Chapter No. 1
"Starting a Git Repository"
In this package, you will find:
The author’s biography
A preview chapter from the book, Chapter no.1 "Starting a Git Repository"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Eric Pidoux has a Master's degree in Computer Science from Miage Aix-Marseille and is currently working as a lead web developer at createur.ch, Lausanne, Switzerland. He started learning PHP 10 years ago and is now a Symfony2 ninja who likes coding as well as drinking beer.

He has worked as a technical reviewer on *GitLab Repository Management* and *Extending Symfony2 Web Application Framework*, both by Packt Publishing.

I would like to dedicate this book to my father and thank all my friends and, of course, my awesome family.

For More Information:
Git Best Practices Guide

Git is a decentralized versioning system that was created by Linus Torvalds (also the creator of Linux Kernel) under the GNU license. It was developed to be simple and efficient. Its aim is to manage the content evolution of a file tree.

This book is an easy-to-follow guide to understand the basic to the deepest levels of Git's abilities. As a Git user (beginner or experienced), you will face some basic questions, such as: how do you find the code you changed just a few weeks ago? Is it possible to work with other team members using Git? In case of conflict, how can I resolve it?

*Git Best Practices Guide* will help you to answer these questions by increasing your skills on Git (learning a practical way to use Git commands with examples).

If you are an SVN user, we will also see how it is possible to easily migrate an SVN repository to Git with a step-by-step guide.

Starting with the basics of Git, this book will lead you to the advanced features, making you more self confident when there are merge conflicts or issues while finding content.

The last part of this book will teach you how to improve your workflow using Git. More and more companies or team members use Agile as a workflow process, leaving behind old-fashioned processes such as waterfall, cascade, iterative enhancement, and so on. As a versioning system, Git has to be a part of this process. In this book, we will see how to take your workflow to another level by creating an efficient branching system, using Continuous Integration, and discovering repository managers.

**What This Book Covers**

*Chapter 1, Starting a Git Repository*, covers the basics of Git, describing how to create a repository and start committing files.

*Chapter 2, Working in a Team Using Git*, explains the best practices to work with other developers as a team by pointing out the useful commands.

*Chapter 3, Finding and Resolving Conflicts*, covers all tips and commands that are useful to fix mistakes, resolve conflicts, search inside the commit history, and so on.

*Chapter 4, Going Deeper into Git*, explains the hard commands or not-so-commonly-used commands such as applying patch, using submodules, and migrating from SVN.

*Chapter 5, Using Git for Continuous Integration*, explains how to improve the team workflow by using Continuous Integration.

For More Information:

Starting a Git Repository

This chapter covers the basics needed to understand the topics discussed in this book, and of course, to improve your skills in Git. Commands in this chapter are used every day by all Git users. Some of them will not be explained in detail; they will be explained in another chapter.

In this chapter, you will learn about:

- Initializing a repository
- Cloning an existing repository
- Adding and committing files
- Pushing commits on remote repositories

Configuring Git

Before you start working on Git, you have to configure your name and e-mail by using the following commands:

```
Erik@local:~$ git config --global user.name "Erik"
Erik@local:~$ git config --global user.email erik@domain.com
```

Initializing a new repository

If you want to create a repository in an existing project, just type the following command line:

```
Erik@local:~$ cd myProject
Erik@local:/myProject$ git init .
```

For More Information:
Starting a Git Repository

Otherwise, you have to create an empty directory and type `git init` inside it, as shown:

```
Erik@local:~$ mkdir myProject
Erik@local:~$ cd myProject
Erik@local:~/myProject$ git init
```

This will create a folder named `.git` inside the current directory that contains the following files used by Git:

- **Config**: This is used with the configuration for the local Git repository
- **HEAD**: This lists a file that is the current head branch
- **Refs directory**: This contains references to a commit for a branch

Cloning an existent repository

With Git, it is possible to clone an existent repository to work on it.

There are several possibilities to clone a repository, but the `http`, `git`, and `ssh` protocols are used the most.

If the repository is public, it will create a folder and everything inside the folder. However, if the repository is private or protected, you have to enter an access information or provide a private `ssh` key. For example, if you want to clone a Symfony2 repository, type this line to clone it using `myProjectName` as the folder name:

```
Erik@local:~/myProject$ git clone https://github.com/symfony/symfony.git myProjectName
Initialized empty Git repository in /var/www/myProjectName/.git/
remote: Counting objects: 7820, done.
remote: Compressing objects: 100% (2490/2490), done.
remote: Total 7820 (delta 4610), reused 7711 (delta 4528)
Receiving objects: 100% (7820/7820), 1.40 MiB | 479 KiB/s, done.
Resolving deltas: 100% (4610/4610), done.
Checking out files: 100% (565/565), done.
```

Note that the name after the `clone` command is optional. If there is no parameter after the repository location, the repository name will be used.

For More Information:

You probably read the line about compressing objects. In fact, before sending any content, Git compresses objects to speed the transmission.

We will see more uses of the `clone` command in the next chapter.

**Working with the repository**

We have to take a few minutes to look at the life cycle of a file inside Git.

A file will go through the following states, and the Git command line will take the file from one state to another. We will explain each state and its command line.

![File state diagram]

The important part of this schema is the triangle between the three states `UNMODIFIED`, `MODIFIED`, and `STAGED`. This triangle is an infinite loop. Indeed, every time you change a file, its state is set to modified, and then staged; when you commit the file, it returns to the unmodified state, and so on.

**UNTRACKED** is the first state where the file is created, but this isn't tracked by Git.

To change the state of a file, you have to add it.

---

For More Information:

Adding a file

When you start an empty repository and add a file, it will be in the untracked state, which means that it isn't in the Git repository.

To track a file, you have to execute this command line:

```
Erik@local:~/myProject$ touch MyFileName.txt
Erik@local:~/myProject$ echo "test" > MyFileName.txt
Erik@local:~/myProject$ git add MyFileName.txt
```

So, your file is now tracked by Git.

If you want to add all files because you already have something inside the directory while you create the repository, add a period (.) just after `git add` to specify to take all files inside the current directory:

```
Erik@local:~/myProject$ echo "hello" > MyFile2.txt
Erik@local:~/myProject$ echo "hello" > MyFile3.txt
Erik@local:~/myProject$ git add .
```

The file is currently staged and ready to be committed inside the repository.

Committing a file

As soon as your file is tracked, all changes will be notified by Git, and you have to commit the change on the repository.

```
Remember to commit your change as soon as possible (not for every line, but it's a marker to validate what you have done).
```

The `commit` command is local to your own repository, nobody except you can see it.

The `commit` command line offers various options. For example, you can commit a file, as shown in the following example:

```
Erik@local:~/myProject$ git commit -m 'This message explains the changes' MyFileName.txt
```

To commit everything, use the following command:

```
Erik@local:~/myProject$ git commit -m 'My commit message'
```

For More Information:

You will create a new commit object in the Git repository. This commit is referenced by an SHA-1 checksum and includes various data (content files, content directories, the commit history, the committer, and so on). You can show this information by executing the following command line:

```
Erik@local:~/myProject$ git log
```

It will display something similar to the following:

```
Commit f658e5f22afel2ce75cde1h671b58d6703ab83f5
Author: Eric Pidoux <contact@eric-pidoux.com>
Date: Mon Jun 2 22:54:04 2014 +0100
My commit message
```

The file is in the unmodified state because you just committed the change; you can push the files in the remote repository.

## Pushing a file

Once committed, you can push the files in the remote repository. It can be on a bare repository, using `init` with the `git init --bare` command, so just type the following command:

```
Erik@local:~/myProject$ git push /home/erik/remote-repository.git
```

If you create a remote repository on another server, you have to configure your local Git repository.

If you use Git 2.0 or later, the previous command will print out something like this on the screen:

```
Warning: push.default is unset; its implicit value is changing in Git 2.0 from 'matching' to 'simple'. To squelch this message and maintain the current behavior after the default changes, use:
gitconfig --global push.default matching
To squelch this message and adopt the new behavior now, use:
gitconfig --global push.default simple
```

The 'matching' value from the `push.default` configuration variable denotes that `git push` will push all your local branches to the branches with the same name on the remote. This makes it easy to accidentally push a branch you didn't intend to.

---

For More Information:

Starting a Git Repository

The 'simple' value from the `push.default` configuration variable denotes that `git push` will push only the current branch to the branch that `git pull` will pull from; it also checks that their names match. This is a more intuitive behavior, which is why the default should be changed to this configuration value.

Firstly, check if a remote repository is defined:

```
Erik@local:~/myProject$ git remote
```

If it's not, define the remote repository named `origin`:

```
Erik@local:~/myProject$ git remote add origin http://github.com/myRepoAddress.git
```

Now, push the changes using the following command:

```
Erik@local:~/myProject$ git push -u origin master
```

After this, you will have a resume of what was pushed.

In fact, the remote repository will check the current Head (the reference to the commit) and compare it with its own. If there are differences between them, it will fail.

Removing a file

If you don't want a file anymore, there are two ways to remove it:

- Delete the file manually and commit the changes. This will delete the file locally and on the repository. Use the following command line

```
Erik@local:~/myProject$ git commit -m 'delete this file'
```

- Delete the file only through Git:

```
Erik@local:~/myProject$ git rm --cached MyFileName.txt
```

Checking the status

There is a way to display the working tree status, that is, the files that have changed and those that need to be pushed, and of course, there is a way to display the conflicts:

```
Erik@local:~/myProject$ git status
```

If everything is correct and up to date, you will get this result:

```
Erik@local:~/myProject$ git status
# On branch master
nothing to commit, working directory clean
```

For More Information:

If you add a file, Git will warn you to track it by using the `git add` command:

```
Erik@local:~/myProject$ touch text5.txt
Erik@local:~/myProject$ git status
# On branch master
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
#   text5.txt
nothing added to commit but untracked files present (use "git add" to track)
```

If you edit `MyFile2.txt` and type `git status` again, then you will have new lines:

```
Erik@local:~/myProject$ echo "I am changing this file" > MyFile2.txt
Erik@local:~/myProject$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#   new file: text5.txt
#
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
#
# modified: MyFile2.txt
```

On these lines, separate paragraphs display all files in each state. The `MyFile2.txt` file is not tracked by Git and `text5.txt` is ready to be committed.

If you add `text5.txt` using the `git add` command, you will notice the following changes:

```
Erik@local:~/myProject$ git add MyFile2.txt
Erik@local:~/myProject$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#   new file: text5.txt
# modified:   MyFile2.txt
```

For More Information:

Starting a Git Repository

Ignoring files
Git can easily ignore some files or folders from your working tree. For example, consider a website on which you are working, and there is an upload folder that you might not push on the repository to avoid having test images in your repository.

To do so, create a .gitignore file inside the root of your working tree:

```
Erik@local:~/myProject$ touch .gitignore
```

Then, add this line in the file; it will untrack the upload folder and its contents:

```
upload
```

Files or folders you define in this file will not be tracked by Git anymore.

You can add some easy regex, such as the following:

- If you want to ignore all PHP files, use the following regex:
  ```
  *.php
  ```
- If you want to ignore all files having p or l at the end of its name, use the following regex:
  ```
  *.\[pl\]
  ```
- If you want to ignore all temporary files (finishing by -), use the following regex:
  ```
  *
  ```

If the file is already pushed on the repository, the file is tracked by Git. To remove it, you will have to use the `git rm` command line by typing this:

```
Erik@local:~/myProject$ git rm --cached MyFileName.txt
```

Summary
In this chapter, we saw the basics of Git: how to create a Git repository, how to put content in it, and how to push data to a remote repository.

In the next chapter, we will see how to use Git with a team and manage all interactions with a remote repository.
Where to buy this book

You can buy Git Best Practices Guide from the Packt Publishing website:

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.