can provide across the whole organization.

## 3. Why Git for your organization

Switching from a centralized version control system to Git changes the way your development team creates software. And, if you're a company that relies on its software for mission-critical applications, altering your development workflow impacts your entire business.



In this article, we'll

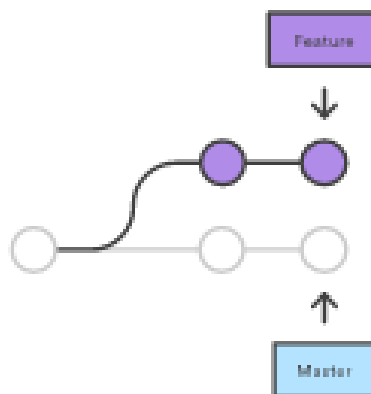
discuss how Git

benefits each aspect of your organization, from your development team to your marketing team, and everything in between. By the end of this article, it should be clear that Git isn't just for agile software development—it's for agile business.

## Git for developers

### Feature Branch Workflow

One of the biggest advantages of Git is its branching capabilities. Unlike centralized version control systems, Git branches are cheap and easy to merge. This facilitates the feature branch workflow popular with many Git users.



Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something—no matter how big or small—they create a new branch. This ensures that the master branch always contains production-quality code.

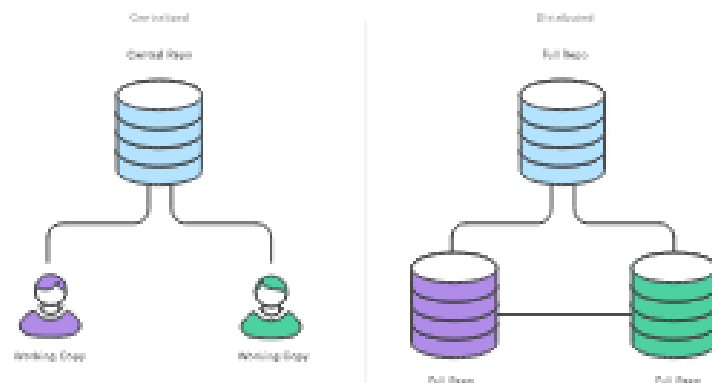
Using feature branches is not only more reliable than directly editing production code, but it also provides organizational benefits. They let you represent development work at the same granularity



as the your agile backlog. For example, you might implement a policy where each JIRA ticket is addressed in its own feature branch.

## Distributed Development

In SVN, each developer gets a working copy that points back to a single central repository. Git, however, is a distributed version control system. Instead of a working copy, each developer gets their own local repository, complete with a full history of commits.



Having a full local history makes Git fast, since it means you don't need a network connection to create commits, inspect previous versions of a file, or perform diffs between commits.

Distributed development also makes it easier to scale your engineering team. If someone breaks the production branch in SVN, other developers can't check in their changes until it's fixed. With Git, this kind of blocking doesn't exist. Everybody can continue going about their business in their own local repositories.

And, similar to feature branches, distributed development creates a more reliable environment. Even if a developer obliterates their own repository, they can simply clone someone else's and start anew.

## Pull Requests

Many source code management tools such as Bitbucket enhance core Git functionality with pull requests. A pull request is a way to ask another developer to merge one of your branches into their repository. This not only makes it easier for project leads to keep track of changes, but also lets developers initiate discussions around their work before integrating it with the rest of the codebase.

Since they're essentially a comment thread attached to a feature branch, pull requests are extremely versatile. When a developer gets stuck with a hard problem, they can open a pull request to ask for help from the rest of the team. Alternatively, junior developers can be confident that they aren't destroying the entire project by treating pull requests as a formal code review.

## 4. Installing Git

Before you start using Git, you have to make it available on your computer. Even if it's already installed, it's probably a good idea to update to the latest version. You can either install it as a package or via another installer, or download the source code and compile it yourself.

### NOTE

This book was written using Git version **2.0.0**. Though most of the commands we use should work even in ancient versions of Git, some of them might not or might act slightly differently if you're using an older version. Since Git is quite excellent at preserving backwards compatibility, any version after 2.0 should work just fine.

### Installing on Linux

If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora for example, you can use yum:

```
$ sudo yum install git-all
```

If you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ sudo apt-get install git-all
```

For more options, there are instructions for installing on several different Unix flavors on the Git website, at <http://git-scm.com/download/linux>.

## Installing on Mac

There are several ways to install Git on a Mac. The easiest is probably to install the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this simply by trying to run *git* from the Terminal the very first time. If you don't have it installed already, it will prompt you to install it.

If you want a more up to date version, you can also install it via a binary installer. An OSX Git installer is maintained and available for download at the Git website, at <http://git-scm.com/download/mac>.

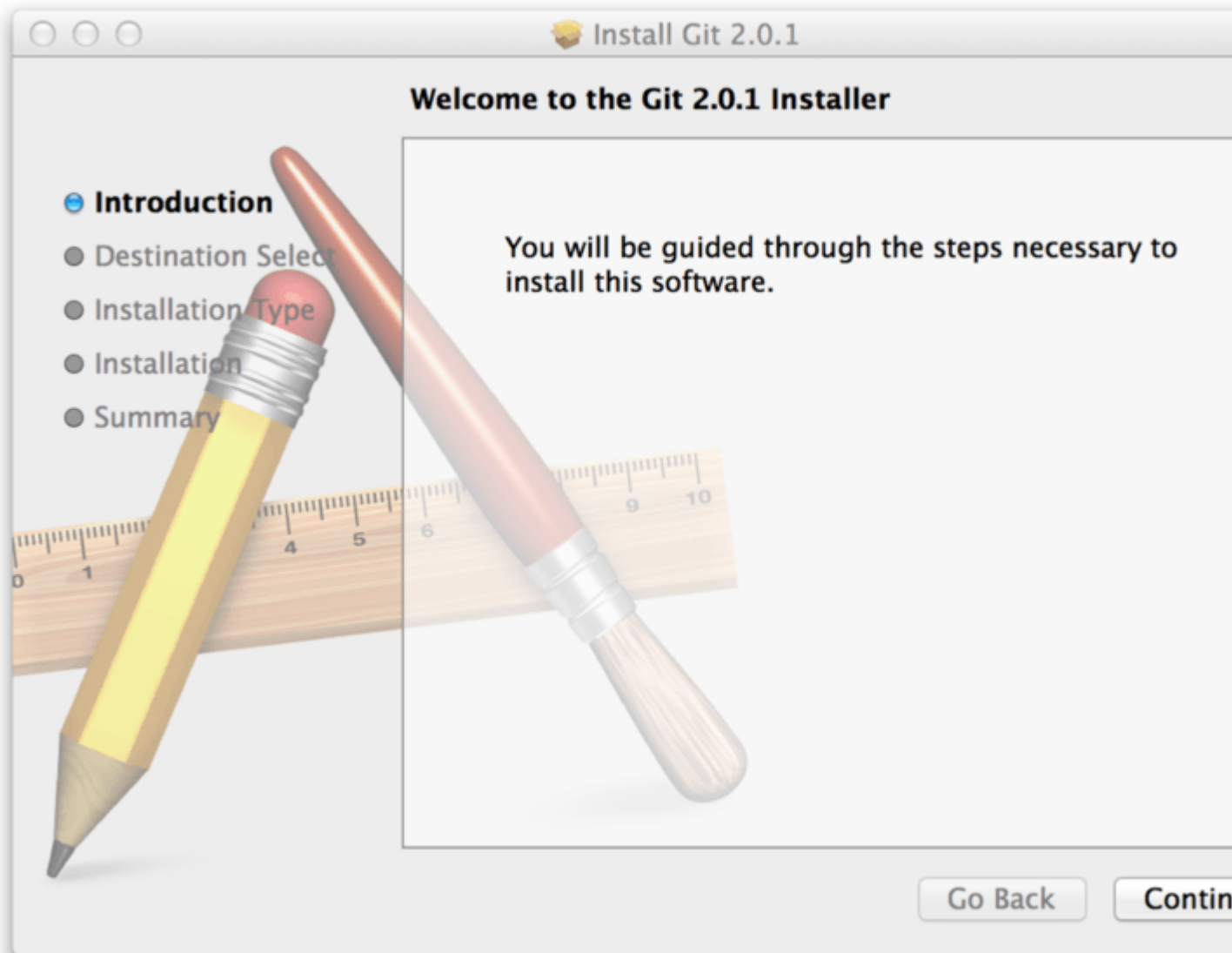


Figure 1-7. Git OS X Installer.

You can also install it as part of the GitHub for Mac install. Their GUI Git tool has an option to install command line tools as well. You can download that tool from the GitHub for Mac website, at <http://mac.github.com>.

## Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to <http://git-scm.com/download/win> and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to <https://git-for-windows.github.io/>.

Another easy way to get Git installed is by installing GitHub for Windows. The installer includes a command line version of Git as well as the GUI. It also works well with Powershell, and sets up solid credential caching and sane CRLF settings. We'll learn more about those things a little later, but suffice it to say they're things you want. You can download this from the GitHub for Windows website, at <http://windows.github.com>.

## 5. Setting up a repository

This tutorial provides a succinct overview of the most important Git commands. First, the Setting Up a Repository section explains all of the tools you need to start a new version-controlled project. Then, the remaining sections introduce your everyday Git commands.

By the end of this module, you should be able to create a Git repository, record snapshots of your project for safekeeping, and view your project's history.

### git init

The `git init` command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new empty repository. Most of the other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing `git init` creates a `.git` subdirectory in the project root, which contains all of the necessary metadata for the repo. Aside from the `.git` directory, an existing project remains unaltered (unlike SVN, Git doesn't require a `.git` folder in every subdirectory).

### Usage

```
$ git init
```

Transform the current directory into a Git repository. This adds a `.git` folder to the current directory and makes it possible to start recording revisions of the project.

```
$ git init <directory>
```

Create an empty Git repository in the specified directory. Running this command will create a new folder called `<directory>` containing nothing but the `.git` subdirectory.

```
$ git init --bare <directory>
```

Initialize an empty Git repository, but omit the working directory. Shared repositories should always be created with the `--bare` flag (see discussion below). Conventionally, repositories initialized with the `--bare` flag end in `.git`. For example, the bare version of a repository called `my-project` should be stored in a directory called `my-project.git`.

## git clone

The `git clone` command copies an existing Git repository. This is sort of like `svn checkout`, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called `origin` pointing back to the original repository. This makes it very easy to interact with a central repository.

## Usage

```
git clone <repo>
```

Clone the repository located at `<repo>` onto the local machine. The original repository can be located on the local filesystem or on a remote machine accessible via HTTP or SSH.

```
git clone <repo> <directory>
```

Clone the repository located at `<repo>` into the folder called `<directory>` on the local machine.

## Example

```
$ git clone https://github.com/wakaleo/game-of-life.git
```

## git config

The `git config` command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info to preferences to the behavior of a repository. Several common configuration options are listed below.

## Usage

```
$ git config user.name <name>
```

Define the author name to be used for all commits in the current repository. Typically, you’ll want to use the `--global` flag to set configuration options for the current user.

```
$ git config --global user.name <name>
```

Define the author name to be used for all commits by the current user.

```
$ git config --global user.email <email>
```

Define the author email to be used for all commits by the current user.

```
$ git config --global alias.<alias-name> <git-command>
```

Create a shortcut for a Git command.

```
$ git config --system core.editor <editor>
```

Define the text editor used by commands like `git commit` for all users on the current machine. The `<editor>` argument should be the command that launches the desired editor (e.g., `vi`).

```
$ git config --global --edit
```

Open the global configuration file in a text editor for manual editing.

## Discussion

All configuration options are stored in plaintext files, so the `git config` command is really just a convenient command-line interface. Typically, you'll only need to configure a Git installation the first time you start working on a new development machine, and for virtually all cases, you'll want to use the `--global` flag.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

- `<repo>/ .git/config` – Repository-specific settings.
- `~/.gitconfig` – User-specific settings. This is where options set with the `--global` flag are stored.
- `$(prefix)/etc/gitconfig` – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you'll see something like the following:

```
[user]
name = John Smith
email = john@example.com
[alias]
st = status
co = checkout
br = branch
up = rebase
ci = commit
[core]
editor = vim
```

You can manually edit these values to the exact same effect as `git config`.

## Example

The first thing you'll want to do after installing Git is tell it your name/email and customize some of the default settings. A typical initial configuration might look something like the following:

```
# Tell Git who you are
$ git config --global user.name "John Smith"
$ git config --global user.email john@example.com
# Select your favorite text editor
$ git config --global core.editor vim
# Add some SVN-like aliases
$ git config --global alias.st status
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.up rebase
$ git config --global alias.ci commit
```

## 6. Saving changes

### git add

The `git add` command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, `git add` doesn't really affect the repository in any significant way—changes are not actually recorded until you run `git commit`.

In conjunction with these commands, you'll also need `git status` to view the state of the working directory and the staging area.

### Usage

```
$ git add <file>
```

Stage all changes in `<file>` for the next commit.

```
$ git add <directory>
```

Stage all changes in `<directory>` for the next commit.

```
$ git add -p
```

Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use `y` to stage the chunk, `n` to ignore the chunk, `s` to split it into smaller chunks, `e` to manually edit the chunk, and `q` to exit.

### Discussion

The `git add` and `git commit` commands compose the fundamental Git workflow. These are the two commands that every Git user needs to understand, regardless of their

team's collaboration model. They are the means to record versions of a project into the repository's history.

Developing a project revolves around the basic edit/stage/commit pattern. First, you edit your files in the working directory. When you're ready to save a copy of the current state of the project, you stage changes with `git add`. After you're happy with the staged snapshot, you commit it to the project history with `git commit`.

The `git add` command should not be confused with `svn add`, which adds a file to the repository. Instead, `git add` works on the more abstract level of changes. This means that `git add` needs to be called every time you alter a file, whereas `svn add` only needs to be called once for each file. It may sound redundant, but this workflow makes it much easier to keep a project organized.

## The Staging Area

The staging area is one of Git's more unique features, and it can take some time to wrap your head around it if you're coming from an SVN (or even a Mercurial) background. It helps to think of it as a buffer between the working directory and the project history.

Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. As in any revision control system, it's important to create atomic commits so that it's easy to track down bugs and revert changes with minimal impact on the rest of the project.

## Example

When you're starting a new project, `git add` serves the same function as `svn import`. To create an initial commit of the current directory, use the following two commands:

```
$ git add .  
$ git commit
```

Once you've got your project up-and-running, new files can be added by passing the path to `git add`:

```
$ git add hello.py  
$ git commit
```

The above commands can also be used to record changes to existing files. Again, Git doesn't differentiate between staging changes in new files vs. changes in files that have already been added to the repository.



## git commit

The `git commit` command commits the staged snapshot to the project history.

Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Along with `git add`, this is one of the most important Git commands.

While they share the same name, this command is nothing like `svn commit`. Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories.

## Usage

```
$ git commit
```

Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you’ve entered a message, save the file and close the editor to create the actual commit. `git commit -m "<message>"`

Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message.

```
$ git commit -a
```

Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with `git add` at some point in their history).

## Discussion

Snapshots are always committed to the local repository. This is fundamentally different from SVN, wherein the working copy is committed to the central repository. In contrast, Git doesn’t force you to interact with the central repository until you’re ready. Just as the staging area is a buffer between the working directory and the project history, each developer’s local repository is a buffer between their contributions and the central repository.

This changes the basic development model for Git users. Instead of making a change and committing it directly to the central repo, Git developers have the opportunity to accumulate commits in their local repo. This has many advantages over SVN-style collaboration: it makes it easier to split up a feature into atomic commits, keep related commits grouped together, and clean up local history before publishing it to the central repository. It also lets developers work in an isolated environment, deferring integration until they’re at a convenient break point.

## Example

The following example assumes you've edited some content in a file called `hello.py` and are ready to commit it to the project history. First, you need to stage the file with `git add`, then you can commit the staged snapshot.

```
$ git add hello.py
$ git commit
```

This will open a text editor (customizable via `git config`) asking for a commit message, along with a list of what's being committed:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#modified: hello.py
```

Git doesn't require commit messages to follow any specific formatting constraints, but the canonical format is to summarize the entire commit on the first line in less than 50 characters, leave a blank line, then a detailed explanation of what's been changed. For example:

Change the message displayed by `hello.py`

- Update the `sayHello()` function to output the user's name
- Change the `sayGoodbye()` function to a friendlier message

Note that many developers also like to use present tense in their commit messages. This makes them read more like actions on the repository, which makes many of the history-rewriting operations more intuitive.

## 7. Syncing

SVN uses a single central repository to serve as the communication hub for developers, and collaboration takes place by passing changesets between the developers' working copies and the central repository. This is different from Git's collaboration model, which gives every developer their own copy of the repository, complete with its own local history and branch structure. Users typically need to share a series of commits rather than a single changeset. Instead of committing a changeset from a working copy to the central repository, Git lets you share entire branches between repositories.

The commands presented below let you manage connections with other repositories, publish local history by "pushing" branches to other repositories, and see what others have contributed by "pulling" branches into your local repository.

## git remote

The `git remote` command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.

### Usage

```
$ git remote
```

List the remote connections you have to other repositories.

```
$ git remote -v
```

Same as the above command, but include the URL of each connection.

```
$ git remote add <name> <url>
```

Create a new connection to a remote repository. After adding a remote, you'll be able to use `<name>` as a convenient shortcut for `<url>` in other Git commands.

```
$ git remote rm <name>
```

Remove the connection to the remote repository called `<name>`.

```
$ git remote rename <old-name> <new-name>
```

Rename a remote connection from `<old-name>` to `<new-name>`.

### Discussion

Git is designed to give each developer an entirely isolated development environment. This means that information is not automatically passed back and forth between repositories. Instead, developers need to manually pull upstream commits into their local repository or manually push their local commits back up to the central repository. The `git remote` command is really just an easier way to pass URLs to these "sharing" commands.

### The origin Remote

When you clone a repository with `git clone`, it automatically creates a remote connection called `origin` pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behavior is also why most Git-based projects call their central repository `origin`.

## Repository URLs

Git supports many ways to reference a remote repository. Two of the easiest ways to access a remote repo are via the HTTP and the SSH protocols. HTTP is an easy way to allow anonymous, read-only access to a repository. For example:

```
http://host/path/to/repo.git
```

But, it's generally not possible to push commits to an HTTP address (you wouldn't want to allow anonymous pushes anyways). For read-write access, you should use SSH instead:

```
ssh://user@host/path/to/repo.git
```

You'll need a valid SSH account on the host machine, but other than that, Git supports authenticated access via SSH out of the box.

## Examples

In addition to origin, it's often convenient to have a connection to your teammates' repositories. For example, if your co-worker, John, maintained a publicly accessible repository on `dev.example.com/john.git`, you could add a connection as follows:

```
$ git remote add john http://dev.example.com/john.git
```

Having this kind of access to individual developers' repositories makes it possible to collaborate outside of the central repository. This can be very useful for small teams working on a large project.

## git fetch

The `git fetch` command imports commits from a remote repository into your local repo. The resulting commits are stored as remote branches instead of the normal local branches that we've been working with. This gives you a chance to review changes before integrating them into your copy of the project.

## Usage

```
$ git fetch <remote>
```

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

```
$ git fetch <remote> <branch>
```

Same as the above command, but only fetch the specified branch.

## Discussion

Fetching is what you do when you want to see what everybody else has been working on. Since fetched content is represented as a remote branch, it has absolutely no effect on

your local development work. This makes fetching a safe way to review commits before integrating them with your local repository. It's similar to `svn update` in that it lets you see how the central history has progressed, but it doesn't force you to actually merge the changes into your repository.

## Remote Branches

Remote branches are just like local branches, except they represent commits from somebody else's repository. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (just like checking out an old commit). You can think of them as read-only branches. To view your remote branches, simply pass the `-r` flag to the `git branch` command. Remote branches are prefixed by the remote they belong to so that you don't mix them up with local branches. For example, the next code snippet shows the branches you might see after fetching from the origin remote:

```
git branch -r
# origin/master
# origin/develop
# origin/some-feature
```

Again, you can inspect these branches with the usual `git checkout` and `git log` commands. If you approve the changes a remote branch contains, you can merge it into a local branch with a normal `git merge`. So, unlike SVN, synchronizing your local repository with a remote repository is actually a two-step process: fetch, then merge. The `git pull` command is a convenient shortcut for this process.

## Examples

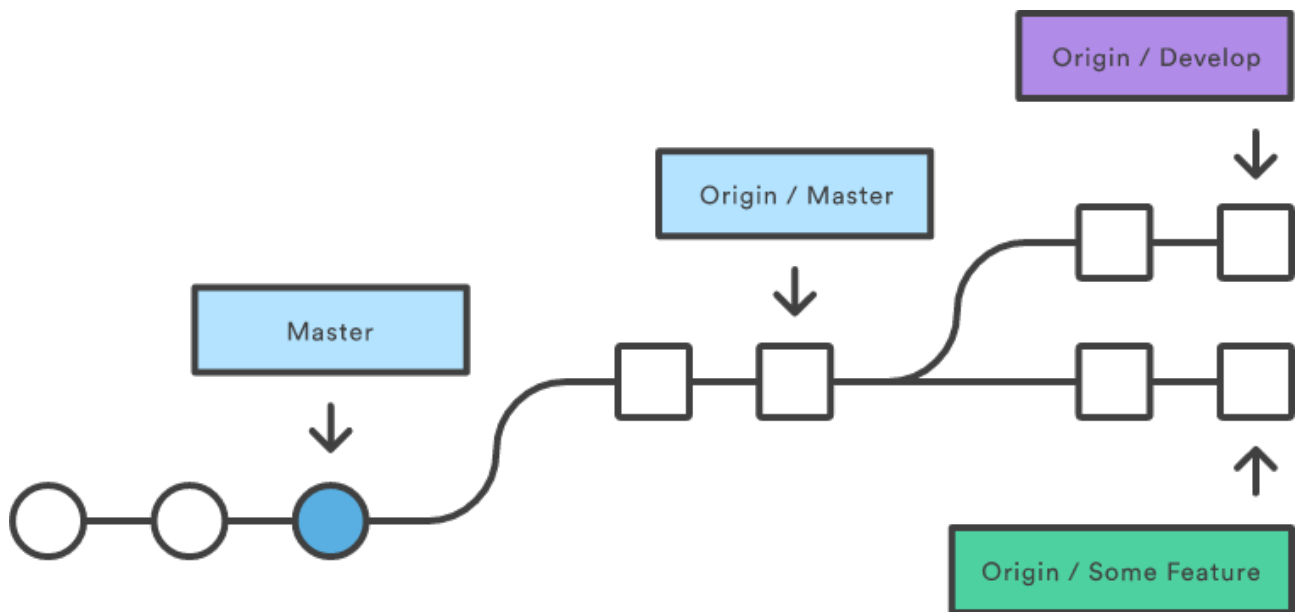
This example walks through the typical workflow for synchronizing your local repository with the central repository's master branch.

```
$ git fetch origin
```

This will display the branches that were downloaded:

```
ale8fb5..45e66a4 master -> origin/master
ale8fb5..9e8ab1c develop -> origin/develop
* [new branch] some-feature -> origin/some-feature
```

The commits from these new remote branches are shown as squares instead of circles in the diagram below. As you can see, `git fetch` gives you access to the entire branch structure of another repository.



To see what commits have been added to the upstream master, you can run a `git log` using `origin/master` as a filter

```
$ git log --oneline master..origin/master
```

To approve the changes and merge them into your local master branch with the following commands:

```
$ git checkout master  
$ git log origin/master
```

Then we can use `git merge origin/master`

```
$ git merge origin/master
```

The `origin/master` and `master` branches now point to the same commit, and you are synchronized with the upstream developments.

## git pull

Merging upstream changes into your local repository is a common task in Git-based collaboration workflows. We already know how to do this with `git fetch` followed by `git merge`, but `git pull` rolls this into a single command.

## Usage

```
$ git pull <remote>
```

Fetch the specified remote's copy of the current branch and immediately merge it into the local copy. This is the same as `git fetch <remote>` followed by `git merge origin/<current-branch>`.

```
$ git pull --rebase <remote>
```

Same as the above command, but instead of using `git merge` to integrate the remote branch with the local one, use `git rebase`.

## Discussion

You can think of `git pull` as Git's version of `svn update`. It's an easy way to synchronize your local repository with upstream changes.

You start out thinking your repository is synchronized, but then `git fetch` reveals that origin's version of `master` has progressed since you last checked it. Then `git merge` immediately integrates the remote `master` into the local one:

## Pulling via Rebase

The `--rebase` option can be used to ensure a linear history by preventing unnecessary merge commits. Many developers prefer rebasing over merging, since it's like saying, "I want to put my changes on top of what everybody else has done." In this sense, using `git pull` with the `--rebase` flag is even more like `svn update` than a plain `git pull`.

In fact, pulling with `--rebase` is such a common workflow that there is a dedicated configuration option for it:

```
$ git config --global branch.autosetuprebase always
```

After running that command, all `git pull` commands will integrate via `git rebase` instead of `git merge`.

## Examples

The following example demonstrates how to synchronize with the central repository's `master` branch:

```
$ git checkout master  
$ git pull --rebase origin
```

This simply moves your local changes onto the top of what everybody else has already contributed.

## git push

Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to `git fetch`, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. This has the potential to overwrite changes, so you need to be careful how you use it. These issues are discussed below.

### Usage

```
$ git push <remote> <branch>
```

Push the specified branch to `<remote>`, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won't let you push when it results in a non-fast-forward merge in the destination repository.

```
$ git push <remote> --force
```

Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the `--force` flag unless you're absolutely sure you know what you're doing.

```
$ git push <remote> --all
```

Push all of your local branches to the specified remote.

```
$ git push <remote> --tags
```

Tags are not automatically pushed when you push a branch or use the `--all` option. The `--tags` flag sends all of your local tags to the remote repository.

### Discussion

The most common use case for `git push` is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you (optionally) clean them up with an interactive rebase, then push them to the central repository.



# Github SSH login

## Generating SSH keys for github

1. Open Terminal.

2. Paste the command below, substituting in your GitHub email address.

```
# ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

3. When you're prompted to "Enter a file in which to save the key,"  
Give below mentioned path

/home/<USERNAME>/.ssh/id\_rsa\_github

**Note:** USERNAME in above path is you linux system user with which you have logged in.

*Generating public/private rsa key pair.*

*Enter a file in which to save the key*

*(/home/<USERNAME>/.ssh/id\_rsa):/home/<USERNAME>/.ssh/id\_rsa\_DO\_github*

4. Hit enter when it asks to enter passphrase

*Enter passphrase (empty for no passphrase): [Type a passphrase]*

*Enter same passphrase again: [Type passphrase again]*

```
imran@DevOps:~$ ssh-keygen -t rsa -b 4096 -C "imranteli0706@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/imran/.ssh/id_rsa): /home/imran/.ssh/id_rsa_DO_github
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/imran/.ssh/id_rsa_DO_github.
Your public key has been saved in /home/imran/.ssh/id_rsa_DO_github.pub.
The key fingerprint is:
SHA256:AacADj0wSRR7tePN93IUS6UyLhPjrpjaFlkg/QUtMGg imranteli0706@gmail.com
The key's randomart image is:
+---[RSA 4096]---+
|B+=+oo+ .      |
|.E.ooo.*       |o
|.O.OO+O.O +    |
|.O= +.+ o      |
|.  *So o       |
|.  . + o       |
|.  . . O       |
|.O . O         |
|.O+ .          |
+---[SHA256]-----+
imran@DevOps:~$
```

## Set SSH private key for github.com login

1. Go to users ssh directory

```
# cd ~/.ssh
```

```
# ls
```

2. Open or create config file and update it with below mentioned content

```
# vi config
```

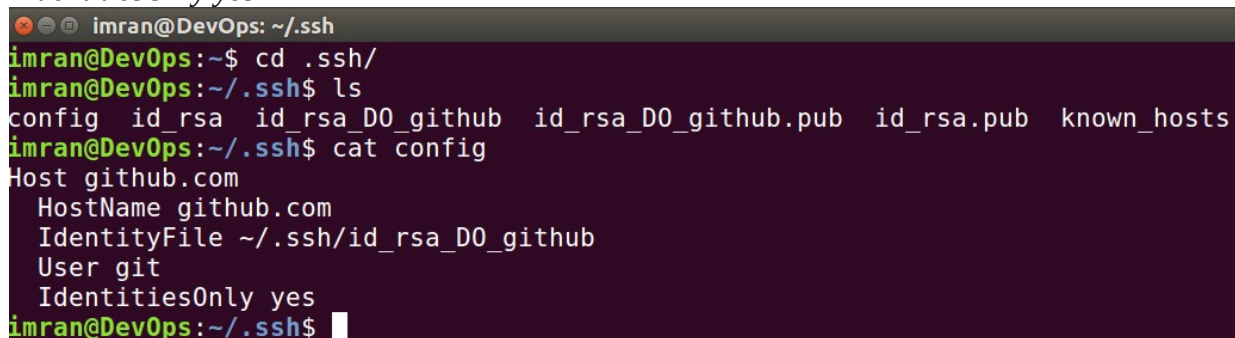
Host github.com

HostName github.com

IdentityFile ~/.ssh/id\_rsa\_DO\_github

User git

IdentitiesOnly yes

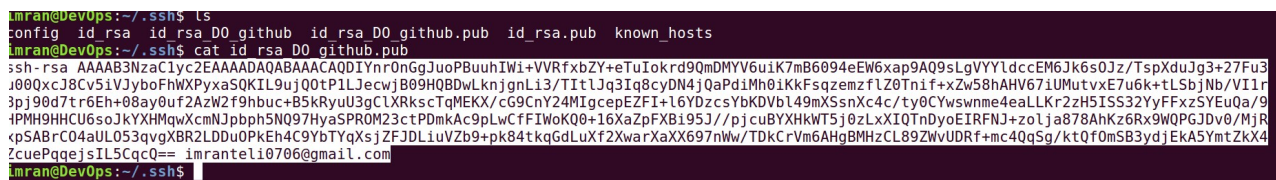
A terminal window titled 'imran@DevOps: ~/.ssh' showing the following commands and output:

```
imran@DevOps:~$ cd .ssh/
imran@DevOps:~/.ssh$ ls
config  id_rsa  id_rsa_DO_github  id_rsa_DO_github.pub  id_rsa.pub  known_hosts
imran@DevOps:~/.ssh$ cat config
Host github.com
  HostName github.com
  IdentityFile ~/.ssh/id_rsa_DO_github
  User git
  IdentitiesOnly yes
imran@DevOps:~/.ssh$
```

## Add SSH public key in github account.

1. Copy public key

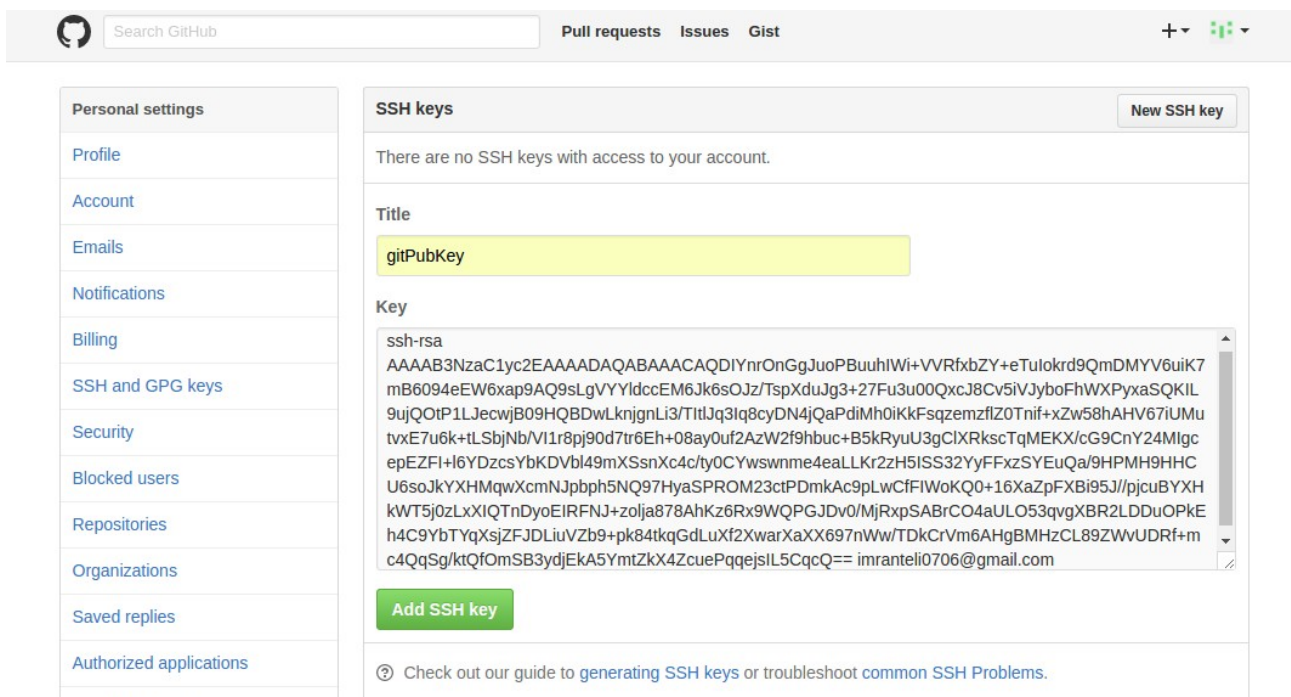
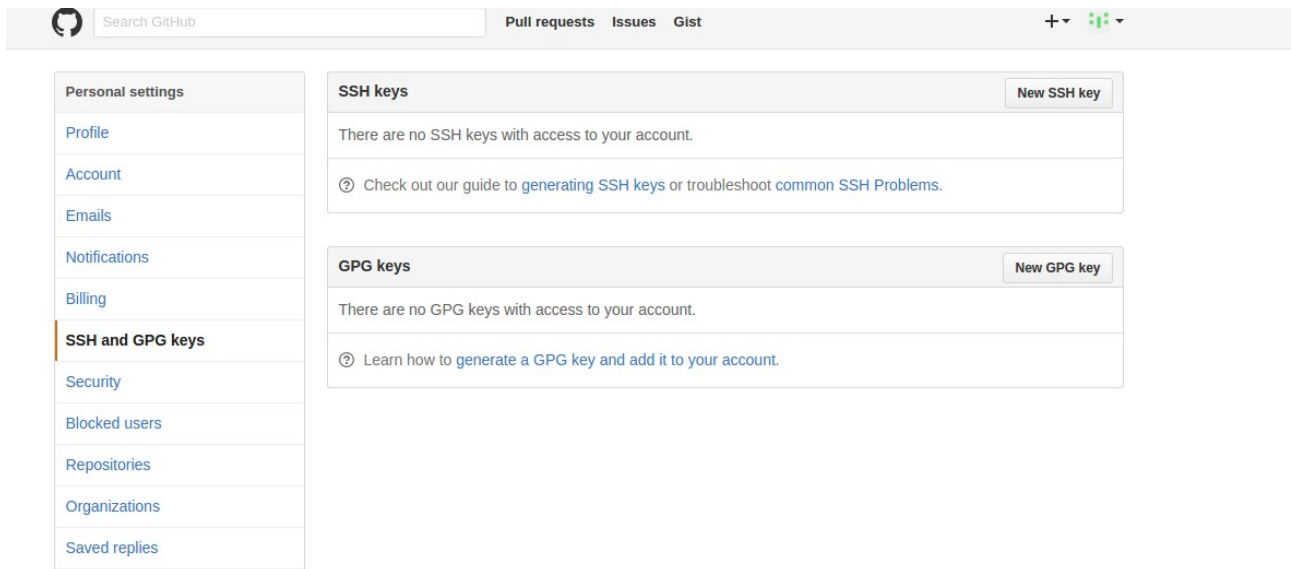
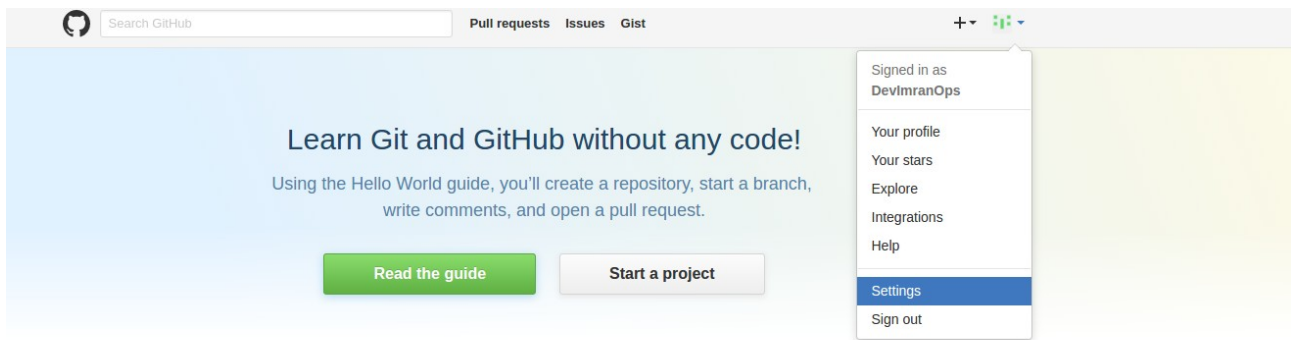
```
# cat ~/.ssh/id_rsa_DO_github.pub
```

A terminal window titled 'imran@DevOps: ~/.ssh' showing the following commands and output:

```
imran@DevOps:~/.ssh$ ls
config  id_rsa  id_rsa_DO_github  id_rsa_DO_github.pub  id_rsa.pub  known_hosts
imran@DevOps:~/.ssh$ cat id_rsa_DO_github.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDAQDIYnR0nGgJuoPBuuhIWi+VVRfxbZY+eTuIokrd9QmDMYV6uiK7mB6094eEW6xap9A09sLgVYYldccEM6Jk6s0Jz/TspXduJg3+27Fu3
J00QxcJ8Cv5iVJyboFhWPyxaSQKIL9uj00tP1LJecwjB09H0BDwLknjgnLi3/TitLJq3Iq8cyDN4jQaPdiMh0iKkFsqzemzfLZ0Tnif+xZw58AHV67iUMutvxE7u6k+tLSbjNb/Vi1r
3pj90d7tr6Eh+08ay0uf2AzW2f9hbuc+B5kRyuU3gCLXRkscTqMEKX/cG9CnY24MIgcepEZF1+l6YDzcsYbKDVbl49mXSsnXc4c/ty0CYwsnme4eaLLKr2zH5ISS32YyFFxzSYEuQa/9
4PMH9HHCU6soJKYXHMqWxcnNjpbph5NQ97HYaSPROM23ctPDmkAc9pLwCffIWoK00+16XaZpFXB195J//pjcubYXHkWT5j0zLxXIQTnDyoEIRFNJ+zolja878AhKz6Rx9WQPGJDv0/MJR
xPSABrC04aUL053qvgXBR2LDu0PkEh4C9YbTYqXsjZFJDLiuVZb9+pk84tkqGdLuXf2XwarXaX697nWw/TDkCrVm6AHgBMH2CL89ZWvUDRf+mc4QqSg/ktQf0mSB3ydjEkA5YmtZkX4
ZcuePqgejsIL5CqcQ== imranteli0706@gmail.com
imran@DevOps:~/.ssh$
```

2. Login to github with the same email id you provided while create ssh keys

Click on settings => SSH and GPG keys => New SSH key => Give a name => Paste the public key content => Add SSH keys



### 3. Test the login

# ssh -T [git@github.com](https://github.com)

You should get a reply as below

*Hi <Username>! You've successfully authenticated, but GitHub does not provide shell access.*

```
imran@DevOps: ~  
imran@DevOps:~$ ssh -T git@github.com  
Warning: Permanently added the RSA host key for IP address '192.30.253.113' to the list of known hosts.  
Hi DevImranOps! You've successfully authenticated, but GitHub does not provide shell access.  
imran@DevOps:~$
```