Jessie and friends

Introduction to Git for beginners

ROMAN MAHOTSKYI

Jessie and friends

Introduction to Git for beginners

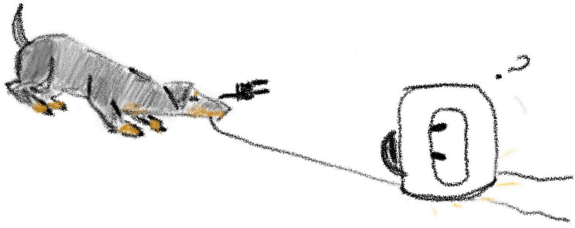
First edition

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

Contents

1

Acquaintance with Jessie and Boopi



In the everyday world, pets are an integral part of our lives. They help us cope with difficulties and support us in difficult situations as much as they can. When we are sad, they are sad with us, when we are happy, they are happy too. This unspoken love can never disappear between us.

Woof-woof!

Wait Jessie, I'm making a nice introduction for our reader. You don't want to ruin their first impression, do you?

Okay-okay! I remember...

Woof-woof!

I apologize for being distracted, apparently, she will not let me finish the introduction.

What happened again? Woof! Did you lose a tennis ball? Another? This is the second ball this week... Alright!

Yes, I'm sorry, I should have introduced you to each other. As you can notice this is Jessie. Jessie, say hi to our reader.

Woof-woof!

Jessie is a virtual dog who loves running, jumping and having fun in between other doggy activities. Usually, she spends her time playing with Boopi.

Boopi is a robot that can make any item we want. Usually, you can give him detailed instructions on how to create things, and Boopi will be able to produce them for you. That's how we created the previous two tennis balls.

Woof-woof!

Jessie don't interrupt me... I know that Boopi can craft you another tennis ball, but it's time you learn to appreciate the things we give you...

These pets can sometimes be very harmful, right, Jessie?

Since she lost the two previous tennis balls that Boopi and I created together, may I ask you, reader, that you create the third one? It shouldn't be hard, but I hope that the gift you make she will appreciate much more than the previous two.

I hope I can count on you, reader

I see that Jessie has started to like you, so let's begin and try to recreate a lost tennis ball.

2

The way Boopi reads instructions

I'm glad you agreed to help us to create another tennis ball for Jessie. As we mentioned earlier, Boopi can only craft things if

we give him instructions. So, what are the instructions and how to provide them to Boopi? Good questions.

Historically, you can't give Boopi instructions directly by writing them down on paper. Instead, you need to create a file and load them by using a Git tool. Yes, it may seem like a complicated process. But let me explain it first.

Git is a special tool that you can install on your computer. The main task of Git is to keep track of changes in any set of files. This tool is especially good when a lot of people are involved in the process of adding, changing and deleting files. Because anyone can accidentally delete or change information within files, Git keeps track of who made those changes and when. Thus, you can be sure that your information will never be lost and will always be ready for recovery.

Most often, this program is used by programmers when they are working on a project together. As soon as each program consists of hundreds or thousands of different files, the use of such a program is simply an integral part.

Boopi doesn't require to create him a hundred files of instructions to craft things. But he follows the same flow of adding, changing or removing files.

I believe that interacting with Boopi can help you learn how to work with Git, and all the knowledge gained in the process will be ready for use in other projects.

Woof-Woof!

Yes, Jessie? Woof! Woof-woof! Ah, I see. You want to play with a tennis ball already, but our reader is just taking the first steps, so be patient.

Let's move forward and install Git on your computer

3

Git installation

As we discussed in the previous chapter, in order to get started with Git, we need to install it on our computer.

Since absolutely any operating system can be installed on your computer (for example, Windows, Linux/Unix, or macOS), and the installation process is different for each operating system, I hope you can handle this task yourself.

This is the link to the official download page where you can get the latest version of Git

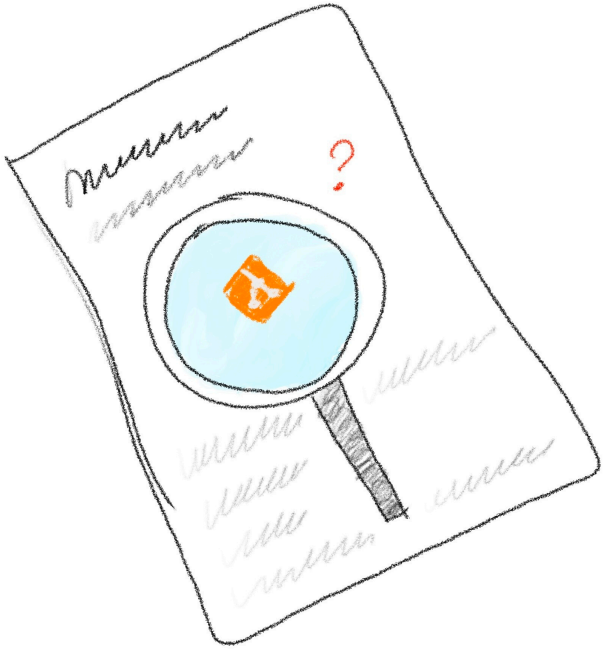
https://git-scm.com/downloads

The only thing I can recommend is to always stay with the default installation. Do not change any settings during installation until you know what you are doing.

While you are installing Git, Jessie and I will be waiting for you in the next chapter. Take your time, don't hurry...

4

Verify Git installation



Woof-woof!

Oh, you're already here. Well done! It means that you installed Git on your computer. But we need to ensure that Git is available inside your console.

I expect that you know what a console is because we will use it through this book very often. Usually, we use a console to execute commands. What commands to execute depends on us. Since we will be working with Git, we will use the commands it provides. Every command is like a button on the interface. Whenever you execute a command, it is something similar to what happens when you click the button on the interface.

Let's use our first command that shows the version of the installed Git on the computer.

Let's open a console and write our first Git command

```
git --version
```

It should print

```
git version 2.37.0
```

> NOTE: That printed version in your console will be equal to the version you installed on your computer. In my case, the version that is installed on my computer is 2.37.0.

If you don't see a similar message in your console, it may mean that Git was not properly installed. Or at least it doesn't work inside a console. Try to restart the console and execute *git —version* command again. If nothing changes, you need to fix this problem before moving to the next chapters.

I hope your installation experience went smoothly and there were no problems. If so, let's move forward

5

Create a folder for Boopi's instructions

Yes, it took some time to get Git ready for use on our computer. But it can not but rejoice that this process needs to be done only once.

Woof-woof! Good point, Jessie!

Let's start writing instructions for Boopi.

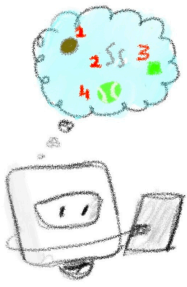
First off, we need to find a place where we would store our files with instructions. It is a good practice to group related files under a folder somewhere on your computer. So let's create it. Jessie and I have no claim in which place the folder should be created. The only advice is that the place should be convenient for you.

Woof! Woof!

Jessie sniffed at the new folder. Good job, Jessie, and well done, reader! Let's enter it and create our first instruction for Boopi!

6

The first instruction for Boopi



So, it looks like we are inside the newly created folder. Could you explain to us, Jessie, what do we need to do next?

Woof!

Okay, I think I understand! Did you understand the reader? She

said that we need to create a file and call it *tennis-ball.txt*. As you may remember, Jessie wants us to create a tennis ball. Since she lost the previous

What next?

Woof-woof!

As Jessie said, we need to open this file and start writing instructions that Boopi can understand. But what kind of instructions can Boopi understand?

Boopi understands the simple steps of how to create objects. By reading line by line, Boopi will be able to create things at the end and pass them to us. So, let's write instructions for creating a tennis ball.

Open the file and write the following instructions.

```
Step 1. Create a rubber ball with a diameter of 6.54
cm
Step 2. Color it green
Step 3. Add curved white lines
Step 4. Craft
```

Great. Now save the file and close it. Let's see what happens.

Hm... Nothing happens. Have we missed something?

Woof-woof!

Ah, exactly! How could I forget?

We just created a file inside our folder and named it *tennis-ball.txt*. After, we added instructions inside this file and saved them. But this file is just a file on your computer and does not differ in any way from other files on your computer.

Do you remember that Boopi can load instructions only by using Git? It seems like we didn't add Git to our folder. But should we? Yes. As we said earlier, Git is a program that can track files and changes inside those files.

Currently, we have Git installed on our computer, but we didn't ask Git to track files in our folder. By default, installed Git does nothing on a computer until you explicitly ask him to do it.

To ask Git to start working with our files, we need to initialize him in our folder. By "initialize" I mean we must say, "Hey Git, could you take care of this folder?"

So, let's do this in the next chapter

7

Initializing the Git inside the folder

In the previous chapter, we created a *tennis-ball.txt* file, but unfortunately, it wasn't enough to let Boopi start to craft it. As I said, we need to initialize the Git inside our folder. To initialize Git inside our folder and make aware of our project, we must open the console in the same place where our *tennis-ball.txt* file

is located and run the next command.

```
git init
```

> NOTE: Whenever I say open the console and write a command, this implies that the console must always be open in the project folder. That is, at the same level as our *tennis-ball.txt* file or any other project file

After running this command, Git should print hint messages to the console that propose you adjust the default configurations.

```
hint: Using 'master' as the name for the initial
branch. This default branch name
hint: is subject to change. To configure the initial
branch name to use in all
hint: of your new repositories, which will suppress
this warning, call:
hint:
hint: git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are
'main', 'trunk' and
hint: 'development'. The just-created branch can be
renamed via this command:
hint:
hint: git branch -m <name>
Initialized empty Git repository in
/Users/author/Documents/project/.git/
```

We are not interested in changing anything, so we can skip it for now.

Let's check our folder. Has something changed?

Yes, there are some changes in our folder, but they are not visible to us.

After running the previous *git init* command, Git created a hidden folder called *.git* inside our folder.

```
project-folder/ ├──────
  .git/ (hidden folder) └──────
  tennis-ball.txt
```

We can draw such a conclusion by reading this message printed as a last sentence in the console

```
...
Initialized empty Git repository in
/Users/author/Documents/project-folder/.git/
```

This hidden folder will store all information about our project, all created, renamed, and deleted files, as well as changes within its files.

But if your OS (operating system) doesn't show hidden files by default, the only file that should exist in your folder is *tennis-ball.txt* that we created previously, which contains instructions for Boopi. Speaking of Boopi, can he make us a tennis ball already?

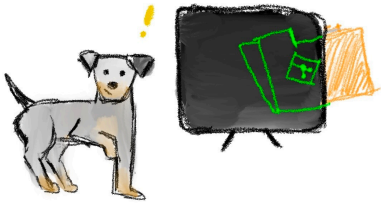
Not yet!

Having initialized Git in our folder, we only said that in the future, we will use some Git commands inside this folder. By default, Git won't do anything unless we ask it to. That is, if we don't ask Git to pass the Boopi instructions, it won't do it itself.

So let's ask Git to pass instructions to Boopi.

8

Check the state of the folder



In the previous chapter we initialized Git inside our project folder, but I would lie if I said that passing instructions to Boopi would be simple as running a single command. So, it should be easy anyway but requires a few steps to be done before it.

Every time you create a file inside a Git-aware folder Git does not automatically start to track the file and its internals. Remember

when I said Git doesn't do anything without you knowing? Every time you want Git to do something, you must ask him explicitly by running a command that Git can understand.

One of those commands that Git can understand is *git status*.

Let's run it and after I explain what it does

Open a console and write the following

```
git status
```

After running this command, Git should print the current state of your folder inside your console (or wherever you run this command).

The output should be similar to something like this

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
tennis-ball.txt

nothing added to commit but untracked files present
(use "git add" to track)
```

There is much information here, but we can skip most of it for now and focus on this important message.

```
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
tennis-ball.txt
```

As I said earlier, Git doesn't track your files automatically, and the *tennis-ball.txt* file you created is one of those untracked files. This means that if you accidentally (or intentionally) deleted a file and want to restore it, Git can't help you because it didn't track it. Even if you remove information inside it (in our case, it is the instruction on how to assemble a tennis ball), Git still is useless.

But how can we ask Git to track it? This is the next command that we will learn.

9

Prepare changes to be committed

To ask Git to track our file, we can run the next command.

```
git commit
```

Woof-woof! I know Jessie, it will not work, but let our reader understand why.

Nothing wrong with this command but something is missing. Let's find it by reading the printed message in our console.

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
tennis-ball.txt

nothing added to commit but untracked files present
(use "git add" to track)
```

If you read the message carefully, you should notice this sentence.

```
nothing added to commit but untracked files are
present (use "git add" to track)
```

It says that to make our *git commit* command work, we need to specify what files will be included in this commit.

But what is a commit? A commit is a group of changes that were added to Git. Simply saying, everything we add to a commit will be stored in Git and Git will take care of it (will start tracking changes of files that you added to a commit).

So, let's add our *tennis-ball.txt* file into a commit.

```
git add tennis-ball.txt
```

As you can see, the *git add* command expects the file name that should be added to the commit.

> TIP: *git add* has much more power than specifying a single file name. It also can add multiple files by listing their names separated by a single space or all files with extensions .txt by specifying a pattern (e.g. *git add *.txt*)

Using the *git add* command, you asked Git to prepare this file to be ready to be committed, but not committed yet.

Let's see what happened with our project after running the previous command.

To see the current state of the repository we need to run this command

```
git status
```

It should print something similar to your console

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   tennis-ball.txt
```

Now Git says that there is a new file called *tennis-ball.txt* that will be committed.

After adding a *tennis-ball.txt* file into a staged area, Git makes a snapshot of our file and stores it as a copy in memory. If we make any changes to this file (that were added by using *git add* command and not committed yet), it will not automatically add changes to the snapshot, instead, it will show that some modifications exist and can be added to the commit if we want.

Let's try to achieve this by opening the *tennis-ball.txt* file and changing the diameter of the ball.

Let's make changes to the first line of the steps we created

```
Step 1. Create a rubber ball with a diameter of 6.55
cm
Step 2. Color it green
Step 3. Add curved white lines
Step 4. Craft
```

We changed the diameter from *6.54* to *6.55*. The changes are minor, but enough for Git to notice them.

Let's again run the status command and see what it shows to us

```
git status
```

The output should be similar to

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   tennis-ball.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be
  committed)
  (use "git restore <file>..." to discard changes in
  working directory)
modified:   tennis-ball.txt
```

As you can see, Git is showing us that some modification has happened to our file.

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   tennis-ball.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be
  committed)
  (use "git restore <file>..." to discard changes in
  working directory)
modified:   tennis-ball.txt
```

This modification is nothing more than a change in diameter.

To confirm our assumptions that these modifications are diameter changes, we can use another useful command called *git diff*. This command requires the name of the file we want to check,

so let's specify the following:

```
git diff tennis-ball.txt
```

It should print something like

```
diff --git a/tennis-ball.txt b/tennis-ball.txt
index 81d5611..3c89c6c 100644
--- a/tennis-ball.txt
+++ b/tennis-ball.txt
@@ -1 +1 @@
-Step 1. Create a rubber ball with a diameter of 6.54
cm
+Step 1. Create a rubber ball with a diameter of 6.55
cm
```

This printed message has a lot of useful information. Let's go through the most important.

The first line says that Git is currently comparing two files, or more accurately, the same file but with two different sets of changes. The fifth line shows the summary of how many lines were affected. It says that one line was removed and one line was added. Actually, we only changed 1 character, but Git treats it as an entire line change (in this example). The last two lines tagged with **-** and **+** show what changes the file has undergone. It shows exactly what we have done, changing the first step and replacing *6.54* with *6.55*.

Cool. This means that Git is doing its job. It tracks changes in our files.

So, at this point, we can do a few more things. Either include our changes to the commit or discard them. In our case it is not big deal will the diameter be 6.54 cm or 6.55 cm. But let's for education purposes discard our changes. The next command will help us to do it

```
git restore tennis-ball.txt
```

It should remove our changes, that were made after adding *git add tennis-ball.txt*. That is, modified *tennis-ball.txt* should disappear from the *git status* message.

Let's verify it by running *git status*

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   tennis-ball.txt
```

Good job! Modification disappeared.

As you can see, we can rely on Git and its tracking capabilities. Whenever we want to get some information about the current state of the project we can ask Git to provide us with this information. The more commands we know, the wider the range of interaction with Git.

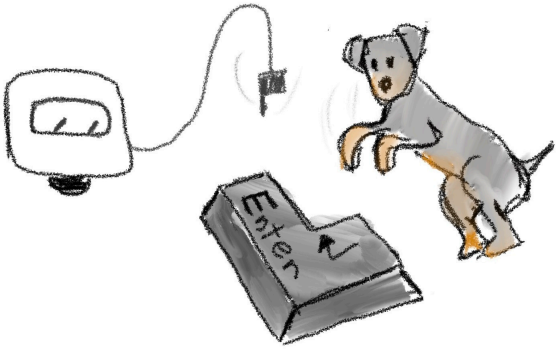
We made changes to the file and asked Git to show them to make

sure that nothing more was changed. And you made sure that everything is exactly right.

So, in the next chapter, we will create a commit of our staged changes and finally create a tennis ball for Jessie

10

Commit changes



In the previous chapter, we added the *tennis-ball.txt* file to the staged area, thus saying that we are ready to commit it.

To make a commit, we need to use the already known *git commit*

command. Previously, we tried to make a commit, but we didn't select any file and Git showed us an error message. Now, we have one file selected, and let's see what happens next.

```
git commit
```

After running these command the default editor should be opened with this content

```
# Please enter the commit message for your changes.
Lines starting
# with '#' will be ignored, and an empty message
aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:   tennis-ball.txt
```

If you read this text you will notice that *git commit* contains the information of the current status of the project and the message above

```
Please enter the commit message for your changes.
Lines starting
# with '#' will be ignored, and an empty message
aborts the commit.
```

It means that we need to provide a review of what we have done. Usually, one line of description is enough.

So, let's put the cursor on the first line and write the description.

> *NOTE: Before adding any information I should mention that usually Git uses Vim as a default editor. You would ask me why I have to make attention on it. Because Vim has special unique features. One of those features is that you are not able to write text directly. By default, when you open Vim, it goes with* ***Normal mode*** *where each key has special meaning:* ***w*** *jumps to the start of the word,* ***e*** *jumps to the end of the word and* ***i*** *switches to* ***Insert mode****.* ***Insert mode*** *is a mode where each key becomes a regular key as we would expect. So let's press the* ***i*** *key on the keyboard. The* ***—INSERT—*** *word should be shown at the left bottom of the screen of your editor. After providing a commit description, you need to press* ***:*** *and write* ***wq*** *(****w*** *- write,* ***q*** *- quite) to save changes.*

```
Add a file with instructions on how to create a
tennis ball
```

After writing the commit message close the editor. If a commit was created, the following message should be shown in the console

```
[master (root-commit) e6d979b] Add a file with
instructions on how to create a tennis ball
 1 file changed, 1 insertion(+)
 create mode 100644 tennis-ball.txt
```

It confirms that a new commit was created.

Once a commit is created, Git starts storing the files added to the commit inside its "internals". By that, I mean a hidden *.git* folder that was automatically created during Git initialization. Thus, from now on, the file *tennis-ball.txt* should be tracked by Git, and any future changes should be reflected without adding them to the staged area by using *git add* command.

What's going on... Boopi starting to craft something. Woof-woof! Look at Boopi! Woof-Woof! This is a tennis ball! Thank you, Boopi, and thank you, reader! Now Jessie has a new tennis ball!

Do you like it, Jessie? Woof! Alright!

Let's make a quick recap of what we have already learned.

We learned that in order to start using Git commands inside our folder, we first had to initialize Git with the *git init* command. It created a hidden *.git* folder inside our project folder (besides the *tennis-ball.txt* file) where Git stores the information that helps him to keep track of our files and their changes.

By default, Git does not automatically add these files to the *.git* folder, so we had to add them explicitly with *git commit*.

Before we created a commit, we had to decide which files should be included in the commit and select them with the *git add* command. After we added the file *tennis-ball.txt* to the staged area, Git took a snapshot of the file and said that this file is

now being tracked by me. After we changed the diameter of the tennis ball inside instructions, Git showed us that some changes were made. We inspected those changes by using *git diff* and discarded them with a *git reset* command.

Finally, when the file was ready to be committed, we used the *git commit* command, which opened the default editor and asked us to provide a description of the current commit. After providing the message and closing the editor, the commit was added to Git and Git started to track our file. Even if we delete some of the internal parts of the file in the future, or delete it altogether, Git will notice those changes and let us know but only when we run the *git status* command.

Well done reader, looks like we did what we wanted. But this is not the end, let's see what else awaits us in the next chapters.

11

Create toys for Jessie's friends in different branches

Woof-woof! Yes, Jessie? Oh, who is it? Woof-woof-woof! Ah, it's your friends? Woof. Hi friends, it's me, reader and Boopi; nice to meet you. Woof-woof! Jessie said that her friends saw her playing with a tennis ball and wanted toys to play with them

too. What a great idea. It would be nice to make some more toys for friends, what do you think, reader and Boopi?

Alright

Let's create a rubber bone and flying disc for them. But before we start creating instruction files, let's think about how we could do this. The easiest way would be to create two separate files for each toy, then provide instructions and commit two files together. Usually, putting different things inside a single commit is not a good practice. You can think of a commit as one atomic change. It makes our Git history clear and our commits independent of each other. So, if we start to follow these recommendations, we would create two different files and put each of those files into a separate commit. Much better. Each commit will contain unique atomic changes that can be easily read and created by Boopi.

But let's think if we had more than two files. Suppose Jessie comes not with two of her friends but a hundred. We would need to create a hundred files and make each commit for them. It would create a mess in our folder. It would be painful to navigate between tons of untracked files and jump between them if needed. As you may have noticed, I'm taking you to a new feature in Git called branches.

So, what branches are? You can think of branches like a separate "universe" in your folder. You can create "universes", jump between them and remove them if you want. Each of those "universes" has different folders, files, and commits. It sounds a lot more complicated than it really is. So, let's create our first

"universe" a.k.a branch.

I expect your console to be open inside your folder and wait for new commands to execute.

To create a new branch, we need to use *git branch* command with the name.

```
git branch rubber-bone
```

After running this command, Git must create a new branch in your folder. You may notice that no messages appeared in the console. It's correct. We have just created a branch named *rubber-bone*. To confirm that we can run another command

```
git branch --list
```

It should list all existing branches in your project.

```
master
rubber-bone
```

You may wonder why *git branch —list* returned two branches instead of one? Good question. The *master* (or *main*, *thunk*) is the standard branch that is created along with the initialization of the project. Remember we started our project with *git init* command? At that moment, Git created a default branch called *master*. Moreover, when we created the tennis ball and made a commit, that commit was added to the *master* branch. I didn't mention it earlier because it wasn't necessary to know. Now,

when we created a new branch, we are still on the *master* branch. We can verify it by running this command

```
git branch --show-current
```

It should print *master*. By default, when you create a branch, Git does not automatically switch to it. Instead, it creates it in the background and lets you switch to it later.

To switch to a newly created branch, we need to run one more command and provide the name of the desired branch. In our case, it should be *rubber-bone*

```
git checkout rubber-bone
```

After running this command you may notice a message in a console

```
Switched to branch 'rubber-bone'
```

It means that we have successfully switched to a new branch. Let's verify it by running an already known command.

```
git branch --show-current
```

If we did everything correctly, this should show the *rubber-bone* name in the console.

After creating a new branch and switching to it, we can finally create a file and provide instructions inside it.

Let's create a file called *rubber-bone.txt*

Open the file and write the following.

```
Step 1. Create a rubber
Step 2. Shape into a bone
Step 3. Color is red
```

Save the file and close it

Good. Now we have a new untracked file. Let's make a commit

As you remember, before making a commit, we need to specify what files will be included inside the commit. To do this, we will use the already known command *git add*

```
git add rubber-bone.txt
```

After running *git status* we can verify that our *rubber-bone.txt* has been added to the staged area.

The final step that we need to do is to create a commit.

Previously, when we created a commit for *tennis-ball.txt*, we used *git commit* command without attributes. It opened the editor where we could specify the commit message. There is a shorter way to provide a commit message without opening any text editor.

Let's see how we can achieve this.

```
git commit -m "Add a file with instructions on how to
create a rubber bone"
```

As you can see, we have added the *-m* attribute with our commit name enclosed in double quotes.

Okay, we created a file with instructions and made a commit. Why doesn't Boopi create an item?

Remember that I said that creating a new branch creates a new "universe" where our files and folders will live? That is exactly what happened. Boopi remained in the previous branch *master*, but we are on the *rubber-bone* branch now. It means that Boopi doesn't have access to the files and folders that were created on the different branches. How we can pass these files, or in our case, a file to the Boopi?

Good question.

To move the file to the *master* branch, we need to merge two branches into one. In other words, you can think of it as merging two universes. Everything that exists in one universe will collide with another universe. But instead of creating a new universe, it continues to exist in one of those universes. How can we decide what universe will preserve changes of both? It depends on what universe we are pouring changes into. It may sound complicated, but in reality, it is nothing more than just saying, "You must merge into me."

Let's do this.

Before merging our "universes" let's verify that our commit exists by running *git log* command in our *rubber-bone* branch.

```
hint: git branch -m
commit 9a7a6d46785732d976813088c55e080db3948166 (HEAD
-> rubber-bone)
Author: Roman Mahotskyi
Date: Tue Jul 12 02:33:17 2022  +0300

Add a file with instructions on how to create a
rubber bone

commit a81dce30f016ee639b4ec62dbfb5c25655697f13
(master)
Author: Roman Mahotskyi
Date: Tue Jul 12 02:32:16 2022 +0300

Add a file with instructions on how to create a
tennis ball
```

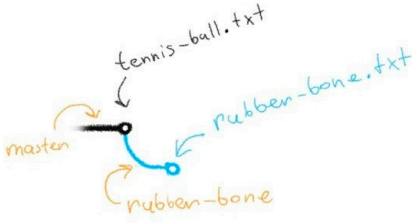
Hah. Have you noticed that *git log* printed two commits instead of one? It printed "*Add a file with instructions on how to create a tennis ball*" as well as our commit with a rubber bone. But how is it possible? We are on the *rubber-bone* branch, and can still see commits from another "universe"?

Yes.

I didn't mention this before because it would make it harder to describe the branches. But in general, there is nothing special. Whenever you create a new branch in your project, the newly

created branch will be based on the current state of the active branch. It means that on the moment when we used a *git branch rubber-bone* Git created a new branch with all commits of the *master* branch and "put it" into a newly created branch - *rubber-bone*.

Therefore, when you run the *git log* command, you see the commits from the current branch and previous commits of the branch that you were based on. If we could visualize this, it would look like the following picture.



I hope it makes sense to you now.

So far, we created a single commit on the *rubber-bone* branch with instructions on how to create a rubber bone toy. But it doesn't help Jessie's friends to start to play with toys yet. Let's take a few more steps in this direction and create a missing

flying disc.

At this moment, we have an unmerged branch with a rubber bone toy, we need to create another branch and create a flying disc there.

Let's do this but with fewer explanations.

Let's switch back to the *master* branch (without merging).

```
git checkout master
```

Run *git log* and verify that the commit from the *rubber-bone* branch didn't leak to the *master* branch.

Good. The *git log* command should show the only commit with a tennis ball.

Let's create a new branch called *flying-disk* based on the master

```
git branch flying-disc
```

And switch to the branch

```
git checkout flying-disc
```

Let's verify that we're on the new branch

```
git branch --show-active
```

It should print the *flying-disc*

If we run the *git log*, it also should show the commit with the tennis ball because, as I said earlier, newly created branches are always based on the state of the currently active branch. In our case, we based on the *master* branch where a commit with a tennis ball was already created.

Let's create a new file called *flying-disc.txt* and add instructions inside it.

Open the file and provide the following instructions.

```
Step 1. Create a circular object
Step 2. Make it thin
Step 3. Color it blue
```

Save changes and close the file.

Let's make a commit with this file

```
git add flying-disc.txt
git commit -m "Add a file with instructions on how to
create a rubber bone."
```

Good. We just created another commit of the instructions for the flying disc. Let's verify that commit was added to the history by running *git log* command.

```
git log
```

Hold on. It shows the incorrect message, *"Add a file with instructions on how to create a rubber bone."*, but we provided instructions on creating a flying disc.

What can we do? Is there any command that can help us to modify existing commits? Of course, there is.

Git provides us a possibility to amend existing commits. If we accidentally committed a wrong file, or we forget to add something to the commit, or we made a mistake in the name of the commit, all of these problems Git allows us to fix.

There are many approaches to modifying the existing commit, but we will use one of them.

Inside your console, write the following.

```
git commit --amend
```

It must open the text editor with the commit message of the latest commit.

```
Add a file with instructions on how to create a
rubber-bone

# Please enter the commit message for your changes.
Lines starting
# with '#' will be ignored, and an empty message
aborts the commit.
#
# Date:      Tue Jul 12 18:52:01 2022 +0300
#
```

```
# On branch flying-disc
# Changes to be committed:
#       new file:   flying-disc.txt
```

It is exactly what we needed. Our latest commit on the *flying-disc* branch contains the commit with the wrong title description.

In the opened text editor, let's change the commit message to *"Add instructions on how to create a flying disc"*. All we need to do is to replace "rubber bone" with a "flying disc".

Save the file and close it.

If we have done everything correctly, Git should replace the old wrong commit message with a new one.

We can verify it by running *git log -1*

```
git log -1
```

Yes, our commit message has the correct description.

```
commit 5f21d4e852c19a65b9ebcce8e1761f84248b2778 (HEAD
-> flying-disc)
Author: Roman Mahotskyi
Date: Tue Jul 12 18:52:01 2022 +0300

Add a file with instructions on how to create a
rubber-bone
```

Good job. We just fixed an incorrect commit message and finally can merge our instructions from both branches into the *master*.

You must remember that merging is colliding two "universes". To collide the *master* branch with two other branches, we need to go back to the *master* branch and specify what branch we want to "collide into us".

Let's back to the *master* branch

```
git checkout master
```

The final step is to ask Git to merge (collide) another branch into the current. We can achieve this by running the next command.

```
git merge rubber-bone
```

Nice. Look at Boopi. It crafts a rubber bone! Good job, reader!

Let's see what happened with our commit history.

```
git log
```

As you can see it shows us two commits. The first commit was created on the *master* and the second one on *rubber-bone* branch (from bottom to top). In the end, they were merged and stored on the *master* branch.

```
commit 6661ee1676f17f3a7e89ef41b6ae21d36d5fdf52 (HEAD
-> master, rubber-bone)
```

```
Author: Roman Mahotskyi
Date: Tue Jul 12 18:57:19 2022 +0300

Add a file with instructions on how to create a
rubber bone

commit b15d1b743fcde3c1fcd26880c7159451f2c2eb3e
Author: Roman Mahotskyi
Date: Tue Jul 12 18:57:00 2022 +0300

Add a file with instructions on how to create a
tennis ball
```

You can open a folder and verify that the *rubber-bone.txt* file has been added to the folder.

Nice! We successfully merged our first branch!

Woof-woof! Yes, Jessie? Woof-woof! Ah, how could we forget? We didn't merge the toy for another Jessie's friend — a flying disc! Let's do the same steps that we did with the *rubber-bone* branch.

To merge the remaining branch, we can run the following command

```
git merge flying-disc
```

At this moment, it should open a text editor with the default merge commit message.

```
Merge branch 'flying-disc'
# Please enter a commit message to explain why this
merge is necessary,
# especially if it merges an updated upstream into a
topic branch.
#
# Lines starting with '#' will be ignored, and an
empty message aborts
# the commit.
```

You can either change the message or leave it as is. In this case, we leave it as it is because it accurately describes what we are doing.

You may ask why Git didn't ask us to provide a commit message when we merged the *rubber-bone* branch? Are there any differences in these branches (*rubber-bone* vs *flying-disc*)? No, two branches are equal in a way we created them and merged. The difference is only in the order of their merging.

Before Git merged *rubber-bone* into *master* it investigated the history of commits of two branches. If the history of these branches is linear, that is, the *master* didn't receive any commits after the *rubber-bone* branch was created, then merging such branches does not raise any issues in Git. Such merging technique is called - fast-forwarding. It's a fancy word to describe that the history after the merge remains linear. But, when we tried to merge a *flying-disc* into *master* it opened an editor with a default merge commit message. After providing a merge commit message and closing the editor a technical commit should have been created. This technical commit is called - a merge commit.

It explicitly shows us in the Git history (*git log*) that merge has happened in this place. When a technical commit is created in history, our history is no longer linear, because there is more to it than just our commits.

Even if this description was not enough to understand the difference, I recommend you not spend much time on it. Because neither fast-forwarding nor additional merge commit doesn't really matter when you learn Git. Git has a lot of settings and nuances that help us to work more efficiently, but this efficiency is not a goal of this book. Let's go back to crafting things!

Look at Boopi! It creates a flying disc!

Woof-woof!

Nice! We merged another branch into our master. Now our history of the master branch should consist of commits from two other branches.

Since everyone received the toys and was happy, we can do the last but not the least thing - remove unused branches.

Usually, branches have a limited lifetime. They make sense only during the development process. Once you've finished working on a branch (as we did) and merged it into another branch, it rarely remains useful the rest of the time. It is a good practice to remove them.

To not guess what branch was already merged into the current active branch we can use another useful Git command

```
git branch --merged
```

It should show the list of all branches that were already merged into the current one.

> NOTE: It includes the currently active branch (it is marked with an asterisk *). Don't delete it by accident.

After reviewing the list of merged branches, we can remove them.

```
git branch -d rubber-bone
```

It should remove the *rubber-bone* branch, and

```
git branch -d flying-disc
```

It should remove the *flying-disc* branch.

Good job. Now we don't have "zombie" branches on our computers. We can verify it by running

```
git branch --list
```

As you can see, it again shows only the *master* branch.

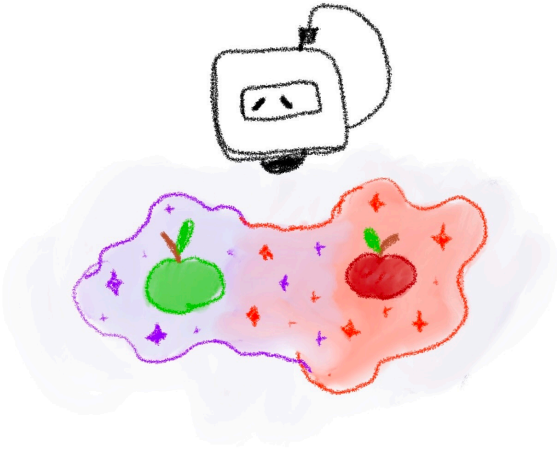
Woof-woof! Yes, Jessie, I see! You are all happy to play with your toys! Say thanks to the reader and Boopi without them

this would not be possible! You can keep having fun with newly created toys, but we need to keep learning Git!

Let's move on to the next challenge!

12

Same things from different universes



In the previous chapter, we created two toys for Jessie's friends. Each of these toys was created and committed to a separate

branch. In order for these toys to appear in the *master* branch, we had to merge each branch into the *master*. So, Boopi was able to create toys for these cuties.

But what if we could create instructions for the same toy in parallel and merge them together? What would happen? What would happen if two same things from different universes appeared in one? Do they merge at all? Let's figure it out!

To do this experiment, we will need to create an apple in two separate branches.

Woof-woof!

I know Jessie, Boopi doesn't like apples. But let's create them anyway.

Firstly off, we have to be on the *master* branch. Secondly, we will create a new branch called *red-apple* and jump into it.

Starting on the *master* branch execute the following commands

```
git branch red-apple
git checkout red-apple
```

Having done this, we created a new branch and moved into it.

Let's create a file and call it *apple.txt*

Open the file and put the following instruction inside it.

```
Step 1. Create a red apple
```

Save the file and close it.

At this moment, we should be on the *red-apple* branch with one new (untracked) file called *apple.txt*. You can verify it on your own by running *git status.* We will skip this optional step and continue

Let's commit the *apple.txt* file with instructions inside of it

Inside the console write the following

```
git add apple.txt
git commit -m "Add a file with instructions on how to
create a red apple"
```

Good. At this point, we created a file with instructions and committed it to the *red-apple* branch.

Let's switch back to the master branch and do the same things.

```
git checkout master
```

After switching to the *master* branch, let's create an *apple.txt* file and pass those instructions inside it

```
Step 1. Create a green apple
```

Save the file and close it.

Woof-woof!

Yes, Jessie, I know. We have just created an apple in a different color than the one we created in the *red-apple* branch. But I will explain it later. For now, let's continue and commit the file.

Let's create a commit on the *master* branch with an *apple.txt* file inside of it.

To achieve this we will execute the following commands

```
git add apple.txt
git commit -m "Add a file with instructions on how to
create a green apple"
```

Well done!

Now we should have another commit on the *master* branch. We have almost reached the climax of our experiment, but before we merge the *red-apple* branch into it, let's try to predict what might happen:

1. The *apple.txt* file from *red-apple* branch will replace *apple.txt* file in the *master branch*
2. The *apple.txt* file on the *master* branch will not be replaced and remains untouched
3. The *apple.txt* file will contain instructions from both branches (e.g. they will be merged)
4. Black hole

None of the above options are correct. Let's figure out what will actually happen

Woof-woof! I know Jessie, you know what will happen, but let our reader go through this on his own.

As far as we are already on the *master* branch, let's merge the *red-apple* branch into it.

```
git merge red-apple
```

Oh, no! What just happened?

Let's see what the console message says.

```
Auto-merging apple.txt
CONFLICT (add/add): Merge conflict in apple.txt
Automatic merge failed; fix conflicts and then commit
the result.
```

You may notice the word in capital letters - CONFLICT. It is exactly what happened - a merge conflict. Merge conflicts are the most common problem of newbies who start to work with Git. But don't be scared. Jessie and I will try to explain when they happen and how to resolve them.

Yes, Jessie? Woof-woof! Alright!

If you run the *git status* command and investigate the result message, you may notice that Git says that you have "unmerged

paths", and under "Unmerged paths:" title, you can see the message - "both added: apple.txt".

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
both added: apple.txt

no changes added to commit (use "git add" and/or "git
commit -a")
```

It is exactly what happened. Two branches were trying to add an *apple.txt* file at once, but there are conflicts inside this file. If we open the file with your favorite editor, you may notice that the file contains something more than the instructions of both files.

```
<<<<<<< HEAD
Step 1. Create a green apple
=======
Step 1. Create a red apple
>>>>>>> red-apple
```

Don't be scared. We will go through this file step by step

The first lines says that everything starting from *«««< HEAD* till ======= belongs to one file, and everything after ======= and until *»»»> red-apple* belongs to the another one.

But what does it all mean anyway? This means that during the merge process, when Git tried to automatically merge files (that is exactly what happens under the hood when we write a git merge command) and their internals from different branches, it could not decide on its own which change to apply.

When Git does not know what changes to apply, he asks the user to help him. So now, you as the user must help him to make a decision. It is easier than you may think.

All we need to do is to leave those changes that we're interested in. If we created two apples with different colors, we must decide what apple in what color to leave. Also, instead of creating a single apple, we could create two apples in two different steps, for instance

```
Step 1. Create a green apple
Step 2. Create a red apple
```

As you can see, there are many options of what to do with the conflicts, and only the user may know what exactly to do. So instead of creating two apples let's leave a single instruction with a green apple, for example

```
Step 1. Create a green apple
```

Save changes and close the file.

Note, that we also removed all techincal lines, like: *«««< HEAD*, `=======` and *»»»> red-apple*. They were only needed when

resolving merge conflicts and are completely redundant after resolving them.

If we go back to the console and find the message that printed a message after CONFLICT occurred, you may notice the instructions

```
...; fix conflicts and then commit the result.
```

We fixed conflicts. The last step that we need to do is to commit results.

Let's add an *apple.txt* file to the staged area

```
git add apple.txt
```

and make a commit

```
git commit
```

It should open a default editor with a predefined message.

```
Merge branch 'red-apple'
```

We can leave it as is. Save and close the file.

Well done! You solved your first conflict!

Look at Boopi! He crafted a green apple! Good job, reader!

As you remember, don't forget to remove branches that you will not use anymore. It is a good practice to keep your repository in a clean state.

```
git branch -d red-apple
```

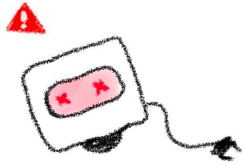
We passed the scariest part for newbies, and it was fun, wasn't it?

Woof-woof!

Let's continue

13

Fixing the Boopi that was accidentally broken



As we remember, every time a commit appears in the history on the *master* branch, it signals Boopi to start to craft things. But what if we accidentally commit the file that does not contain instructions for Boopi. Will Boopi understand how to handle it on his own? Or will he break? Let's figure out what happens.

Let's create a file called *banana.txt* and put something inside

that Boopi does not know.

Let's open a banana.txt file and write the following.

```
Step -1: abcdefg
```

Save the file and close it.

As you can see, we created a step with a negative number (`-1`), and also added Latin characters, which makes no sense for Boopi. At least we can assume that these characters are something that Boopi will not understand.

Now, let's commit this file.

```
git add banana.txt
git commit -m "Add a file with instructions on how to
create a banana"
```

Commit was created. Let's see what happened with Boopi.

Boopi, can you hear us? If yes, give us a sign!

Woof-woof!

Oops, looks like we broke Boopi

As you may have noticed, we did it on purpose. I would never ask you to do something that would totally break Boopi or anyone else. So let's think about what we can do to bring Boopi back to

“life”.

How can we undo the instructions that we gave to Boopi? The simplest solution would be to remove the file that we committed. That is, remove the file in our folder and create a new commit (e.g. “Remove a file with incorrect instructions on how to create banana”).

Instead of doing it manually, Git has a special command that can do it for us. The name of the command is *git revert*.

Whenever you make a commit that doesn’t work as expected, you can use a special *git revert <commit_hash>* command to roll back those changes. As you can see, it requires the hash of the commit. We can find this hash inside Git logs by running the *git log* command. As far as we want to undo the latest commit in our history, we can specify an additional *-1* parameter with the *git log* command to prevent printing all the existing commits on the branch (except the latest one). Let’s write it

```
git log -1
```

It should print the description of the lastest commit in our Git history

```
commit e6d979b1498e7d27cba0780e23c7bfbeca84790e (HEAD
-> master)
Author: Roman Mahotskyi <roman.mahotskyi@gmail.com>
Date:   Tue Jul 5 19:14:41 2022 +0300

    Add a file with instructions on how to create a
```

```
    banana
```

Let's copy the hash on the first line of the log message. Yes, all these numbers and characters combined into a sequence of symbols is called a hash.

```
e6d979b1498e7d27cba0780e23c7bfbeca84790e
```

Now, we can write a revert command with the already copied hash of the latest commit. (I remind you that this "latest commit" broked Boopi)

```
git revert e6d979b1498e7d27cba0780e23c7bfbeca84790e
```

It should open an editor with a default commit message.

```
Revert "Add a file with instructions on how to create
a banana"

This reverts commit
e6d979b1498e7d27cba0780e23c7bfbeca84790e.
```

If you want, you can change this commit message and its description or leave it as is.

After saving changes and closing the editor, Git must automatically revert the recently added commit. Or, in other words, Git must remove the *banana.txt* file with incorrect instructions.

We can verify this by opening the folder of our project.

It looks like the file with the bad instruction no longer exists.

If you run a *git log -2* command again, you should notice that the commit where we have added a *banana.txt* file still exists in history as well as a new commit with the message *Revert "Add a file with instructions on how to create a banana"*.

I know it's not what you expected. You might expect that commit with the message "*Add a file with instructions on how to create a banana*" to disappear from the Git history. Or in other words, the history will be cleared of the unnecessary commit. In our case, the commit that broke Boopi.

But let's see what actually happened.

Whenever you revert changes, Git instead of re-writing the history and removing the existing commit creates a new commit with mirrored changes. In our case, in that commit, we created a single file called *banana.txt* and added it to the commit. After running *git revert* Git removed this file for us and created a new commit with changes. If we didn't create the file but only made some changes to it, Git would remove those changes and create a new commit without removing a file.

I hope this makes sense and you will be able to undo any changes that could be harmful in some way.

Woof-woof! Yes, it seems like Boopi is working again! Are you alright Boopi?

Boop-boop!

Alright!

Let's move on to the next chapter, where we'll learn how not to break Boopi with files that weren't exactly intended for it.

14

Ask Git to ignore files

In the previous chapter, we created a file called *banana.txt* and provided incorrect instructions on how to craft a thing. In the end, we broke Boopi and had to fix him. To prevent such cases in the future we can use another feature that Git provides to us.

But before I name this feature, let's think about how we could tell Boopi not to read unknown files in the folder.

The first thing that comes to my mind is to create the file but not include it in the commit. In other words, don't use the *git add* command on such files that may break Boopi. But if we had more than one file, say twenty or forty files. They will mess up the output of the *git status* command and make a noise like many unused "untracked files".

The second thing we could do is create a separate branch where we put all the files that might break Boopi. But what if some of those files would be needed in the current branch? Not all files need to be committed to Git, but they can also be useful to us. For example, a file with passwords. It would be unnecessary to put files with sensitive information into the Git history. What if Boopi accidentally creates a thing that will have your password written on it. This will reveal your password to others. Not what you expect, right?

As you can see, all of the above solutions can work with a little tweaking, but they were not designed to solve such problems.

Woof-woof! Ah-ha! Jessie asked me to finally name the feature and stop beating around the bush.

This feature is called - a *.gitignore* file.

So, the *.gitignore* is a special file in which we can specify which files will not be tracked by Git and accidentally committed by us, even though we run *git add .* (to include all files). If the file is not added to the commit, then Boopi will not be able to craft it and accidentally break. This is exactly what we want to achieve.

> NOTE: Please don't forget to add the . (dot) at the start of the file (e.g. .gitignore). It is required.

Usually, the *.gitignore* file contains a list of other files and folders that are excluded from the tracking area of Git. If you create a file with the name which is listed in this *.gitignore* this file will not be visible for *git status* or *git add* commands. It would be the same if you said: "Hey Git, everything that is called like this and that is ignored".

Let's create this file with the name *.gitignore* inside our folder and open it.

Inside opened *.gitignore* file we will list all files that we want to ignore in the future

```
passwords.txt
bad-instructions.txt
```

Save and close the file.

Starting from this point we asked Git not to track files that were listed in the *.gitignore* file (e.g. *password.txt* and *bad-instructions.txt).*

Let's verify my words and create one of those files besides the recently created *.gitignore* file.

We can create a *password.txt* file.

Open the file and provide some fake passwords inside.

```
Password 123
```

Save and close the file.

This file is created in our folder and visible to us. But this file is not "visible" for Git. Let's run *git status* in the console and verify that Git doesn't say that we have a new untracked file called *password.txt*.

```
git status
```

It should print

```
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
.gitignore

nothing added to commit but untracked files present
(use "git add" to track)
```

As you can see, *git status* doesn't say that we created a *password.txt* file. It only shows that a new *.gitignore* file was added to the folder. As far *.gitignore* is a special file that Git and Boopi know how to handle we can make a commit with this file. Yes, it is good practice (or required) to commit a *.gitignore* file.

Let's add this file to the staged area and create a commit message like the following

```
git add .gitignore
git commit -m "Add special .gitignore file to the
project"
```

From now on, if you are going to create any technical file (or folder) that should not be included in future commits, you can adjust the *.gitignore* file as you want. But do not forget to commit the *.gitignore* file after each update. By the way, the *.gitignore* file starts working from the moment it is created, and not from the moment it is committed, but it is still recommended to commit it.

Woof-woof-woof! Ah-ha! Jessie thanked you for helping us prevent future breakdowns of Boopi.

Woof-woof! Oh, how could I forget?

Jessie said Boopi's birthday is coming soon, let's surprise him in the next chapter!

15

Make a surprise for Boopi

Woof-woof. Yes, Jessie? Woof-woof. I remember, Jessie, that Boopi has a birthday today. Are you thinking about what I am? Let's make a gift for Boopi!

Woof! Exactly! We can create a big cake for him. But wait.

How we can prepare instructions for the cake without ruining

the surprise?

Woof-woof! Good point, Jessie, we can create a cake in another branch. Let's do this.

So, let's create a new branch called *cake* where we prepare instructions on how to bake it!

```
git checkout -b cake
```

We have not used this shorter form of creating a branch before. Let's understand what it does. Actually, this command consists of two parts: create a new branch and switch to it.

It would be the same as if we could create a branch and make it active in two separate steps.

```
git branch cake
git checkout cake
```

Now that this is clear, let's get back to our cake.

Our cake will consist of several parts: dough, jam, and cherry on top of it. Let's create files for each of these parts and write instructions for them.

First off, we will provide instructions for the dough. Let's create a file with the name *dough.txt*.

Open this file and provide instructions inside.

```
Step 1. Knead the dough
Step 2. Wait until the dough rises
Step 3. Bake until delicious
```

Save the file and close it.

Secondly, create another file called *jam.txt.*

Open this file and provide instructions inside.

```
Step 1. Crush ripe cherries
Step 2. Add sugar
Step 3. Stir
```

Save the file and close it.

So, next...

Woof-woof! Yes, Jessy? WOOF-WOOF-WOOF!!! It can't be! One of Jessie's friends falls into the water, and we need to help him.

But what can we do? Think think!

Woof-woof!

Excellent idea, Jessie! We need a lifebuoy!

But where do we get it?

Woof-woof!

Of course! Boopi can craft it for us. Let's craft it!

But wait, we're creating a cake for Boopi. If we switch to the *master* branch, all the files with instructions that we created so far will be moved to the *master* branch. In the best case, Boopi will know that we are preparing a surprise, and in the worst case, this will create confusion along with the problems that we have right now!

Do we have any commands in Git that could help us solve this problem?

Woof-woof! That's right, Jessie!

Of course, we have! We can use the *git stash* command.

Let's use it!

Before we use this command, you can think of stashing as a process of moving modified or untracked files out of the project into a place where Git can hold them for a while. This place is not visible neither for us nor for Boopi. In the end, we will be able to get those files back when we need them. This option suits us! What do we need to do?

Nothing much. We need to write the following command in the already opened console

```
git stash --include-untracked
```

Usually, we can use *git stash* without *—include-untracked* option. But as far as we didn't add our files to the staged area (that is, didn't use *git add* on them) yet, we need to provide an extra option that tells Git to hide even files that Git didn't care about before.

Check your folder. The *jam.txt* and *doubt.txt* files should have disappeared. It means that you successfully stashed them. We will return them soon, but firstly let's rescue Jessie's friend!

Now we need to create a new branch called *lifebuoy*. But remember that we need to take into account which branch we are branching off from. As a general rule, we usually want to create a branch based on the branch we want to apply changes to later. This means that in order to merge the *lifebouy* branch into the *master* branch in the future, we need to branch from the *master* now.

Let's switch back to the *master* branch and create a new branch based on it.

```
git checkout master
git checkout -b lifebuoy
```

Let's verify that we're on a new branch by writing.

```
git branch --show-current
```

It should print *lifebuoy*

If everything is correct, we can create a file, *lifebouy.txt*.

Open this file and provide instructions.

```
Step 1. Create a circle with a radius of 10 cm
Step 2. Paint it in orange and white colors
Step 3. Add rope around the circle
```

Save the file and close it.

Our instructions are ready! Let's create a commit and give it to Boopi as soon as possible

```
git add lifebouy.txt
git commit -m "Add lifebouy.txt"
```

Nice! Let's move back to *master* and merge it

```
git checkout master
git merge lifebuoy
```

The *git merge* command tells Git to take all the unique commits from the *lifeline* branch and copy them to the *master* branch if there are no conflicts.

Oh, look at Boopi! Woof-woof! Yes, good job Boopi! Jessie, show us where we need to go. Let's go!

I see him, throw it there Boopi! Take a lifebuoy friend!

Good job, Boopi, and good job everyone!

What is your name, little fluffy?

Woof-woof.

Ah, Thomas, nice to meet you, Thomas. This is the reader, Boopi, and Me. Next time be careful, please!

While Jessie and Boopi continue to sit with wet Thomas we can go back and finish our surprise for Boopi. But shhh! Boopi doesn't suspect a thing!

Let's quietly switch back to the branch cake

```
git checkout cake
```

At this moment, previously created files (*dough.txt* and *jam.txt*) should not exist here, that's correct because we asked Git to stash them!

Ahem. Why am I still whispering?

We can actually check if our files are still in the stash by running

```
git stash list
```

It should show a single record with our files.

```
stash@{0}: WIP on master: 57da989...
```

Instead of showing our files, it shows the record which holds our files.

Let's bring our files back by using

```
git stash pop
```

This command will take the latest record from the stash and apply it (or in other words returns your previously stashed files back).

In general, we can stash as many times as we want. All our files will be stored in the stack structure. Where the latest stashed files will be stored on top of it. If we want to bring it back we can either specify an id of the record or take the latest one from the stash. As far we stashed only once and our record is the only record on the stack we could simply use git stash pop.

```
git stash pop
```

Let's write *git status* and ensure that Git returned our files.

```
git status
```

Yes, it shows that *jam.txt* and *dough.txt* are untracked files as we had before.

```
On branch cake

Untracked files:
  (use "git add <file>..." to include in what will be
```

```
  committed)
  dough.txt
  jam.txt


nothing added to commit but untracked files present
(use "git add" to track)
```

Okay, our two of three files are ready. Let's create the missing part of the cake by creating a *cherry.txt* file and providing instructions inside it

Open the newly created file *cherry.txt* and provide instructions inside it

```
Step 1. Create a cherry
```

Save the file and close it.

As a result, we should have three files: *jam.txt*, *dough.txt*, and *cherry.txt*. Let's add them to the staged area and create a commit

```
git add .
```

This is another shortened version of how to add files to the staged area. The . (dot) means add all files untracked files or files with unstaged changes to the staged area. We could achieve the same result by manually specifying each file (e.g. *git add jam.txt dough.txt cherry.txt*), but it doesn't change the point.

The last thing that we need to do is to create a commit with all

three files. We usually create commits with a short title, but today is Boopi's special day, let's add congratulations to him.

By running the following command

```
git commit
```

Git should open your already familiar window inside your default editor

On the first line, we will put a title

```
Surprise!
```

We will skip the second line, and on the third, we will put our congratulations for Boopi

```
Happy birthday! We hope all your birthday wishes and
dreams come true

Sincerely,
Jessie, Thomas, Reader, Athor and others
```

The full commit message should look like this

```
Surprise

Happy birthday! We hope all your birthday wishes and
dreams come true

Sincerely,
Jessie, Thomas, Reader, Athor and others
```

Okay, save changes and close the file.

Let's switch back to the *master*

```
git checkout master
```

Woof-woof. I know, I know Jessie. We're back...

Dear Boopi, birthdays are a new start, a fresh beginning, and a time to pursue new endeavors with new goals. Move forward with confidence and courage. You are a very special person. May today and all of your days be amazing!

Reader, do the last thing that you have to do.

```
git merge cake
```

HAPPY BIRTHDAY!

16

Last words

It's been a long journey where we mainly covered the most basic Git commands. Using those commands, we created toys for two of Jessie's friends and rescued a fallen Thomas with a lifebuoy. We even managed to break the Boopi by accident and fix him back. The *.gitignore* file helped us not to make such mistakes in the future. In addition, we have overcome merge conflicts, which were considered the scariest topic while learning Git.

Yes, it was fun!

Here is the list of all commands that we used as a tool for achieving our goals:

- *git init*
- *git add*
- git diff
- *git commit*
- *git status*
- *git log*

- *git branch*
- *git revert*
- *git stash*
- *git merge*

with many other optional attributes that can be applied to existing commands and modify their execution:

- *git log -1*
- *git checkout -b red-apple*
- *git stash —include-untracked*

and many more...

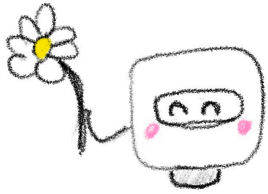
It would make no sense to overload the introduction book with other equally useful commands that have remained untouched. So I hope this book will help you start your new journey of learning Git.

So, Jessie and others, it's time to say bye to the reader!

Woof-woof!

Oh, hold on!

It seems like Boopi, what's to give you something.



Bi-boop!