

# فصل پنجم

## غواصی درون گیت

اکنون که شما با دستورات اساسی **Git** آشنا هستید ، ما به سایر ویژگی های موجود در آن می پردازیم. در این فصل ویژگی هایی را که در فصل 2 به شما قول داده ام ، کشف خواهید کرد.






### نادیده گرفتن پرونده ها

همه چیز در فهرست کار نباید توسط **Git** ردیابی شود. فایل های خاصی (پیکربندی ، گذرواژه ها ، کد بد) وجود دارند که عموماً توسط نویسندگان یا برنامه نویسان بدون کنترل باقی می مانند.

این پرونده ها (یا فهرستها) در یک پرونده ساده با نام **"gitignore"** ذکر شده اند. به دوره قبل از **"git ignore"** توجه کنید. آن مهم است. برای نادیده گرفتن پرونده ها ، پرونده ای را ایجاد کنید. **gitignore** پرونده ها یا پوشه ها را لیست کنید تا در آن نادیده بگیرد. بیایید از قسمت قبلی ، لیست توگوها به مخزن ما برگردیم. بیایید تصور کنیم که می خواهید یک پرونده خصوصی و غیرمستقیم با نام **PRIVATE.txt** را درج کنید. ابتدا باید فایل **gitignore** را با استفاده از ویرایشگر متن مورد علاقه خود ایجاد کرده و سپس **PRIVATE** را بنویسید.



اگر فایل **PRIVATE.txt** را ایجاد و اصلاح کنید (مانند شکل 5-2) اگر وضعیت را بررسی کنید ، توسط **Git** در نظر گرفته نمی شود.

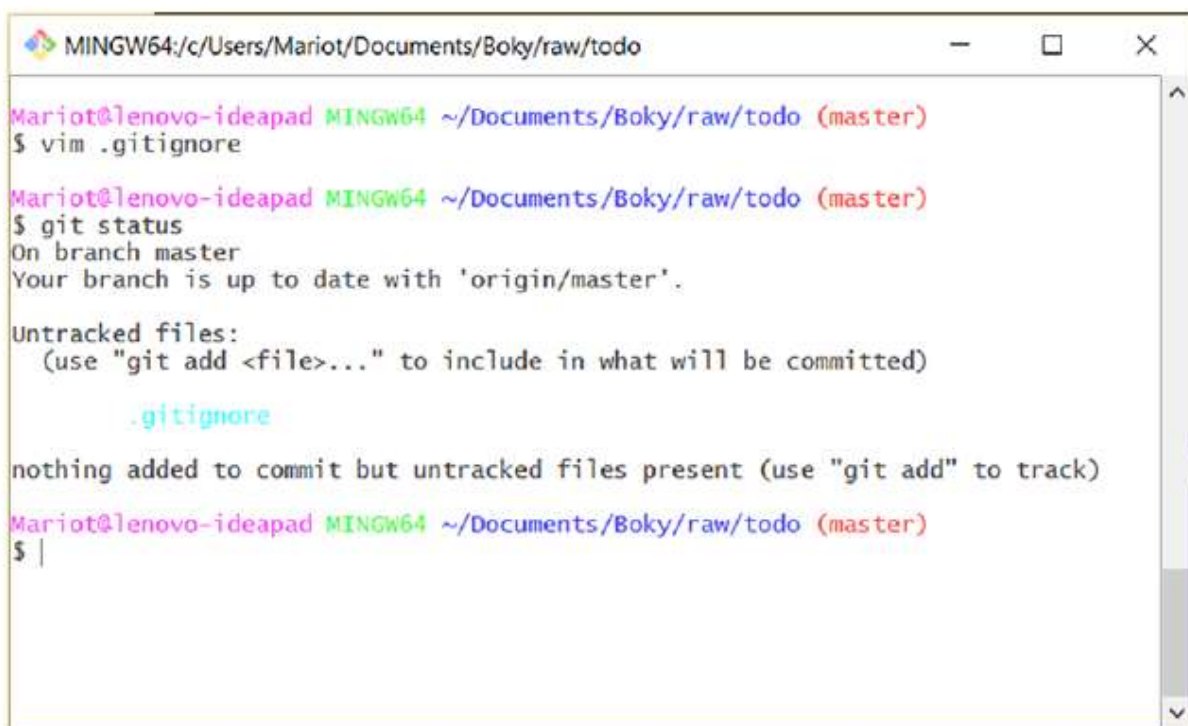
Name	Date modified	Type	Size
 .gitignore	2019-05-23 20:51	Git Ignore Source ...	1 KB
 DOING.txt	2019-05-23 20:17	Text Document	0 KB
 DONE.txt	2019-05-23 20:17	Text Document	1 KB
 PRIVATE.txt	2019-05-23 20:10	Text Document	0 KB
 TODO.txt	2019-05-23 20:15	Text Document	1 KB

**Figure 4-2. Adding PRIVATE.txt**

بیایید سعی کنیم وضعیت را بررسی کنیم.

```
$ git status
```

نتیجه مشابهی را که در شکل 3-5 نشان داده شده است دریافت خواهید کرد.



```

MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ vim .gitignore
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
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)
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |

```

**Figure 4-3. Status of the working directory**

همانطور که در وضعیت نشان داده شده در شکل 5-3 مشاهده می کنید ، PRIVATE.txt ردیابی نمی شود. همچنین می توانید مشاهده کنید که پرونده .gitignore ردیابی شده است. بنابراین ، شما مجبور خواهید بود پس از اصلاح کردن ، آن را اضافه کرده.

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

مانند همیشه ، تشکیل پرونده و سپس اجرای پروژه منجر به ارسال پیام می شود که خلاصه تغییرات انجام شده را نشان می دهد (در شکل 5-4 نشان داده شده است).



```
MINGW64/c:/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
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)
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git add .gitignore
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git commit
[master b2eccfb] Add .gitignore
1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
```

**Figure 5-4. Committing .gitignore**

به یاد داشته باشید که پرونده جهانی .gitignore باید در ریشه مخزن شما قرار بگیرد. اگر آن را در یک فهرست قرار دهید ، فقط پرونده های مربوط به آن فهرست را نادیده می گیرند. به طور کلی ، داشتن چندین فایل .gitignore در چندین دایرکتوری یک حرکت بد محسوب می شود مگر اینکه پروژه شما بسیار زیاد باشد. ترجیح می دهید آنها را در یک gitignorefile منفرد قرار دهید که در ریشه مخزن شما قرار دارد.

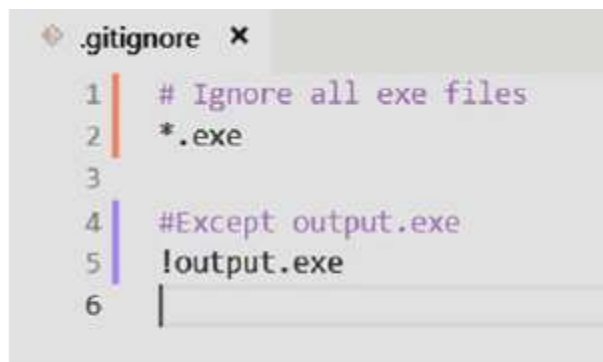
ممکن است از خود پرسید که هنگام استفاده از Git چه نوع فایل هایی را نادیده بگیرید. خوب ، قانون انگشت نادیده گرفتن کلیه پرونده های تولید شده توسط پروژه است. به عنوان مثال ، اگر پروژه شما کد منبع نرم افزاری است ، باید خروجی های کامپایل شده (پرونده های اجرایی یا ترجمه شده) را نادیده بگیرید. پرونده ها و پرونده های موقت نیز باید همراه با کتابخانه های بزرگ (node\_modules) کنار گذاشته شوند. فراموش نکنید که تمام پیکربندی های شخصی و پرونده های دمای ویرایشگر متن خود را حذف نکنید.

پرونده .gitignore تنها پرونده هایی را که با نام مشخص شده اند ، نادیده نمی گیرند. همچنین می توانید فهرستها و پرونده های منطبق با توضیحات را نادیده بگیرید. در جدول 5-1 یک یادآوری مفید از کلیه قالب هایی که می توانید استفاده کنید ، خواهید یافت.

<b>.gitignore line</b>	<b>What is ignored</b>	<b>Example</b>
config.txt	config.txt in any directory	config.txt local/config.txt
build/	Any build directory and all files in it. But not a file named build	build/target.bin build/output.exe NOT output/build
build	Any build directory, all files in it, and any file named build	build/target.bin output/build
*.exe	All files with the extension .exe	target.exe output/res.exe
bin/*.exe	All files with the extension .exe in the bin/ directory	bin/output.exe
temp*	All files with name beginning by temp	Temp temp.bin temp_output.exe
**/configs	Any directory named configs	configs/prod.py local/configs/preprod.py
**/configs/local.py	Any file named local.py in any directory named configs	configs/local.py server/configs/local.py NOT configs/tr/local.py
output/**/result.exe	Any file named result.exe in any directory inside output	output/result.exe output/latest/result.exe output/1991/12/16/result.exe

اینها رایج ترین خطوط مورد استفاده با **gitignore** هستند. برخی دیگر نیز وجود دارند اما فقط در موقعیت های خاص مورد استفاده قرار می گیرند و تقریباً هرگز در پروژه های مشترک مورد استفاده قرار نمی گیرند. اگر از زبان رایانه ای یا چارچوبی استفاده می کنید ، می توانید به <https://github.com/Github/git> نادیده بگیرید تا یک الگوی پرونده **gitignore** مورد استفاده خود را دریافت کنید.

اما اگر می خواهید تمام پرونده های منطبق با توضیحات را به جز یک مورد نادیده بگیرید ، چه می شود؟ خوب ، شما می توانید به **Git** بگویید که تمام پرونده ها را نادیده گرفته و سپس بلافاصله استثنائی ایجاد کنید. برای حذف پرونده از لیست نادیده گرفته شده ، از "!" استفاده می کنید. به عنوان مثال ، اگر می خواهید تمام فایل های **exe** به جز **outout.exe** را نادیده بگیرید ، می توانید **gitignore** خود را مانند شکل 5-5 بنویسید.



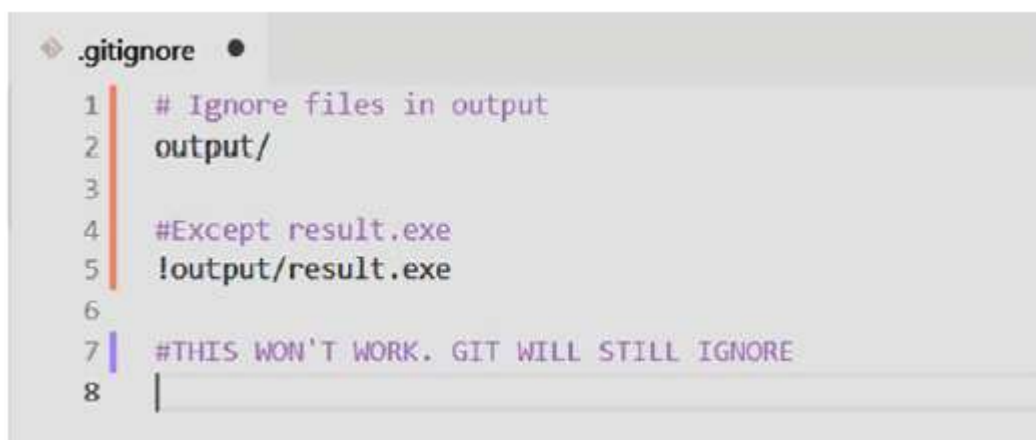
```

1 | # Ignore all exe files
2 | *.exe
3 |
4 | #Except output.exe
5 | !output.exe
6 |

```

### How to make an exception

توجه داشته باشید که ترتیب خطوط. استثناء بعد از قانون به وجود می آید! این استثنائات فقط برای خطوطی که نام فایل ها را توصیف می کنند ، کار می کند. شما نمی توانید با استفاده از خطوط نادیده گرفته شده از فهرست ، از آن استفاده کنید. پرونده Agitignore همانطور که در شکل 5-6 نشان داده شده است کار نمی کند.



```

1 | # Ignore files in output
2 | output/
3 |
4 | #Except result.exe
5 | !output/result.exe
6 |
7 | #THIS WON'T WORK. GIT WILL STILL IGNORE
8 |

```

استثناء با پرونده هایی که توسط تطبیق دایرکتوری نادیده گرفته نمی شوند کار می کنند مخزن خود را از تمرین قبلی بگیرید و چندین فایل و فهرست را ایجاد کنید. جدول 5-1 را بررسی کنید و سعی کنید پرونده هایی که با استفاده از هر خط ایجاد کرده اید را نادیده بگیرید. هر تعداد الگوی مورد نیاز خود را ایجاد کنید و متوقف نشوید. نیازی نیست همه چیز را به خاطر بسپارید ، اما حداقل باید تصویری از زمان استفاده از آنها داشته باشید.

این موارد به چه معنی است

شکل 5-7 را بررسی کنید. بدون نگاهی به بخش قبلی ، هر خط چه چیزی را نادیده می گیرد؟



**Figure 5-7.** Guess what each line ignores

و اینگونه است که پرونده ها را نادیده می گیرید! تقریباً به آسانی نادیده گرفتن مسئولیتهای شماست! اما به یاد داشته باشید: پرونده **gitignore** ردیابی و نسخه شده است ، بنابراین پیراشکی را فراموش نکنید که قبل از ارتکاب آن را مرحله بزنید!

## Checking logs and history

اگر تمرینات را دنبال کردید (همانطور که باید) یا شروع به استفاده از **Git** برای پروژه های خود کرده اید ، اکنون مشکل کمی وجود دارد که من قول داده ام به راحتی با **Git** حل شود: چگونگی مشورت با سابقه تاریخ. این یکی از پرکاربردترین ویژگی های **Git** و همچنین یکی از ساده ترین دستورات **Git** است:

Git log

\$ git log

امتحان کن مخزن خود را باز کرده و دستور را اجرا کنید. شما باید نمایی مانند آنچه در شکل 5-8 نشان داده شده است را ببینید.

```
MINGW64~/c:/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log
commit b2eccfbf5b54c0f5b6d34b2432245a1a582a96f6 (HEAD -> master)
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 21:20:51 2019 +0200

    Add .gitignore

commit 5f57824bdc7b704d17e8a9cbf36146f43eb0269a (origin/master)
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:18:12 2019 +0200

    Finish task 1: mittens

commit 9f180aae6d70f83a5252b0d1be2d68321f5b2146
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:17:11 2019 +0200

    Doing task 1: mittens

commit 1c3f05747ab8a5416d1be8efbbd3865206681275
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:15:26 2019 +0200

    Create TODO

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |
```

گزارش مربوط به تعهدات ، تمام عکسهای شما یا سایر افراد مرتکب شده (از جدیدترین تا قدیمی ترین) را درج می کند. این شامل همچنین ، برای هر مرتکب

- نام (منحصر به فرد ، به دست آمده توسط هش)

- نویسنده

- تاریخ

- توضیحات

از آنجا که نام های متعهد بسیار طولانی هستند ، ما فقط از پنج حرف اول به عنوان نام استفاده خواهیم کرد. این برای بخش بعدی مهم خواهد بود.

اگر سابقه تعهد شما بسیار طولانی است ، می توانید از صفحه کلید استفاده کرده و یک خط را به جلو یا عقب بروید: کلید بالا و پایین را وارد کنید یا کلیدهای j و k

- یک پنجره رو به جلو یا عقب: f و b

- در پایان ورود: G

- در ابتدای ورود: g

- کمک بگیرید: h

- از ورود به سیستم خارج شوید: q



پارامترهای بسیاری وجود دارد که می توانید با استفاده از `log git` از آنها استفاده کنید. جدول 2-5 آنها را به شما ارائه می دهد.

Command	Use	Example
<code>git log --reverse</code>	Reverse the order of commits	
<code>git log -n &lt;number&gt;</code>	Limit the number of commits shown	<code>git log -n 10</code>
<code>git log --since=&lt;date&gt;</code> <code>git log -after=&lt;date&gt;</code>	Only show commits after a certain date	<code>git log --since=2018/11/11</code>
<code>git log --until=&lt;date&gt;</code> <code>git log --before=&lt;date&gt;</code>	Only show commits before a certain date	
<code>git log --author=&lt;name&gt;</code>	Show all commits from a specific author	<code>git log --author=Mariot</code>
<code>git log --stat</code>	Show change statistics	
<code>git log --graph</code>	Show commits in a simple graph	

### مشاهده نسخه های قبلی

اکنون که می دانید چگونه تاریخ را بررسی کنید و `commit` ها را چک ، زمان آن است که پرونده ها را بررسی کنید تا ابتدا ببینید چه پرونده هایی تغییر یافته اند. آن نامهای طولانی را که با هر `commit` ایجاد می شوند ، به خاطر دارید؟ می توانید از آنهایی استفاده کنید تا بین تعهدات یا عکس های فوری حرکت کنند. برای بررسی اینکه چگونه پرونده های شما در یک عکس فوری خاص بوده اند ، فقط باید نام آن را بدانید. بهترین روش برای شناخت نام هر تعهد ، بررسی تاریخچه مانند شکل 2-9 است.



```
MINGW64/c:/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log
commit b2eccbf5b54c0f5b6d34b2432245a1a582a96f6 (HEAD -> master)
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 21:20:51 2019 +0200

    Add .gitignore

commit 5f57824bdc7b704d17e8a9cbf36146f43eb0269a
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:18:12 2019 +0200

    Finish task 1: mittens

commit 9f180aae6d70f83a5252b0d1be2d68321f5b2146
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:17:11 2019 +0200

    Doing task 1: mittens

commit 1c3f05747ab8a5416d1be8efb3865206681275
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:15:26 2019 +0200

    Create TODO

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |
```

برای نشان دادن و یادگیری تغییراتی که در پروژه شما ایجاد شده است ، فقط از دستور "git show" استفاده می کنید که به دنبال آن نام **commit** است. شما حتی لازم نیست که نام کامل ، فقط هفت حرف اول را بنویسید.

\$ git show <name>

با مخزن خود امتحان کنید! شما باید نتیجه مطابق شکل 5-10 دریافت کنید.

MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo

```
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git show 9f180aa
commit 9f180aae6d70f83a5252b0d1be2d68321f5b2146
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:17:11 2019 +0200

    Doing task 1: mittens

diff --git a/DOING.txt b/DOING.txt
new file mode 100644
index 0000000..c6a584e
--- /dev/null
+++ b/DOING.txt
@@ -0,0 +1 @@
+-Put the mittens on the kittens
\ No newline at end of file
diff --git a/TODO.txt b/TODO.txt
index cb72b4b..02c3043 100644
--- a/TODO.txt
+++ b/TODO.txt
@@ -1,4 +1,3 @@
--Put the mittens on the kittens
-Buy a hat for the bat
-Clear the fogs for the frogs
-Bring a box to the fox
\ No newline at end of file

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |
```

همانطور که مشاهده می کنید ، این **commit** به روشی بسیار دقیق نشان داده شده است. تفاوت بین **commit** انتخاب شده و نمونه قبلی را خواهید دید. موارد اضافی با رنگ سبز و حذف آنها به رنگ قرمز نشان داده شده است. می توانید جزئیات مربوط به هرگونه ارتکاب را با دستور "**git show**" نشان دهید.

### بررسی تغییرات فعلی

بررسی نسخه های قبلی خوب است ، اما اگر فقط بخواهید تغییراتی را که تازه ایجاد کرده اید بررسی کنید؟ بررسی تفاوت بین آخرین **commit** و فهرست کار فعلی یک ویژگی اساسی **Git** است. شما از آن استفاده خواهید کرد! دستور بررسی تفاوتها ساده است: **git diff**.

**\$ git diff**

یک یا چند فایل را در دایرکتوری خود اصلاح کنید و سپس دستور را اجرا کنید. نتیجه مطابق شکل 5-11 دریافت خواهید کرد که بسیار شبیه به نتیجه دستور `git show` از قسمت قبلی است. آنها در واقع همان دیدگاه هستند زیرا اطلاعات نشان داده شده یکسان هستند.

```
MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ ls
another/ build/ DOING.txt folder/ PRIVATE.txt TODO.txt

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ vim TODO.txt

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git diff
diff --git a/TODO.txt b/TODO.txt
index 02c3043..1fbd05d 100644
--- a/TODO.txt
+++ b/TODO.txt
@@ -1,3 +1,4 @@
- Buy a hat for the bat
- Clear the fogs for the frogs
-- Bring a box to the fox
\ No newline at end of file
+ Bring a box to the fox
+ Make a combo with the dodo

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$
```

بررسی تمام تغییرات انجام شده در فهرست کار

بیشتر اوقات ، شما فقط باید تغییرات ایجاد شده در یک پرونده را بررسی کنید ، نه برای کل پروژه. می توانید نام فایل را به عنوان پارامتر منتقل کنید تا اختلافات آن در مقایسه با آخرین تعهد بررسی شود.

`$ git diff TODO.txt`

نکته اصلی که باید بخاطر بسپارید این است که `git Diff` تغییرات ایجاد شده در پرونده های موجود در فهرست را بررسی می کند. پرونده های مرحله بندی شده را بررسی نمی کند! برای بررسی تغییرات ایجاد شده در پرونده های مرحله بندی شده ، باید از پارامتر `--staged` استفاده کنید.

`$ git diff --staged`

شما همیشه باید قبل از انجام یک پروژه ، تفاوت در پرونده های مرحله بندی شده را بررسی کنید ، بنابراین می توانید یک بررسی نهایی انجام دهید. می دانم که یک روز چنین کاری را فراموش خواهید کرد ، بنابراین به فصل بعد بروید تا یاد بگیرید که چگونه می توانید تعهد خود را خنثی سازی یا اصلاح کنید.

این پایان این فصل است و ما چیزهای زیادی را آموخته ایم. لطفاً قبل از رفتن به فصل بعد، اطمینان حاصل کنید که از این ویژگی ها راحت هستید:

- نادیده گرفتن پرونده ها
- بررسی سوابق تاریخ
- بررسی تغییرات محلی و مرحله ای

## خلاصه

این فصل در مورد تاریخچه پروژه بود. ما در مورد ورود به سیستم با ورود به سیستم `git log` و `git show` آشنا شدیم اما یاد گرفتیم که تغییرات فعلی را با `git diff` بررسی کنیم. `Git log` و تفاوت آن در آینده بسیار مفید خواهد بود، بنابراین مطمئن شوید که آنها را به خوبی درک کرده اید.

`Git Diff` مربوط به مقایسه پرونده های اصلاح شده فعلی با پرونده های آخرین مرتبه است، در حالی که ورود آن فقط لیستی از تمام تعهدات قبلی است.

امکان نادیده گرفتن پرونده ها با `gitignore` همچنین یک مهارت خوب است که وضعیت `git` شما با پرونده های تغییر یافته اشخاصی که شما علاقه ای به ارتکاب آن ندارند، اشباع نشود. همچنین این یک راه خوب برای اطمینان از عدم اجرای یک پرونده خاص (احتمالاً حاوی کلیدهای مخفی) است.

ما هنوز در فصل بعد چیزهای زیادی برای یادگیری در مورد تعهدات داریم. ما ابتدا سه حالت پرونده های `Git` را مرور خواهیم کرد و سپس خواهیم دید که چگونه می توان نسخه های قبلی را به فهرست کارگردانی برگرداند. و شما حداقل یاد خواهید گرفت که چگونه خنثیسازی و اصلاح تعهدات را تغییر دهید.

# فصل پنجم

# Commits

فصل قبل کمی در مورد ویژگی های اساسی Git کمی به شما آموخت. شما باید بدانید که چگونه تاریخچه تاریخ را بررسی کنید و تغییرات ایجاد شده در نسخه فعلی را ببینید ، اما این امر باعث شده است که یک مشکل برای نیش زدن وجود داشته باشد ، بنابراین می خواهیم در این فصل بیشتر درباره آنها صحبت کنیم. اول ، ما به بررسی (دوباره) کار درونی گیت و اصطلاحات آن خواهیم پرداخت. سپس ، می آموزیم که چگونه نسخه های قبلی را مشاهده و بررسی کنیم. بیا بریم!

## The three states of Git

قبل از صحبت کردن در مورد جزئیات بیشتر ، ما باید به اصول اولیه برگردیم و درباره نحوه کار Git صحبت کنیم. مطمئناً سه حالت را به خاطر دارید که یک پرونده می تواند خودش را پیدا کند. اگر این کار را نکردید ، از این فصل پرش نکنید. برای هر کاری که با Git انجام دهید ضروری است. اگر به خاطر دارید ، آن را از دست ندهید ، زیرا من وقت زیادی را برای نوشتن آن صرف کردم.

همانطور که در فصل گذشته دیدید ، همه پرونده ها توسط Git ردیابی نمی شوند. برخی پرونده ها نادیده گرفته می شوند (توسط پرونده `.gitignore`). و سپس فایل هایی نیز وجود دارند که توسط Git نادیده گرفته نشده اما هنوز ردیابی نشده اند. آنها پرونده های تازه ایجاد شده هستند که هرگز جزئی از عکس فوری (متعهد) نشده اند.

پرونده های ردیابی شده می توانند در سه حالت باشند:

- اصلاح شده: شما پرونده را تغییر داده اید.
  - Staged: شما پرونده را تغییر داده و آن را آماده تصویربرداری کردید.
  - committed: شما یک عکس فوری از کل پروژه گرفتید و پرونده در آن قرار داشت.
- پرونده های غیرقابل استفاده تا زمانی که تصمیم به صحنه آمدن و ارتکاب آنها بگیرید یا صریحاً از آنها چشم پوشی کنید ، به همین ترتیب باقی می مانند.
- به یاد داشته باشید: Git تغییرات را ردیابی نمی کند ، عکس های فوری را ردیابی می کند. هر بار که `commit` می زنید ، وضعیت کل پروژه صرفه جویی می شود ، نه فقط تغییرات کمی که ایجاد شده اند.

واقعیت : **Git** سریع است زیرا شما همیشه روی آخرین وضعیت پروژه کار می کنید. وقتی می خواهید **commit** قبلی را ببینید ، فقط وضعیت پروژه را در آن نشان می دهد

بسیاری از **VCS** ها هر تغییری را که در یک پرونده انجام شده بود ذخیره می کردند و وقتی می خواستید به حالت قبلی برگردید ، آنها دوباره به صورت معکوس تغییر می کنند. هنگامی که پروژه بزرگ می شود ، این باعث مشکلات زیادی یا سرعت و حافظه می شود. آیا روش تفکر **Git** نمی تواند پایگاه داده های فوق العاده بزرگی ایجاد کند؟ نه ، زیرا وقتی یک عکس فوری می گیرید و یک پرونده تغییر نمی دهد ، دوباره ذخیره نمی شود. در عوض ، از یک مرجع به پرونده استفاده می شود.

بیا یاد دوباره به سه ایالت برگردیم و رابطه بین آنها را ببینیم:

- شما در فهرست کار می کنید. این فقط فهرست شماست

قبل از شروع مخزن موجود است. جایی که فایل های خود را خوانده و ویرایش خواهید کرد.

- منطقه مرحله بندی جایی است که شما قبل از گرفتن عکس فوری از کل پروژه ، پرونده های تغییر یافته خود را قرار می دهید. اگر فایل های تغییر یافته خود را روی صحنه بریزید ، نمی توانید عکس فوری بگیرید. فقط پرونده های مرحله بندی شده (و پرونده های بدون تغییر) در عکس فوری مورد توجه قرار می گیرند. پرونده های غیر فعال (ردیابی یا غیرفعال) و پرونده های نادیده گرفته شده فقط در همان حالت باقی می مانند.

- دیتابیس یا دایرکتوری **git**. هر عکس را که شما گرفتید ، ذخیره می کند. به عکس های فوری **commit** گفته می شود.

## پیمایش بین نسخه ها

در بسیاری از مواقع ، شما نه تنها می خواهید بدانید که در پروژه شما چه تغییری کرده است ، بلکه می توانید ببینید که در چه وضعیتی بوده است ، تا عکس فوری را که گرفتید ببینید. با گیت آسان است.

هنگامی که شما می خواهید وضعیت قبلی پروژه را به فهرست کار بیاورید ، ما باید تعهد را با "پرداخت **git check**" بررسی کنیم. از آنجا که این پرونده ها را در دایرکتوری کار تغییر می دهد ، شما باید مطمئن شوید که هیچ پرونده غیرفعال در آن وجود ندارد.

پرونده های بدون دسترسی خوب هستند زیرا **Git** هنوز وضعیت آنها را ردیابی نمی کند. برای بررسی عکس فوری از پروژه ، از دستور "**git checkout**" استفاده می کنیم و نام **commit** را به عنوان پارامتر منتقل می کنیم.

```
$ git checkout <name>
```

بیا یاد امتحان کنیم! پروژه فعلی خود را در یک ویرایشگر متن باز کنید و از مطالب آن یادداشت کنید.

اکنون **commit** قبلی را مانند شکل 6-1 بررسی کنید.



```
MINGW64/c/Users/Mariot/Documents/Boky/raw/todo

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log
commit 8ba74a5546782e38d1c2d6dafd2386e814034c69 (HEAD -> master)
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Mon May 27 21:31:51 2019 +0200

    Rearrange .gitignore

commit b2eccbf5b54c0f5b6d34b2432245a1a582a96f6
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 21:20:51 2019 +0200

    Add .gitignore

commit 5f57824bdc7b704d17e8a9cbf36146f43eb0269a
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:18:12 2019 +0200

    Finish task 1: mittens

commit 9f180aae6d70f83a5252b0d1be2d68321f5b2146
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:17:11 2019 +0200

    Doing task 1: mittens

commit 1c3f05747ab8a5416d1be8efbbd3865206681275
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:15:26 2019 +0200

    Create TODO

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git checkout 9f180aa
Note: checking out '9f180aa'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

HEAD is now at 9f180aa Doing task 1: mittens
```

اگر ویرایشگر متن خود را بررسی کنید ، متوجه می شوید که این پروژه اکنون دقیقاً مثل زمانی است که شما هنگام گرفتن عکس فوری اقدام کرده اید. این بهترین چیزی است که با Git وجود دارد. هیچ چیزی که از شما عکس فوری گرفته اید از بین رفته است!

اکنون بیایید برخی اصطلاحات Git را بیاموزیم. اول "head" است "head" فقط اشاره ای به commit است. به جای گفتن "name" ، هنگام صحبت در مورد commit ، می گویم "head".

هنگام جابجایی بین تعهدات مختلف ، به راهی نیاز داریم تا بدانیم که "head" کجا هستیم. head فعلی (شخصی که در حال بررسی است) فقط "HEAD" نامیده می شود. و همین پیش گفته اشاره به تعهد است (می توانید چندین سر در یک مخزن وجود داشته باشد) ، و هد که به تعهد بررسی شده در حال حاضر اشاره دارد ، HEAD نام دارد.

اما چگونه به فهرست کار معمول و فعلی برگردیم؟ از آنجا که ما هیچ تغییر بزرگی در مخزن خود ایجاد نکردیم ، بازگشت به دایرکتوری مشغول بررسی تنها شاخه ای است که ما داریم. طبق کنوانسیون ، آن شاخه "master" خوانده می شود.



## \$ git checkout master

آن را امتحان کنید! و به یاد داشته باشید دو قانون طلایی سفر زمان:

- در زمان تمیز فقط زمان سفر کنید (چیزی که در فهرست کار قرار ندارد).
  - گذشته را تغییر ندهید (تا زمانی که تجربه بیشتری داشته باشید).
- فراموش نکنید که شاخه فعلی (استاد) را پس از پیمایش بین نسخه ها بررسی کنید.

## Undo a commit

زمان آن فرا می رسد که پرونده ها **commit** کنید اما بعداً نظر خود را تغییر دهید.

برای همه اتفاق می افتد اما با روش های سنتی (بدون نسخه سازی) ، برگرداندن تغییرات بسیار دشوار است به خصوص اگر این تغییرات در سنین قبل انجام شده باشد. با **Git** فقط یک دستور واحد است: **git revert**.

چرا فقط تعهد را حذف نمی کنیم؟ به دلیل قانون گذر زمان از قسمت قبلی: هرگز گذشته را تغییر ندهید. هر تغییری که انجام شود باید به خاطر تاریخ بماند. تغییر آنچه در گذشته اتفاق افتاده بسیار خطرناک و ضد کارآمیز است. در عوض ، شما از **git revert** برای ایجاد یک تعهد جدید استفاده خواهید کرد که حاوی خلاف واقع تعهدی است که می خواهید آن را خنثی سازی کنید.

بنابراین ، خنثی کردن تعهد دقیقاً برعکس است. ساده است! برای استفاده از آن ، شما باید نام تعهدی را که به عنوان پارامتر برگردانده شده است ، وارد کنید.

## \$ git revert <commit name>

شما می توانید هر **commit**ی را برگردانید. فقط مطمئن شوید که در یک فهرست کار تمیز کار کنید. بنابراین ، فراموش نکنید که قبل از بازگشت به **commit** ، پرونده های خود را **commit** کنید. بیا یاد امتحان کنیم! ابتدا مطمئن شوید که فهرست کار مانند شکل 2-6 تمیز است.



```
MINGW64;C:/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
nothing to commit, working tree clean

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |
```

اکنون که می دانیم فهرست کارها پاک است ، باید تاریخ را بررسی کنیم تا بدانیم کدام یک از افراد متعهد به خنثی سازی هستند. باید نتیجه ای بگیریم مانند آنچه در شکل 3-6 نشان داده شده است.

```
MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log
commit 8ba74a5546782e38d1c2d6dafd2386e814034c69 (HEAD -> master)
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Mon May 27 21:31:51 2019 +0200

    Rearrange .gitignore

commit b2eccbf5b54c0f5b6d34b2432245a1a582a96f6
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 21:20:51 2019 +0200

    Add .gitignore

commit 5f57824bdc7b704d17e8a9cbf36146f43eb0269a
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:18:12 2019 +0200

    Finish task 1: mittens

commit 9f180aae6d70f83a5252b0d1be2d68321f5b2146
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:17:11 2019 +0200
```

```
MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log --oneline
8ba74a5 (HEAD -> master) Rearrange .gitignore
b2eccfb Add .gitignore
5f57824 Finish task 1: mittens
9f180aa Doing task 1: mittens
1c3f057 Create TODO

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |
```

بیا باید سومین تعهد را برگردانیم ما فقط از **git revert** استفاده می کنیم و نام تعهد را دنبال می کنیم.

```
$ git revert 5f57824
```

از آنجا که بازگشت **git** فقط تعهد جدیدی را ایجاد می کند که حاوی تغییرات متضاد است ، بقیه مراحل مشابه هر تعهد جدید است. همانطور که در شکل 5-6 نشان داده شده است ، از شما خواسته می شود تعهد جدید خود را توصیف کنید. پیشنهاد می کنم همیشه توضیحات مربوط به تعهد پیش فرض را حفظ کنید زیرا شناسایی آن آسان می شود.





```
MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
nothing to commit, working tree clean

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log --oneline
8ba74a5 (HEAD -> master) Rearrange .gitignore
b2eccfb Add .gitignore
5f57824 Finish task 1: mittens
9f180aa Doing task 1: mittens
1c3f057 Create TODO

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git revert 5f57824
[master 2b7e227] Revert "Finish task 1: mittens"
2 files changed, 1 insertion(+), 1 deletion(-)
delete mode 100644 DONE.txt

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$
```

**Figure 6-6. Summary of the revert**

همانطور که مشاهده می کنید ، خنثی سازی تغییرات با **Git** بسیار آسان است. نکته قابل ذکر این است که برگشت **git** فقط تعهد جدیدی را ایجاد می کند که حاوی تغییرات متضاد است. این بدان معنی است که شما می توانید یک برگرداندن برگردانید! برگرداندن یک بازگرداندن ، فقط تعهد اصلی شما را مجدداً اعمال می کند ، و دو "برگشت" یکدیگر را لغو می کنند. با این حال ، این تعهدات همچنان که نمی توانید گذشته را تغییر دهید ، بر روی سابقه تاریخ شما باقی خواهند ماند.

## اصلاح یک تعهد

همانطور که در آخرین فصل به شما قول داده ام ، خواهید آموخت که چگونه یک تعهد را در این فصل اصلاح کنید. این مورد زمانی استفاده می شود که شما یک مرحله را فراموش کرده اید یا می خواهید پیام متعهد را تغییر دهید. این نباید برای اصلاح بسیاری از پرونده ها مورد استفاده قرار گیرد زیرا این کار خلاف واقع است. در فصل بعد به تفصیل بحث خواهیم کرد که از کجا و کجا می توان از این استفاده کرد. و من دوباره این را می گویم: هرگز سعی نکنید گذشته را تغییر دهید.

برای تغییر یک تعهد ، شما باید از دستور **git commit** اما با **--amend** به عنوان پارامتر استفاده کنید. این ویرایشگر متن پیش فرض شما را مانند یک تعهد عادی اما با پرونده های مرحله بندی شده و پیغام های متعهد شده در آنجا باز می کند.

## \$ git commit --amend

شما فقط ویرایشگر متن را مانند هر متعه ای ذخیره و بسته می کنید. کلمه "اصلاح" که من استفاده کردم کمی گمراه کننده است زیرا شما در حال تغییر یک تعهد نیستید. شما در حال ایجاد یک تعهد جدید هستید و جایگزین فعلی هستید. بنابراین ، از این پس ، من از کلمه "اصلاح" استفاده خواهم کرد.

اصلاح یک تعهد ، همه چیز را در حوزه نمایش داده می کند و باعث می شود مجدداً با آن متعهد شود. بنابراین ، اگر می خواهید یک پرونده جدید به متعهد اضافه کنید یا پرونده ای را از آن حذف کنید ، می توانید آنها را به خواست خود مرحله بندی کنید. یادآوری: برای مرحله بندی کردن پرونده ، باید از `git reset HEAD <file>` استفاده کنید. این یک مثال کوچک است. بیایید دوباره از برنامه `TODO` استفاده کنیم. یک فایل موجود را ویرایش کنید. سپس دو پرونده جدید با نام `fileforgotten.txt` و `filenottocommit.txt` مانند شکل 6-7 ایجاد کنید.

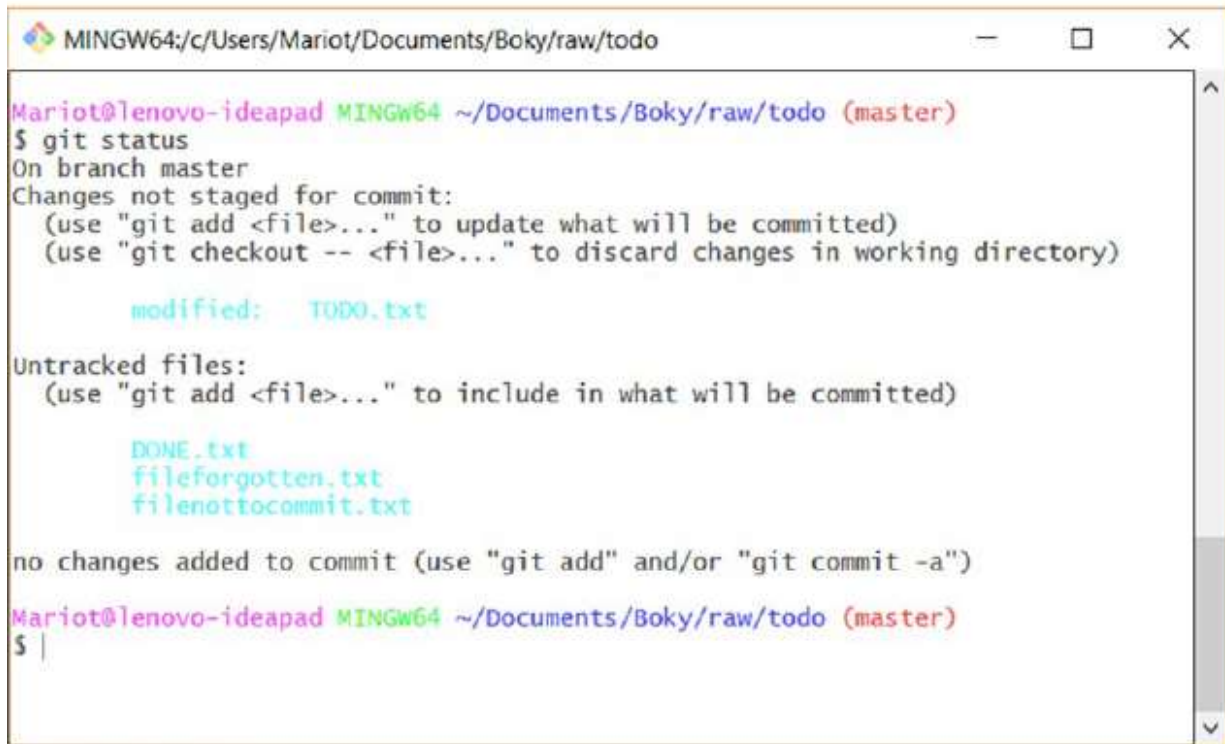
Name	Date modified	Type	Size
another	2019-05-27 22:20	File folder	
build	2019-05-23 21:57	File folder	
folder	2019-05-23 22:08	File folder	
.gitignore	2019-05-27 22:20	Text Document	1 KB
DOING.txt	2019-06-17 23:11	Text Document	1 KB
DONE.txt	2019-08-12 23:45	Text Document	0 KB
fileforgotten.txt	2019-08-12 23:46	Text Document	0 KB
filenottocommit.txt	2019-08-12 23:46	Text Document	0 KB
PRIVATE.txt	2019-05-27 22:20	Text Document	0 KB
TODO.txt	2019-06-17 23:31	Text Document	1 KB

**Figure 6-7.** All the files in our Working Directory

می توانید با اجرای دستور وضعیت `git` وضعیت فعلی پروژه را بررسی کنید:

## \$ git status

بسته به اینکه چند پرونده به پروژه اضافه کرده اید ، ممکن است نتیجه کمی متفاوت داشته باشید اما هنوز شبیه به شکل 6-8 است.



```
MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   TODO.txt

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

        DONE.txt
        fileforgotten.txt
        filenottocommit.txt

no changes added to commit (use "git add" and/or "git commit -a")
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |
```

**Figure 6-8.** The modified and untracked files are highlighted

مورد بعدی که باید انجام دهیم این است که پرونده ها را قسمتی از تعهد قرار دهیم. پرونده های تغییر یافته و **filenottocommit.txt** را اضافه کنید.

**\$ git add TODO.txt DONE.txt filenottocommit.txt**

شما از آخرین فصل می دانید که همیشه باید آنچه را که با "git Diff --staged" انجام داده اید قبل از ارتکاب بررسی کنید. اما بیایید وانمود کنیم که بلافاصله فراموش کرده اید و مرتکب می شوید.

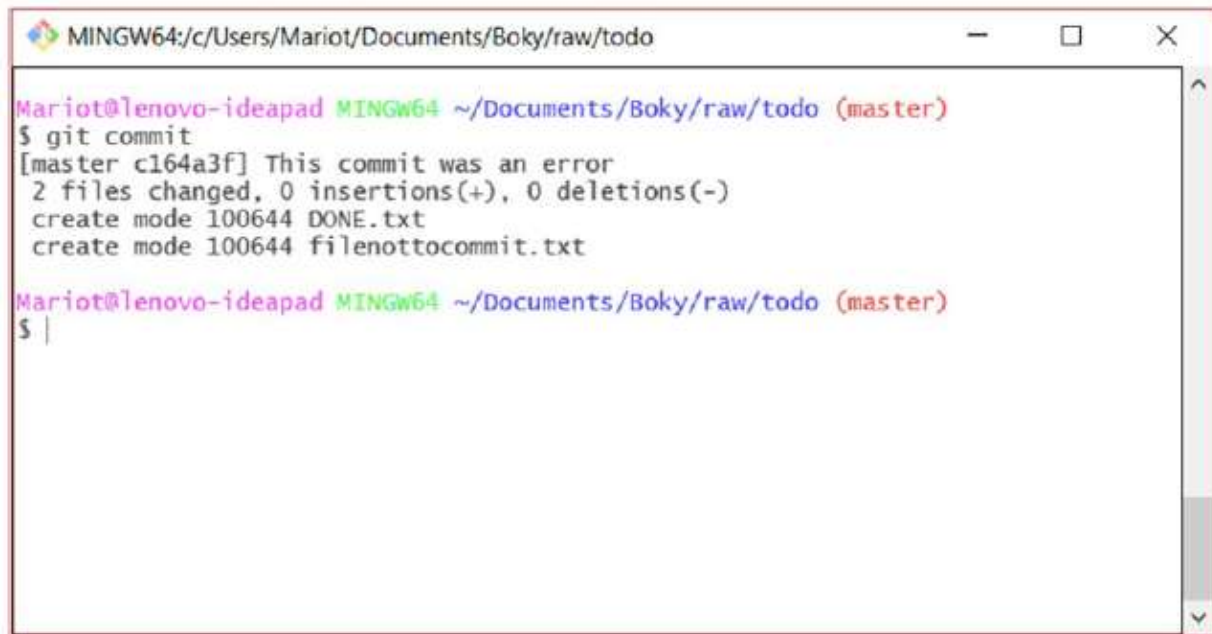
**\$ git commit**

حتی در این صورت ، به صفحه پیام متعهد می شوید که تغییری را که مانند شکل 6-9 مرتکب شده اید ، تشریح می کند.









```
MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git commit
[master c164a3f] This commit was an error
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 DONE.txt
create mode 100644 filenottocommit.txt

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |
```

**Figure 6-10.** The commit summary. We messed up

اکنون که متوجه شده اید پرونده ای را اشتباهی **commit** کرده اید. ابتدا باید آخرین تعهد خود را با تنظیم مجدد **git** حذف کنید. ما از گزینه "**soft--**" استفاده خواهیم کرد تا ویرایش هایی که ساخته ایم در فهرست کار بماند. **HEAD ~ 1** بدان معنی است که **commit** قبلی پیش رو ، مرجعی برای عملکرد فعلی است.

```
$ git reset --soft HEAD~1
```

پس از این ، می توانید دوباره پرونده را با بازنشانی **git** مرحله بندی کنید:

```
$ git reset HEAD filenottocommit.txt
```

بررسی کنید آیا دستورات با بررسی وضعیت فعلی پروژه در نظر گرفته شده اند.

```
$ git status
```

نتیجه ای خواهید گرفت مانند آنچه در شکل 6-11 نشان داده شده است.

```
MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   DONE.txt

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

    modified:   TODO.txt

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

    fileforgotten.txt
    filenottocommit.txt

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$
```

همانطور که مشاهده می کنید، `filenottocommit.txt` اکنون بدون دسترسی است، زیرا ما آن را از قسمت صحنه حذف کردیم. به طور طبیعی `fileforgotten.txt` موسیقی متن فیلم است زیرا ما آن را تنها صحنه انجام نداده ایم. فقط در مرحله صحنه قرار داریم زیرا ما پس از ارتکاب آن را لمس نکرده ایم.

```
$ git add fileforgotten.txt
```

اکنون که فایل‌های صحیح را ترسیم کرده اید، می توانید پروژه را `commit` کنید.

```
$ git commit
```

خطای دستوری را در پیام `commit` قرار دهید تا ویژگی دیگری از `Git` را مشاهده کنید.

## اصلاح یک تعهد

برای اشتباهات ساده مانند خطایی در پیام متعهد، نیازی به اصلاح کل تعهد نیست. شما فقط باید آن را اصلاح کنید. بیا یاد بگیریم!

```
$ git commit -amend
```

روند اصلاح درست مانند یک تعهد عادی به نظر می رسد، اما در عوض پیام متعهد قبلاً نوشته شده است، همانطور که در شکل 6-12 مشاهده می کنید.

```
MINGW64/c:/Users/Mariot/Documents/Boky/raw/todo
This commit was an error
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Tue Aug 13 00:28:47 2019 +0200
#
# On branch master
# Changes to be committed:
#   new file:   DONE.txt
#   new file:   fileforgotten.txt
#
# Changes not staged for commit:
#   modified:   TODO.txt
#
# Untracked files:
#   filenottocommit.txt
#
<ot/Documents/Boky/raw/todo/.git/COMMIT_EDITMSG [unix] (00:36 13/08/2019)1,1 All
<ariot/Documents/Boky/raw/todo/.git/COMMIT_EDITMSG" [unix] 18L, 413C
```

می توانید پیغام **commit** را به خواست خود تغییر دهید و ویرایشگر را مثل همیشه ذخیره و بسته کنید. ساده است! به نام تعهد جدید نگاهی بیندازید و آن را با نمونه قدیمی مقایسه کنید. متوجه خواهید شد که آنها متفاوت هستند. دلیل این امر این است که نام متعهد اطلاعات مهمی در عکس فوری است. بنابراین حالات مختلف پروژه منجر به نامهای مختلفی می شود.

یادداشت جدائی درباره اصلاح تعهدات: از آن سوءاستفاده نکنید! بله ، خطا هنگام نوشتن کد ایده آل نیست و بیشتر اوقات می خواهیم بلافاصله آنها را اصلاح کنیم. اما خطاها همچنین به ما کمک می کند که بهتر باشیم. و پیگیری از اشتباهات ما یک روش عالی برای یادگیری است.

خلاصه

در این فصل به طور عمده به مرورگر ، خنثی سازی و اصلاح نسخه های پروژه پرداخته شده است. اکنون نباید با اصلاحات جزئی در تعهدات خود مشکلی ایجاد کنید. اطمینان حاصل کنید که بخش اول این فصل را دوباره بخوانید ، زیرا برای هر کاری که در **Git** انجام می دهید ضروری است.

باید تفاوت های بین **Git byheart**((state سه را بدانید.

فصل بعدی یک مقاله کوچک است زیرا ما فقط در مورد تئوری صحبت خواهیم کرد. شما یاد خواهید گرفت که چگونه یک پیام متعهد خوب بنویسید ، چه چیزی را در تعهدات گنجانده و نادیده بگیرید ، و چه خطاهای متداولی مبتدیان انجام می دهند. حتما فصل بعد را با دقت بخوانید زیرا به شما و تیم شما کمک بزرگی خواهد کرد.

# فصل هفتم

## بهترین روشها

فصل قبل یکی از مهمترین مطالب این کتاب بود. اطمینان حاصل کنید که هر بار که به تعهدات خود شک دارید دوباره به آن برگردید. پس از خواندن آن ، باید بتوانید بدون هیچ مشکلی عکس فوری های پروژه را تهیه ، بررسی و اصلاح کنید. اکنون که ویژگی های اصلی **Git** را می دانید ، زمان آن فرا رسیده است که شما بهترین شیوه ها را برای آسانتر کردن زندگی (و هم تیمی هایتان) بیاموزید. اینها مطالبی هستند که آرزو می کنم هنگام اولین بار از **Git** می دانستم. ما پیامها ، موارد ضعیف و ارزشمندی های **Git** و لیستی از رایج ترین اشتباهاتی که مبتدیان انجام می دهیم را نادیده می گیریم. سپس با کمی یادآوری نحوه کار **Git** به پایان خواهیم رسید.

### Commit messages

پیام های تعهد یکی از مهمترین جنبه های **VersionControl** و یکی از مهمترین آنها نادیده گرفته می شود. این پیام ها برای کمک به شما (و دیگران) وجود دارد که می توانید درک کنید که چه عواملی در تعهد ایجاد شده و از همه مهمتر ، چرا این تغییرات ایجاد شده است. پیام های متعهد تمیز و قابل خواندن برای تجربه بهتر **Git** ضروری است.

بباید با شناسایی مشکل شروع کنیم. رایج ترین مشکلات با **Git** این است که پیام های متعهد غالباً بی معنی هستند و هیچ اطلاعات معناداری را منتقل نمی کنند. و بیشتر اوقات ، پیام ها با هر تعهد کمتر و واضح تر می شوند. این به دلیل سوء تفاهم از مفاهیم گیت است: هر تعهدی باید در کنار خود بایستد. اگر یک متعهد برای معقول شدن به تعهدات دیگری نیاز داشته باشد ، نباید وجود داشته باشد. شما هرگز نباید پروژه ای را انجام دهید که نیمه تمام باشد. اگر یک کار خیلی بزرگ است ، آن را در چند بخش منطقی تقسیم کنید ، جایی که هر قسمت به خودی خود حس می کند. یک راه خوب برای دانستن اینکه هنگام تقسیم وظایف در مسیر اشتباه هستید ، بررسی پیام متعهد احتمالی است: اگر به استفاده از یک پیام متعهد بسیار مشابه فکر می کنید ، احتمالاً هنگام تقسیم کار خطایی مرتکب شده اید. به عنوان مثال ، اگر وظیفه شما انجام بسیاری از اصلاحات کوچک در یک وب سایت بزرگ است ، منطقی است که آن را به کارهای کوچکتر مانند تعهد برای هر صفحه یا یک تعهد برای هر دسته از صفحه تقسیم کنید. بنابراین به یاد داشته باشید: تعهدات شما باید مستقل ، اتمی و کامل باشند.

مشکلی که بسیاری از مبتدیان نیز در آنها وجود دارد انتقال اطلاعات بیش از حد در پیام متعهد است ، بنابراین بیشتر صفحه ها را با جزئیات غیر ضروری مسدود می کنید. یک پیام متعهد باید مختصر و مستقیم باشد. لازم نیست همه چیزهایی را که تغییر کرده است بگویید ، فقط باید توضیح دهید که چرا این تغییرات انجام شده است. اگر شخصی می خواست ببیند چه چیزی تغییر کرده است ، از دستور `git show` استفاده می کند ، که نشانگر جمع بندی کامل پرونده های تغییر یافته در تعهد است.

به یاد داشته باشید که شما تنها کسی نیستید که کد یا متن خود را بخوانید. شما باید کمی زمان سرمایه گذاری کنید تا زمینه تحولات را توضیح دهید و چرا انجام شده است. گفتن به خودتان "یادم می آید" دروغ است و هرگز نباید عملی شود. برای هر متعهد ، باید از خود پرسید: "اگر شخص دیگری به پروژه من نگاه کند ، آیا آنها با نگاه کردن به پیامهای متعهد من ، می توانند جدول زمانی تغییرات در پروژه ها را درک کنند؟" و همچنین به یاد داشته باشید که آن شخص ممکن است در عرض چند ماه شما باشد. کدها به راحتی فراموش می شوند.

نکته پایانی این است که پیام `Git` شما باید بگوید که چرا تغییرات ایجاد شده اند. اگر کسی بخواهد ببیند چه چیزی تغییر کرده است ، آنها به `Git diff` نگاه می کنند.

## Git commit best practices

برای پیام بهتر متعهد و جلوگیری از بروز مشکلات قبلی ، در اینجا نکاتی ذکر شده است که از این پس باید دنبال کنید. این نکات به همکاران شما کمک می کند و به احتمال زیاد در آینده ، شما دیدگاه روشنی درباره چرایی ارتکاب تعهد دارید. همانطور که پروژه می گذرد ، ما تمایل داریم مراحل قبلی خود را فراموش کنیم ، بنابراین داشتن یک سابقه تاریخ خوب در پیشرفت سریع ضروری است.

- پیام های `commit` باید آسان باشد.

وقتی از `log git` استفاده می کنید ، هیچ خط جدیدی ایجاد نمی شود که پیام ها خیلی طولانی باشند. بنابراین کاربر برای مشاهده همه موارد باید پیمایش کند. این ایده آل نیست زیرا باید بتوانید تعهدات را به راحتی جستجو و بازیابی کنید.

- شما نباید پیام ها را طولانی تر از 50 نویسه بنویسید.

- پیام را با یک نامه بزرگ شروع کنید.

- پیام را با یک دوره پایان ندهید.

- از زمان حال و مقالات غیرضروری خندق استفاده کنید.

- پیام های متعهد باید سازگار باشند.

از آنجا که پیام های `Git` در هر پروژه اساسی هستند ، باید سازگار باشند و نباید تحت تأثیر تغییرات وحشیانه قرار گیرند. شما همیشه باید برای هر متعهد ، از همان زبان استفاده کنید و منطق درونی آن را دنبال کنید. تغییر سبک نوشتن در اواسط پروژه ، جستجوی تعهدات را بسیار دشوار خواهد کرد.

- پیام ها باید واضح و متنی باشند.

هنگامی که بسیاری از نویسندگان روی بخش های مختلف کار می کنند ، متن اصلی است. به عنوان مثال ، بسیاری از توسعه دهندگان پیام های متعهد خود را با توجه به متن یا محدوده پروژه تحت تأثیر تغییرات شروع می کنند. اما این فقط مربوط به پروژه های بسیار بزرگ است

از پیام های ناشناخته یا مبهم مانند "تغییر CSS" ، "آزمایش تست" ، "عیب یابی" ، "اصلاحات اندک" و "به روز رسانی" باید به هر قیمتی جلوگیری شود. آنها غالباً گمراه کننده هستند و کاربر را مجبور می کنند تا به تفاوت ها نگاه کند. همیشه اطمینان حاصل کنید که چرا تغییرات ایجاد شده اند. و هرگز کاربران را مجبور نکنید که برای درک تعهد ، تغییرات کد شما را بررسی کنند.

- با جزئیات دیوانه نشوید.

می توانید پیام ارتکاب خود را در بدن گسترش دهید اما در دادن اطلاعات بیش از حد خطا نکنید.

تنها چیزی که باید توضیح دهید این است که چرا تغییرات ایجاد شده اند ،

به یاد داشته باشید: پیام متعهد شما باید بگوید در صورت اعمال ، چه اتفاقی برای پروژه خواهد افتاد. بنابراین شما همیشه باید از یک زبان روشن ، زمان حال و ضروری استفاده کنید. بهترین پیام های متعهد معمولاً کوتاه ، مستقیم و قابل ذکر هستند.

هیچ راهی بهتر از مثال برای روشن تر شدن وجود ندارد ، بنابراین اجازه دهید این کار را انجام دهیم. جدول 7-1 ابزاری مفید برای نشان دادن شما در جهت صحیح است.

Best	Bad	Worst
[login] Fix typo in DB call	Fixed typo in DB call	Fix typo
Refactor login function for reuse	Changing login function by moving declarations to parameters	Code refactoring
Add new API for user program check	Adding a new API for user program check	New user API

نمونه های ارائه شده در جدول 7-1 باید نشان دهند که آیا هنگام نوشتن پیام متعهد در مسیر خوبی هستید یا خیر.

توجه داشته باشید که این اقدامات توصیه می شوند و به صورت سنگ نوشته نمی شوند. اگر واقعاً مجبور هستید ، اگر این پیام را واضح تر کند ، می توانید برخی از آنها را نادیده بگیرید.

## چه کاری انجام دهیم

ببایید با شمارش شیوه های خوبی که همیشه هنگام استفاده از Git باید به خاطر بسپارید شروع کنیم. برای موفقیت شما بسیار مهم است زیرا باعث می شود تا شما در زمان جدی صرفه جویی کنید.

مهمترین چیزی که باید به خاطر داشته باشید این است که یک تعهد تغییر در پروژه است که باید روی خود بایستد. همیشه باید تعهدات را کوچک و مستقل نگه دارید. نقش متعهد (بیشتر اوقات) معرفی یک ویژگی یا رفع اشکال است. این برای پیگیری هر تغییری که ایجاد کرده اید نیست. اگر یک ویژگی یا رفع اشکال به مراحل مستقل بزرگی نیاز دارد ، آنها را در چند مورد جدا کنید. به عنوان مثال ، یک ویژگی به یک نقطه پایانی API و تماس تلفنی نیاز دارد. نیازی به ایجاد همه این تغییرات در یک تعهد واحد نیست زیرا آنها مستقل هستند و به هیچ وجه با یکدیگر ارتباط ندارند. اگر در کد باطن خطایی ایجاد کردید ، می توانید بدون ایجاد مزاحمت در کد جلویی ، تغییرات را برگردانید. جدا کردن آنها توسط چندین تعهد همچنین باعث می شود سابقه تاریخ قابل خواندن و پیام تعهد روشن تر شود.



ما قبلاً در مورد این موضوع صحبت کرده ایم ، اما از آنجا که بسیار مهم است ، اجازه دهید به آن برگردیم. هر پیام متعهد باید به سؤال "چرا؟" پاسخ دهد. چرا تعهد ایجاد شد؟ چه مشکلی را حل می کند؟ به یاد داشته باشید که در **Git** ، تعهدات بین بسیاری از کاربران قابل تبادل است. بنابراین ، پیام متعهد باید به این سؤال پاسخ دهد: اگر من این تعهد را انتخاب و استفاده کنم ، چه خواهد کرد؟ به همین دلیل است که آنها مرتکب تنش می شوند باید به شکل فعلی باشد. تکان دادن نیاز به نوشتن آن در تنش های گذشته دشوار است ، اما پس از چند هفته ، باید با آن راحت باشید.

و همین لیست کارهایی که باید انجام شود با **Git** بسیار اندک است. فقط اطمینان حاصل کنید که برای تعهدات کوچک و مستقل خود پیام های واضحی بنویسید. لیست کارهایی که نباید انجام شود ، از طرف دیگر به شرح زیر است.

## چه کار نکنیم

این لیست کمی طولانی تر از لیست قبلی است. به این دلیل است که **Git** ابزاری بسیار قدرتمند است و کارهایی را که می توانید انجام دهید محدود نمی کند. بنابراین ، اشتباه کردن بسیار آسان است ، به خصوص هنگامی که فکر می کنید باعث صرفه جویی در وقت شما می شود. این نمی شود شیوه های بد همیشه در طول مسیر به شما مشکلات بیشتری می رساند. بهتر است از انجام این کارها در کل خودداری کنید.

یک خطای متداول که بیشتر مبتدیان تمایل به ایجاد آن دارند ، حل چندین مشکل در یک تعهد است. به عنوان مثال ، آنها در هنگام رفع اشکال در هنگام مشاهده یک مورد دیگر ، در حال کار هستند. آنها هر دو مشکل را حل می کنند و سپس پروژه را مرتکب می شوند. این به نظر می رسد خوب است تا زمانی که کشف شود که این ارتکاب مشکلات زیادی را در مبنای کد ایجاد کرده است. از آنجا که تنها یک تعهد وجود دارد ، آنها نمی دانند کدام تغییرات باعث ایجاد مشکلات شده است. این تنها یک جنبه از مشکلات مربوط به تعهدات مسدود شده است. نکته دیگر این است که نوشتن پیام های متعهد و شفاف را دشوار می کند. اگر خود را مرتکب تغییر در زمینه های مختلف می شوید ، تقسیم تعهدات را به موارد کوچکتر در نظر بگیرید.

یکی دیگر از اشتباهات مشابه قبلی ، ترکیب تعهداتی است که هیچ ارتباطی با یکدیگر ندارند. به عنوان مثال ، پالایش مجدد کد نباید متعهد باشد

به عنوان رفع اشکالات یا ویژگی های جدید اما در تعهد خاص خود. این ، دوباره ، به منظور تسهیل در تعقیب اشکال و تمیز کردن ورود به سیستم تاریخ است.

اشتباه بعدی ناشی از سوءاستفاده اساسی از **Git** و مطالبات برخی شرکت ها است. این خطا در استفاده از **Git** به عنوان یک سیستم پشتیبان است. از آنجا که **Git** یک سیستم کنترل نسخه توزیع شده است ، مخزن را می توان در یک سرور از راه دور ذخیره کرد. این امر برخی از توسعه دهندگان را وادار می سازد تغییرات خود را در هر آخر روز انجام دهند ، چه معقول باشد ، چه نباشد. این همچنین به دلیل نیاز به نشان دادن پیشرفت روزانه شما ایجاد می شود زیرا برخی از شرکت ها برای اندازه گیری بهره وری به تعداد خطوط کد تولید شده نگاه می کنند. این یک روش بسیار ضد انعطاف پذیر برای کار است زیرا بسیاری از تعهدات را ایجاد می کند که سعی در حل همان مشکل دارند. همچنین منجر به سردرگمی پیامهای متعهد می شود که با گذشت زمان ، پیام های کمتری و واضح نشان می دهند. به هر قیمتی از این کار خودداری کنید. شما باید متعهد شوید که کار آماده باشد ، نه به این دلیل که مجبور شوید. اگر نیاز به ارتکاب داشته باشید زیرا وظیفه انجام کار بر روی چیز دیگری را دارید ، فرصتی برای انجام این کار با کمک مفاهیمی مانند انشعاب یا **stashing** خواهید داشت. این موارد را بعد از چند فصل خواهید آموخت.

یکی دیگر از ویژگی های سوء استفاده از **Git** ، فرمان اصلاح است. از ایجاد تعهدات برای ایجاد تغییرات بزرگ در آن خودداری کنید. برای اصلاح علائم و اضافه کردن پرونده های فراموش شده یا تغییرات بسیار ناچیز باید فقط از اصلاح استفاده شود. اگر تغییرات آنقدر بزرگ هستند که لزوم به روزرسانی پیام متعهد را احساس می کنید ، فقط یک متعهد دیگر



انجام دهید. اما آیا این اشتباهات من را در پایه کد نمی گذارد؟ بله ، اما **Git** از این رو نسخه ها را ردیابی می کند و نشان می دهد چه چیزی تغییر کرده است.

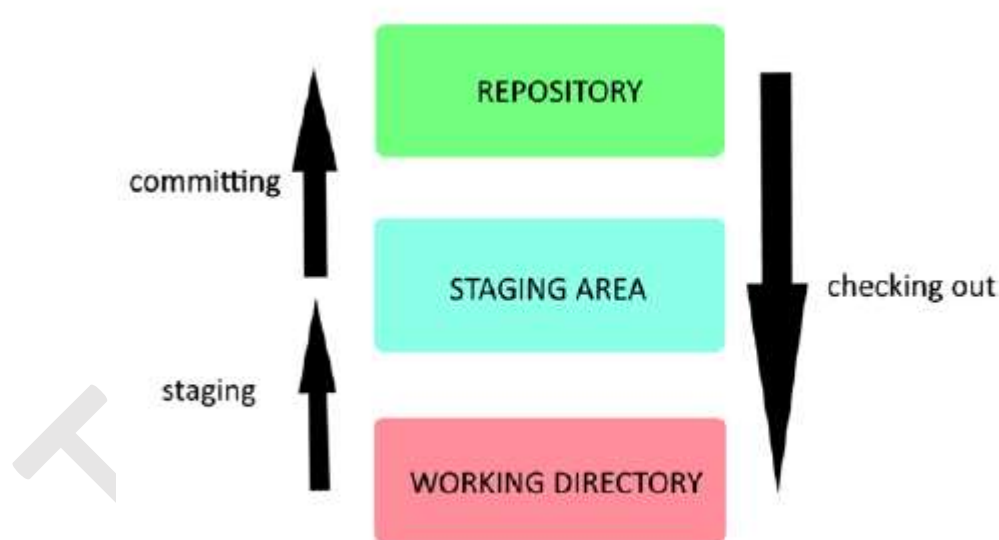
شما باید خطاهای خود را نیز ردیابی کنید ، زیرا فراموش کردن آنها آسان است. شرمنده از اشتباهات خود نباشید. تلاش برای پاک کردن آنها به هیچ کس کمک نخواهد کرد و در صورت مواجهه با همان مشکل دوباره وقت زیادی را برای شما صرفه جویی می کند.

این آخرین اشتباه رایج قبلاً در این کتاب و در فیلم های بی شماری درباره آن صحبت شده است: هرگز سعی نکنید تاریخ را تغییر دهید. بازگشت به نسخه های قبلی و تغییر چیزها بسیار وسوسه انگیز است. این یک ایده بسیار بد و یکی از خطرناک ترین کارهایی است که می توانید انجام دهید. همکاران شما اگر این کار را انجام دهند از شما متنفر خواهند شد و احتمالاً کل مخزن را مخدوش خواهید کرد. روش صحیح برای تغییر چیزی ، تعهد جدید است. گذشته ها گذشته. رهاپش کن.

## Git چگونه کار می کند (دوباره)

میدونم میدونم. ما قبلاً این را پشت سر گذاشته ایم. اما می خواهم قبل از حرکت به قسمت دوم این کتاب اطمینان حاصل کنم که شما کاملاً راحت از این کار هستید.

سه حالت گیت را به خاطر دارید؟ از آنها به سه درخت نیز یاد می شود (در واقع این اسم رسمی در اسناد است). بگذارید یکبار دیگر آنها را مرور کنیم. شکل 1-7 به شما در شناسایی سریع درختان کمک می کند.



*The relationship between the three states of Git*

همانطور که در شکل 1-7 مشاهده می کنید ، در اینجا هیچ چیز جدیدی وجود ندارد ، فقط یک یادآوری. برای ردیابی تغییرات در یک پروژه ، باید از کلیت آن عکس بگیرید. **Git** تغییرات را ردیابی نمی کند؛ نسخه های آن را ردیابی می کند.

شما فقط با دایرکتوری کار تعامل خواهید کرد زیرا در اینجا امکان ویرایش پرونده های شما آزادانه وجود دارد. به ویژه ، چیزی برای گفتن در این مورد وجود ندارد: این فقط وضعیت فعلی پرونده های شما است.

هنگامی که آماده عکس گرفتن از عکس خود از پروژه هستید ، پرونده های خود را قرار می دهید. هر پرونده تغییر یافته ای که در منطقه **Staging** (یا شاخص مرحله بندی) قرار نگرفته باشد جزئی از عکس فوری نخواهد بود. هرچند که این تغییرات هنوز در دایرکتوری کار موجود خواهند بود. بنابراین ، لازم است قبل از اضافه کردن پرونده ها به صفحه اصلی **Staging Index** وضعیت **WorkDirectory** را بررسی کنید تا مطمئن شوید همه چیز درست نیست.

**Repository** بانک اطلاعاتی معماری **Git** است. شما در آنجا تمام تعهدات و سابقه تاریخی خود را پیدا خواهید کرد. می توانید آن را در پوشه ".git" بیابید (که هرگز نباید آنرا لمس کنید ، مگر اینکه تنظیمات را تنظیم کنید). عمل مرتکب همه چیز را در منطقه صحنه می گیرد و عکس فوری از آن می گیرد. به همین دلیل می گوئیم "انجام یک پروژه" ، نه "مرتکب پرونده" یا "انجام تغییرات". پرونده های غیرقابل تغییر که در گذشته مرتکب شده اند ، در حال حاضر در منطقه استیشن قرار دارند. به همین دلیل لازم نیست همه موارد را مرحله بندی کنید ، فقط پرونده های ویرایش شده. به یاد داشته باشید که فایل های جدید یا حذف شده را نیز مرحله بندی کنید

سرانجام ، چک کردن وضعیت یک پروژه را به یک پروژه قبلی باز می گرداند. دایرکتوری کار تغییر خواهد کرد تا تغییرات را منعکس کند ، بنابراین مطمئن شوید که هیچ پرونده غیرقابل قبول وجود ندارد.

بنابراین مراحل اساسی هنگام استفاده از **Git** هستند

- ایجاد تغییرات (در فهرست کار)
- هر پرونده تغییر یافته را مرحله بندی کنید (در فهرست مرحله بندی)
- انجام پروژه (در مخازن)

این ساده است اما لطفاً قبل از شروع به فصل بعدی اطمینان حاصل کنید که رابطه بین آن **state** ها را درک کرده اید. هر بخش بعد از این فرض می کند که شما با آن ها آشنا هستید.

اما چگونه تعهدات به داخل مخزن نگاه می کنند؟ این ساده است: آنها مانند لیست های پیوندی به نظر می رسند. تعهد حاوی اطلاعات زیادی است: مطالب و ابر داده. مطالب فقط پرونده های پروژه هستند (پرونده ها تغییر یافته و منابع موجود در پرونده های بدون تغییر). فوق داده

شامل داده های دیگری نیز که بسیار مهم هستند: تاریخ انجام ، هویت مرتکب و پیام های **Git**. یکی دیگر از ابر داده های موجود در این تعهد ، نشانگر یا مرجع والدین است. این فقط نام تعهد قبلی است. و اگر خالی باشد ، به این معنی است که مرتکب اولین نفر است. بنابراین ، هر تعهد با روابط والدین و فرزند به بعد پیوند می خورد.

# فصل هشتم

## Git از راه دور

### Remote Git

تبریک می‌گویم برای تکمیل قسمت اول این کتاب! حالا سرگرمی شروع می‌شود. بخش اول ویژگی‌های اساسی Git را به شما آموخت. شما باید در ایجاد تغییرات و ردیابی آنها با Git راحت باشید. نوشتن پیام‌های متعهد معنی دار کمی دشوار است، اما در صورت پیروی از توصیه‌های فصل آخر با هر تعهد بهتر می‌شوید. همچنین می‌توانید در نسخه قبلی به زیرچشمی نگاهی بیندازید و بازدید از سابقه را مشاهده کنید که اینها ویژگیهای بسیار مهمی هستند که برای کلیه فصلهای بعدی مورد نیاز است.

اکنون شما آماده مقابله با یک چالش کاملاً جدید هستید: مخزن محلی خود را ترک کرده و با مخازن از راه دور بازی کنید. در این فصل خواهید آموخت که چرا کار بر روی ریموت و از همه مهمتر چگونه کارکرد آن مهم است. همچنین با یک گردش کار معمولی در کار تیمی و نحوه استفاده صحیح از مخازن از راه دور آشنا می‌شوید. از آنجا که مفهوم از راه دور Git کمی چالش برانگیز است، به ابزاری آسان ارائه خواهید شد که در این راه به شما بسیار کمک خواهد کرد (اشاره: این به اسم این کتاب است). بیایید آنلاین شویم.

### Why work on remote

از ابتدای این کتاب، ما فقط به تنهایی در مخزن محلی خود کار کرده ایم. اما Git ابزاری عالی برای کار تیمی است. شرم آور خواهد بود که از آن فقط در یک مخزن محلی استفاده کنید. ما در این بخش قصد داریم ببینیم Git از راه دور چیست و چرا کسی مایل به استفاده از آن است.

در ابتدای این کتاب گفتیم که Git یک سیستم VersionControl توزیع شده است. این بدان معناست که مخازن در یک سرور واحد ذخیره نمی‌شوند بلکه در بسیاری از مخازن محلی ذخیره می‌شوند. هر مشتری با تعهدات و تاریخچه خود مخزن محلی خود را دارد. این تعهدات می‌توانند آزادانه رد و بدل شوند و همه پرونده‌ها همیشه آماده ویرایش در هر زمان هستند. اینگونه است که Git قادر به پشتیبانی از کار تیم است.

از آنجا که کار تیمی مبتنی بر تبادل تعهد است ، راهی برای اطمینان از وجود همه تعهدات در همه زمانها باید پیدا شود. بسیار ناخوشایند خواهد بود که صبر کنید تا همکاران شما قبل از دسترسی به تعهدات خود به محل کار و راه اندازی رایانه های خود بپردازند. راه حل بارز این است که یک سرور میزبان مخزن باشد و همه افراد فقط تعهدات خود را فشار داده و از آن خارج کنند. اما آیا به اندازه کافی به جریان کاری مرکزی VCS نزدیک نیست؟ اصلاً (خوب ، کمی). همانطور که قبلاً بحث کردیم ، توزیع VCS توزیع شده بودند

ایجاد شده برای جلوگیری از مشکلات ناشی از داشتن یک مخزن مرکزی. هر مشتری مخزن خود را دارد و می توانند در هر زمان دلخواه روی آن کار کنند. تقریباً تمام اقدامات Git به صورت محلی انجام می شود. سرور از راه دور فقط به عنوان مشتری تعیین می شود که دارای مخزنی است که در آن هرکس تعهدات خود را تحت فشار قرار می دهد. به این ترتیب ، همه تغییرات در هر زمان ممکن است. این روش کار فقط برای تسهیل ارز مبادله ای مورد استفاده قرار می گیرد. در Git ساخته نشده است. برای Git ، تمام مخازن برابر ایجاد می شوند. توسعه دهندگان فقط تصمیم گرفتند که برخی از مخازن از سایرین برابر هستند.

حتی اگر به تنهایی کار کنید ، هنوز ایده خوبی است که علاوه بر محلی که در آن وجود دارد ، یک مخزن از راه دور نیز داشته باشید. به این ترتیب ، شما با تمام تاریخچه خود در یک مکان امن ، نسخه پشتیبان تهیه کرده اید. شما همچنین می توانید در هر زمان به پروژه خود دسترسی داشته باشید ، به شرطی که به شبکه سرور که مخزن را در اختیار دارد دسترسی داشته باشید.

## مختر راه دور چگونه کار می کند

استفاده از سرور راه دور فقط داشتن رایانه ای است که کپی از پروژه شما و سابقه آن را در اختیار دارد. لازم نیست همه تعهدات خود را به آن فشار بیاورید ، فقط تعهدی را که می خواهید به اشتراک بگذارید فشار می آورید. همکاران شما تعهدی را که مورد علاقه آنهاست ، بکشند و آنها را در مخازن شخصی خود اعمال کنند. و این در اصل این است! شما برای کپی مخازن و فشار و کشش تغییرات با یک سرور از راه دور کار می کنید. بیا بید با جزئیات ببینیم که چگونه این همه کار می کند.

برای راه اندازی یک مخزن از راه دور ، ابتدا به سرور قادر خواهید بود تا نرم افزار Git را اجرا کند. هر رایانه ای که نمک آن ارزش داشته باشد می تواند Git را اجرا کند زیرا یک نرم افزار بسیار کوچک است. برای اجرای صحیح آن نیز به انرژی زیادی نیاز ندارید. حتی یک کامپیوتر بسیار کوچک مانند تمشک پی بیش از حد کافی برای Git است.

اکنون که سرور دارید ، باید راهی برای برقراری ارتباط با آن پیدا کنید. دسترسی به شبکه به سرور ضروری است به طوری که چندین مشتری بتوانند به همان مخزن فشار وارد کرده و از آن خارج شوند. این ارتباط با سرور باید بسیار امن باشد. اگر کسی که به سرور دسترسی دارد می تواند مخزن را بخواند و ویرایش کند ، بسیار ناامید کننده خواهد بود. برای تعامل با مخزن ، کاربران باید خود را با هر عملیاتی از Git تأیید کنند. می توان از یک نوع احراز هویت HTTPS برای ورود / رمز عبور استفاده کرد ، اما از آنجا که تأیید اعتبار باید قبل از هر عملی باشد ، خیلی سریع خسته می شود. راه حل این امر استفاده از احراز هویت SSH است. اصل احراز هویت SSH ساده است: فقط مشتریانی که از پیش تعیین شده اند می توانند به مخزن دسترسی پیدا کنند.

و این در اصل این است! راه اندازی یک سرور Git از راه دور کار بسیار آسانی است. از طرف دیگر حفظ و ایمن سازی آن ...

استفاده از سرور خود برای میزبانی پروژه های Git خود ایده خوبی است اگر تنها کار کنید یا می خواهید آنها را خصوصی نگه دارید. به هر حال هنگام کار با یک تیم دردناک می شود هر یک از اعضای تیم باید از طریق شبکه به سرور Git دسترسی داشته باشند ، بنابراین اگر تیم شما همان فضای کار است ، باید یک شبکه محلی ایجاد کنید. همچنین سرور باید 24/7 کار کند تا در عملکرد Git هیچ تاخیری نداشته باشد.

چه اتفاقی می افتد اگر برخی از همکاران شما در یک منطقه کار دور افتاده یا در یک محل کار متفاوت باشند؟ خوب ، شما باید سرور خود را به اینترنت وصل کنید. بنابراین ، شما همچنین نیاز به اجرای بازی امنیتی خود دارید. هرچه همکار بیشتری داشته باشید ، استثناء تأیید اعتبار بیشتری نیز برای مدیریت خواهید داشت.

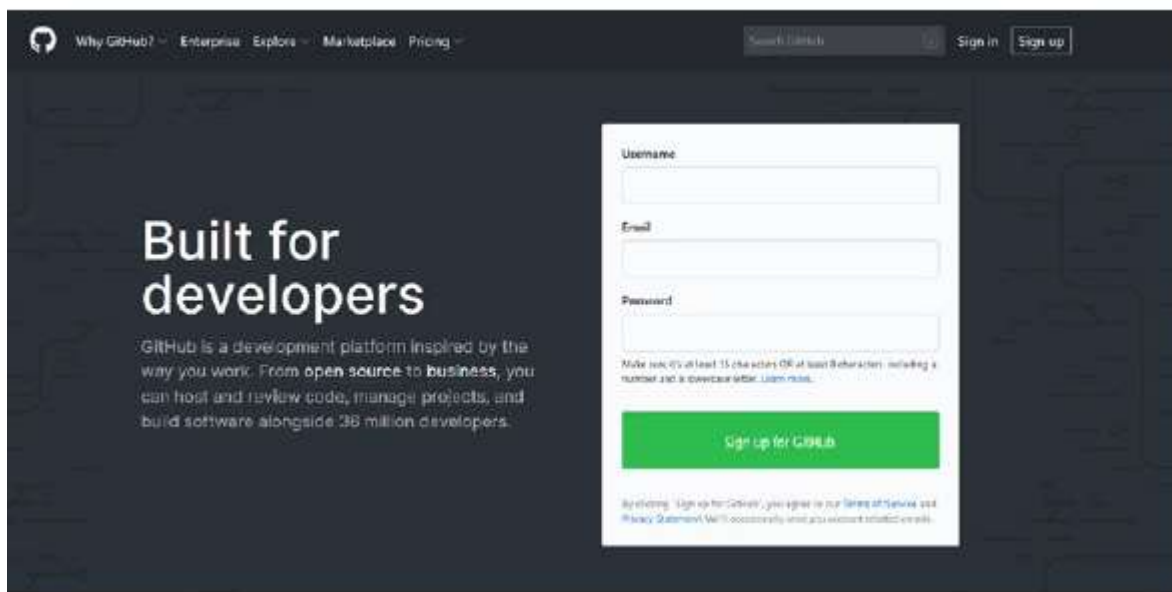
مشکل دیگر استفاده از سرور Git خود این است که شما باید با مجوزها مقابله کنید. همانطور که در فصل 2 مشاهده می شود ، همه برنامه نویسان نباید از نوشتن دسترسی به مخزن داشته باشند. به عنوان مثال اعضای جوان قبل از فشار آوردن به مخزن ، به تعهدات خود توسط اعضای ارشد بررسی شده نیاز دارند. توجه به آنها دسترسی مستقیم به پروژه ایده بدی است (به دلیل نیاز سیری ناپذیر آنها به تغییر تاریخ).

اینها مشکلاتی هستند که با حفظ Gitserver شخصی شما ایجاد می شود. اگر فقط ابزاری وجود داشته باشد که بتوانیم از آن استفاده کنیم که از آنهایی که برای ما مراقبت می کنند وجود دارد ...

## راه آسان

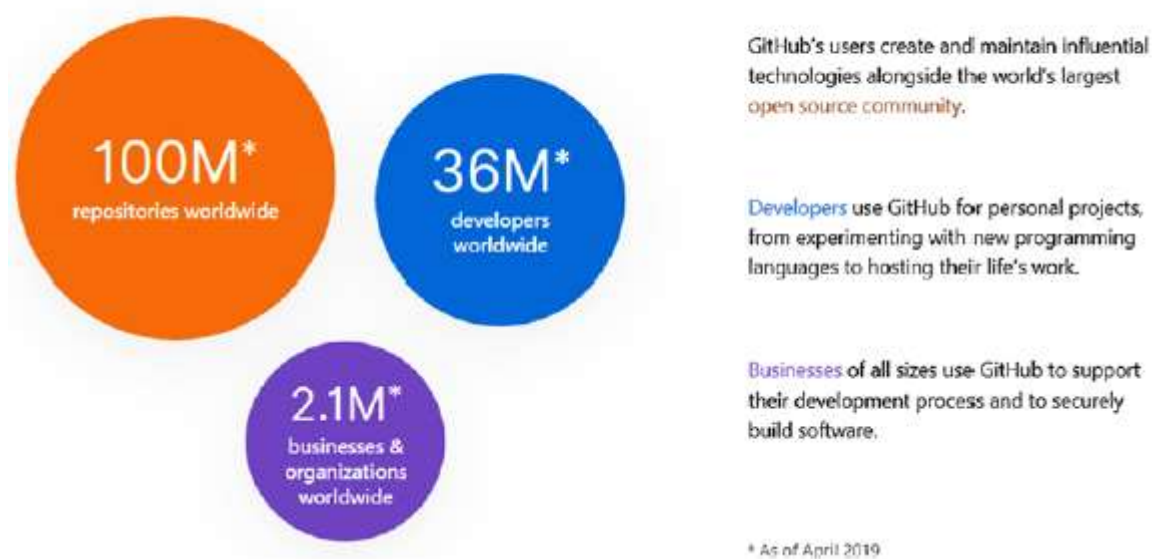
حدس بزن چی شده؟ ابزاری وجود دارد که از همه آن چیزها برای ما مراقبت می کند! و نام آن GitHub است! GitHub ابزاری برای انتخاب هنگام برخورد با مخازن از راه دور است. می توانید از GitHub به عنوان یک سرور میزبان کد برای پروژه هایی که از Git استفاده می کنند فکر کنید. این دقیقاً مانند سرور Git شما کار می کند اما سردرد کمتری دارد.

این برنامه در سال 2008 برای میزبانی پروژه های Git ایجاد شده است و اکنون یک شرکت تابعه مایکروسافت است که سرمایه گذاری زیادی در انجمن های منبع باز سرمایه گذاری کرده است. شکل 1-8 صفحه اصلی آنها را در [github.com](https://github.com) نشان می دهد.



**Figure 8-1.** GitHub homepage

حالا بیایید در مورد اعداد صحبت کنیم. GitHub دارای بیش از 100 میلیون مخزن است که توسط بیش از 36 میلیون کاربر ساخته شده است. همانطور که در شکل 2-8 مشاهده می کنید ، آنها به این اعداد بسیار افتخار می کنند.



**Figure 8-2. The users of GitHub**

GitHub تقریباً همه نیازهای توسعه دهندگان را تحت پوشش قرار می دهد ، خواه توسعه دهندگان متن باز باشند که می خواهند نرم افزار یا تیم های حرفه ای خود را که می خواهند بصورت خصوصی کار کنند بدون دردسر استفاده از سرور خودشان به اشتراک بگذارند.

تقریباً مانند یک رسانه اجتماعی ، GitHub همچنین فضایی را برای توسعه دهندگان فراهم می کند تا پروژه های خود را بسازند ، به اشتراک بگذارند و مستند کنند. دیگر نیازی به ابزار خارجی یا وب سایت ندارید.

GitHub همچنین ابزاری بسیار مهم برای پروژه های منبع باز است زیرا به منظور تسهیل روابط توسعه دهنده و انتشار کد ارائه شده است. کاربران می توانند تغییری را در پروژه های یکدیگر بررسی و پیشنهاد دهند. شما حتی می توانید در مخازن مورد علاقه خود دنبال کنید و مشارکت کنید! و فقط به پروژه های منبع باز محدود نمی شود! شرکت ها و توسعه دهندگان همچنین می توانند مخازن خصوصی ایجاد کنند که فقط توسط آنها قابل دسترسی است. آنها از ویژگی های معمول Git بهره می برند ، بلکه موارد بسیار دیگری نیز دارند. این GitHub بسیار پرتعداد است: چیزی برای همه وجود دارد!

همچنین بسیاری از شرکت های نرم افزاری وجود دارند که خدمات بسیار مشابه GitHub را ارائه می دهند و محبوب ترین آنها GitLab و BitBucket هستند.

GitLab در اکثر ویژگی های آن بسیار شبیه به GitHub است و در دو نسخه عرضه می شود:

جامعه و سازمانی. GitLab Community Edition از متن باز و به همین ترتیب مشابه GitHub است که می توانید بدون هیچ مشکلی تقریباً کلیت این کتاب را دنبال کنید. GitLab در محافل DevOps نیز بسیار مورد توجه است ، بنابراین اگر به آن مسیر شغلی علاقه مند هستید ، حتماً باید آن را بررسی کنید.

BitBucket که در ابتدا برای میزبانی پروژه های Mercurial ایجاد شده است ، از سال 2011 پشتیبانی پروژه های Git را اضافه کرده است. مدل تجاری آن که توسط Atlassian ساخته شده است بسیار شبیه به Git است و مزایای شرکت های مشابهی را ارائه می دهد.

استفاده از یک سرور محلی دارای جوانب مثبت و منفی است. اما تعداد منفی آنقدر زیاد است که ما در این کتاب می خواهیم راهی آسان را انتخاب کنیم. با این حال ، شما انتظار می رود حداقل بدانید که چگونه یک مخزن از راه دور کار می کند و چرا مورد نیاز است. اگر هنوز می خواهید از سرور خود استفاده کنید ، در یکی از پیوست های این کتاب راهنمایی وجود دارد که چگونه این کار را انجام دهید.

## خلاصه

این فصل فقط یک نمایش بسیار ساده از مخازن از راه دور بود. کار در محل سرگرم کننده است ، اما کار تیمی نیازمند به اشتراک گذاشتن تعهدات با دقت خود است. می توانید مخازن yourGit را روی سرورهای راه دور مورد نظر خود میزبان کنید ، اما ساده ترین راه استفاده از خدماتی مثل GitHub است که در میزبانی کد تخصص دارد.

اما GitHub خیلی بیشتر از همه این کارها را می کند! در فصل بعد ، ما به تفصیل بحث خواهیم کرد که ویژگی های بزرگ آن چیست و چگونه می توانیم از آنها استفاده کنیم. ما قصد داریم در مورد ردیابی اشکال ، کنترل دسترسی ، درخواست ویژگی ها و موارد دیگر بیشتر بدانیم. بیایید حرکت کنیم!