



Free Sample

C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Mastering Git

Attain expert-level proficiency with Git for enhanced productivity and efficient collaboration by mastering advanced distributed version control features

Jakub Narębski

[PACKT]
PUBLISHING

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 '**Git Basics in Practice**'
- A synopsis of the book's content
- More information on **Mastering Git**

About the Author

Jakub Narębski followed Git development from the very beginning of its creation. He is one of the main contributors to the gitweb subsystem (the original web interface for Git), and is an unofficial gitweb maintainer. He created, announced, and analyzed annual Git User's Surveys from 2007 till 2012 – all except the first one (you can find his analysis of those surveys on the Git Wiki). He shares his expertise with the technology on the StackOverflow question-and-answer website.

He was one of the proofreaders of the *Version Control by Example* by Eric Sink, and was the reason why it has chapter on Git.

He is an assistant professor in the faculty of mathematics and computer science at the Nicolaus Copernicus University in Toruń, Poland. He uses Git as a version control system of choice both for personal and professional work, teaching it to computer science students as a part of their coursework.

Preface

Mastering Git is meticulously designed to help you gain deeper insights into Git's architecture and its underlying concepts, behavior, and best practices.

Mastering Git starts with a quick implementation example of using Git for the collaborative development of a sample project to establish the foundation knowledge of Git's operational tasks and concepts. Furthermore, as you progress through the book, subsequent chapters provide detailed descriptions of the various areas of usage: from the source code archaeology, through managing your own work, to working with other developers. Version control topics are accompanied by in-detail description of relevant parts of Git architecture and behavior.

This book also helps augment your understanding to examine and explore your project's history, create and manage your contributions, set up repositories and branches for collaboration in centralized and distributed workflows, integrate work coming from other developers, customize and extend Git, and recover from repository errors. By exploring advanced Git practices, and getting to know details of Git workings, you will attain a deeper understanding of Git's behavior, allowing you to customize and extend existing recipes, and write your own.

What this book covers

Chapter 1, Git Basics in Practice, serves as a reminder of version control basics with Git. The focus will be on providing the practical aspects of the technology, showing and explaining basic version control operations for the development of an example project, and the collaboration between two developers.

Chapter 2, Exploring Project History, introduces the concept of the Directed Acyclic Graph (DAG) of revisions and explains how this concept relates to the ideas of branches, tags, and the current branch in Git. You will learn how to select, filter, and view the range of revisions in the history of a project, and how to find revisions using different criteria.

Chapter 3, Developing with Git, describes how to create such history and how to add to it. You will learn how to create new revisions and new lines of development. This chapter introduces the concept of the staging area for commits (the index), and explains how to view and read differences between the working directory, the index, and the current revision.

Chapter 4, Managing Your Worktree, focuses on explaining how to manage the working directory (the worktree) to prepare contents for a new commit. This chapter will teach the reader how to manage their files in detail. It will also show how to manage files that require special handling, introducing the concepts of ignored files and file attributes.

Chapter 5, Collaborative Development with Git, presents a bird's eye view of the various ways to collaborate, showing different centralized and distributed workflows. It will focus on the repository-level interactions in collaborative development. You will also learn here the concept of the chain of trust, and how to use signed tags, signed merges, and signed commits.

Chapter 6, Advanced Branching Techniques, goes deeper into the details of collaboration in a distributed development. It explores the relations between local branches and branches in remote repositories, and describes how to synchronize branches and tags. You will learn here branching techniques, getting to know various ways of utilizing different types of branches for distinct purposes (including topic branch workflow).

Chapter 7, Merging Changes Together, teaches you how to merge together changes from different parallel lines of development (that is, branches) using merge and rebase. This chapter will also explain the different types of merge conflicts, how to examine them, and how to resolve them. You will learn how to copy changes with cherry-pick, and how to apply a single patch and a patch series.

Chapter 8, Keeping History Clean, explains why one might want to keep clean history, when it can and should be done, and how it can be done. Here you will find step-by-step instructions on how to reorder, squash, and split commits. This chapter also demonstrates how can one recover from a history rewrite, and explains what to do if one cannot rewrite history: how to revert the effect of commit, how to add a note to it, and how to change the view of project's history.

Chapter 9, Managing Subprojects – Building a Living Framework, explains and shows different ways to connect different projects in the one single repository of the framework project, from the strong inclusion by embedding the code of one project in the other (subtrees), to the light connection between projects by nesting repositories (submodules). This chapter also presents various solutions to the problem of large repositories and of large files.

Chapter 10, Customizing and Extending Git, covers configuring and extending Git to fit one's needs. You will find here details on how to set up command line for easier use, and a short introduction to graphical interfaces. This chapter explains how to automate Git with hooks (focusing on client-side hooks), for example, how to make Git check whether the commit being created passes specified coding guidelines.

Chapter 11, Git Administration, is intended to help readers who are in a situation of having to take up the administrative side of Git. It briefly touches the topic of serving Git repositories. Here you will learn how to use server-side hooks for logging, access control, enforcing development policy, and other purposes.

Chapter 12, Git Best Practices, presents a collection of version control generic and Git-specific recommendations and best practice. Those cover issues of managing the working directory, creating commits and a series of commits (pull requests), submitting changes for inclusion, and the peer review of changes.

1

Git Basics in Practice

This book is intended for novice and advanced Git users to help them on their road to mastering Git. Therefore the following chapters will assume that the reader knows the basics of Git, and has advanced past the beginner stage.

This chapter will serve as a reminder of version control basics with Git. The focus will be on providing practical aspects of the technology, showing and explaining basic version control operations in the example of the development of a sample project, and collaboration between two developers.

In this chapter we will recall:

- Setting up a Git environment and Git repository (`init`, `clone`)
- Adding files, checking status, creating commits, and examining the history
- Interacting with other Git repositories (`pull`, `push`)
- How to resolve a merge conflict
- Creating and listing branches, switching to a branch, and merging
- How to create a tag

An introduction to version control and Git

A **version control system** (sometimes called **revision control**) is a tool that lets you track the history and attribution of your project files over time (stored in a **repository**), and which helps the developers in the team to work together. Modern version control systems help them work simultaneously, in a non-blocking way, by giving each developer his or her own sandbox, preventing their work in progress from conflicting, and all the while providing a mechanism to merge changes and synchronize work.

Distributed version control systems such as Git give each developer his or her own copy of the project's history, a **clone** of a repository. This is what makes Git fast: nearly all operations are performed locally, and are flexible: you can set up repositories in many ways. Repositories meant for developing also provide a separate **working area** (or a **worktree**) with project files for each developer. The branching model used by Git enables cheap local branching and flexible branch publishing, allowing to use branches for context switching and for sandboxing different works in progress (making possible, among other things, a very flexible **topic branch** workflow).

The fact that the whole history is accessible allows for long-term undo, rewinding back to last working version, and so on. Tracking ownership of changes automatically makes it possible to find out who was responsible for any given area of code, and when each change was done. You can compare different revisions, go back to the revision a user is sending a bug report against, and even automatically find out which revision introduced a regression bug. The fact that Git is tracking changes to the tips of branches with **reflog** allows for easy undo and recovery.

A unique feature of Git is that it enables explicit access to the staging area for creating commits (new revisions of a project). This brings additional flexibility to managing your working area and deciding on the shape of a future commit.

All this flexibility and power comes at a cost. It is not easy to master using Git, even though it is quite easy to learn its basic use. This book will help you attain this expertise, but let's start with a reminder about basics with Git.

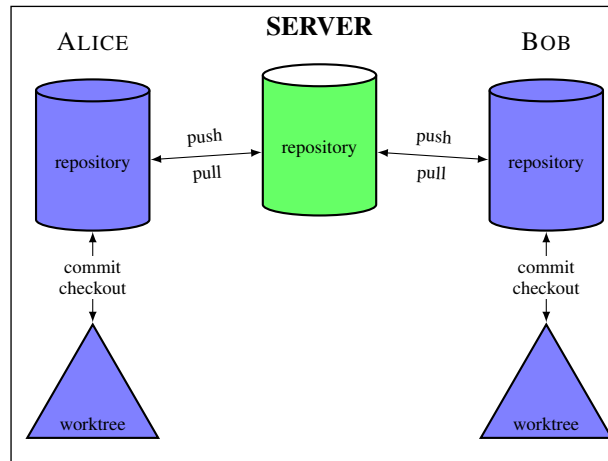
Git by example

Let's follow step by step a simple example of two developers using Git to work together on a simple project. You can download the example code files from <http://www.packtpub.com>. You can find there all three repositories (for two developers, and the bare server repository) with the example code files for this chapter, where you can examine code, history, and reflog..


Repository setup

A company has begun work on a new product. This product calculates a random number – an integer value of specified range.

The company has assigned two developers to work on this new project, Alice and Bob. Both developers are telecommuting to the company's corporate headquarters. After a bit of discussion, they have decided to implement their product as a command-line app in C, and to use Git 2.5.0 (<http://git-scm.com/>) for version control. This project and the code are intended for demonstration purposes, and will be much simplified. The details of code are not important here – what's important is how the code changes:



With a small team, they have decided on the setup shown in the preceding diagram.


 This is one possible setup, with the central *canonical* repository, and without a dedicated maintainer responsible for this repository (all developers are equal in this setup). It is not the only possibility; other ways of configuring repositories will be shown in *Chapter 5, Collaborative Development with Git*.

Creating a Git repository

Alice gets the project started by asking Carol, an administrator, to create a new repository specifically for collaborating on a project, to share work with all the team:



Command line examples follow the Unix convention of having `user@host` and directory in the command prompt, to know from the first glance who performs a command, on what computer, and in which directory. This is the usual setup on Unix (for example, on Linux).

You can configure your command prompt to show Git-specific information like the name of the repository name, the subdirectory within the repository, the current branch, and even the status of the working area, see *Chapter 10, Customizing and Extending Git*.

```
carol@server ~$ mkdir -p /srv/git
carol@server ~$ cd /srv/git
carol@server /srv/git$ git init --bare random.git
```



I consider the details of server configuration to be too much for this chapter. Just imagine that it happened, and nothing went wrong. Or take a look at *Chapter 11, Git Administration*.

You can also use a tool to manage Git repositories (for example Gitolite); creating a public repository on a server would then, of course, look different. Often though it involves creating a Git repository with `git init` (without `--bare`) and then pushing it with an explicit URI to the server, which then automatically creates the public repository.

Or perhaps the repository was created through the web interface of tools, such as GitHub, Bitbucket, or GitLab (either hosted or on-premise).

Cloning the repository and creating the first commit

Bob gets the information that the project repository is ready, and he can start coding.

Since this is Bob's first time using Git, he first sets up his `~/.gitconfig` file with information that will be used to identify his commits in the log:

```
[user]
  name = Bob Hacker
  email = bob@company.com
```

Now he needs to get his own repository instance:

```
bob@hostB ~$ git clone https://git.company.com/random
Cloning into random...
Warning: You appear to have cloned an empty repository.
```

done.

```
bob@hostB ~$ cd random
```

```
bob@hostB random$
```



All examples in this chapter use the command-line interface. Those commands might be given using a Git GUI or IDE integration. The *Git: Version Control for Everyone* book, published by Packt Publishing, shows GUI equivalents for the command-line.

Bob notices that Git said that it is an empty repository, with no source code yet, and starts coding. He opens his text editor and creates the starting point for their product:

```
#include <stdio.h>
#include <stdlib.h>

int random_int(int max)
{
    return rand() % max;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int max = atoi(argv[1]);

    int result = random_int(max);
    printf("%d\n", result);

    return EXIT_SUCCESS;
}
```

Typically, like for most initial implementations, this version is missing a lot of features. But it's a good place to begin. Before committing his code, Bob wants to make sure that it compiles and runs:

```
bob@hostB random$ gcc -std=c99 random.c
```

```
bob@hostB random$ ls -l
```

```
total 43
```

```
-rwxr-xr-x  1 bob    staff   86139  May 29 17:36 a.out
-rw-r--r--  1 bob    staff    331   May 19 17:11 random.c
bob@hostB random$ ./a.out
Usage: ./a.out <number>
bob@hostB random$ ./a.out 10
1
```

Alright! It's time to add this file to the repository:

```
bob@hostB random$ git add random.c
```

Bob uses the status operation to make sure that the pending changeset (the future commit) looks proper:



We use here a short form of the `git status` to reduce the amount of space taken by examples; you can find an example of full status output further in the chapter.

```
bob@hostB random$ git status -s
A  random.c
?? a.out
```

Git is complaining because it does not know what to do about the `a.out` file: it is neither tracked nor ignored. That's a compiled executable, which as a generated file should not be stored in a version control repository. Bob can just ignore that issue for the time being.

Now it's time to commit the file:

```
bob@hostB random$ git commit -a -m "Initial implementation"
[master (root-commit) 2b953b4] Initial implementation
 1 file changed, 22 insertions(+)
Create mode 100644 random.c
```




Normally, you would create a commit message not by using the `-m <message>` command-line option, but by letting Git open an editor. We use this form here to make examples more compact.

The `-a / --all` option means to take all changes to the tracked files; you can separate manipulating the staging area from creating a commit — this is however a separate issue, left for *Chapter 4, Managing Your Worktree*.

Publishing changes

After finishing working on the initial version of the project, Bob decides that it is ready to be published (to be made available for other developers). He pushes the changes:

```
bob@hostB random$ git push
warning: push.default is unset; its implicit value has changed in
Git 2.0 from 'matching' to 'simple'. To squelch this message [...]
To https://git.company.com/random
* [new branch]      master -> master
bob@hostB random$ git config --global push.default simple
```

 Note that, depending on the speed of network, Git could show progress information during remote operations such as `clone`, `push`, and `fetch`. Such information is omitted from examples in this book, except where that information is actually discussed while examining history and viewing changes.

Examining history and viewing changes

Since it is Alice's first time using Git on her desktop machine, she first tells Git how her commits should be identified:

```
alice@hostA ~$ git config --global user.name "Alice Developer"
alice@hostA ~$ git config --global user.email alice@company.com
```

Now Alice needs to set up her own repository instance:

```
alice@hostA ~$ git clone https://git.company.com/random
Cloning into random...
done.
```

Alice examines the working directory:

```
alice@hostA ~$ cd random
alice@hostA random$ ls -al
total 1
drwxr-xr-x  1 alice staff    0 May 30 16:44 .
drwxr-xr-x  4 alice staff    0 May 30 16:39 ..
drwxr-xr-x  1 alice staff    0 May 30 16:39 .git
-rw-r--r--  1 alice staff 353 May 30 16:39 random.c
```



The `.git` directory contains Alice's whole copy (clone) of the repository in Git internal format, and some repository-specific administrative information. See `gitrepository-layout(5)` manpage for details of the file layout, which can be done for example with `git help repository-layout` command.

She wants to check the `log` to see the details (to examine the project history):

```
alice@hostA random$ git log
commit 2b953b4e80abfb77bdcd94e74dedeeebf6aba870
Author: Bob Hacker <bob@company.com>
Date:   Thu May 29 19:53:54 2015 +0200
```

Initial implementation



Naming revisions:

At the lowest level, a Git version identifier is a SHA-1 hash, for example `2b953b4e80`. Git supports various forms of referring to revisions, including the unambiguously shortened SHA-1 (with a minimum of four characters) — see *Chapter 2, Exploring Project History*, for more ways.

When Alice decides to take a look at the code, she immediately finds something horrifying. The random number generator is never initialized! A quick test shows that the program always generates the same number. Fortunately, it is only necessary to add one line to `main()`, and the appropriate `#include`:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int random_int(int max)
{
    return rand() % max;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number>\n", argv[0]);
        return EXIT_FAILURE;
    }
}
```

```

    int max = atoi(argv[1]);

    srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);

    return EXIT_SUCCESS;
}

```

She compiles the code, and runs it a few times to check that it really generates random numbers. Everything looks alright, so she uses the `status` operation to see the pending changes:

```

alice@hostA random$ git status -s
M random.c

```

No surprise here. Git knows that `random.c` has been modified. She wants to double-check by reviewing the actual changes with the `diff` command:



From here on, we will not show untracked files, unless they are relevant to the topic being discussed; let's assume that Alice set up an ignore file, as described in *Chapter 4, Managing Your Worktree*.

```

alice@hostA random$ git diff
diff --git a/random.c b/random.c
index cc09a47..5e095ce 100644
--- a/random.c
+++ b/random.c
@@ -1,5 +1,6 @@
    #include <stdio.h>
    #include <stdlib.h>
+   #include <time.h>

    int random_int(int max)
    {
@@ -15,6 +16,7 @@ int main(int argc, char *argv[])

    int max = atoi(argv[1]);

+   srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);

```

Now it's time to commit the changes and push them to the public repository:

```
alice@hostA random$ git commit -a -m "Initialize random number generator"
[master db23d0e] Initialize random number generator
1 file changed, 2 insertions(+)
alice@hostA random$ git push
To https://git.company.com/random
3b16f17..db23d0e master -> masterRenaming and moving files
```

Renaming and moving files

Bob moves on to his next task, which is to restructure the tree a bit. He doesn't want the top level of the repository to get too cluttered so he decides to move all their source code files into a `src/` subdirectory:

```
bob@hostA random$ mkdir src
bob@hostA random$ git mv random.c src/
bob@hostA random$ git status -s
R random.c -> src/random.c
bob@hostA random$ git commit -a -m "Directory structure"
[master 69e0d3d] Directory structure
1 file changed, 0 insertions(+), 0 deletions(-)
rename random.c => src/random.c (100%)
```

While at it, to minimize the impact of reorganization on the `diff` output, he configures Git to always use rename and copy detection:

```
bob@hostB random$ git config --global diff.renames copies
```

Bob then decides the time has come to create a proper Makefile, and to add a README for a project:

```
bob@hostA random$ git add README Makefile
bob@hostA random$ git status -s
A Makefile
A README
bob@hostA random$ git commit -a -m "Added Makefile and README"
[master abfee4] Added Makefile and README
2 files changed, 15 insertions(+)
create mode 100644 Makefile
create mode 100644 README
```


Bob decides to rename `random.c` to `rand.c`:

```
bob@hostA random$ git mv src/random.c src/rand.c
```

This of course also requires changes to the Makefile:

```
bob@hostA random$ git status -s
M Makefile
R src/random.c -> src/rand.c
```

He then commits those changes.

Updating your repository (with merge)

Reorganization done, now Bob tries to publish those changes:

```
bob@hostA random$ git push
$ git push
To https://git.company.com/random
 ! [rejected]          master -> master (fetch first)
error: failed to push some refs to 'https://git.company.com/random'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository
hint: pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

But Alice was working at the same time and she had her change ready to commit and push first. Git is not allowing Bob to publish his changes because Alice has already pushed something to the master branch, and Git is preserving her changes.



Hints and advices in Git command output will be skipped from here on for the sake of brevity.

Bob uses `pull` to bring in changes (as described in hint in the command output):

```
bob@hostB random $ git pull
From https://git.company.com/random
+ 3b16f17...db23d0e master      -> origin/master
```

```
Auto-merging src/rand.c
Merge made by the 'recursive' strategy.
 src/rand.c | 2 ++
 1 file changed, 2 insertions(+)
```

Git pull fetched the changes, automatically merged them with Bob's changes, and committed the merge.

Everything now seems to be good:

```
bob@hostB random$ git show
commit ba5807e44d75285244e1d2eacb1c10cbc5cf3935
Merge: 3b16f17 db23d0e
Author: Bob Hacker <bob@company.com>
Date: Sat May 31 20:43:42 2015 +0200
```

```
Merge branch 'master' of https://git.company.com/random
```

The merge commit is done. Apparently, Git was able to merge Alice's changes directly into Bob's moved and renamed copy of a file without any problems. Marvelous!

Bob checks that it compiles (because *automatically merged* does not necessarily mean that the merge output is correct), and is ready to push the merge:

```
bob@hostB random$ git push
To https://git.company.com/random
 db23d0e..ba5807e master -> master
```

Creating a tag

Alice and Bob decide that the project is ready for wider distribution. Bob creates a tag so they can more easily access/refer to the released version. He uses an **annotated tag** for this; an often used alternative is to use **signed tag**, where the annotation contains a PGP signature (which can later be verified):

```
bob@hostB random$ git tag -a -m "random v0.1" v0.1
bob@hostB random$ git tag --list
```

v0.1

```
bob@hostB random$ git log -1 --decorate --abbrev-commit
commit ba5807e (HEAD -> master, tag: v0.1, origin/master)
Merge: 3b16f17 db23d0e
Author: Bob Hacker <bob@company.com>
Date: Sat May 31 20:43:42 2015 +0200
```

Merge branch 'master' of https://git.company.com/random

Of course, the v0.1 tag wouldn't help if it was only in Bob's local repository. He therefore pushes the just created tag:

```
bob@hostB random$ git push origin tag v0.1
Counting objects: 1, done.
Writing objects: 100% (1/1), 162 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
To https://git.company.com/random
* [new tag] v0.1 -> v0.1
```

Alice updates her repository to get the v0.1 tag, and to start with up-to-date work:

```
alice@hostA random$ git pull
From https://git.company.com/random
f4d9753..be08dee master -> origin/master
* [new tag] v0.1 -> v0.1
Updating f4d9753..be08dee
Fast-forward
 Makefile          | 11 ++++++++
 README           |  4 +++
 random.c => src/rand.c |  0
3 files changed, 15 insertions(+)
create mode 100644 Makefile
create mode 100644 README
rename random.c => src/rand.c (100%)
```

Resolving a merge conflict

Alice decides that it would be a good idea to extract initialization of a pseudo-random numbers generator into a separate subroutine. This way, both initialization and generating random numbers are encapsulated, making future changes easier.

She codes and adds `init_rand()`:

```
void init_rand(void)
{
    srand(time(NULL));
}
```

Grand! Let's see that it compiles.

```
alice@hostA random$ make
gcc -std=c99 -Wall -Wextra -o rand src/rand.c
alice@hostA random$ ls -F
Makefile  rand*  README  src/
```

Good. Time to commit the change:

```
alice@hostA random$ git status -s
M src/rand.c
alice@hostA random$ git commit -a -m "Abstract RNG initialization"
[master 26f8e35] Abstract RNG initialization
1 files changed, 6 insertions(+), 1 deletion(-)
```

No problems here.

Meanwhile, Bob notices that the documentation for the `rand()` function used says that it is a weak pseudo-random generator. On the other hand, it is a standard function, and it might be enough for the planned use:

```
bob@hostB random$ git pull
Already up-to-date.
```

He decides to add a note about this issue in a comment:

```
bob@hostB random$ git status -s
M src/rand.c
bob@hostB random$ git diff
diff --git a/src/rand.c b/src/rand.c
index 5e095ce..8fddf5d 100644
```

```
--- a/src/rand.c
+++ b/src/rand.c
@@ -2,6 +2,7 @@
#include <stdlib.h>
#include <time.h>

+// TODO: use a better random generator
int random_int(int max)
{
    return rand() % max;
```

He has his change ready to commit and push first:

```
bob@hostB random$ git commit -m 'Add TODO comment for random_int()'
[master 8c4ceca] Use Add TODO comment for random_int()
1 files changed, 1 insertion(+)
bob@hostB random$ git push
To https://git.company.com/random
ba5807e..8c4ceca master -> master
```

So when Alice is ready to push her changes, Git rejects it:

```
alice@hostA random$ git push
To https://git.company.com/random
! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'https://git.company.com/random'
[...]
```

Ah. Bob must have pushed a new changeset already. Alice once again needs to pull and merge to combine Bob's changes with her own:

```
alice@hostA random$ git pull
From https://git.company.com/random
ba5807e..8c4ceca master -> origin/master
Auto-merging src/rand.c
CONFLICT (content): Merge conflict in src/rand.c
Automatic merge failed; fix conflicts and then commit the result.
```

The merge didn't go quite as smoothly this time. Git wasn't able to automatically merge Alice's and Bob's changes. Apparently, there was a conflict. Alice decides to open the `src/rand.c` file in her editor to examine the situation (she could have used a graphical merge tool via `git mergetool` instead):

```
<<<<<<< HEAD
void init_rand(void)
{
    srand(time(NULL));
}

=====
// TODO: use a better random generator
>>>>>>> 8c4ceca59d7402fb24a672c624b7ad816cf04e08
int random_int(int max)
```

Git has included both Alice's code (between `<<<<<<< HEAD` and `=====` conflict markers) and Bob's code (between `=====` and `>>>>>>>`). What we want as a final result is to include both blocks of code. Git couldn't merge it automatically because those blocks were not separated. Alice's `init_rand()` function can be simply included right before Bob's added comment. After resolution, the changes look like this:

```
alice@hostA random$ git diff
diff --cc src/rand.c
index 17ad8ea,8fddf5d..0000000
--- a/src/rand.c
+++ b/src/rand.c
@@@ -2,11 -2,7 +2,12 @@@
    #include <stdlib.h>
    #include <time.h>

+void init_rand(void)
+{
+    srand(time(NULL));
+}
+
```

```
+ // TODO: use a better random generator
    int random_int(int max)
    {
        return rand() % max;
```

That should take care of the problem. Alice compiles the code to make sure and then commits the merge:

```
alice@hostA random$ git status -s
UU src/rand.c
alice@hostA random$ git commit -a -m 'Merge: init_rand() + TODO'
[master 493e222] Merge: init_rand() + TODO
```

And then she retries the push:

```
alice@hostA random$ git push
To https://git.company.com/random
8c4ceca..493e222 master -> master
```

And... done.

Adding files in bulk and removing files

Bob decides to add a `COPYRIGHT` file with a copyright notice for the project. There was also a `NEWS` file planned (but not created), so he uses a bulk add to add all the files:

```
bob@hostB random$ git add -v
add 'COPYRIGHT'
add 'COPYRIGHT~'
```

Oops. Because Bob didn't configure his **ignore patterns**, the backup file `COPYRIGHT~` was caught too. Let's remove this file:

```
bob@hostB random$ git status -s
A  COPYRIGHT
A  COPYRIGHT~
bob@hostB random$ git rm COPYRIGHT~
error: 'COPYRIGHT~' has changes staged in the index
(use --cached to keep the file, or -f to force removal)
bob@hostB random$ git rm -f COPYRIGHT~
rm 'COPYRIGHT~'
```

Let's check the status and commit the changes:

```
bob@hostB random$ git status -s
A  COPYRIGHT
bob@hostB random$ git commit -a -m 'Added COPYRIGHT'
[master ca3cdd6] Added COPYRIGHT
 1 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 COPYRIGHT
```

Undoing changes to a file

A bit bored, Bob decides to indent `rand.c` to make it follow a consistent coding style convention:

```
bob@hostB random$ indent src/rand.c
```

He checks how much source code it changed:

```
bob@hostB random$ git diff --stat
src/rand.c | 40 ++++++-----
1 files changed, 22 insertions(+), 18 deletions(-)
```

That's too much (for such a short file). It could cause problems with merging. Bob calms down and undoes the changes to `rand.c`:

```
bob@hostB random$ git status -s
M src/rand.c
bob@hostB random$ git checkout -- src/rand.c
bob@hostB random$ git status -s
```



If you don't remember how to revert a particular type of change, or to update what is to be committed (using `git commit` without `-a`), the output of `git status` (without `-s`) contains information about what commands to use. This is shown as follows:

```
bob@hostB random$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changed but not updated:
```



```
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
# modified:   src/rand.c
```

Creating a new branch

Alice notices that using a modulo operation to return random numbers within a given span does not generate uniformly distributed random numbers, since in most cases it makes lower numbers slightly more likely. She decides to try to fix this issue. To isolate this line of development from other changes, she decides to create her own named branch (see also *Chapter 6, Advanced Branching Techniques*), and switch to it:

```
alice@hostA random$ git checkout -b better-random
Switched to a new branch 'better-random'
alice@hostA random$ git branch
* better-random
  master
```



Instead of using the `git checkout -b better-random` shortcut to create a new branch and switch to it in one command invocation, she could have first created a branch with `git branch better-random`, then switched to it with `git checkout better-random`.

She decides to shrink the range from `RAND_MAX` to the requested number by rescaling the output of `rand()`. The changes look like this:

```
alice@hostA random$ git diff
diff --git a/src/rand.c b/src/rand.c
index 2125b0d..5ded9bb 100644
--- a/src/rand.c
+++ b/src/rand.c
@@ -10,7 +10,7 @@ void init_rand(void)
 // TODO: use a better random generator
 int random_int(int max)
 {
```

```
-         return rand() % max;
+         return rand()*max / RAND_MAX;
}
```

```
int main(int argc, char *argv[])
```

She commits her changes, and pushes them, knowing that the push will succeed because she is working on her private branch:

```
alice@hostA random$ git commit -a -m 'random_int: use rescaling'
[better-random bb71a80] random_int: use rescaling
1 files changed, 1 insertion(+), 1 deletion(-)
alice@hostA random$ git push
fatal: The current branch better-random has no upstream branch.
To push the current branch and set the remote as upstream, use
```

```
git push --set-upstream origin better-random
```

Alright! Git just wants Alice to set up a remote origin as the upstream for the newly created branch (it is using a simple push strategy); this will also push this branch explicitly.

```
alice@hostA random$ git push --set-upstream origin better-random
To https://git.company.com/random
* [new branch]        better-random -> better-random
```



If she wants to make her branch visible, but private (so nobody but her can push to it), she needs to configure the server with hooks, or use Git repository management software such as Gitolite to manage it for her.

Merging a branch (no conflicts)

Meanwhile, over in the default branch, Bob decides to push his changes by adding the COPYRIGHT file:

```
bob@hostB random$ git push
To https://git.company.com/random
! [rejected]        master -> master (non-fast-forward)
[...]
```

OK. Alice was busy working at extracting random number generator initialization into a subroutine (and resolving a merge conflict), and she pushed the changes first:

```
bob@hostB random$ git pull
From https://git.company.com/random
   8c4ceca..493e222  master       -> origin/master
   * [new branch]      better-random -> origin/better-random
Merge made by 'recursive' strategy.
 src/rand.c | 7 ++++++
 1 file changed, 6 insertions(+), 1 deletion(-)
```

Well, Git has merged Alice's changes cleanly, but there is a new branch present. Let's take a look at what is in it, showing only those changes exclusive to the `better-random` branch (the double dot syntax is described in *Chapter 2, Exploring Project History*):

```
bob@hostB random$ git log HEAD..origin/better-random
commit bb71a804f9686c4bada861b3fcd3cfb5600d2a47
Author: Alice Developer <alice@company.com>
Date:   Sun Jun 1 03:02:09 2015 +0200
```

```
    random_int: use rescaling
```

Interesting. Bob decides he wants that. So he asks Git to merge stuff from Alice's branch (which is available in the respective remote-tracking branch) into the default branch:

```
bob@hostB random$ git merge origin/better-random
Merge made by the 'recursive' strategy.
 src/rand.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Undoing an unpublished merge

Bob realizes that it should be up to Alice to decide when the feature is ready for inclusion. He decides to undo a merge. Because it is not published, it is as simple as rewinding to the previous state of the current branch:

```
bob@hostB random$ $ git reset --hard @{1}
HEAD is now at 3915cef Merge branch 'master' of https://git.company.com/
random
```



This example demonstrates the use of `reflog` for undoing operations; another solution would be to go to a previous (pre-merge) commit following the first parent, with `HEAD^` instead of `@{1}`.

Summary

This chapter walked us through the process of working on a simple example project by a small development team.

We have recalled how to start working with Git, either by creating a new repository or by cloning an existing one. We have seen how to prepare a commit by adding, editing, moving, and renaming files, how to revert changes to file, how to examine the current status and view changes to be committed, and how to tag a new release.

We have recalled how to use Git to work at the same time on the same project, how to make our work public, and how to get changes from other developers. Though using a version control system helps with simultaneous work, sometimes Git needs user input to resolve conflicts in work done by different developers. We have seen how to resolve a merge conflict.

We have recalled how to create a tag marking a release, and how to create a branch starting an independent line of development. Git requires tags and new branches to be pushed explicitly, but it fetches them automatically. We have seen how to merge a branch.

[Get more information Mastering Git](#)

Where to buy this book

You can buy Mastering Git from the [Packt Publishing website](#).

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

[Click here](#) for ordering and shipping details.



www.PacktPub.com

