Version Control

Pervasive electronics

See http://git-scm.com/book chapters 2 and 3

What is version control

- Version management allows you to control and monitor changes to files
  - What changes were made?
  - Revert to previous versions
  - When were changes made
  - What code was present in release 2.7?
- Earliest tools were around 1972 (SCCS)
- Older tools – RCS, CVS, Microsoft Source Safe, PVCS Version Manager, etc…
- Current tools – Subversion, Mercurial, Git, Bazaar

Version control systems

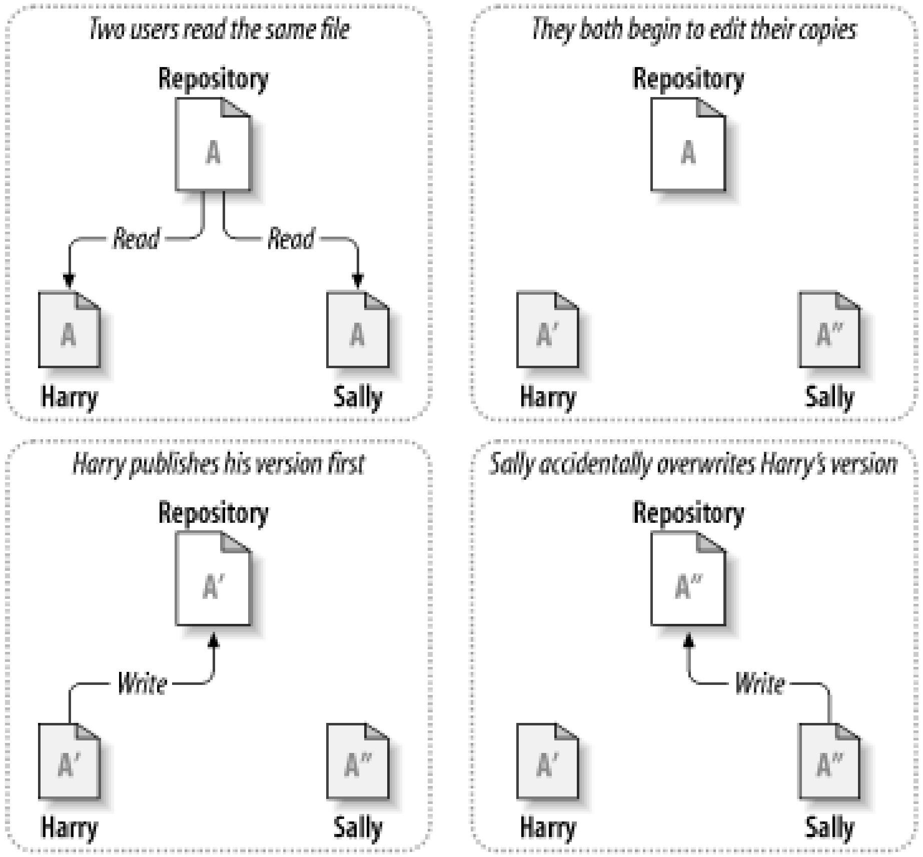
- Version control (or revision control, or source control) is all about managing multiple versions of documents, programs, web sites, etc.
  - Almost all "real" projects use some kind of version control
  - Essential for team projects, but also very useful for individual projects
- Some well-known version control systems are CVS, Subversion, Mercurial, and Git
  - CVS and Subversion use a "central" repository; users "check out" files, work on them, and "check them in"
  - Mercurial and Git treat all repositories as equal
- Distributed systems like Mercurial and Git are newer and are gradually replacing centralized systems like CVS and Subversion

Why version control?

- For working by yourself:
  - Gives you a “time machine” for going back to earlier versions
  - Gives you great support for different versions (standalone, web app, etc.) of the same basic project
- For working with others:
  - Greatly simplifies concurrent work, merging changes
- For getting an internship or job:
  - Any company with a clue uses some kind of version control
  - Companies without a clue are bad places to work

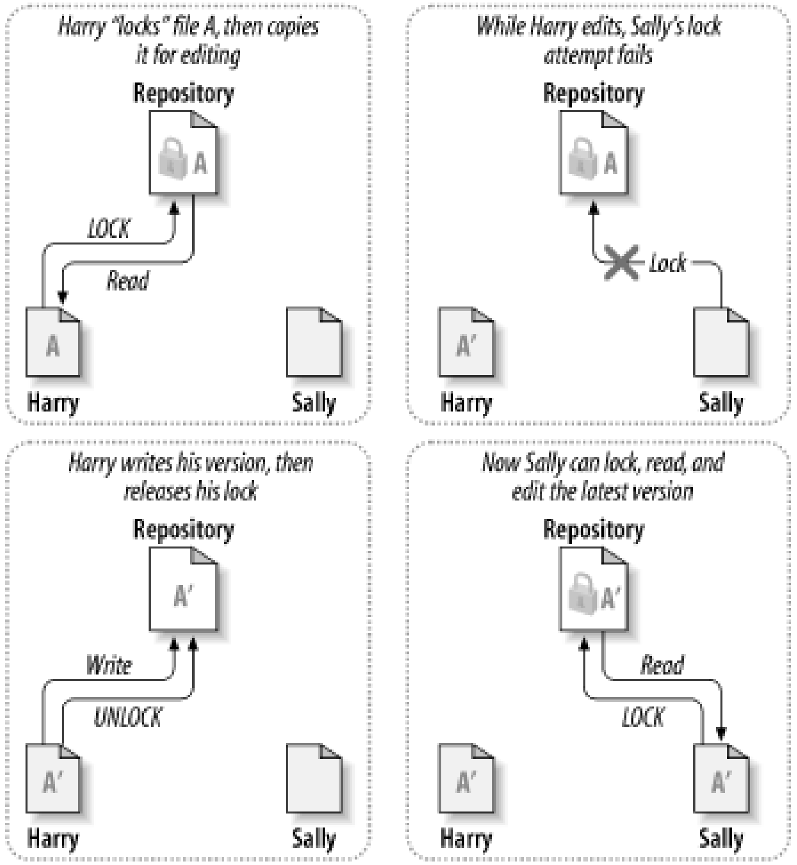
The problem to avoid

- How will the system allow users to share information, but prevent them from accidentally stepping on each other's feet?


Two users read the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Harry publishes his version first
Repository
A′
Write
A′
A″
Harry
Sally
Sally accidentally overwrites Harry's version
Repository
A″
Write
A′
A″
Harry
Sally

Lock-Modify-Unlock Solution

- In this model, the repository allows only one person to change a file at a time


Harry "locks" file A, then copies it for editing
Repository
A
LOCK
Read
A
Harry
Sally
While Harry edits, Sally's lock attempt fails
Repository
A
Lock
A'
Harry
Sally
Harry writes his version, then releases his lock
Repository
A'
Write
UNLOCK
A'
Harry
Sally
Now Sally can lock, read, and edit the latest version
Repository
A'
Read
LOCK
A'
A'
Harry
Sally

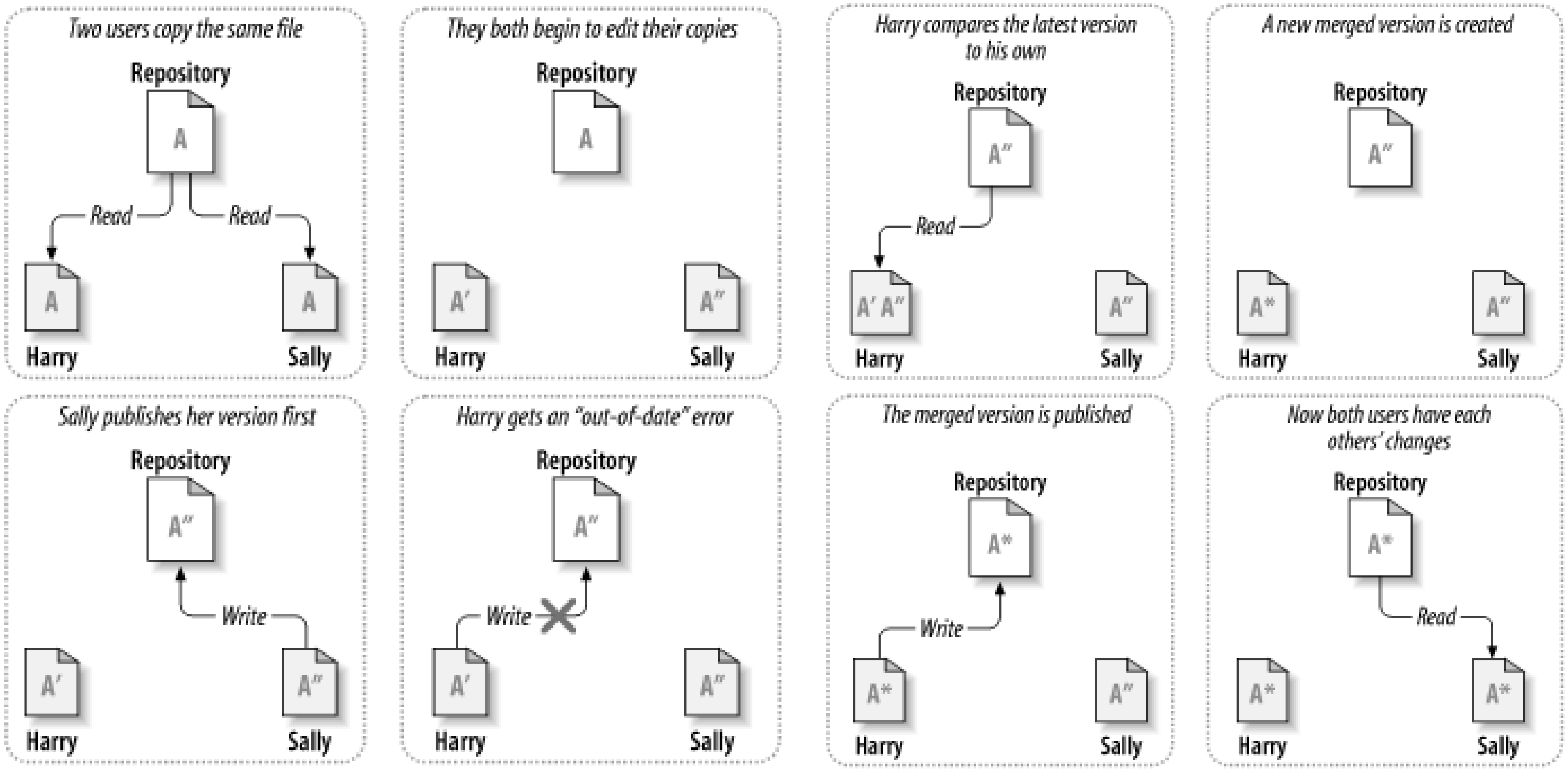
Problems of Lock-Modify-Unlock

- This model is restrictive and often becomes a roadblock for users:
- Locking may cause administrative problems: who locks a file forgets about it
- Locking may cause unnecessary serialization: modifications are made in different parts of the same file
- Locking may create a false sense of security: different files might depend one from the other

Copy-modify-merge solution

- Each user's client contacts the project repository and creates a personal working copy.
- Users then work simultaneously and independently, modifying their private copies.
- Finally, the private copies are merged together into a new, final version.
- The version control system often assists with the merging, but ultimately, a human being is responsible for making it happen correctly.

Copy-modify-merge solution


Two users copy the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Harry compares the latest version to his own
Repository
A″
Read
A′ A″
A″
Harry
Sally
A new merged version is created
Repository
A″
A*
A″
Harry
Sally
Sally publishes her version first
Repository
A″
Write
A′
A″
Harry
Sally
Harry gets an "out-of-date" error
Repository
A″
Write
A′
A″
Harry
Sally
The merged version is published
Repository
A*
Write
A*
A″
Harry
Sally
Now both users have each others' changes
Repository
A*
Read
A*
A*
Harry
Sally

Conflicts

- But what if Sally's changes do overlap with Harry's changes?
- This situation is called a **conflict**: when Harry asks his client to merge the latest repository changes into his working copy, his copy of file A is somehow flagged as being in a state of conflict: he'll be able to see **both sets of conflicting** changes and manually choose between them.
- Note that software **can't automatically resolve conflicts**; only humans are capable of understanding and making the necessary intelligent choices.
- Once Harry has manually resolved the overlapping changes—perhaps after a **discussion** with Sally—he can safely save the merged file back to the repository.

Drawbacks of centralized systems

- High risk of loosing everything for a server failure
- Slow interactions with the centralized database (dependent on the speed of the network)
- Nearly impossible to work when offline

- “When I say I hate CVS with a passion, I have to also say that if there are any SVN [Subversion] users in the audience, you might want to leave. Because my hatred of CVS has meant that I see Subversion as being the most pointless project ever started. The slogan of Subversion for a while was "CVS done right", or something like that, and if you start with that kind of slogan, there's nowhere you can go. There is no way to do CVS right.”

--Linus Torvalds, as quoted in Wikipedia

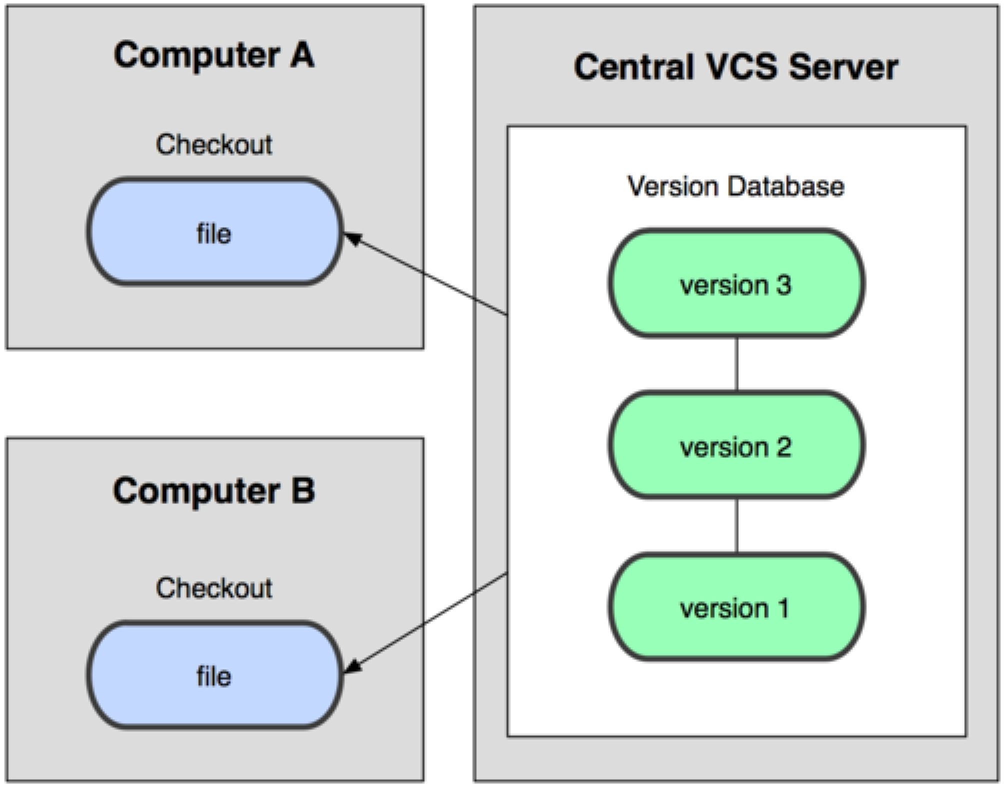
History

Created by Linus Torvalds for work on the Linux kernel ~2005

Some of the companies that use git:

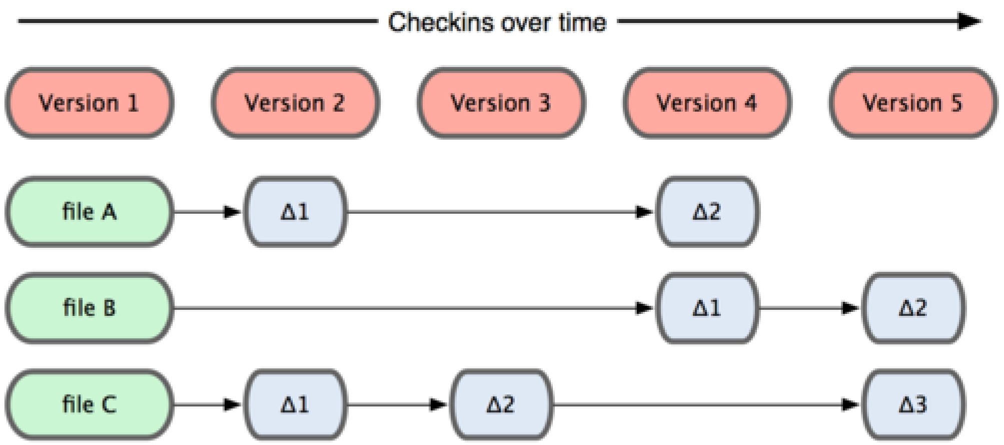
Google
facebook
Microsoft
twitter
LinkedIn
NETFLIX
PostgreSQL
android
RAILS
Qt
GNOME
eclipse
K

Centralized Version Control

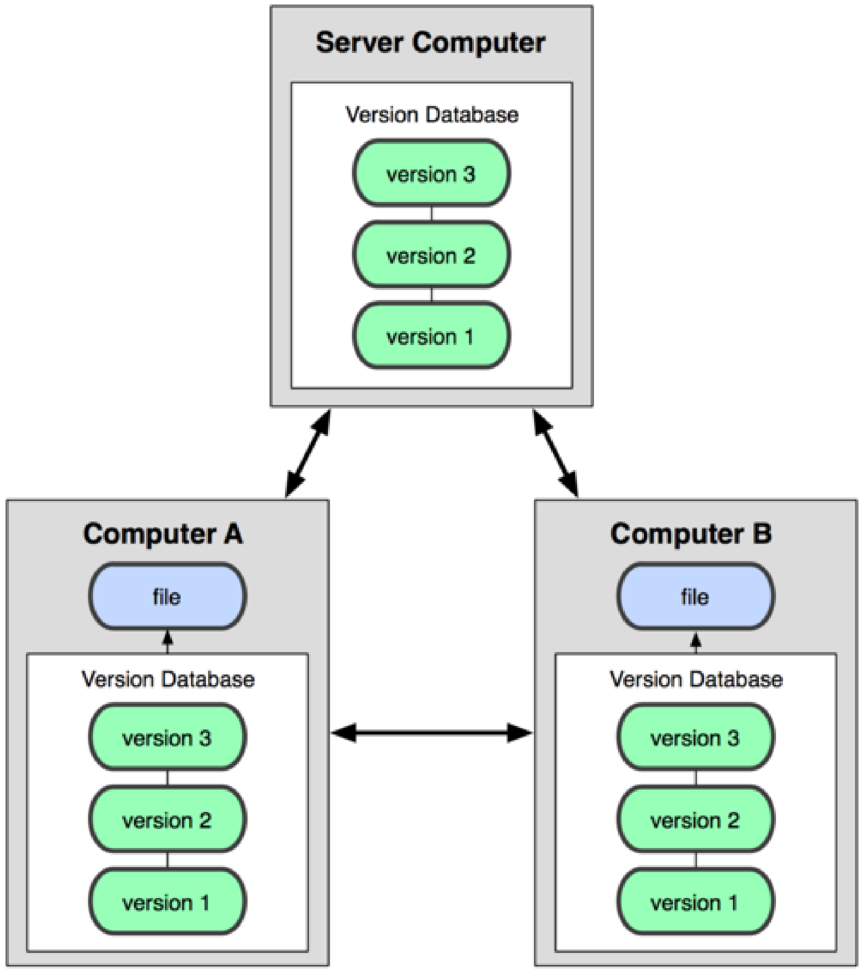


Subversion is like this

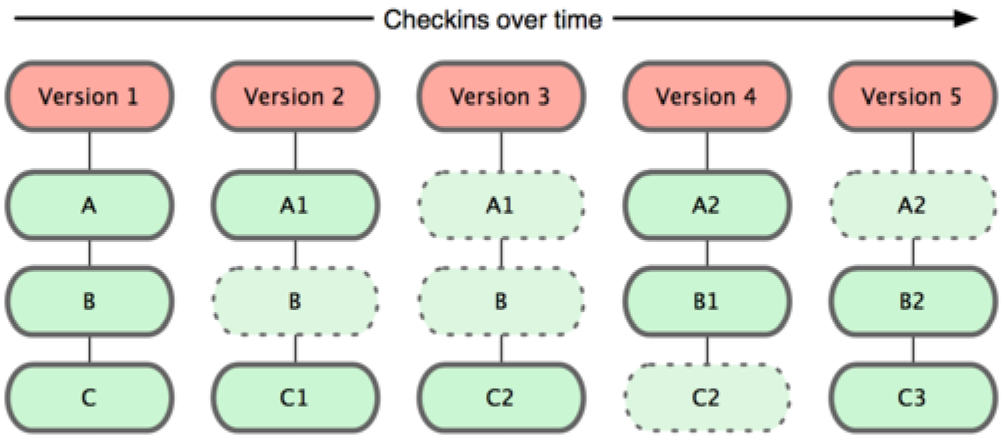
Centralized - Differences



Distributed Version Control

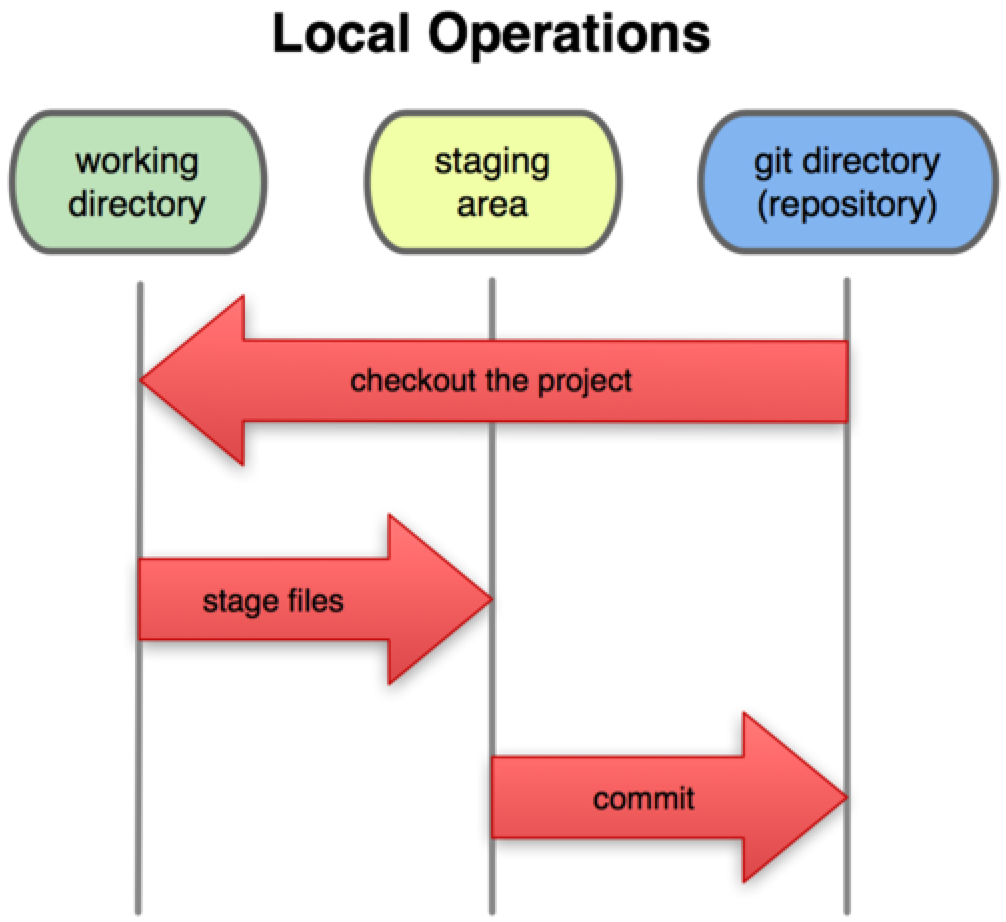


Distributed - Snapshots



- Files are stored by SHA-1 hash rather than filename
- Stored in git database in compressed format
- Database is stored on your *local* machine
- Must “checkout” from database into working directory to edit
- In this example, files A, B and C are *tracked*

Local Operations



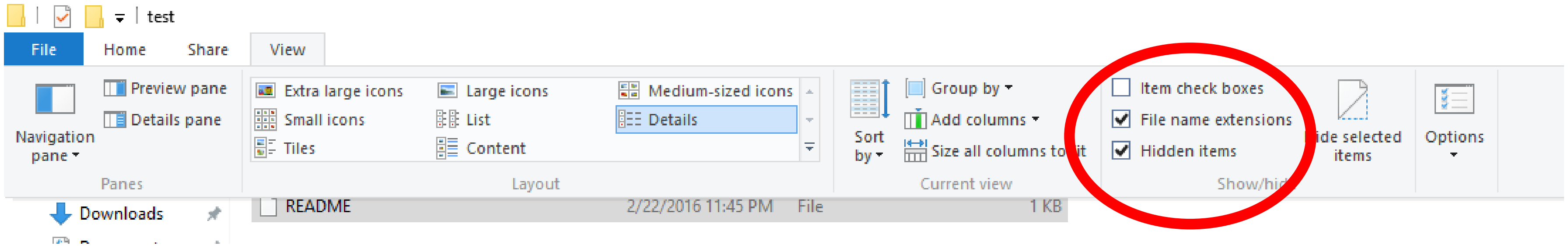
Why might you want to stage files?

Download and install Git

- Download and install latest version from http://git-scm.com/
- Latest version is: Git-2.7.2 (22 February 2016)
- Optionally install a GUI (e.g., TortoiseGIT https://tortoisegit.org/)
- Optionally install GitHub client (e.g. on windows https://windows.github.com/)

Configure file explorer

- In File explorer: **enable visualization** of
  - File name extensions
  - Hidden items

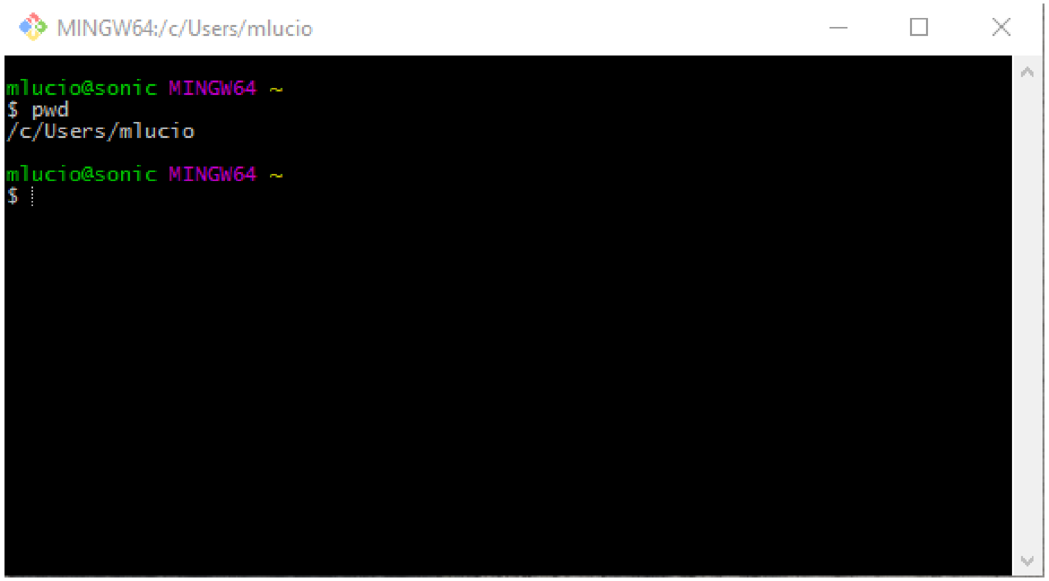


- [ ] Item check boxes
- [✓] File name extensions
- [✓] Hidden items

README 2/22/2016 11:45 PM File 1 KB

Git Bash

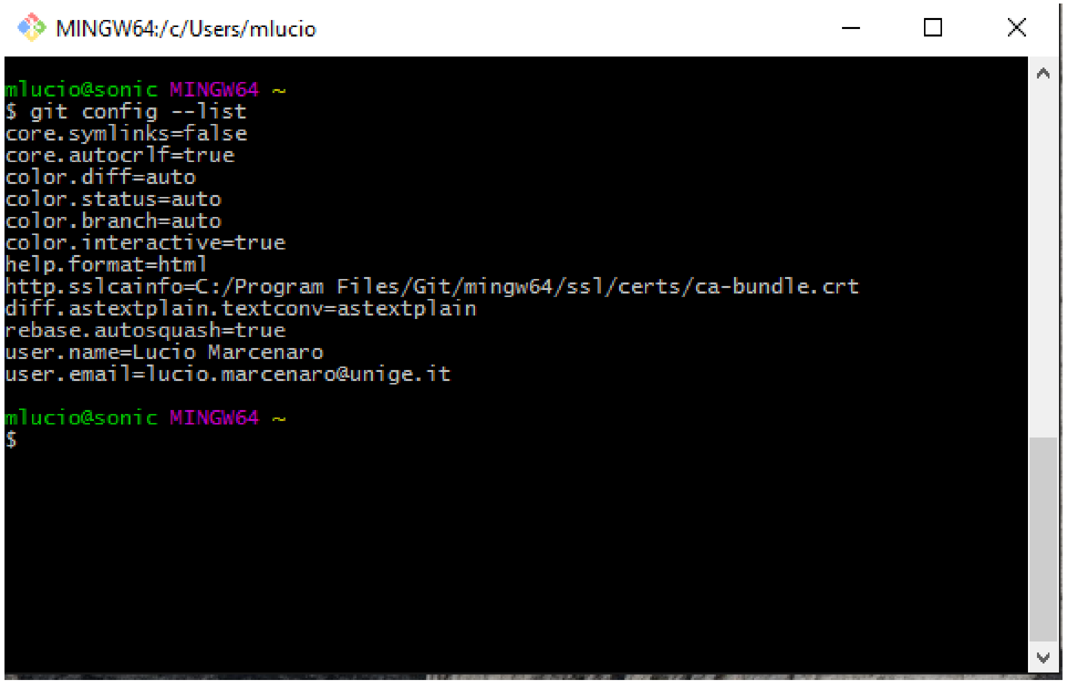
- Work with Git bash (GUIs also available)
  - pwd = show current directory
  - cd = change directory (navigate to your files)
  - ls = list the directory contents
  - Use TAB key to speed-up writing names of files/folders
  - Arrow keys to go through the command history



```
mlucio@sonic MINGW64 ~
$ pwd
/c/Users/mlucio

mlucio@sonic MINGW64 ~
$ 
```

Tell Git who you are

- Update your config, one time only
  - git config --global user.name “Lucio Marcenaro”
  - git config --global user.email lucio.marcenaro@unige.it
  - git config --list



MINGW64:/c/Users/mlucio

```
mlucio@sonic MINGW64 ~
$ git config --list
core.symlinks=false
core.autocrlf=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
rebase.autosquash=true
user.name=Lucio Marcenaro
user.email=lucio.marcenaro@unige.it

mlucio@sonic MINGW64 ~
$
```

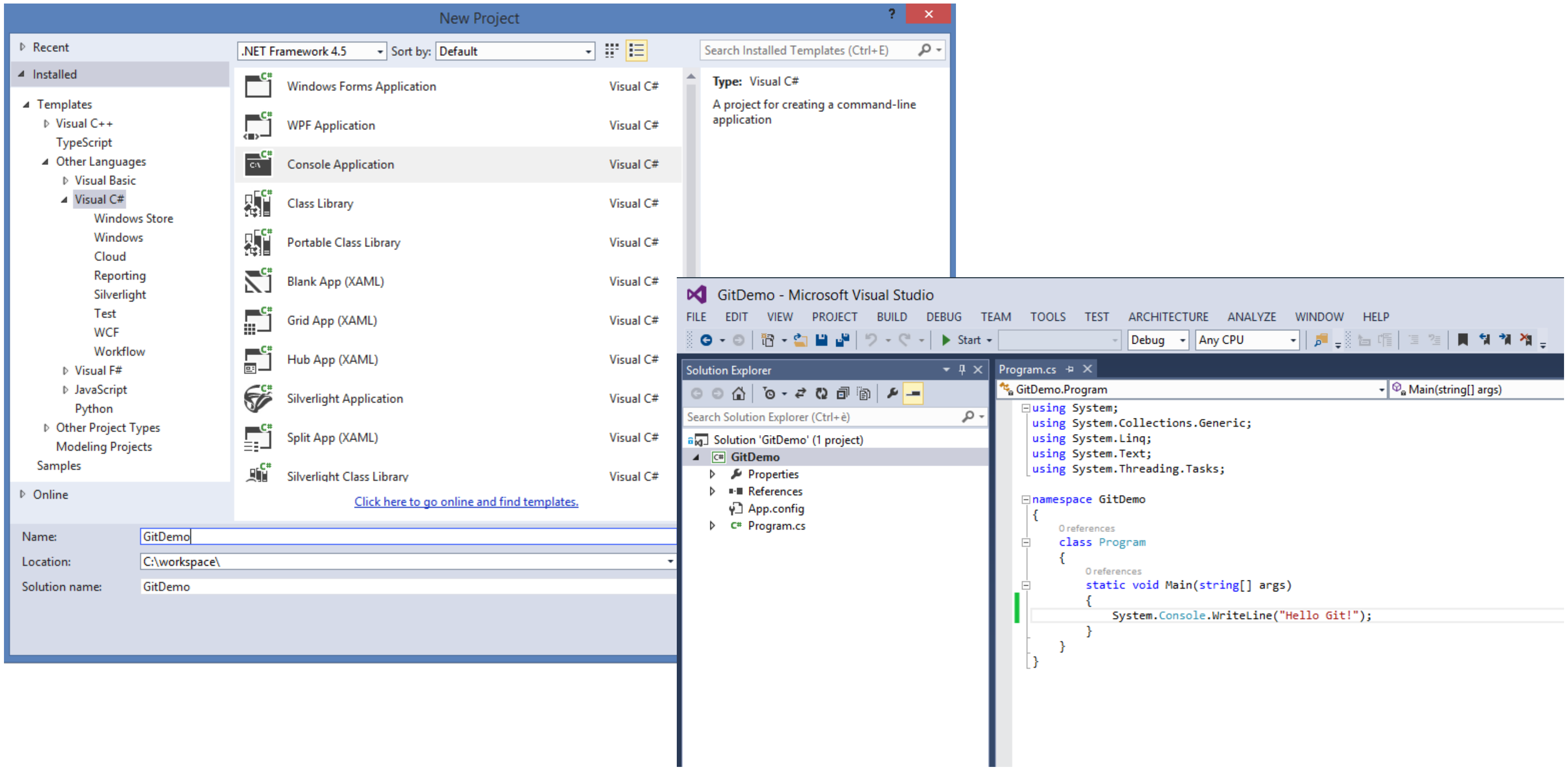
.gitignore

- It’s important to tell Git what files you do *not* want to track
- Temp files, executable files, etc. do not need version control (and can cause major issues when merging!)
- Several examples of .gitignore files can be found at: https://github.com/github/gitignore
- Example (place in root of repo):

```
*.suo
*.user
*.sln.docstates
[Dd]ebug/
[Dd]ebugPublic/
[Rr]elease/
[Rr]eleases/
x64/
x86/
build/
bld/
[Bb]in/
[Oo]bj/
```

Getting started

- Open VisualStudio and create C# Project (mine is GitDemo)


New Project
Recent
Installed
Templates
Visual C++
TypeScript
Other Languages
Visual Basic
Visual C#
Windows Store
Windows
Cloud
Reporting
Silverlight
Test
WCF
Workflow
Visual F#
JavaScript
Python
Other Project Types
Modeling Projects
Samples
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Class Library Visual C#
Portable Class Library Visual C#
Blank App (XAML) Visual C#
Grid App (XAML) Visual C#
Hub App (XAML) Visual C#
Silverlight Application Visual C#
Split App (XAML) Visual C#
Silverlight Class Library Visual C#
Click here to go online and find templates.
GitDemo - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Start
Debug
Any CPU
Solution Explorer
Search Solution Explorer (Ctrl+è)
Solution 'GitDemo' (1 project)
GitDemo
Properties
References
App.config
Program.cs
Program.cs
GitDemo.Program
Main(string[] args)

Type: Visual C#

A project for creating a command-line application

Name: GitDemo

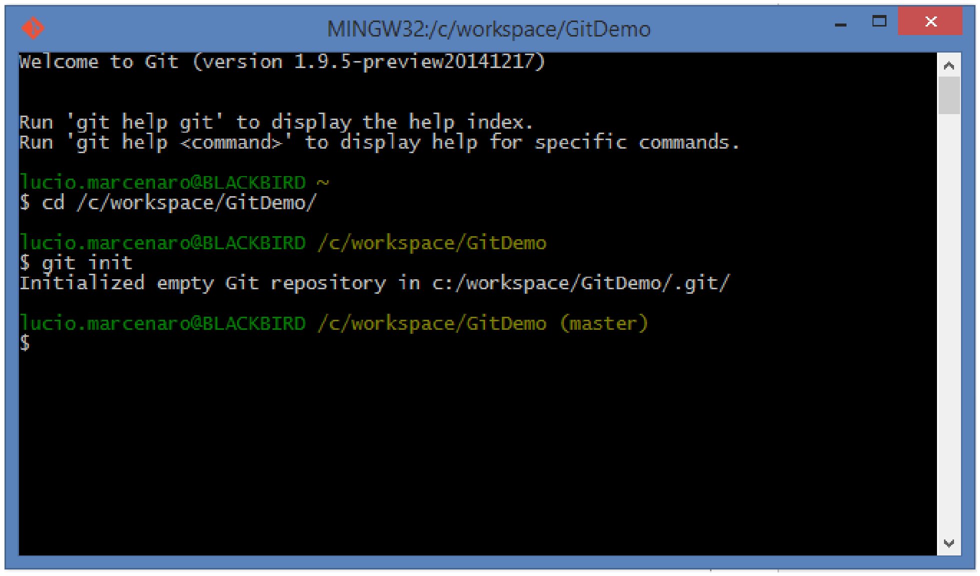
Location: C:\workspace\

Solution name: GitDemo

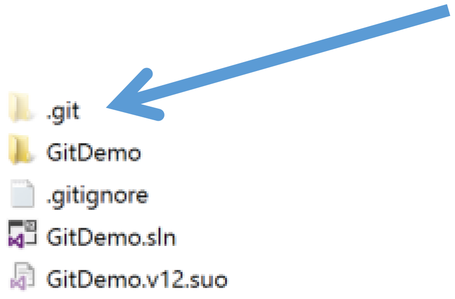
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GitDemo
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello Git!");
        }
    }
}
```

Getting started

- Download .gitignore file for VisualStudio from github (i.e. https://raw.githubusercontent.com/github/gitignore/master/VisualStudio.gitignore) and copy in the C# Project folder (rename it to .gitignore)
- Open git bash and **cd** into directory.
- git init
- Notice the .git directory. That's the git database (yep, all of it!)

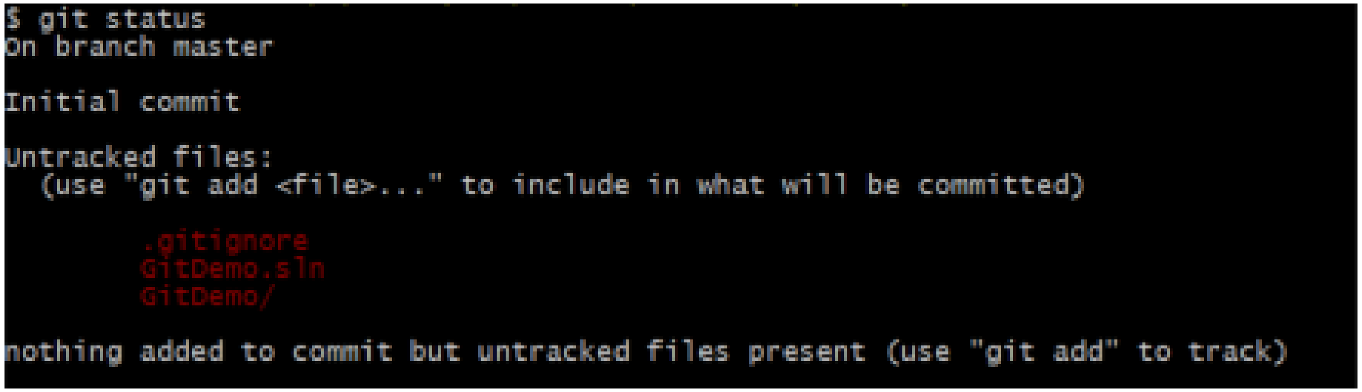


```
Welcome to Git (version 1.9.5-preview20141217)


Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

lucio.marcenaro@BLACKBIRD ~
$ cd /c/workspace/GitDemo/

lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo
$ git init
Initialized empty Git repository in c:/workspace/GitDemo/.git/

lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$
```



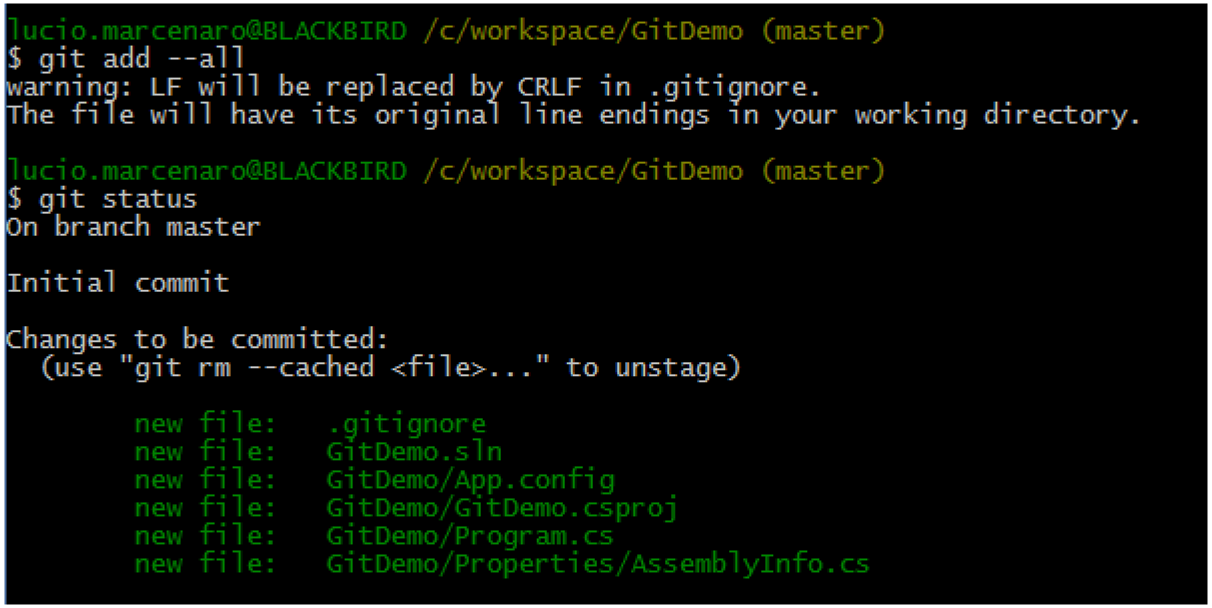
Track those files

Nothing tracked yet.



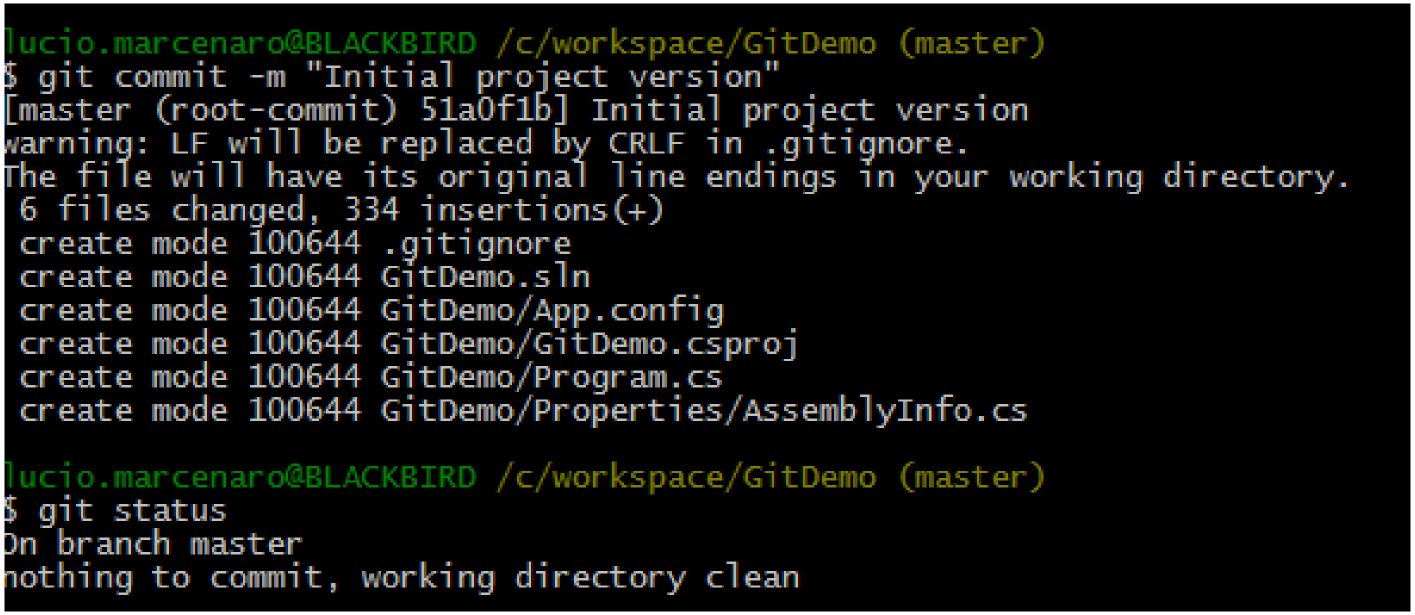
```
$ git status
On branch master

Initial commit

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

        .gitignore
        GitDemo.sln
        GitDemo/

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

Tell Git to track all files



```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git add --all
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory.

lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git status
On branch master

Initial commit

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

        new file:   .gitignore
        new file:   GitDemo.sln
        new file:   GitDemo/App.config
        new file:   GitDemo/GitDemo.csproj
        new file:   GitDemo/Program.cs
        new file:   GitDemo/Properties/AssemblyInfo.cs
```

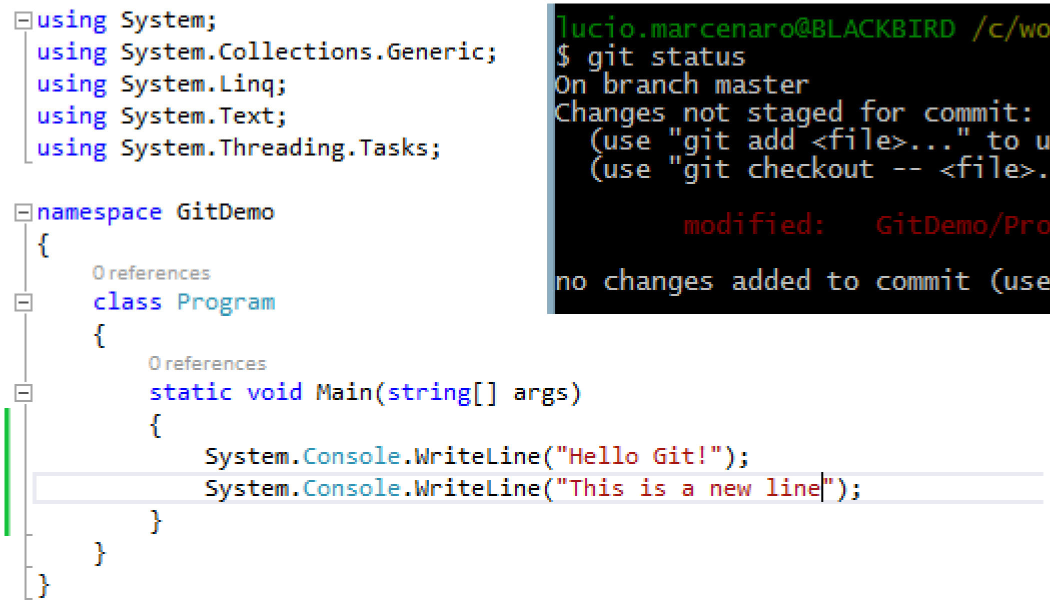
Commit the file to the Git database



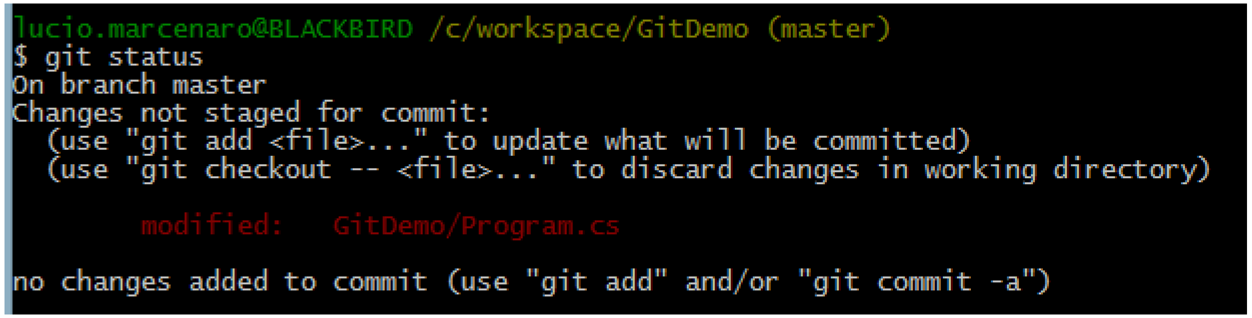
```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git commit -m "Initial project version"
[master (root-commit) 51a0f1b] Initial project version
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory.
 6 files changed, 334 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 GitDemo.sln
 create mode 100644 GitDemo/App.config
 create mode 100644 GitDemo/GitDemo.csproj
 create mode 100644 GitDemo/Program.cs
 create mode 100644 GitDemo/Properties/AssemblyInfo.cs

lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git status
On branch master
nothing to commit, working directory clean
```

When you commit, you must provide a comment (if you forget, Git will open a text editor so you can write one.

What if you change the file?

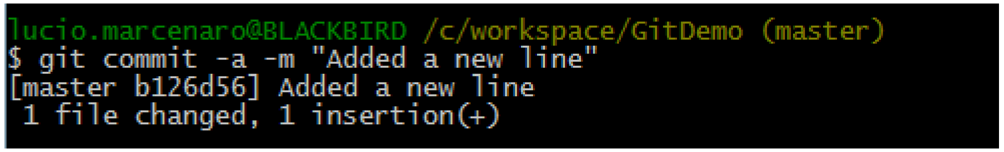


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GitDemo
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello Git!");
            System.Console.WriteLine("This is a new line");
        }
    }
}
```



```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (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:   GitDemo/Program.cs

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

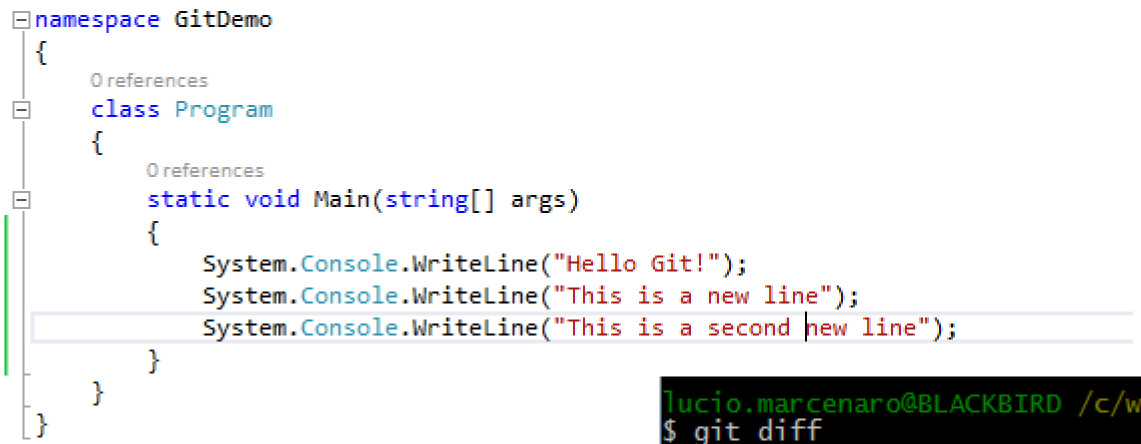
Notice the helpful hints Git provides.
You could add to the staging area, OR add & commit in one step.



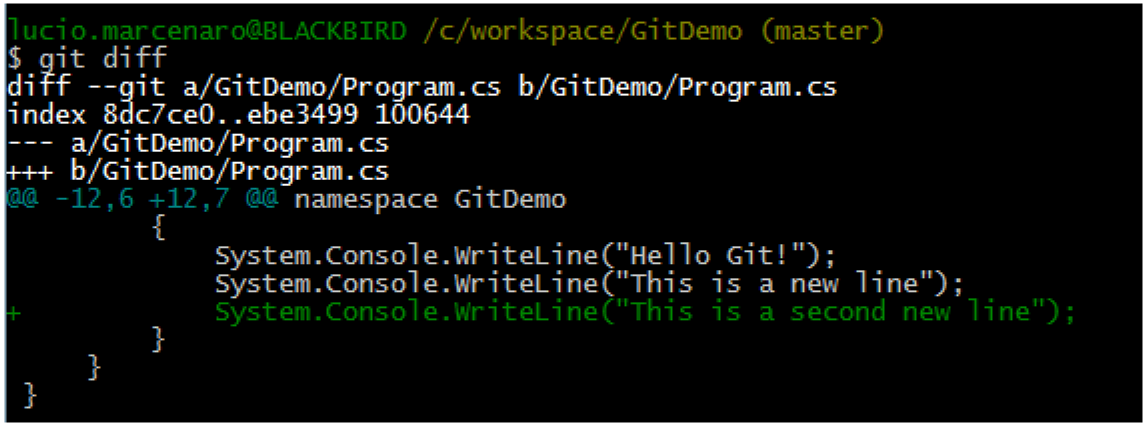
```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git commit -a -m "Added a new line"
[master b126d56] Added a new line
 1 file changed, 1 insertion(+)
```

Be careful if you add to the staging area and then make more changes – the file can appear as both staged and unstaged. For now, I suggest doing –a -m

You made some changes – but what did you do?



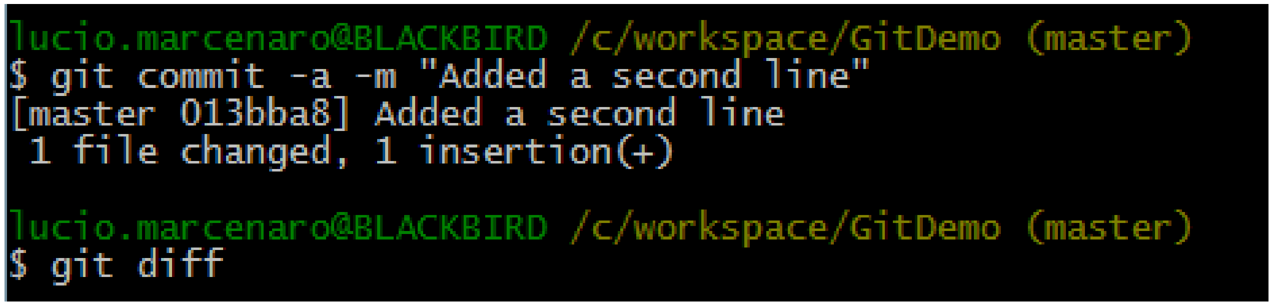
```
namespace GitDemo
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello Git!");
            System.Console.WriteLine("This is a new line");
            System.Console.WriteLine("This is a second new line");
        }
    }
}
```



```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git diff
diff --git a/GitDemo/Program.cs b/GitDemo/Program.cs
index 8dc7ce0..ebe3499 100644
--- a/GitDemo/Program.cs
+++ b/GitDemo/Program.cs
@@ -12,6 +12,7 @@ namespace GitDemo
         {
             System.Console.WriteLine("Hello Git!");
             System.Console.WriteLine("This is a new line");
+            System.Console.WriteLine("This is a second new line");
         }
     }
 }
```

This command compares your working directory with your staging area. These are the changes that are not yet staged.

What if you've committed all your changes?

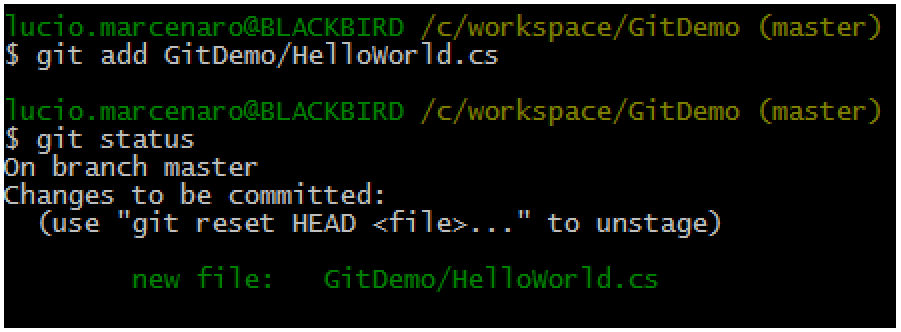


```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git commit -a -m "Added a second line"
[master 013bba8] Added a second line
 1 file changed, 1 insertion(+)

lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git diff
```

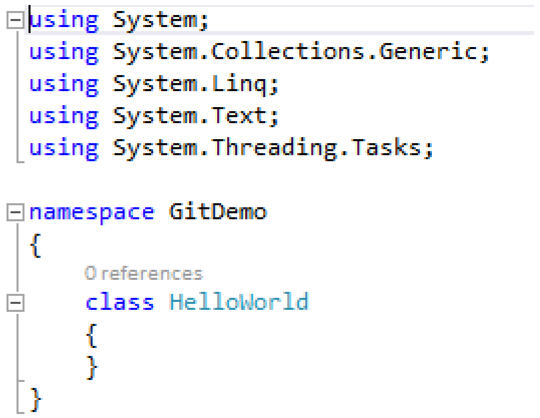
diff doesn't have anything to display

What if I remove a file?

File added not committed

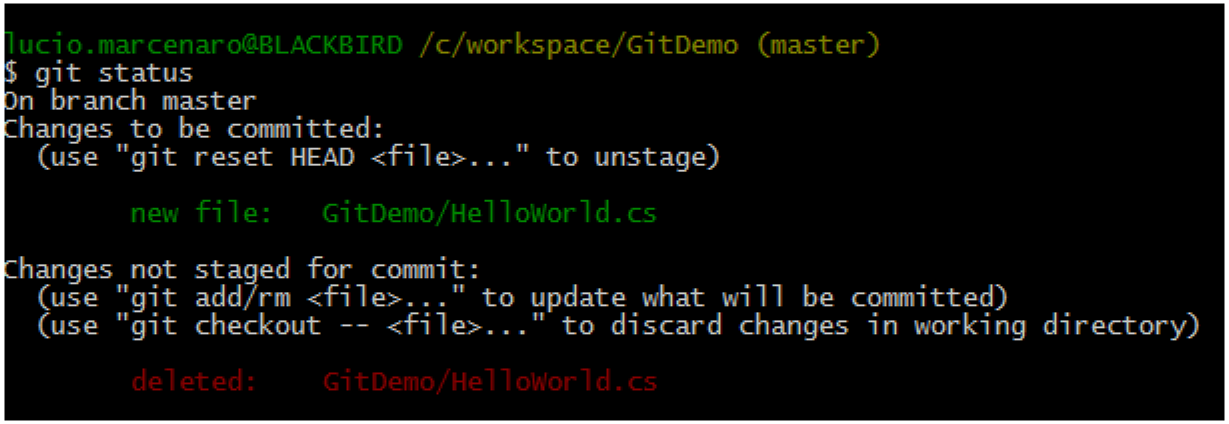


```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git add GitDemo/HelloWorld.cs

lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   GitDemo/HelloWorld.cs
```



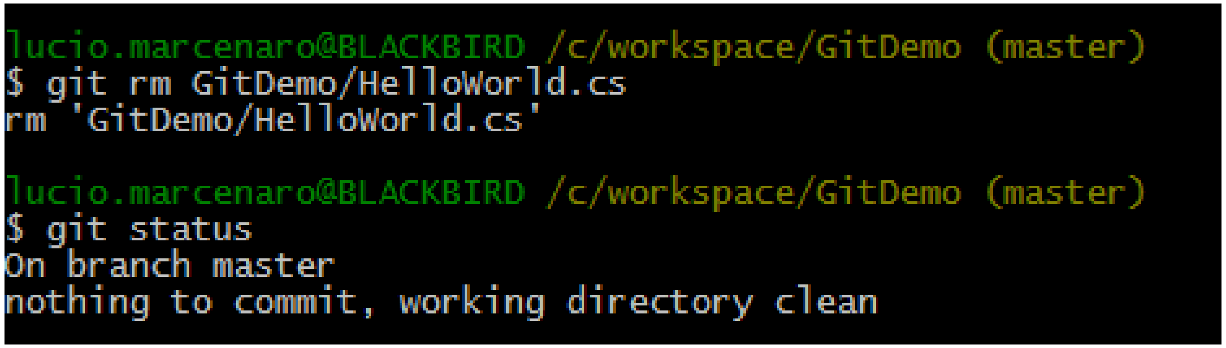
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GitDemo
{
    0 references
    class HelloWorld
    {
    }
}
```

Now I remove HelloWorld.cs from inside Visual Studio



```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   GitDemo/HelloWorld.cs

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

        deleted:    GitDemo/HelloWorld.cs
```

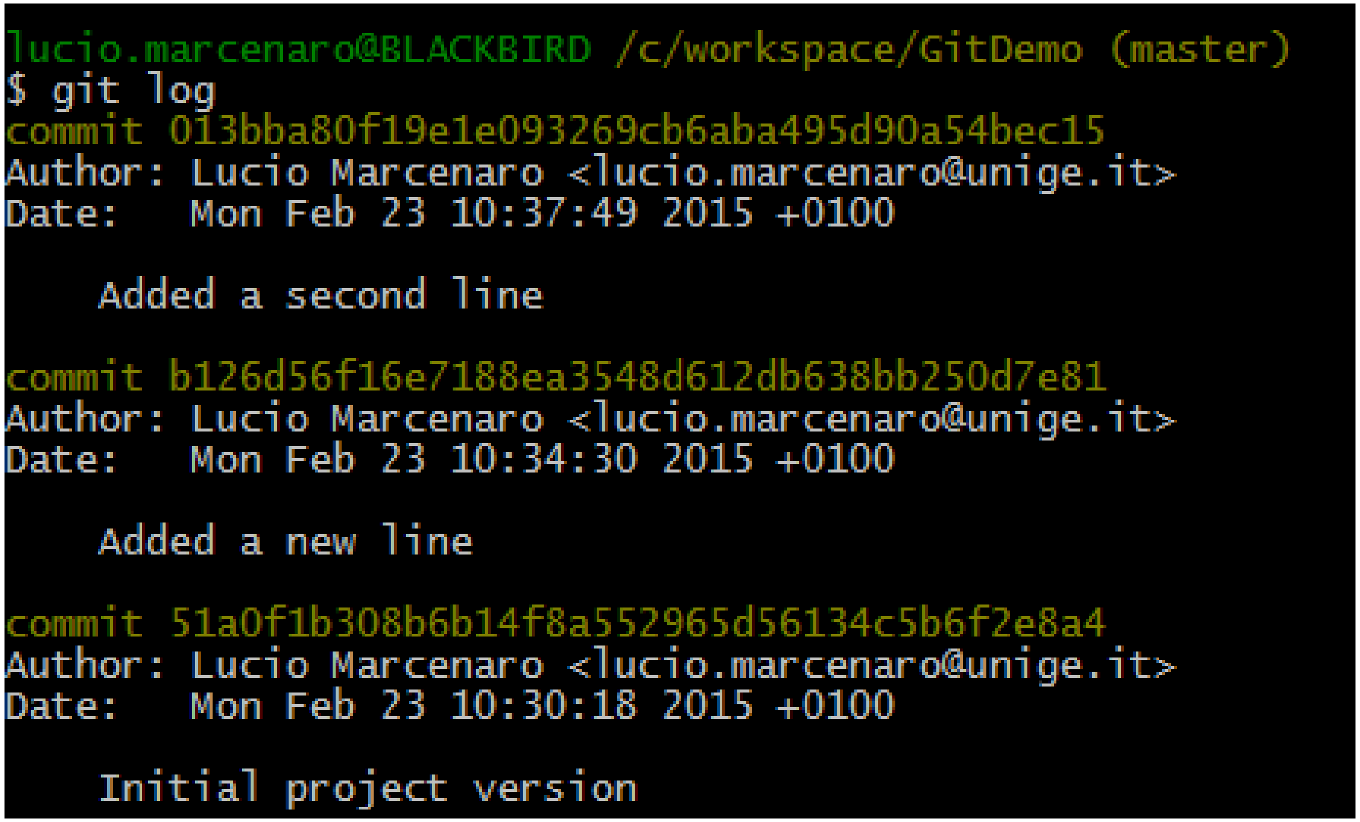
removal, continued



```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git rm GitDemo/HelloWorld.cs
rm 'GitDemo/HelloWorld.cs'

lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git status
On branch master
nothing to commit, working directory clean
```

This removes the file from being tracked. If you've already committed, the file is still in the database. -- see github tutorials for info on this.

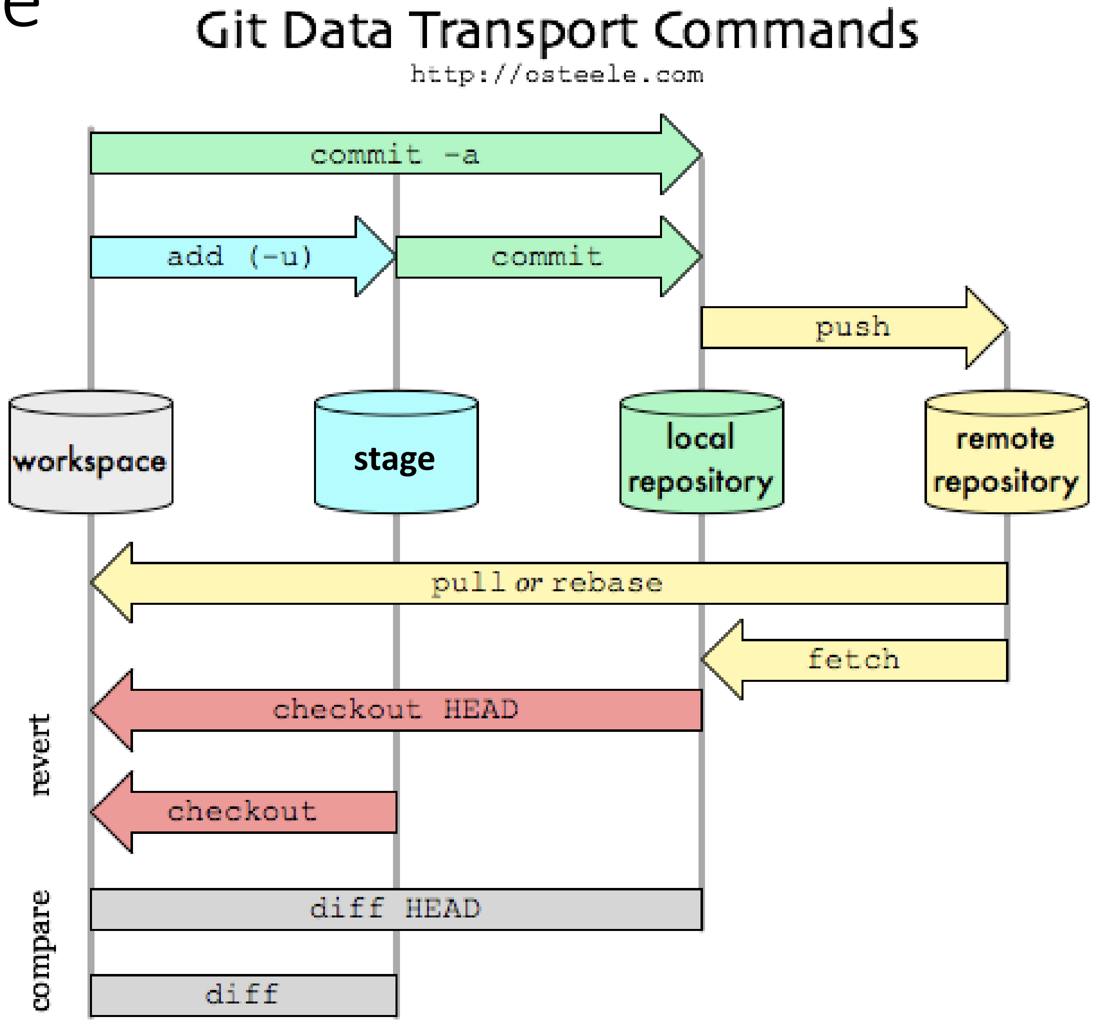
So what all have I done?



```
lucio.marcenaro@BLACKBIRD /c/workspace/GitDemo (master)
$ git log
commit 013bba80f19e1e093269cb6aba495d90a54bec15
Author: Lucio Marcenaro <lucio.marcenaro@unige.it>
Date:   Mon Feb 23 10:37:49 2015 +0100

    Added a second line

commit b126d56f16e7188ea3548d612db638bb250d7e81
Author: Lucio Marcenaro <lucio.marcenaro@unige.it>
Date:   Mon Feb 23 10:34:30 2015 +0100

    Added a new line

commit 51a0f1b308b6b14f8a552965d56134c5b6f2e8a4
Author: Lucio Marcenaro <lucio.marcenaro@unige.it>
Date:   Mon Feb 23 10:30:18 2015 +0100

    Initial project version
```

There are many useful options for git log.

The Big Picture



http://blog.mikepearce.net/2010/05/18/the-difference-between-git-pull-git-fetch-and-git-clone-and-git-rebase/

Quick Summary for git bash

- cd to your project or source directory
  - cd = change directory.

  Example: if files stored c:\workspace\GitDemo
  - cd workspace
  - cd GitDemo
  - OR /c/workspace/GitDemo
- ls (shows a list of files in that directory)
- git init (creates .git repo)
- git add --all (tracks all files recursively considering .gitignore)
- git commit -am “Initial commit” (adds files to repo)
- git status (see what files have been modified, etc.)
- git log (see list of commits)

git log options

- Can specify a format, such as:
  - git log –pretty=format: "%h - %an, %ar : %s"
  - %h = abbreviated hash
  - %an = author name
  - %ar = author date, relative
  - %s = subject
  - MORE options, see documentation
- Can filter, such as:
  - git log –pretty=format: "%h - %an%s" -- since "2013-12-01“
  - includes filters like –since, --after, --until, --before, --author etc
- Can redirect output to file, such as:
  - git log >> gitlog.txt

.

Initializing a Repository in an Existing Directory

- Create an empty directory
- Start «git bash» program, cd to the created directory and issue the following command:

```
git init
```

- Creates a new subdirectory named .git that contains all of your necessary repository files
- Create a README file with some simple text

```
git add README
git commit –m "initial project version"
```

- At this point, you have a Git repository with tracked files and an initial commit.

Default editor

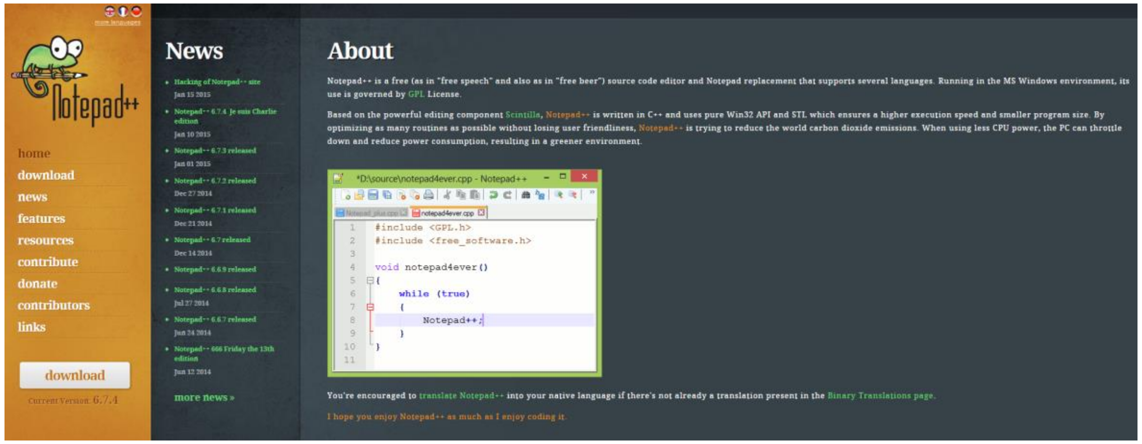
- If we forget the `–m` option git will start the `vi` editor
- `vi` has 2 modes:
  - command mode (initial or "default" mode)
  - insert mode
- [`Esc`] is used to switch to command mode.
- To insert text:
  - `i` inserts text before cursor.
  - `a` appends text after cursor.
- To save file and quit the vi:
  - First check that you are in command mode by pressing [`ESC`]
  - `ZZ`
  - `:wq`

A more user friendly editor

- Notepad++ is a great free editor http://notepad-plus-plus.org/
- For being able to use notepad++ as the default text editor on Windows the following command can be used:

```
git config --global core.editor "'C:/Program Files
(x86)/Notepad++/notepad++.exe' -multiInst -
notabbar -nosession -noPlugin"
```

- Git will start automatically notepad++ when needed



News

- Hacking of Notepad++ site
  Jan 15 2015
- Notepad++ 6.7.4 Je suis Charlie edition
  Jan 10 2015
- Notepad++ 6.7.3 released
  Jan 01 2015
- Notepad++ 6.7.2 released
  Dec 27 2014
- Notepad++ 6.7.1 released
  Dec 21 2014
- Notepad++ 6.7 released
  Dec 14 2014
- Notepad++ 6.6.9 released
- Notepad++ 6.6.8 released
  Jul 27 2014
- Notepad++ 6.6.7 released
  Jun 24 2014
- Notepad++ 666 Friday the 13th edition
  Jun 12 2014

more news »

About

Notepad++ is a free (as in "free speech" and also as in "free beer") source code editor and Notepad replacement that supports several languages. Running in the MS Windows environment, its use is governed by GPL License.

Based on the powerful editing component Scintilla, Notepad++ is written in C++ and uses pure Win32 API and STL which ensures a higher execution speed and smaller program size. By optimizing as many routines as possible without losing user friendliness, Notepad++ is trying to reduce the world carbon dioxide emissions. When using less CPU power, the PC can throttle down and reduce power consumption, resulting in a greener environment.

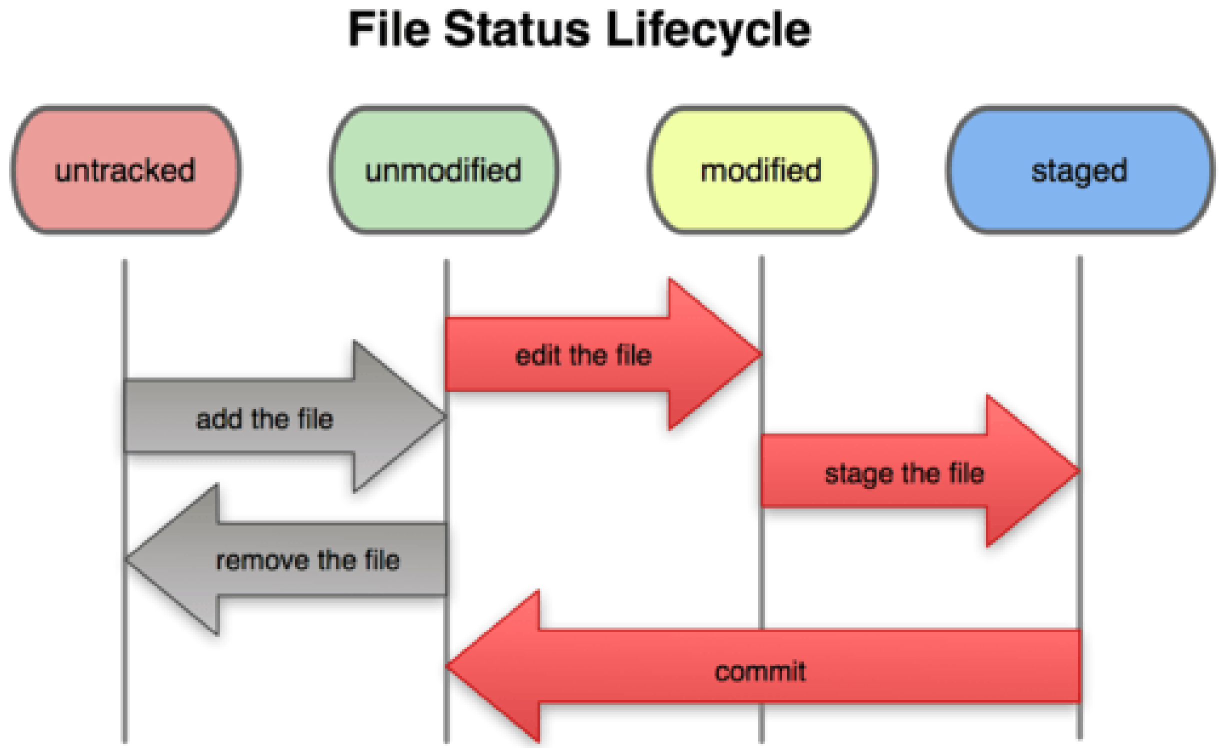
```
#include <GPL.h>
#include <free_software.h>

void notepad4ever()
{
    while (true)
    {
        Notepad++;
    }
}
```

You're encouraged to translate Notepad++ into your native language if there's not already a translation present in the Binary Translations page.

I hope you enjoy Notepad++ as much as I enjoy coding it.

File status lifecycle

- Each file can be in one of two states: tracked or untracked
- Tracked files can be unmodified, modified, or staged
- Untracked files are everything else



Checking the status of your files

- The main tool you use to determine which files are in which state is the git status command.

```
$ git status
# On branch master
nothing to commit, working directory clean
```

- Create a new empty file README and try again with git status

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will
be committed)
#
#       README
nothing added to commit but untracked files present
(use "git add" to track)
```

Tracking new files

- In order to begin tracking a new file, you use the command git add

```
git add README
```

- Try again with git status

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
```

Staging modified files (1/3)

- Try to change a file that is already tracked (HelloWorld.cs)
- Try again with git status

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#
# 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:   HelloWorld.cs
```

Staging modified files (2/3)

- To stage the modified file run the git add command (it’s a multipurpose command—you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved)

```
$ git add HelloWorld.cs
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   HelloWorld.cs
#
```

Staging modified files (3/3)

- Try to change again HelloWorld.cs (suppose we forgot a minor modification)

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   HelloWorld.cs
#
# 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:   HelloWorld.cs
```

View staged and unstaged changes (1/2)

- To see what you’ve changed but not yet staged, type git diff with no other arguments (compares the working directory with the staging area)

```
$ git diff
diff --git a/HelloWorld.cs b/HelloWorld.cs
index ebe3499..de87b57 100644
--- a/HelloWorld.cs
+++ b/HelloWorld.cs
@@ -11,8 +11,6 @@ namespace GitDemo
         static void Main(string[] args)
         {
             System.Console.WriteLine("Hello Git!");
-            System.Console.WriteLine("This is a new line");
-            System.Console.WriteLine("This is a second new line");
         }
     }
 }
```

View staged and unstaged changes (2/2)

- To compare your staged changes to your last commit type git diff --staged

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index ebe3499..de87b57 100644
--- a/HelloWorld.cs
+++ b/HelloWorld.cs
@@ -11,8 +11,6 @@ namespace GitDemo
         static void Main(string[] args)
         {
+            System.Console.WriteLine("Hello Git!");
         }
     }
 }
```

Committing your changes

- After your staging area is set up as you want it, you can commit your changes

```
git commit
```

- Will start the default editor

```
$ git commit -m "added hello git program"
[master cade440] added hello git program
 2 files changed, 1 insertion(+)
 create mode 100644 README
```

Removing files (1/3)

- To remove a file from Git it must be removed from the tracked files and then commit
- Remove README from the hard disk

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    README
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Removing files (2/3)

- Then run git rm to stage the file’s removal

```
$ git rm README
rm 'README'
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    README
#
```

- Next commit, the file will be gone and no longer tracked.
- If you modified the file and added it to the index already, you must force the removal with the –f option.

Removing files (3/3)

- To keep the file in your working tree but remove it from your staging area

```
$ git rm --cached README
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#       README
```

Moving files

- If you want to rename a file in Git, you can run something like

```
$ git mv README README.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README -> README.txt
#
```

View the commit history (1/2)

- Use the git log command

```
$ git log
commit cb31124f00ff7ea85269cad7622f06b1940d32df
Author: Lucio Marcenaro <lucio.marcenaro@unige.it>
Date:   Tue Feb 25 21:55:14 2014 +0100
    file rename test
commit cade4403240ed5198991180b6cef4361274de8bd
Author: Lucio Marcenaro <lucio.marcenaro@unige.it>
Date:   Tue Feb 25 21:36:25 2014 +0100
    added hello git program
commit 491f23b6cf9a9afca07896e158f594d5cc9c9284
Author: Lucio Marcenaro <lucio.marcenaro@unige.it>
Date:   Tue Feb 25 20:46:40 2014 +0100
    Initial project version
```

View the commit history (2/2)

- One of the more helpful options is –p, which shows the diff introduced in each commit
- You can also use -2, which limits the output to only the last two entries
- The –stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed
- Another really useful option is –pretty. This option changes the log output to formats other than the default:

```
git log --pretty=oneline
$ git log --pretty=format:"%h - %an, %ar : %s"
cb31124 - Lucio Marcenaro, 6 minutes ago : file
rename test
cade440 - Lucio Marcenaro, 25 minutes ago : added
hello git program
491f23b - Lucio Marcenaro, 74 minutes ago : Initial
project version
```

Undoing things: changing your last commit

- You commit too early and possibly forget to add some files, or you mess up your commit message

```
git commit --amend
```

- As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ git commit
$ git add forgotten.txt
$ git commit --amend
[master 0fac616] file rename test added new file
 2 files changed, 0 insertions(+), 0 deletions(-)
 rename README => README.txt (100%)
 create mode 100644 forgotten.txt
```

Undoing things: unstaging a staged file

- You commit too early and possibly forget to add some files, or you mess up your commit message

```
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#       modified:   HelloWorld.cs
#
$ git reset HEAD HelloWorld.cs
```

Undoing things: unmodifying a modified file

- You realize that you don't want to keep your changes to the HelloWorld.cs

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.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:   HelloWorld.cs
$ git checkout -- HelloWorld.cs
```

Working with remotes

- Remote repositories are versions of your project that are hosted on the Internet or network somewhere

```
$ git clone https://github.com/schacon/ticgit.git
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB |
338.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
$ cd ticgit/
$ git remote -v
origin  https://github.com/schacon/ticgit.git (fetch)
origin  https://github.com/schacon/ticgit.git (push)
```

Adding remote repositories

- To add a new remote Git repository as a shortname you can reference easily, run git remote add [shortname] [url]

```
$ git remote add pb
https://github.com/paulboone/ticgit.git

$ git remote -v

origin https://github.com/schacon/ticgit.git (fetch)

origin https://github.com/schacon/ticgit.git (push)

pb https://github.com/paulboone/ticgit.git (fetch)

pb https://github.com/paulboone/ticgit.git (push)
```

- Now you can use the string pb on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don’t yet have in your repository, you can run git fetch pb

Fetching and pulling from remotes

- The command fetch goes out to that remote project and pulls down all the data from that remote project that you don’t have yet
- It’s important to note that the fetch command pulls the data to your local repository—it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready
- The git pull command to automatically fetch and then merge a remote branch into your current branch
- The git clone command automatically sets up your local master branch to track the remote master branch on the server you cloned from

Pushing to your remotes

- When you have your project at a point that you want to share, you have to push it upstream.
- The command for this is simple: git push [remote-name] [branch-name]
- If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push your work back up to the server:

```
git push origin master
```

- This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime

Inspecting a remote

- If you want to see more information about a particular remote, you can use the git remote show [remote-name] command.

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit.git
  Push  URL: https://github.com/schacon/ticgit.git
  HEAD branch: master
  Remote branches:
    master tracked
    ticgit tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Removing and renaming remotes

- If you want to rename a reference you can run git remote rename to change a remote’s shortname
- It’s worth mentioning that this changes your remote branch names, too. What used to be referenced at pb/master is now at paul/master.
- If you want to remove a reference for some reason — you’ve moved the server or are no longer using a particular mirror, or perhaps a contributor isn’t contributing anymore — you can use git remote rm
- `$ git remote rm pb`
- `$ git remote`
- `origin`

Tagging (1/3)

- Git has the ability to tag specific points in history as being important (e.g., release points)
- Listing the available tags in Git is straightforward. Just type git tag
- A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.
- Annotated tags, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, e-mail, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).
- Creating annotated tags:

```
$ git tag -a v1.4 -m 'my version 1.4‘
```

- Creating lightweight tags:

```
$ git tag v1.4-lw
```

Tagging (2/3)

- You can see the tag data along with the commit that was tagged by using the git show command
- You can also tag commits after you’ve moved past them

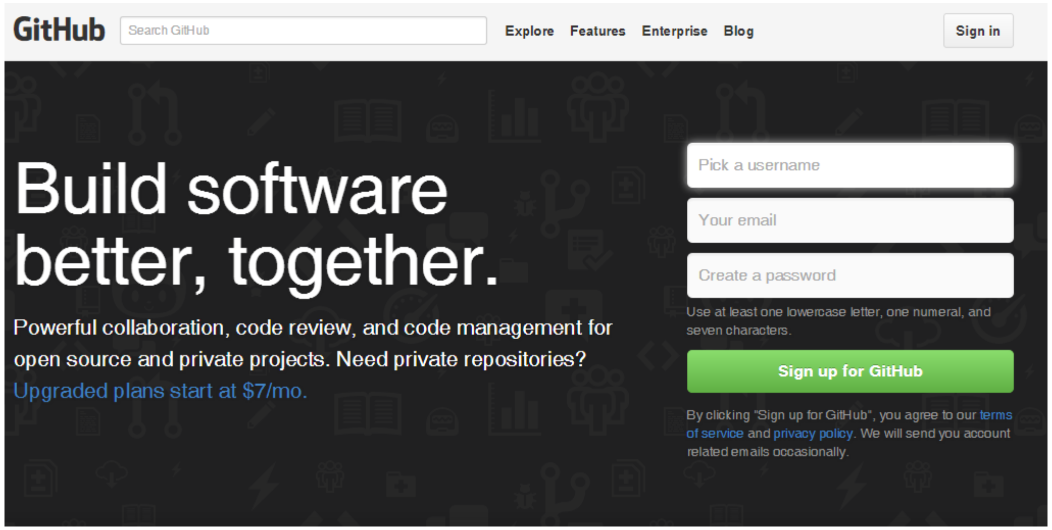
```
$ git log --pretty=oneline
0fac616a11a86f4c4172f64ead008f67ce2543e5 file
rename test added new file
cade4403240ed5198991180b6cef4361274de8bd added
hello git program
491f23b6cf9a9afca07896e158f594d5cc9c9284 Initial
project version
$ git tag -a v1.2 -m 'version 1.2' cade440
```

Tagging (3/3)

- By default, the git push command doesn’t transfer tags to remote servers.
- You will have to explicitly push tags to a shared server after you have created them.
- This process is just like sharing remote branches — you can run git push origin [tagname]
- If you have a lot of tags that you want to push up at once, you can also use the --tags option to the git push command.
- This will transfer all of your tags to the remote server that are not already there.

GitHub (https://github.com/)

- GitHub is a famous remote git server
- Many shareware projects are stored on GitHub
- Create new user
  - Pick a username
  - Type your email
  - Create a password



Build software better, together.

Powerful collaboration, code review, and code management for open source and private projects. Need private repositories? Upgraded plans start at $7/mo.

Pick a username

Your email

Create a password

Use at least one lowercase letter, one numeral, and seven characters.

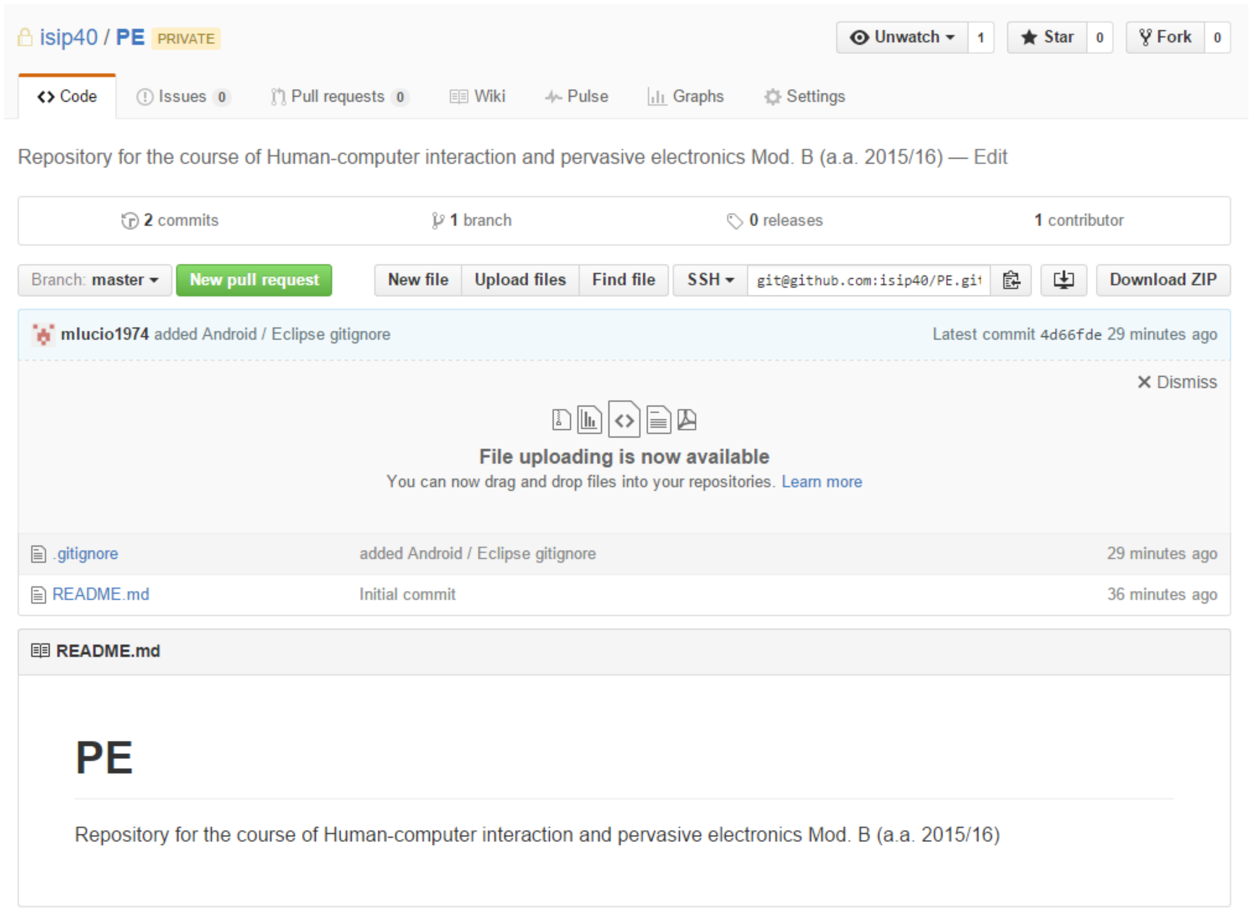
By clicking "Sign up for GitHub", you agree to our terms of service and privacy policy. We will send you account related emails occasionally.

Clone PE repo

- Send me (lucio.marcenaro@unige.it) your GitHub user name
- You will receive an automatic email after I add you to the PEStudents team under «isip40» organization (i.e. the research group name)
- After refreshing your github page you can click on the repository https://github.com/isip40/PE

Clone PE repo

- Copy the HTTPS clone url



isip40 / PE PRIVATE

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

Repository for the course of Human-computer interaction and pervasive electronics Mod. B (a.a. 2015/16) — Edit

2 commits 1 branch 0 releases 1 contributor

Branch: master New pull request New file Upload files Find file SSH git@github.com:isip40/PE.git Download ZIP

mlucio1974 added Android / Eclipse gitignore Latest commit 4d66fde 29 minutes ago

✕ Dismiss

File uploading is now available
You can now drag and drop files into your repositories. Learn more

.gitignore	added Android / Eclipse gitignore	29 minutes ago
README.md	Initial commit	36 minutes ago

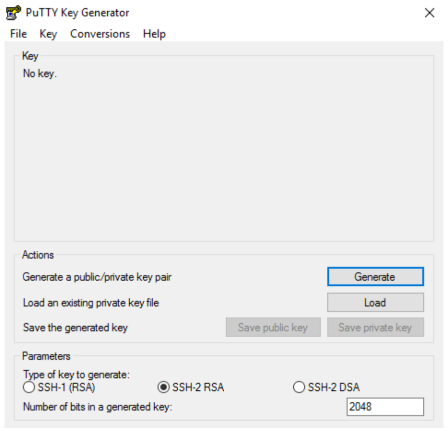
README.md

PE

Repository for the course of Human-computer interaction and pervasive electronics Mod. B (a.a. 2015/16)

Working with GITHub

- Right click on an empty folder and **clone** the PE repo (through TortoiseGIT clone)
- TortoiseGIT has a credential helper for avoiding writing username and password each time
- You can also use **SSH authentication** by selecting the SSH repository link and generating/using an SSH public/private key pair
  - Use PuTTYgen to generate SSH-2 RSA keys
  - Load public key into your user profile in GitHUB
  - Check «Load Putty Key» when cloning


PuTTY Key Generator

File Key Conversions Help

Key

No key.

Actions

Generate a public/private key pair — Generate

Load an existing private key file — Load

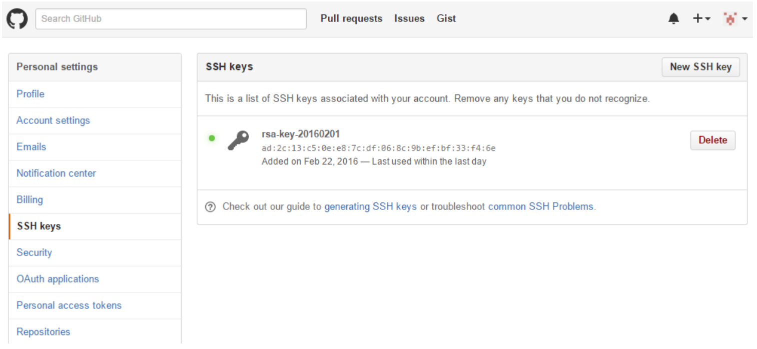
Save the generated key — Save public key — Save private key

Parameters

Type of key to generate:

SSH-1 (RSA) [✓ SSH-2 RSA] SSH-2 DSA

Number of bits in a generated key: 2048


Search GitHub — Pull requests — Issues — Gist

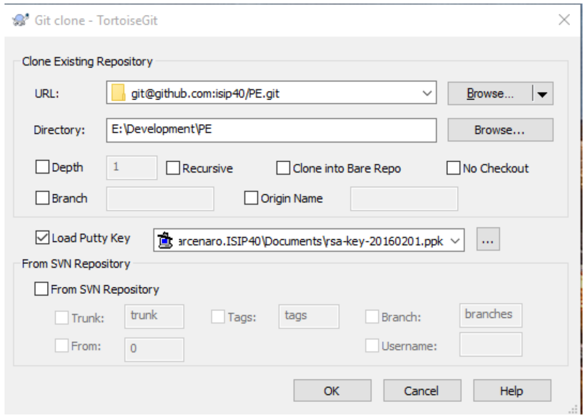
Personal settings: Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security, OAuth applications, Personal access tokens, Repositories

SSH keys — New SSH key

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

rsa-key-20160201
`ad:2c:13:c5:0e:e8:7c:df:06:8c:9b:ef:bf:33:f4:6e`
Added on Feb 22, 2016 — Last used within the last day — Delete

Check out our guide to generating SSH keys or troubleshoot common SSH Problems.


Git clone - TortoiseGit

Clone Existing Repository

URL: git@github.com:isip40/PE.git — Browse...

Directory: E:\Development\PE — Browse...

☐ Depth 1 ☐ Recursive ☐ Clone into Bare Repo ☐ No Checkout

☐ Branch ☐ Origin Name

☑ Load Putty Key: arcenaro.ISIP40\Documents\rsa-key-20160201.ppk — ...

From SVN Repository

☐ From SVN Repository

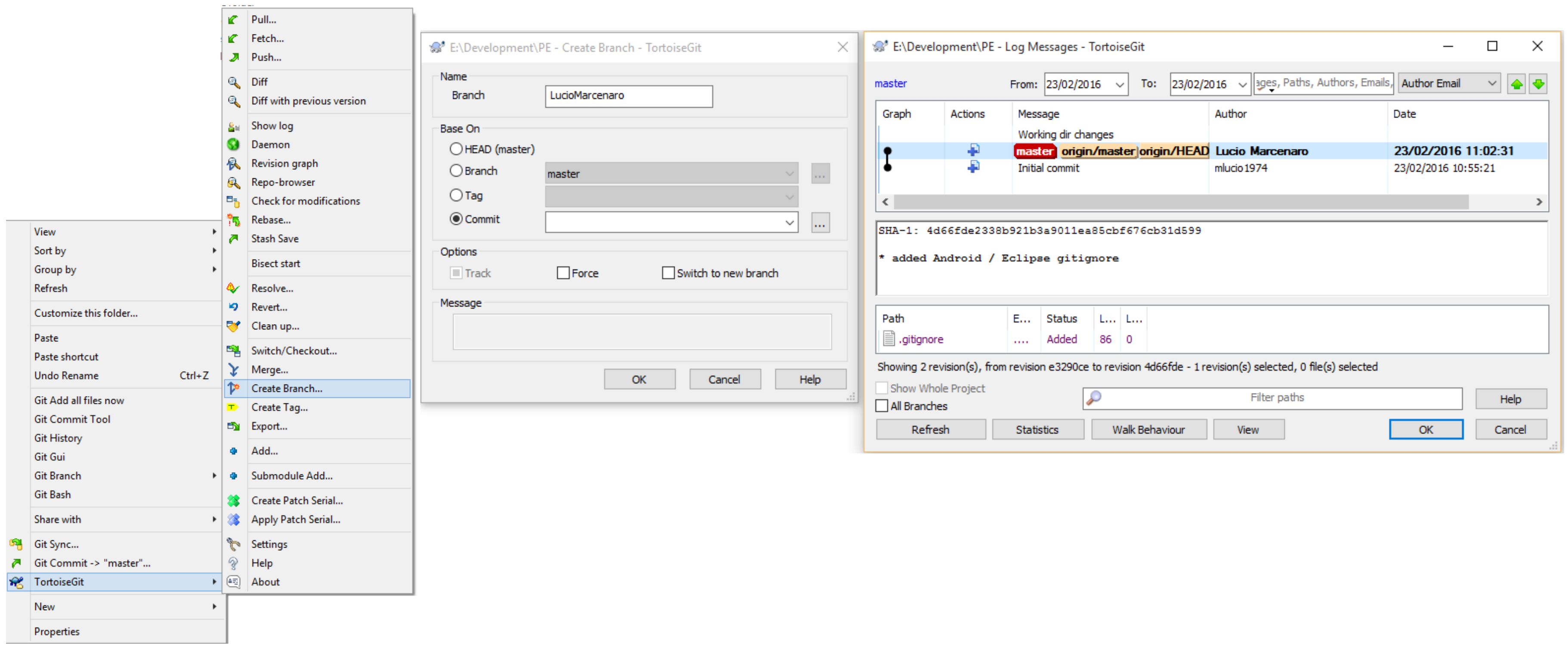
☐ Trunk: trunk ☐ Tags: tags ☐ Branch: branches

☐ From: 0 ☐ Username:

OK — Cancel — Help

Working with GITHub

- Decide a **nickname** for your group (can be your username)
- Create a **new branch** (use the same name as your group) and **switch to it**
- **IMPORTANT:** create the **branch** based on the **second commit** (SHA-1 4d66fde2338b921b3a9011ea85cbf676cb31d599) not on the HEAD



Name
Branch: LucioMarcenaro

Base On
( ) HEAD (master)
( ) Branch: master
( ) Tag
(●) Commit

Options
☐ Track ☐ Force ☐ Switch to new branch

Message

master From: 23/02/2016 To: 23/02/2016 ages, Paths, Authors, Emails, Author Email

Graph	Actions	Message	Author	Date
		Working dir changes		
●		master origin/master origin/HEAD	Lucio Marcenaro	23/02/2016 11:02:31
●		Initial commit	mlucio1974	23/02/2016 10:55:21

```
SHA-1: 4d66fde2338b921b3a9011ea85cbf676cb31d599

* added Android / Eclipse gitignore
```

Path	E...	Status	L...	L...
.gitignore	....	Added	86	0

Showing 2 revision(s), from revision e3290ce to revision 4d66fde - 1 revision(s) selected, 0 file(s) selected

Working with GITHub

- Create a subfolder named after the **nickname**
- Work in **that folder** (add new files/projects)
- Always work in your branch and **merge** with master branch only when needed (i.e. send me your exercises)
- The master branch is typically protected: I will unprotect it when approaching the deadline for exercises

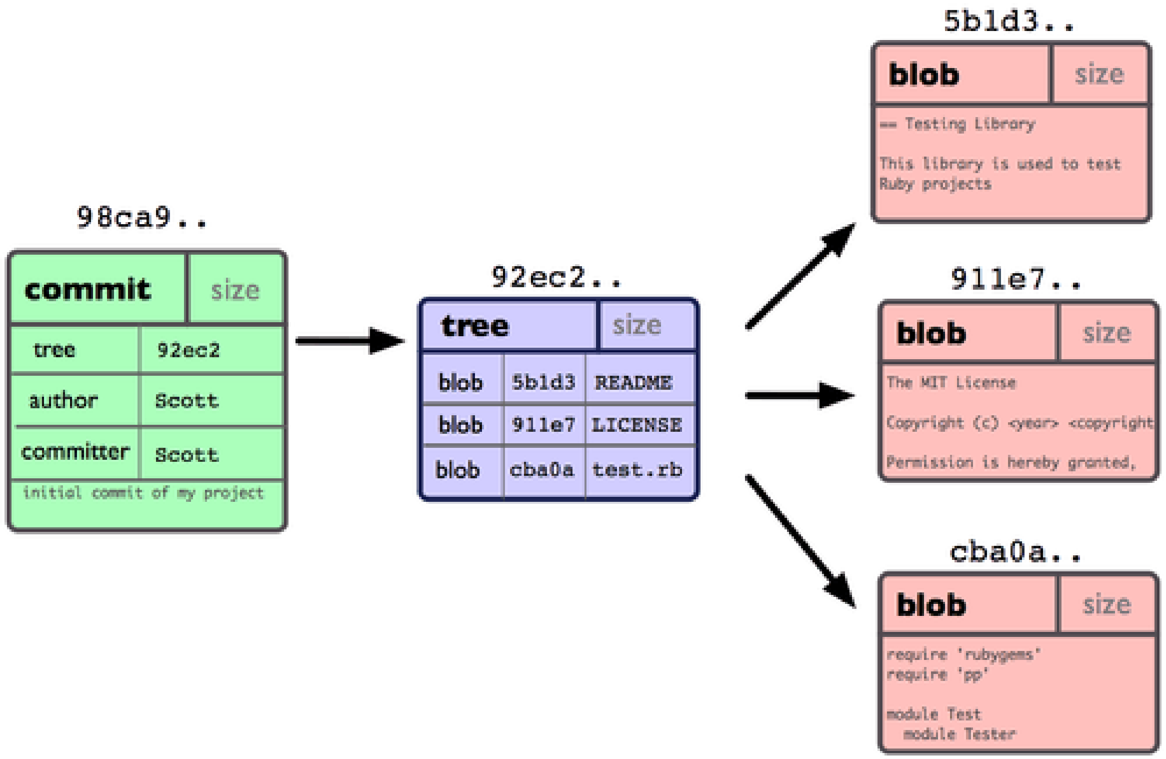
Git branching

- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- Git branches is incredibly **lightweight**, making branching operations nearly **instantaneous** and switching back and forth between branches generally just as **fast**.
- Recall: Git doesn’t store data as a series of changesets or deltas, but instead as a series of **snapshots**

Git branching

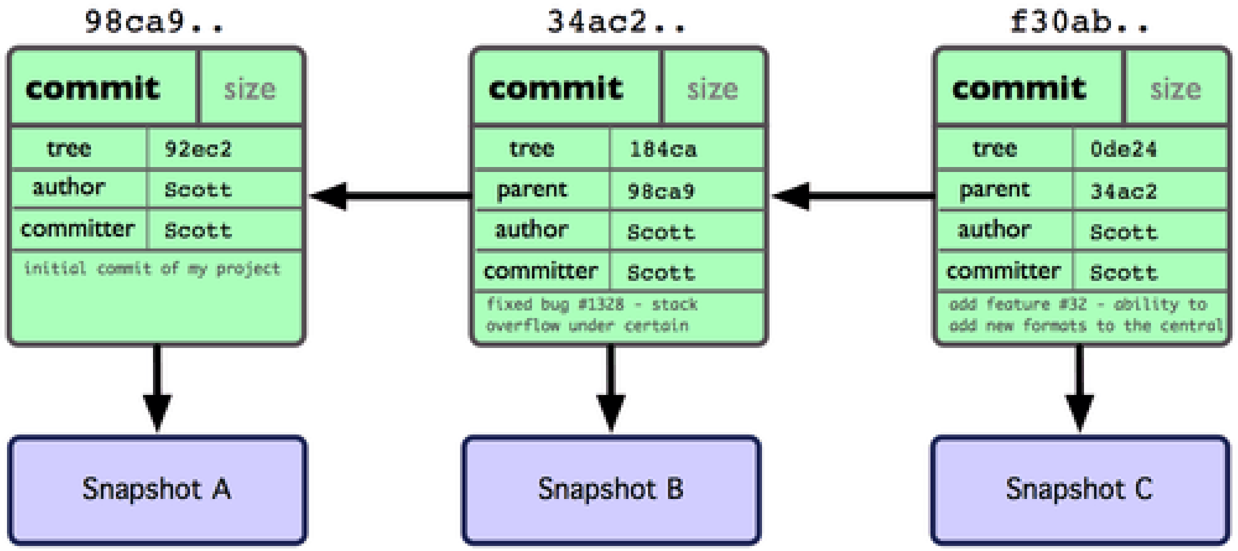
- Let’s suppose to stage and commit three files

```
$ git add README HelloWorld.cs LICENSE
$ git commit -m 'initial commit of my project'
```



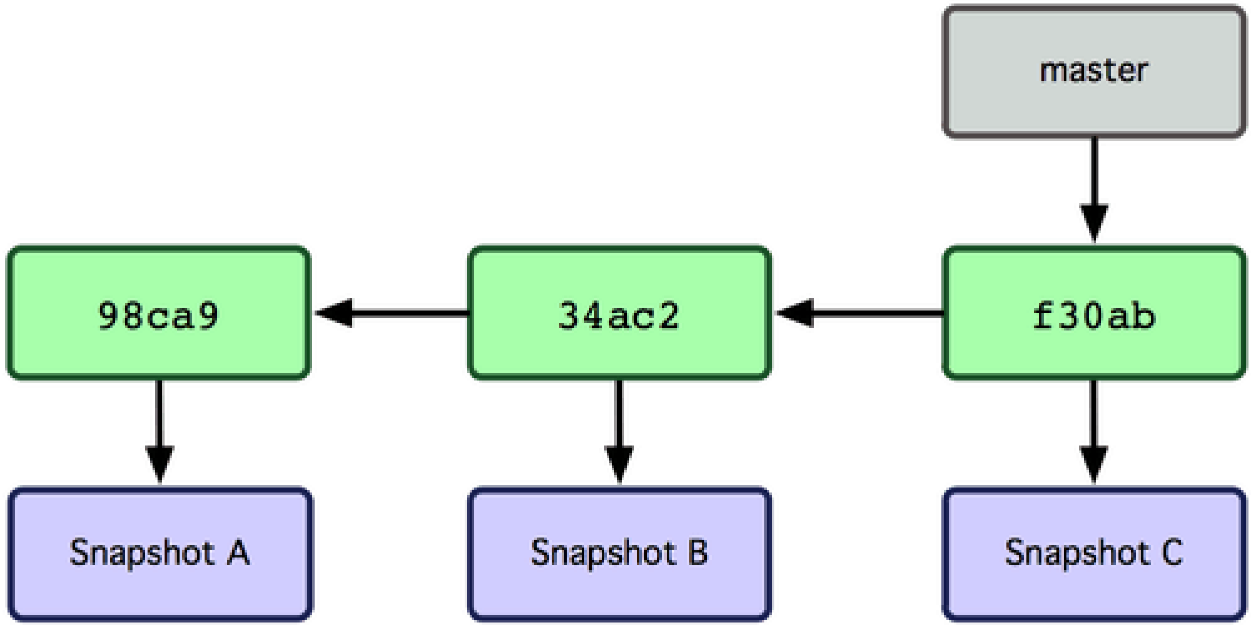
Git branching

- After some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.
- After two more commits, your history might look something like



Git branching

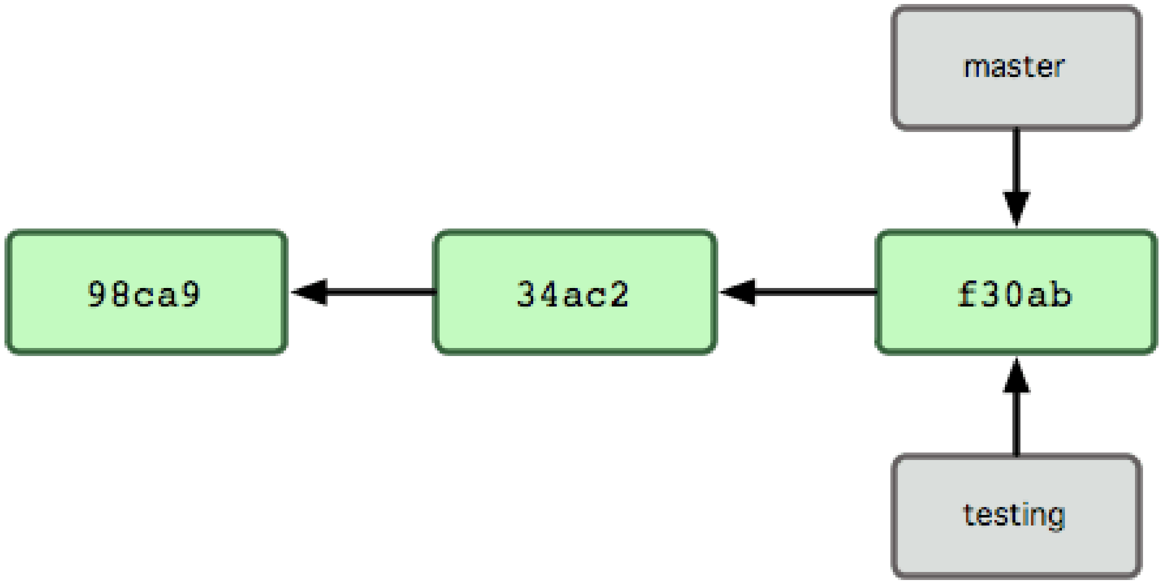
- A branch in Git is simply a lightweight movable pointer to one of these commits
- The default branch name in Git is master



Git branching

- What happens if you create a new branch?

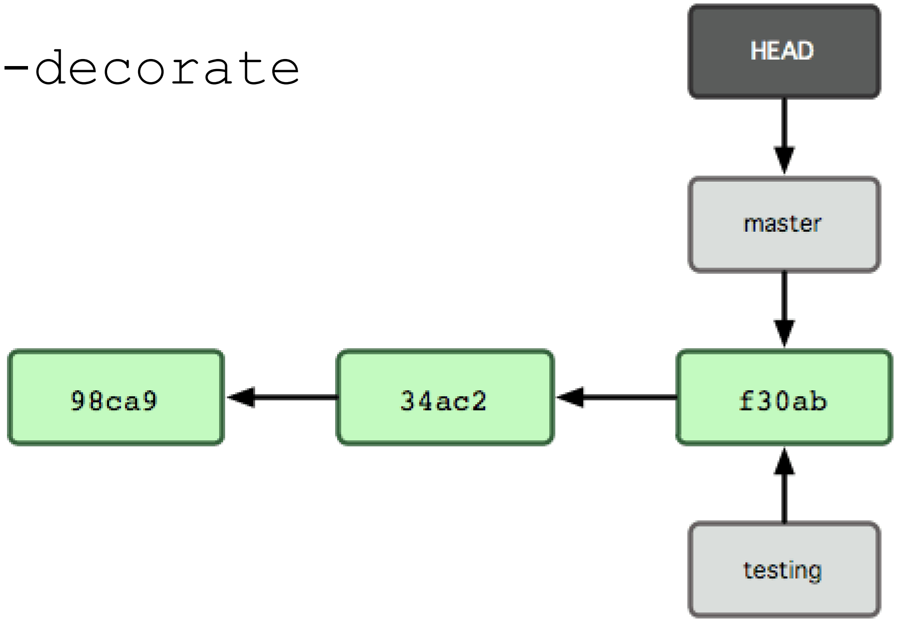
```
$ git branch testing
```



Git branching

- How does Git know what branch you're currently on? It keeps a special pointer called **HEAD**.
- This is a pointer to the local branch you're currently on. In this case, you're still on master.
- The git branch command only created a new branch — it didn't switch to that branch.

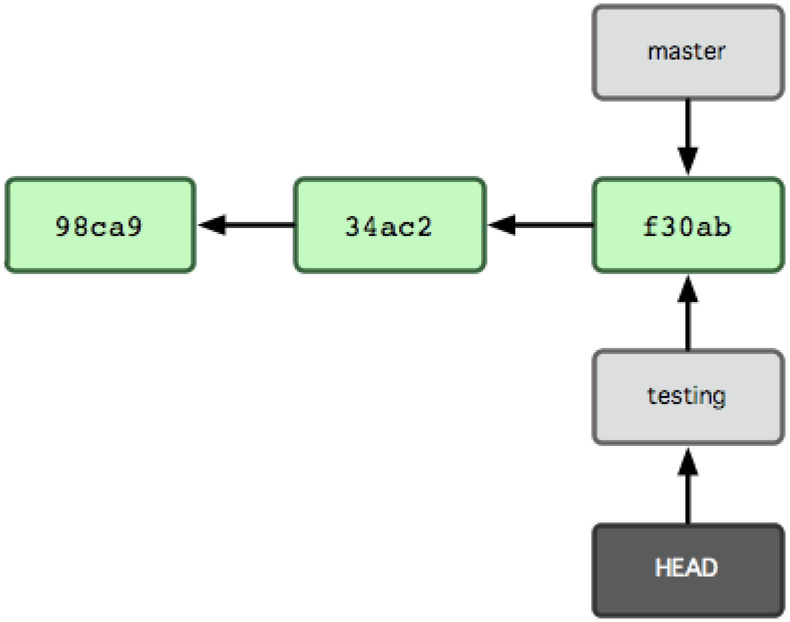
```
$ git log --oneline --decorate
```



Git branching

- To switch to an existing branch, you run the git checkout command.

```
$ git checkout testing
```

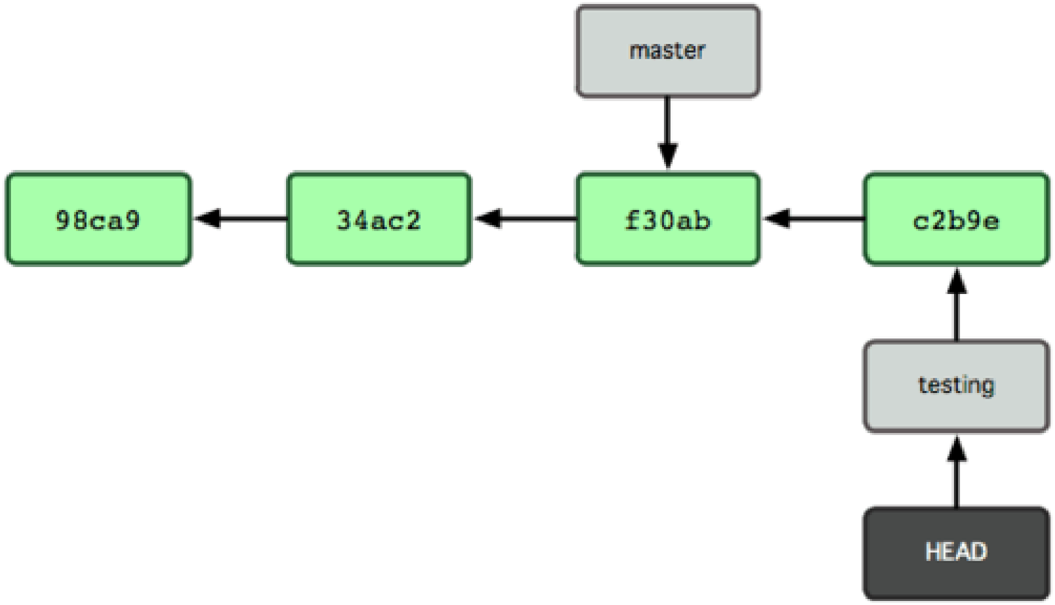


Git branching

Modify one of the committed files

- Let’s do another commit

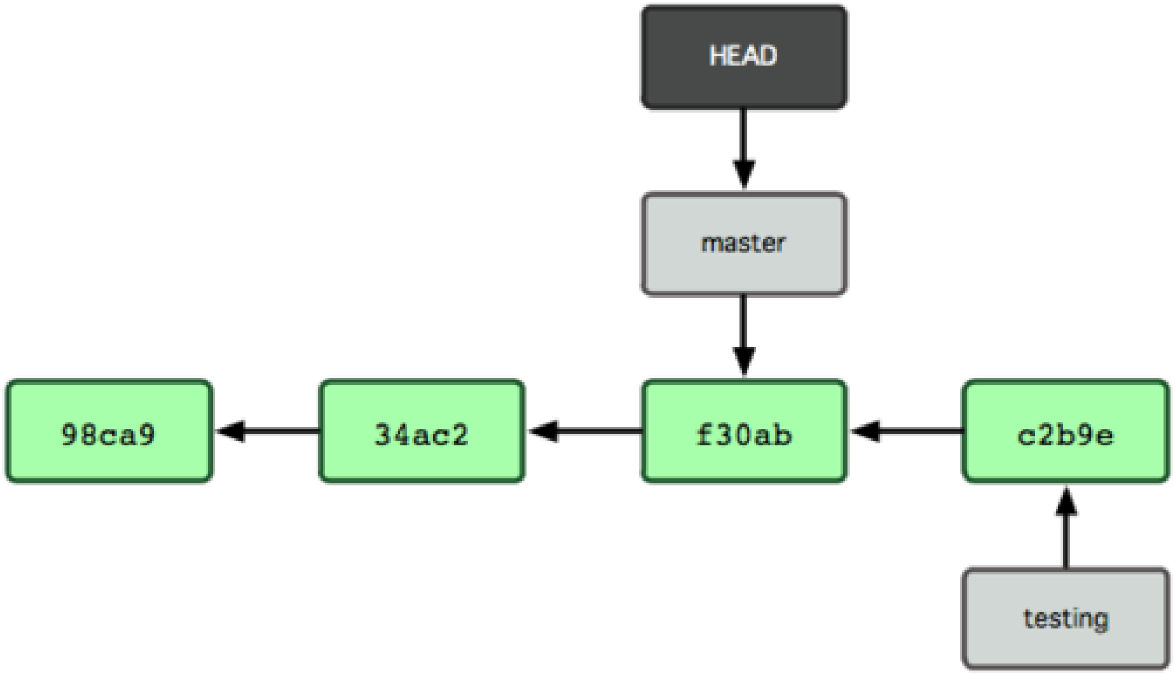
```
$ vim test.rb
$ git commit -a -m 'made a change'
```



Git branching

- Let’s switch back to the master branch

```
$ git checkout master
```



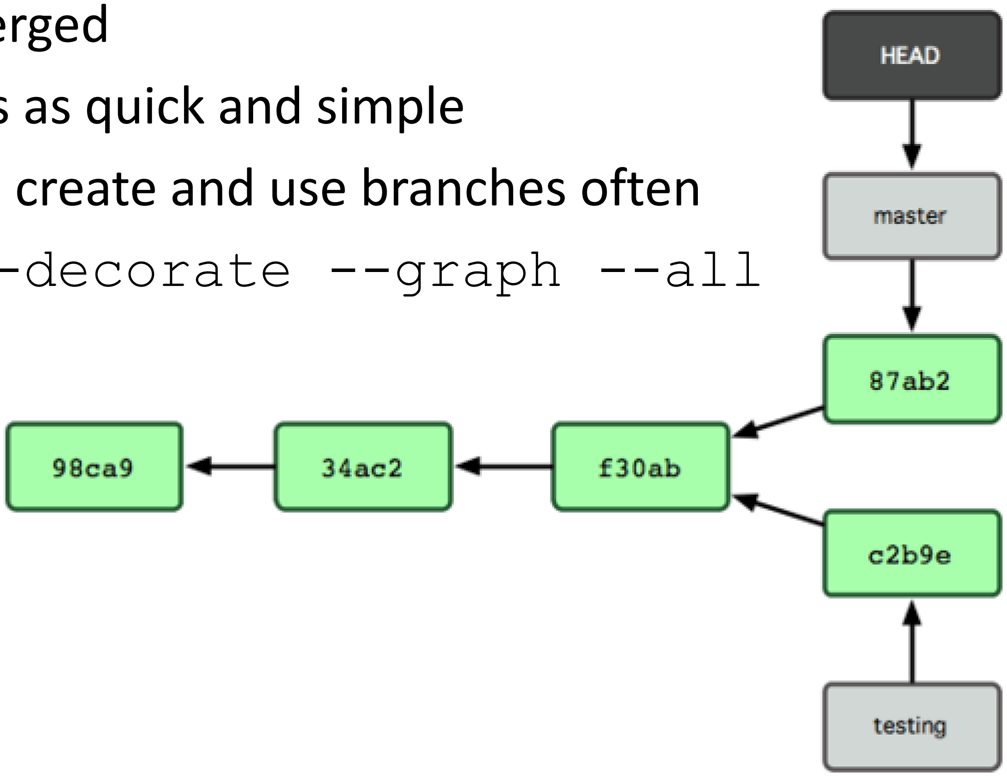
Git branching

- Let’s make a few changes and commit again

```
$ vim test.rb
$ git commit -a -m 'made other changes`
```

- The branch histories have diverged
- Creating a new branch in Git is as quick and simple
- Developers are encouraged to create and use branches often

```
$ git log --oneline --decorate --graph --all
```

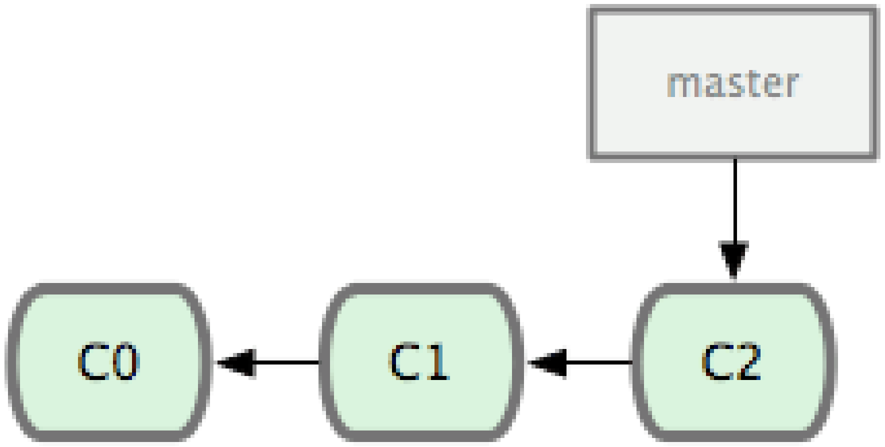


Basic branching and merging

- Let’s see a simple example of branching and merging with a workflow that you might use in the real world:
  - Do work on a web site.
  - Create a branch for a new story you’re working on.
  - Do some work in that branch.
- At this stage, you’ll receive a call that another issue is critical and you need a hotfix:
  - Switch back to your production branch.
  - Create a branch to add the hotfix.
  - After it’s tested, merge the hotfix branch, and push to production.
  - Switch back to your original story and continue working.

Basic branching

- Working on a project with 2 commits already
- Decide to work on issue #53 in the issue tracking system



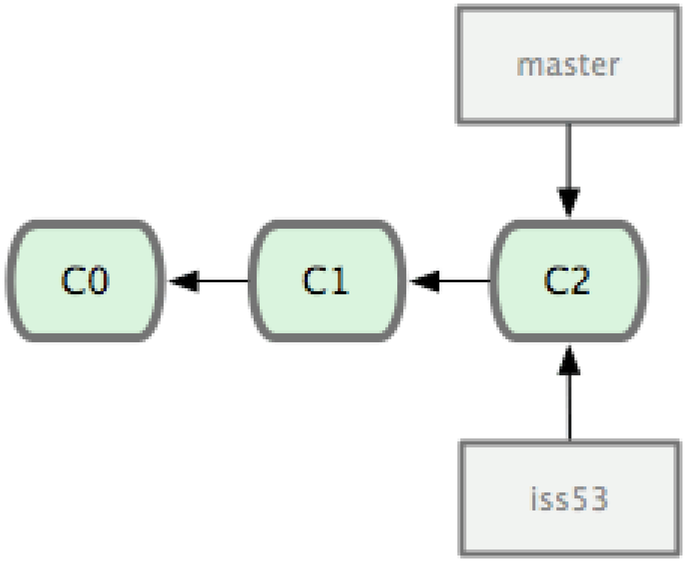
Basic branching

- To create a branch and switch to it at the same time, you can run the git checkout command with the -b switch:

```
$ git checkout -b iss53
Switched to a new branch 'iss53'
```

- This is shorthand for:

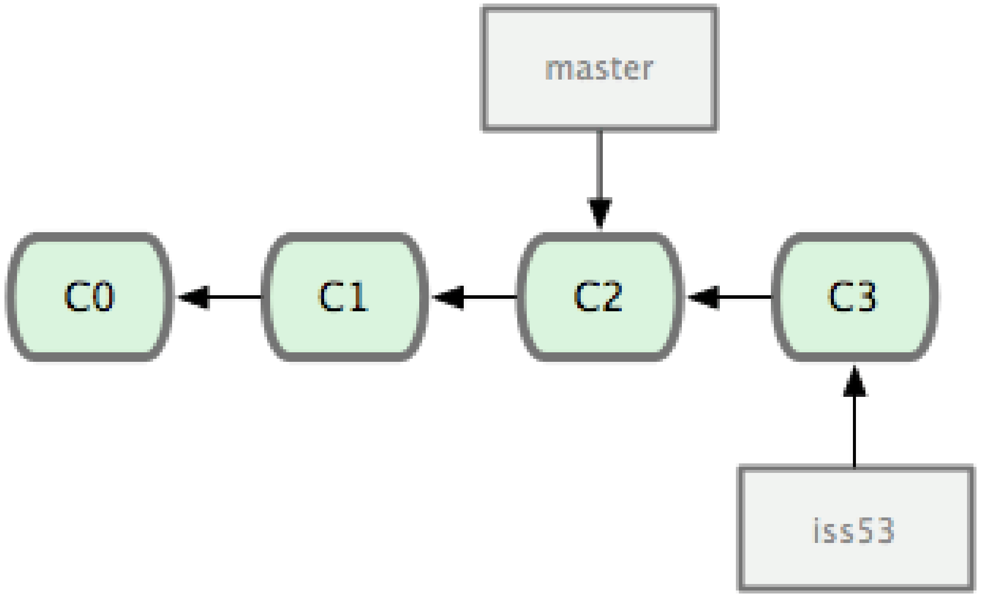
```
$ git branch iss53
$ git checkout iss53
```



Basic branching

- You work on your project and do some commits. Doing so moves the iss53 branch forward, because you have it checked out (that is, your HEAD is pointing to it):
- Try to modify HelloWorld.cs and then commit

```
$ git commit -a -m 'added a new class[issue 53]'
```



Basic branching

- Now you get the call that there is an issue with the project, and you need to fix it immediately.
- All you have to do is switch back to your master branch.
- However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you’re checking out, Git won’t let you switch branches.
- It’s best to have a clean working state when you switch branches.
- There are ways to get around this (namely, stashing and commit amending) that we’ll cover later.
- For now, you’ve committed all your changes, so you can switch back to your master branch:

```
$ git checkout master
Switched to branch 'master'
```

Basic branching

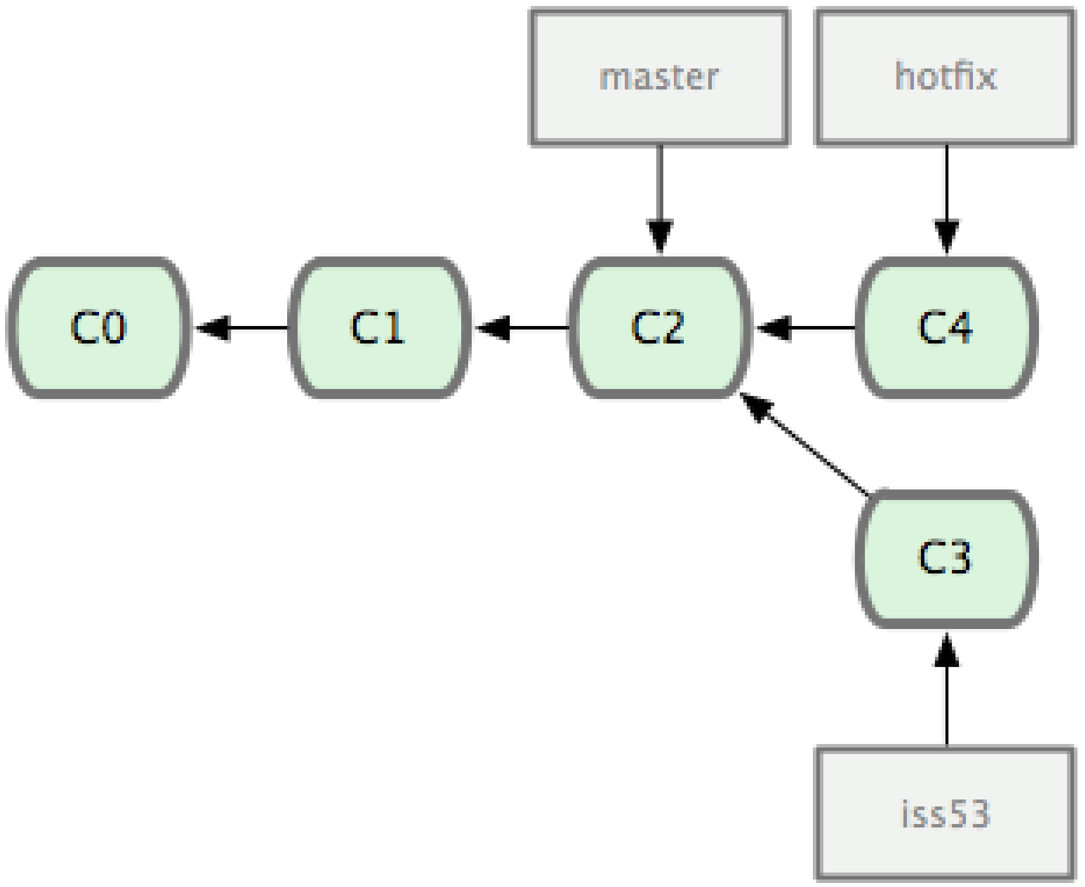
- Next, you have a hotfix to make. Let’s create a hotfix branch on which to work until it’s completed:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
```

- Modify again HelloWorld.cs

```
$ git commit -a -m 'fixed the divide by zero
bug'
[hotfix 3a0874c] fixed the divide by zero bug
 1 files changed, 1 deletion(-)
```

Basic branching

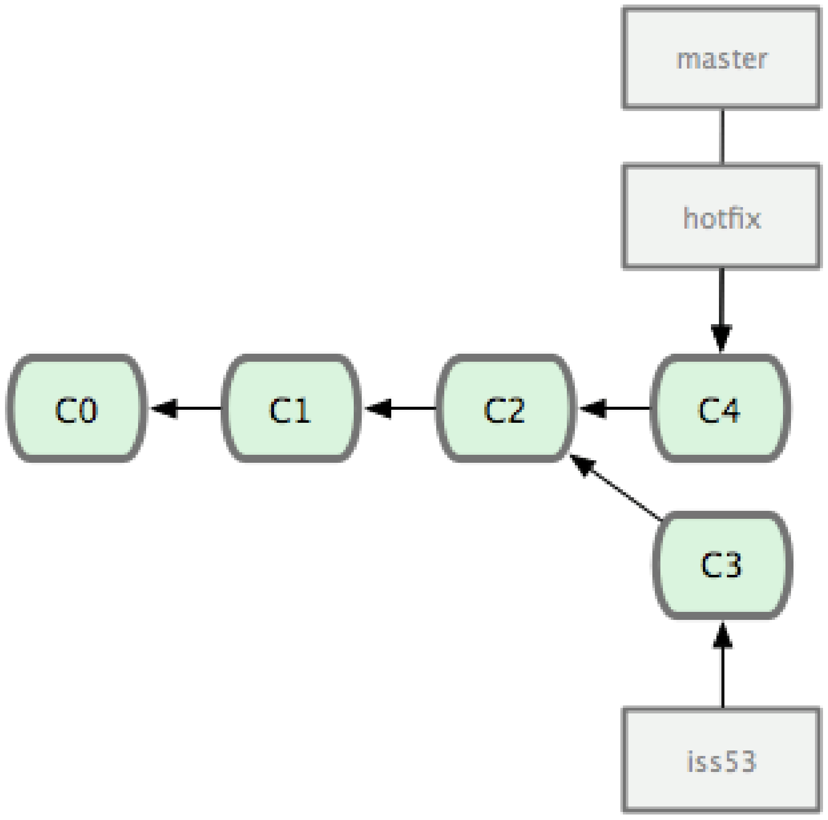


Basic branching

- You can run your tests, make sure the hotfix is what you want, and merge it back into your master branch to deploy to production. You do this with the git merge command:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```

Basic branching



Basic branching

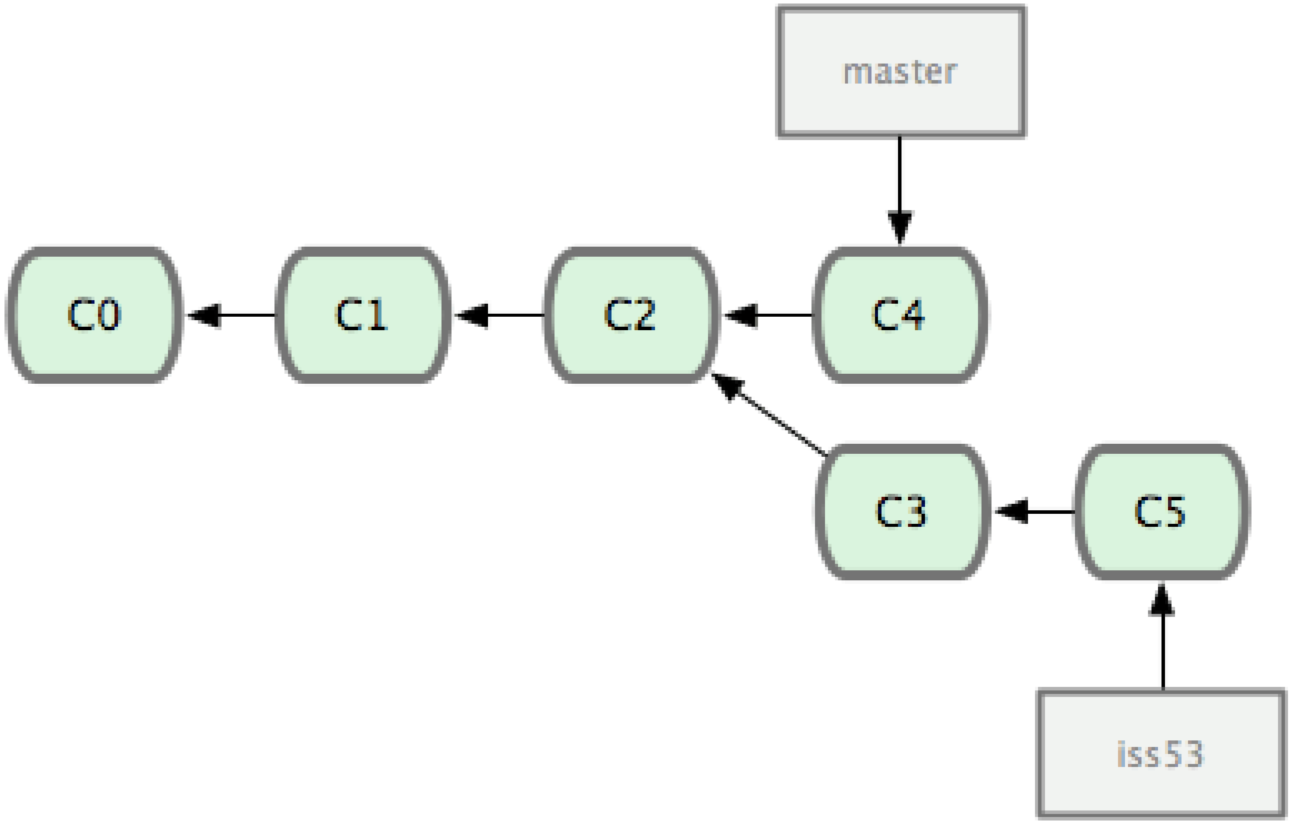
- After your super-important fix is deployed, you're ready to switch back to the work you were doing before you were interrupted.
- However, first you'll delete the hotfix branch, because you no longer need it — the master branch points at the same place. You can delete it with the -d option to git branch:

```
$ git branch -d hotfix
Deleted branch hotfix (was 3a0874c).
```

- Now you can switch back to your work-in-progress branch on issue #53 and continue working on it (see Figure 3-15):

```
$ git checkout iss53
Switched to branch 'iss53'
$ vim index.html
$ git commit -a -m 'finished the new class[issue 53]'
[iss53 ad82d7a] finished the new class [issue 53]
 1 file changed, 1 insertion(+)
```

Basic branching



Basic branching

- It’s worth noting here that the work you did in your hotfix branch is not contained in the files in your iss53 branch.
- If you need to pull it in, you can merge your master branch into your iss53 branch by running git merge master, or you can wait to integrate those changes until you decide to pull the iss53 branch back into master later.

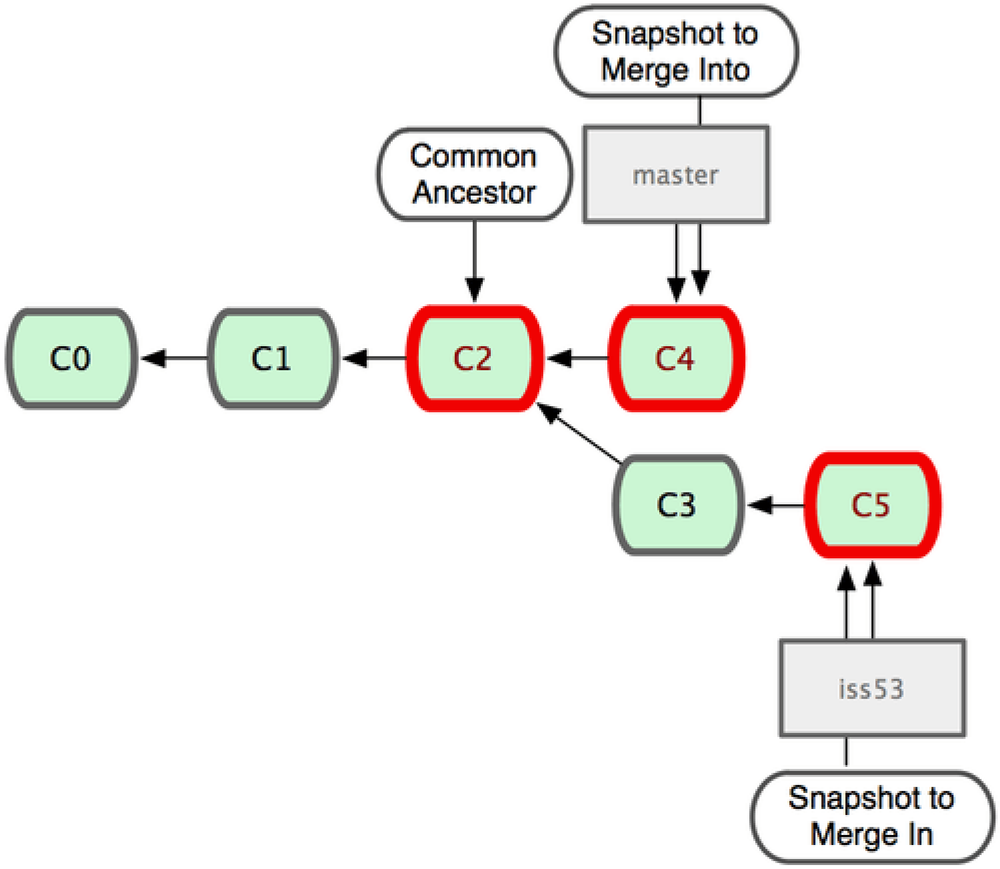
Basic merging

- Suppose you’ve decided that your issue #53 work is complete and ready to be merged into your master branch. In order to do that, you’ll merge in your iss53 branch, much like you merged in your hotfix branch earlier. All you have to do is check out the branch you wish to merge into and then run the git merge command:

```
$ git checkout master
$ git merge iss53
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

Basic merging

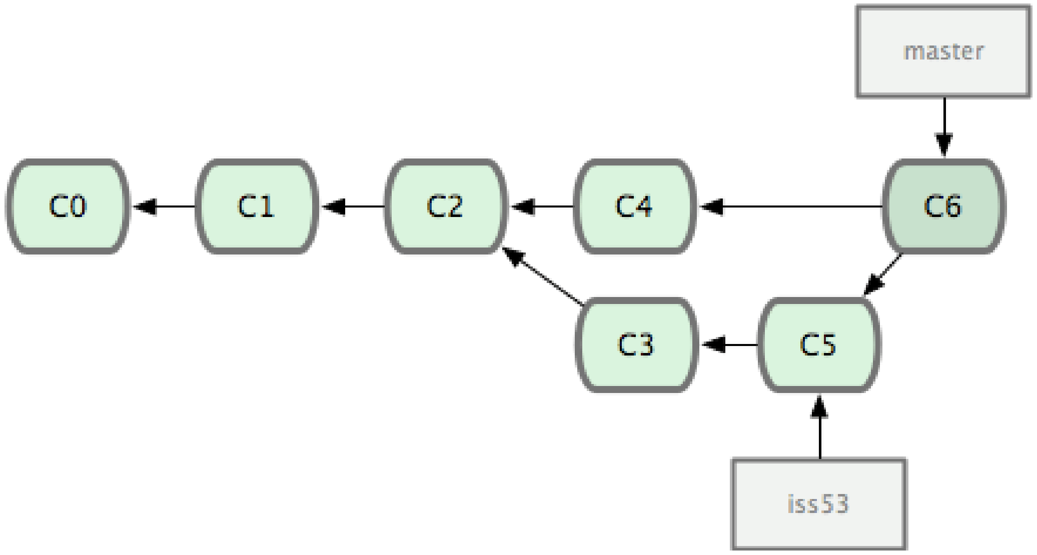
- Different than the hotfix merge you did earlier.
- In this case, your development history has diverged from some older point.



Basic merging

- Git automatically creates a new commit object that contains the merged work.
- Now that your work is merged in, you have no further need for the iss53 branch. You can delete it and then manually close the ticket in your ticket-tracking system:

```
$ git branch -d iss53
```



Basic merge conflicts

- Occasionally, this process doesn’t go smoothly.
- If you changed the same part of the same file differently in the two branches you’re merging together, Git won’t be able to merge them cleanly.
- If your fix for issue #53 modified the same part of a file as the hotfix, you’ll get a merge conflict that looks something like this:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then
commit the result.
```

Basic merge conflicts

- Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

        both modified:      index.html

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

Basic merge conflicts

- Anything that has merge conflicts and hasn’t been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
<<<<<<< HEAD
<div id="footer">contact :
email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53
```

Basic merge conflicts

- In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself.
- For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

- This resolution has a little of each section, and I've fully removed the <<<<<<<, =======, and >>>>>>> lines.
- After you've resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved.
- Staging the file marks it as resolved in Git.

Basic merge conflicts

- You can run git status again to verify that all conflicts have been resolved:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html
```

- If you’re happy with that, and you verify that everything that had conflicts has been staged, you can type git commit to finalize the merge commit.

Branch management

- The git branch command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

```
$ git branch
  iss53
* master
  testing
```

- Notice the * character that prefixes the master branch: it indicates the branch that you currently have checked out.
- This means that if you commit at this point, the master branch will be moved forward with your new work.

Branch management

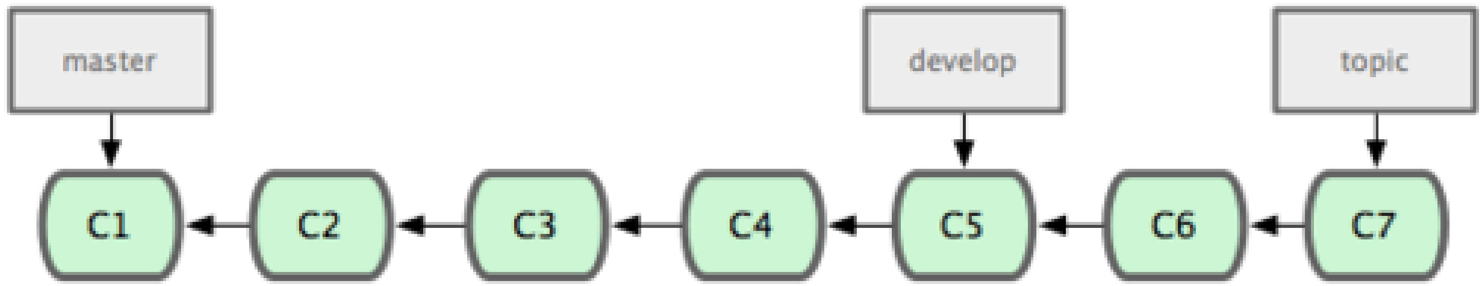
- To see the last commit on each branch, you can run git branch -v:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list
in the readmes
```

- The useful `--merged` and `--no-merged` options can filter this list to branches that you have or have not yet merged into the branch you're currently on.
- To see all the branches that contain work you haven't yet merged in, you can run git branch `--no-merged`. Trying to delete one of these with `git branch -d` will fail:

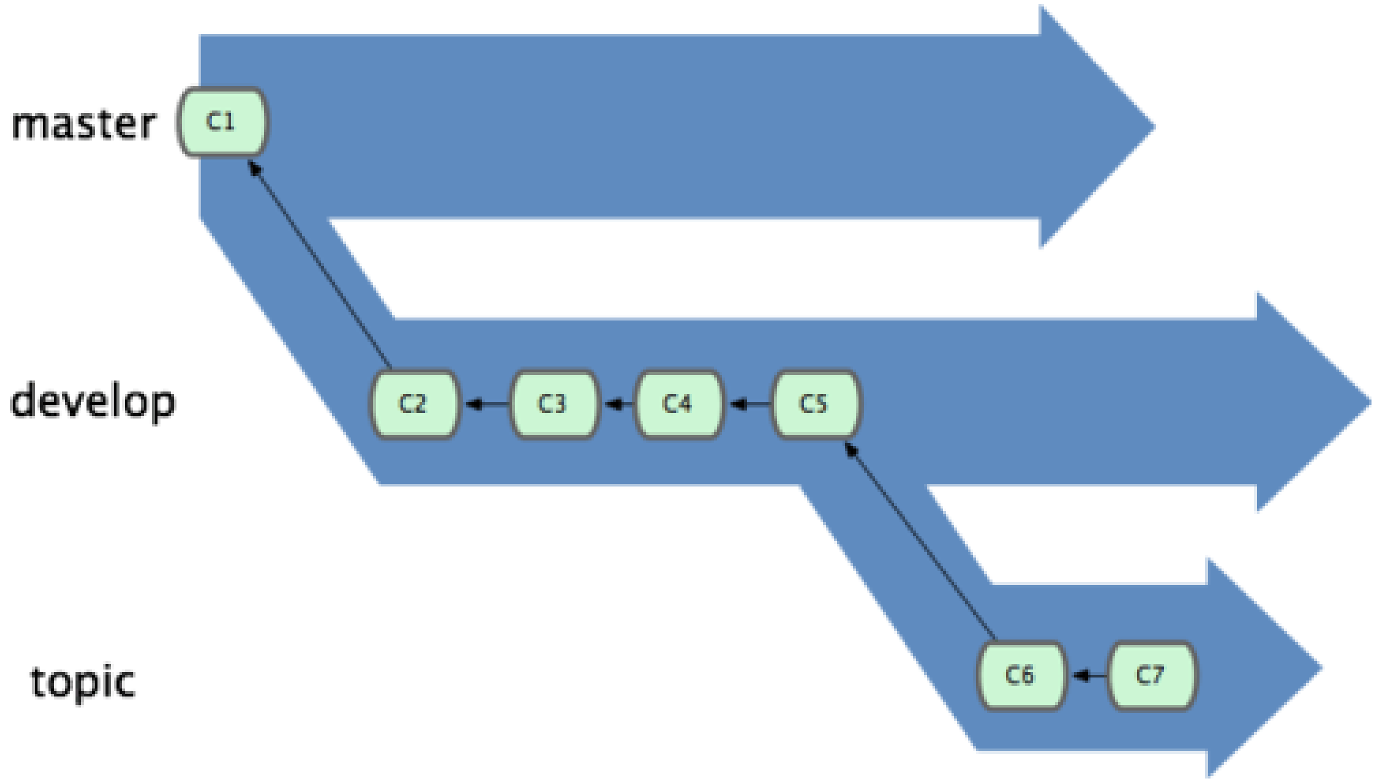
Branching workflows

- Long running branches:
  - several branches are always open and used for different stages of the development cycle
  - merge regularly from some of them into others
  - only code that is entirely stable in the **`master`** branch
  - another parallel branch named **`develop`** or **`next`** is used to develop and test stability: it isn't necessarily always stable, but whenever it gets to a stable state, it can be **merged into master**



Branching workflows

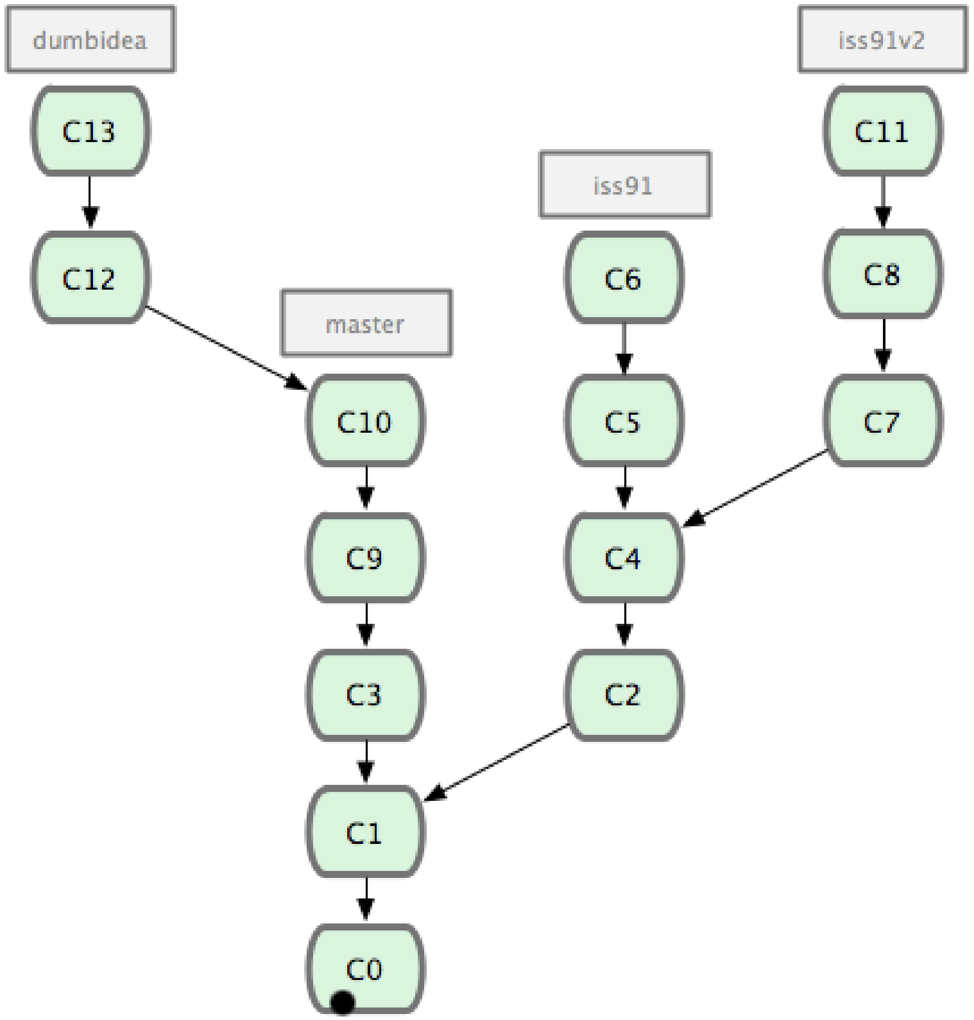
- It's generally easier to think about branches as work silos, where sets of commits graduate to a more stable silo when they're fully tested



Branching workflows

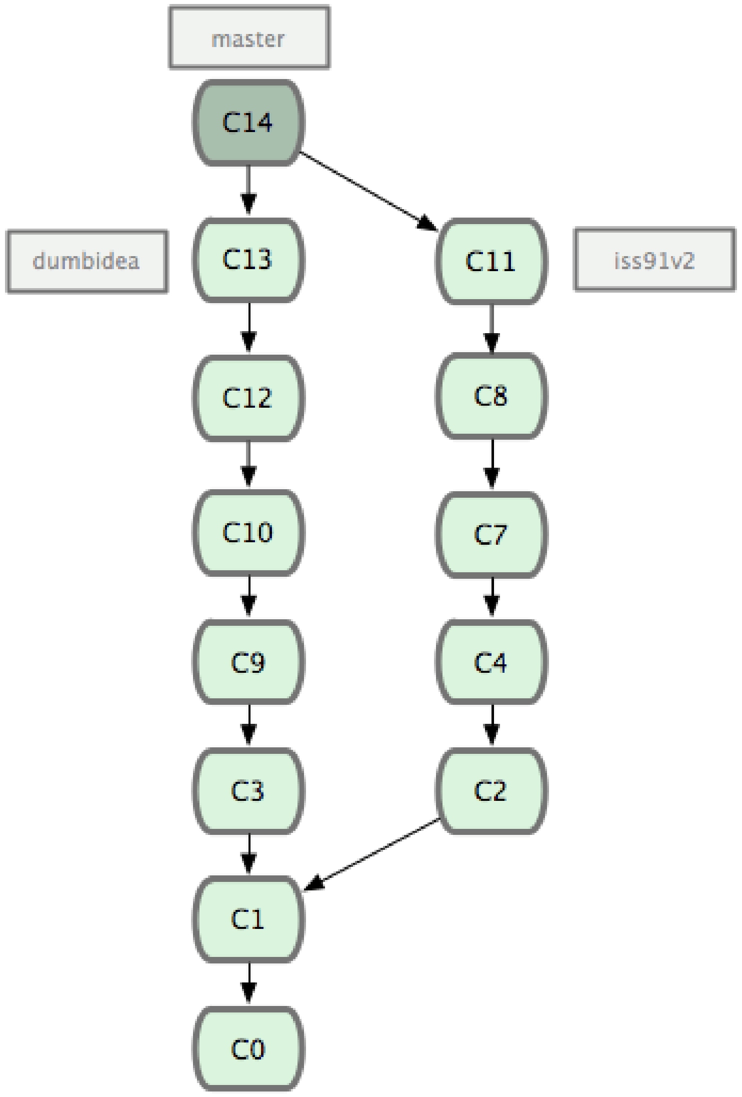
- Topic Branches
  - A topic branch is a short-lived branch that you create and use for a single particular feature or related work.
  - In Git it's common to create, work on, merge, and delete branches several times a day.
  - Consider an example of doing some work (on master), branching off for an issue (iss91), working on it for a bit, branching off the second branch to try another way of handling the same thing (iss91v2), going back to your master branch and working there for a while, and then branching off there to do some work that you're not sure is a good idea (dumbidea branch). Your commit history will look something like the following figure

Branching workflows



Branching workflows

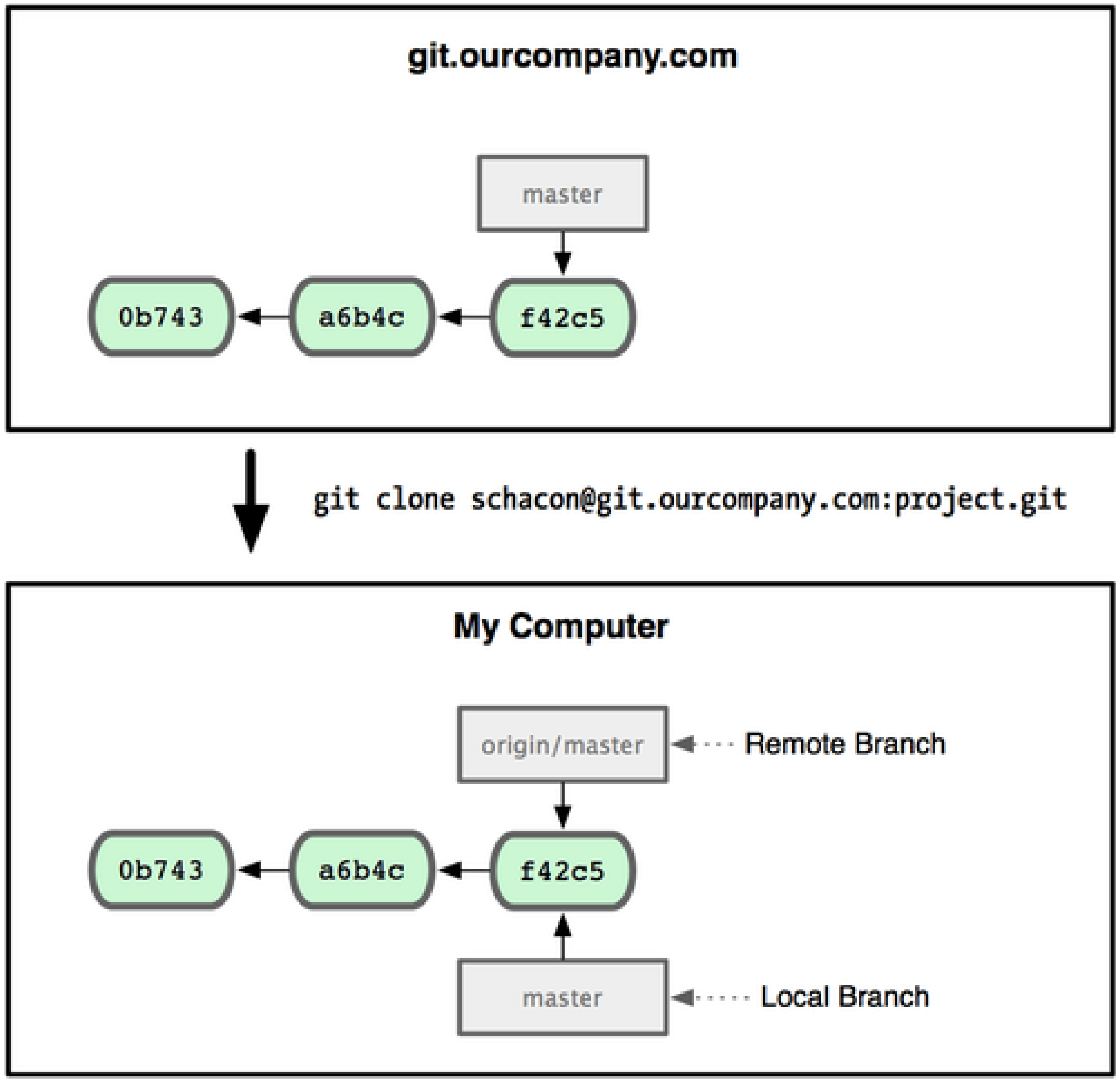
- Now, let’s say you decide you like the second solution to your issue best (iss91v2)
- And you showed the dumbidea branch to your coworkers, and it turns out to be genius.
- You can throw away the original iss91 branch (losing commits C5 and C6) and merge in the other two. Your history then looks like



Remote branches

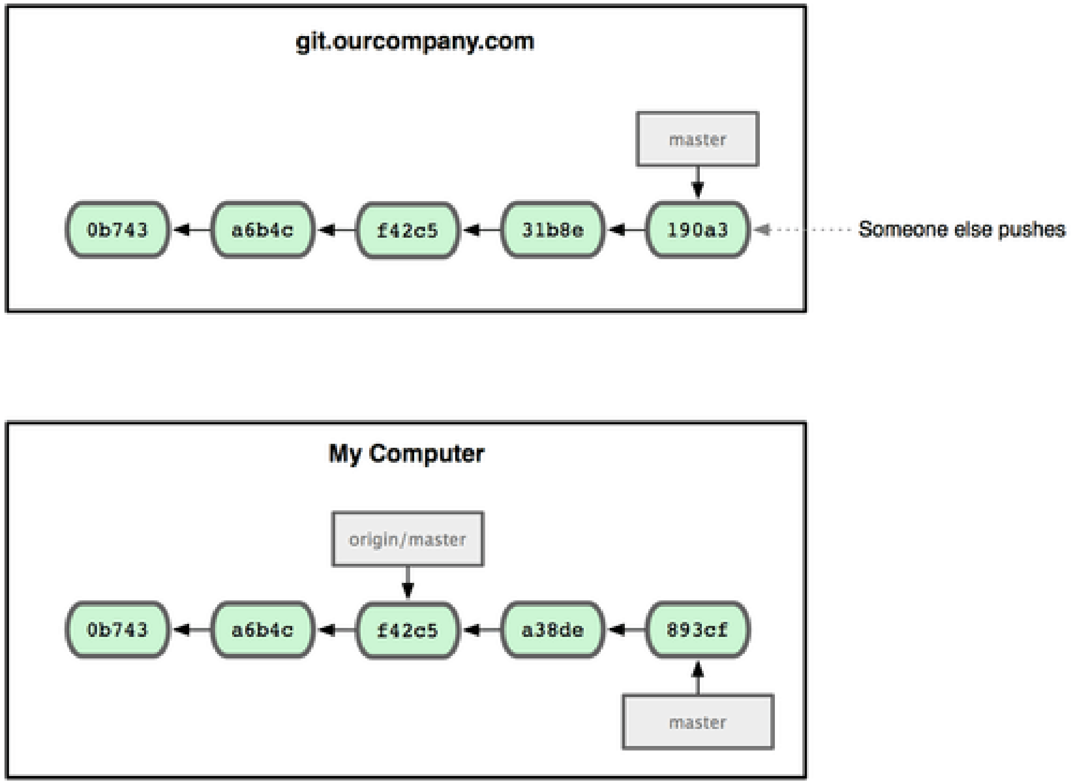
- Remote branches are references to the state of branches on your remote repositories.
- They’re local branches that you can’t move; they’re moved automatically whenever you do any network communication.
- Remote branches act as bookmarks to remind you where the branches on your remote repositories were the last time you connected to them.
- They take the form (remote)/(branch). For instance, if you wanted to see what the master branch on your origin remote looked like as of the last time you communicated with it, you would check the origin/master branch.
- If you were working on an issue with a partner and they pushed up an iss53 branch, you might have your own local iss53 branch; but the branch on the server would point to the commit at origin/iss53.

Remote branches



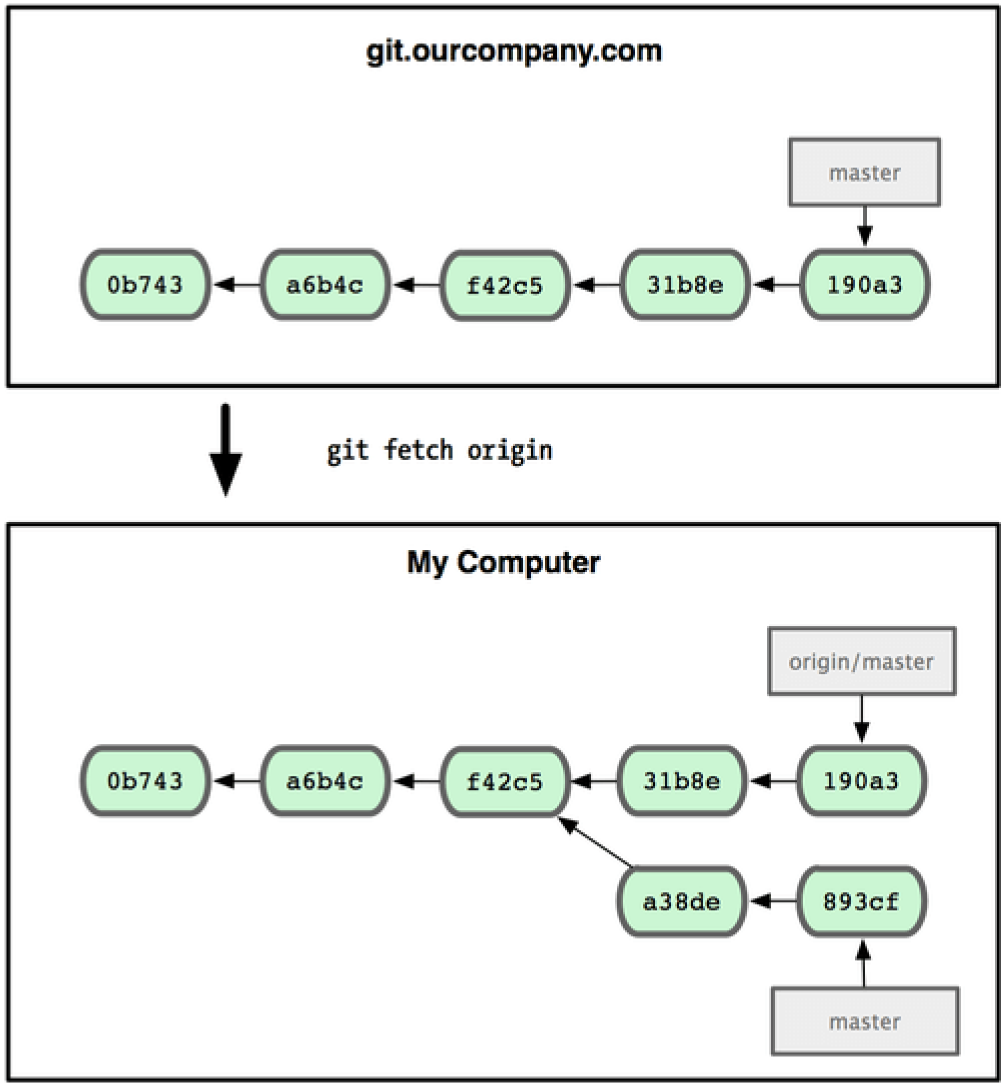
Remote branches

- If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently.
- Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn't move



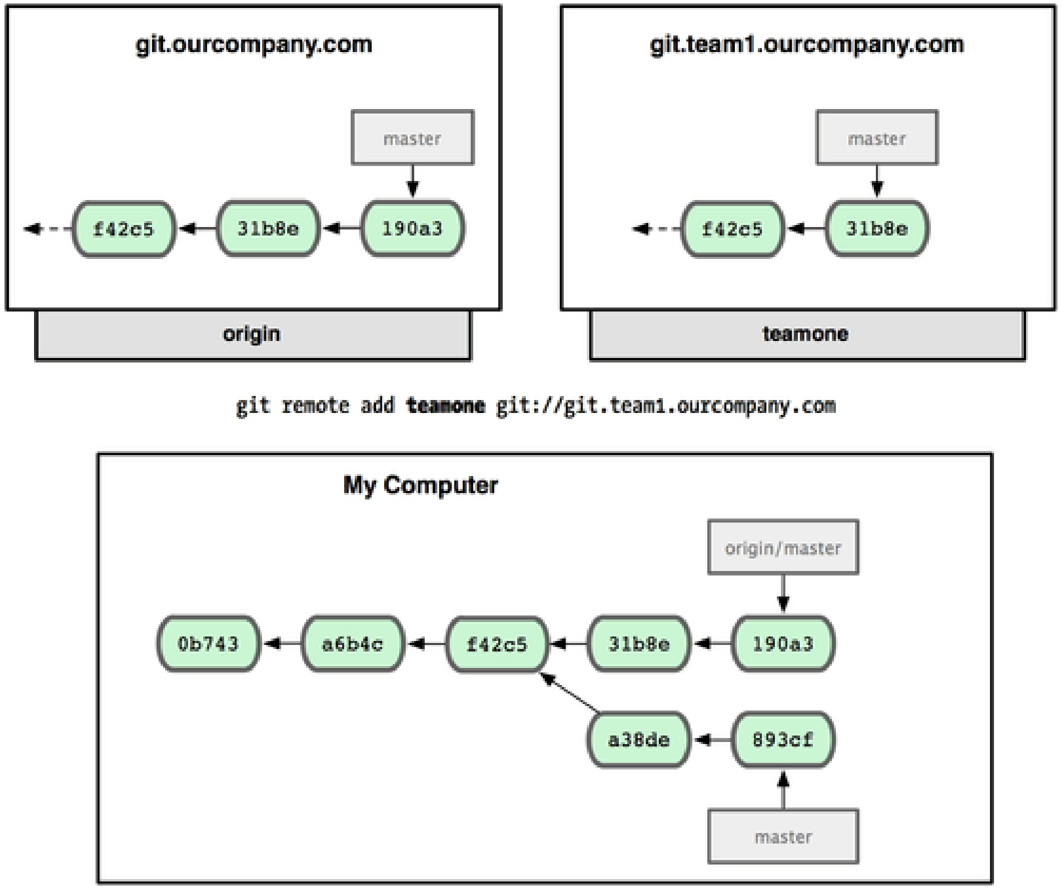
Remote branches

- To synchronize your work, you run a `git fetch origin` command.
- This command looks up which server origin is (in this case, it's git.ourcompany.com), fetches any data from it that you don't yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position



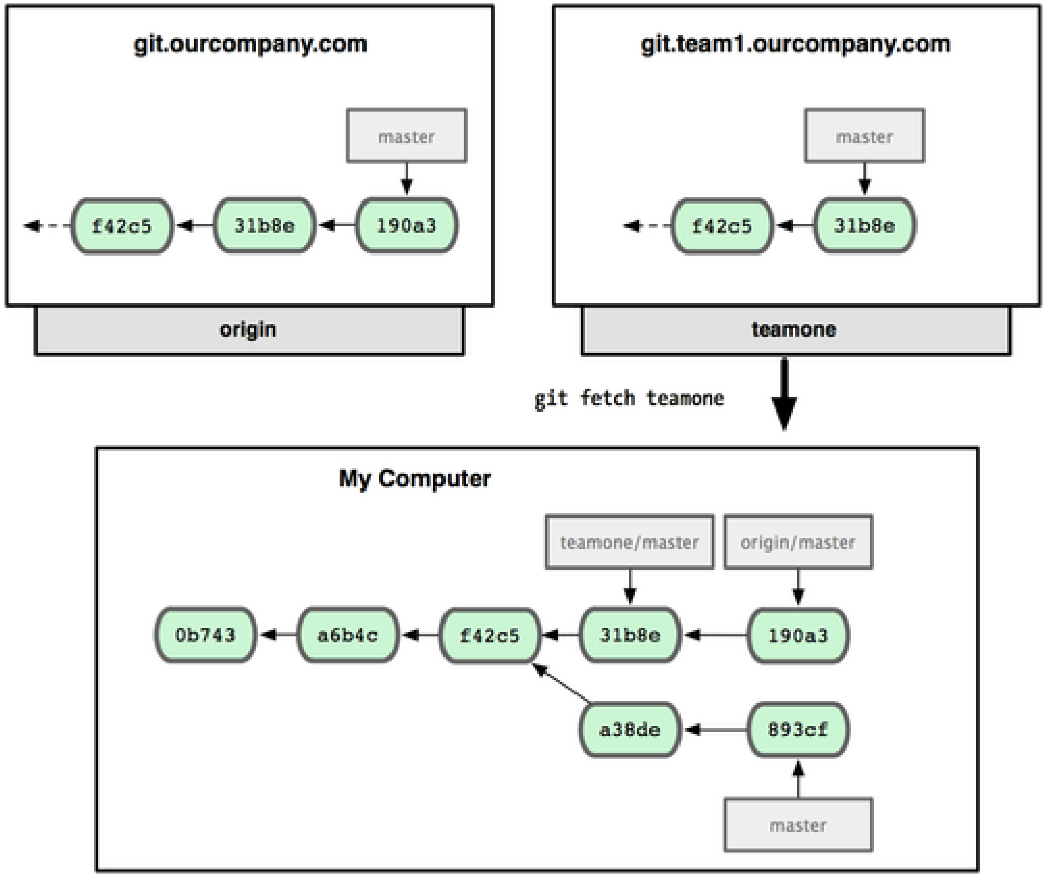
Remote branches

- To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let’s assume you have another internal Git server that is used only for development by one of your sprint teams.
- This server is at git.team1.ourcompany.com



Remote branches

- Now, you can run git fetch teamone to fetch everything the remote teamone server has that you don't have yet.
- Because that server has a subset of the data your origin server has right now, Git fetches no data but sets a remote branch called teamone/master to point to the commit that teamone has as its master branch



Pushing

- When you want to share a branch with the world, you need to push it up to a remote that you have write access to.
- Your local branches aren’t automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share.
- That way, you can use private branches for work you don’t want to share, and push up only the topic branches you want to collaborate on.

Pushing

- If you have a branch named `serverfix` that you want to work on with others, you can push it up the same way you pushed your first branch. Run `git push (remote) (branch)`:

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

Pushing

- It’s important to note that when you do a fetch that brings down new remote branches, you don’t automatically have local, editable copies of them.
- In other words, in this case, you don’t have a new `serverfix` branch — you only have an `origin/serverfix` pointer that you can’t modify.
- To merge this work into your current working branch, you can run `git merge origin/serverfix`. If you want your own serverfix branch that you can work on, you can base it off your remote branch:

```
$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch serverfix
from origin.

Switched to a new branch 'serverfix'
```

- This gives you a local branch that you can work on that starts where `origin/serverfix` is.

Credential helper

- Don’t type your password every time
- If you’re using an HTTPS URL to push over, the Git server will ask you for your username and password for authentication.
- By default it will prompt you on the terminal for this information so the server can tell if you’re allowed to push.
- If you don’t want to type it every single time you push, you can set up a “credential cache”.
- The simplest is just to keep it in memory for a few minutes (default 15 mins), which you can easily set up by running

```
git config --global credential.helper cache
```

Tracking branches

- Checking out a local branch from a remote branch automatically creates what is called a **tracking branch**.
- Tracking branches are local branches that have a direct relationship to a remote branch.
- If you're on a tracking branch and type `git push`, Git automatically knows which server and branch to push to.
- Also, running `git pull` while on one of these branches fetches all the remote references and then automatically merges in the corresponding remote branch.
- When you clone a repository, it generally automatically creates a `master` branch that tracks `origin/master`.
- You can also use the --track shorthand:

```
$ git checkout --track origin/serverfix
```

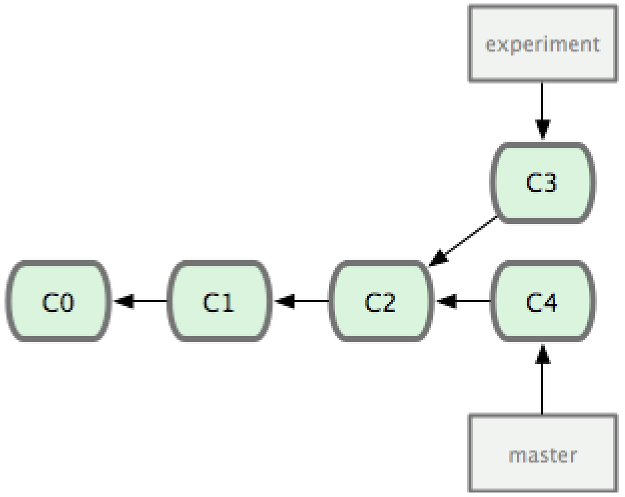
Deleting remote branches

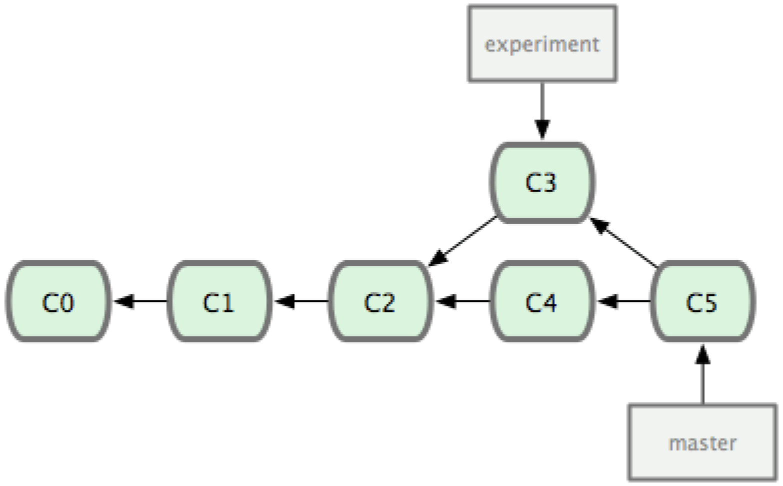
- Suppose you're done with a remote branch.
- You can delete a remote branch using `git push [remotename] :[branch]`. If you want to delete your serverfix branch from the server, you run the following:

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
 - [deleted]         serverfix
```

Rebasing

- Alternative to merge
- With merge C5 is the new commit





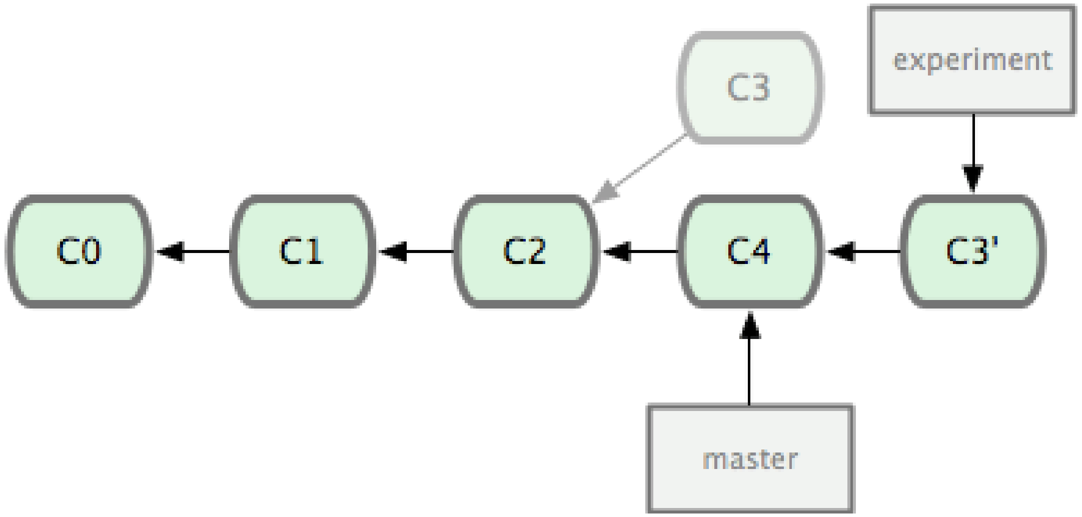
Rebasing

- The patch of the change that was introduced in C3 can be reapplied it on top of C4. In Git, this is called **rebasing**.
- With the rebase command, you can take all the changes that were committed on one branch and replay them on another one.

```
$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top
of it...

Applying: added staged command
```

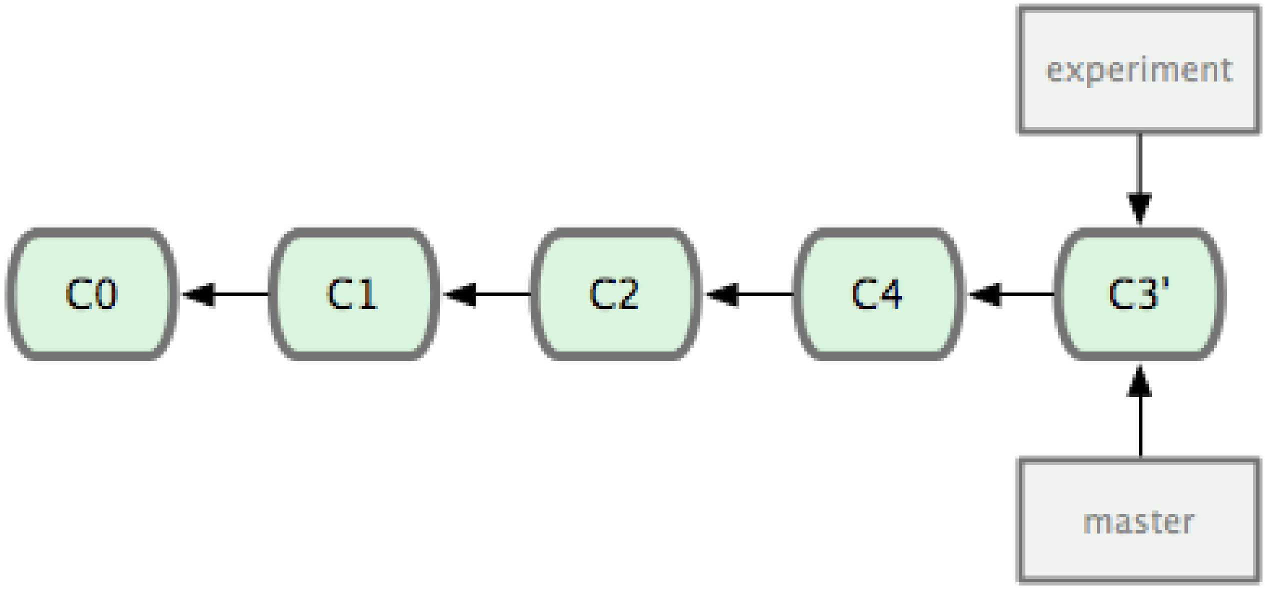
Rebasing

- It works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

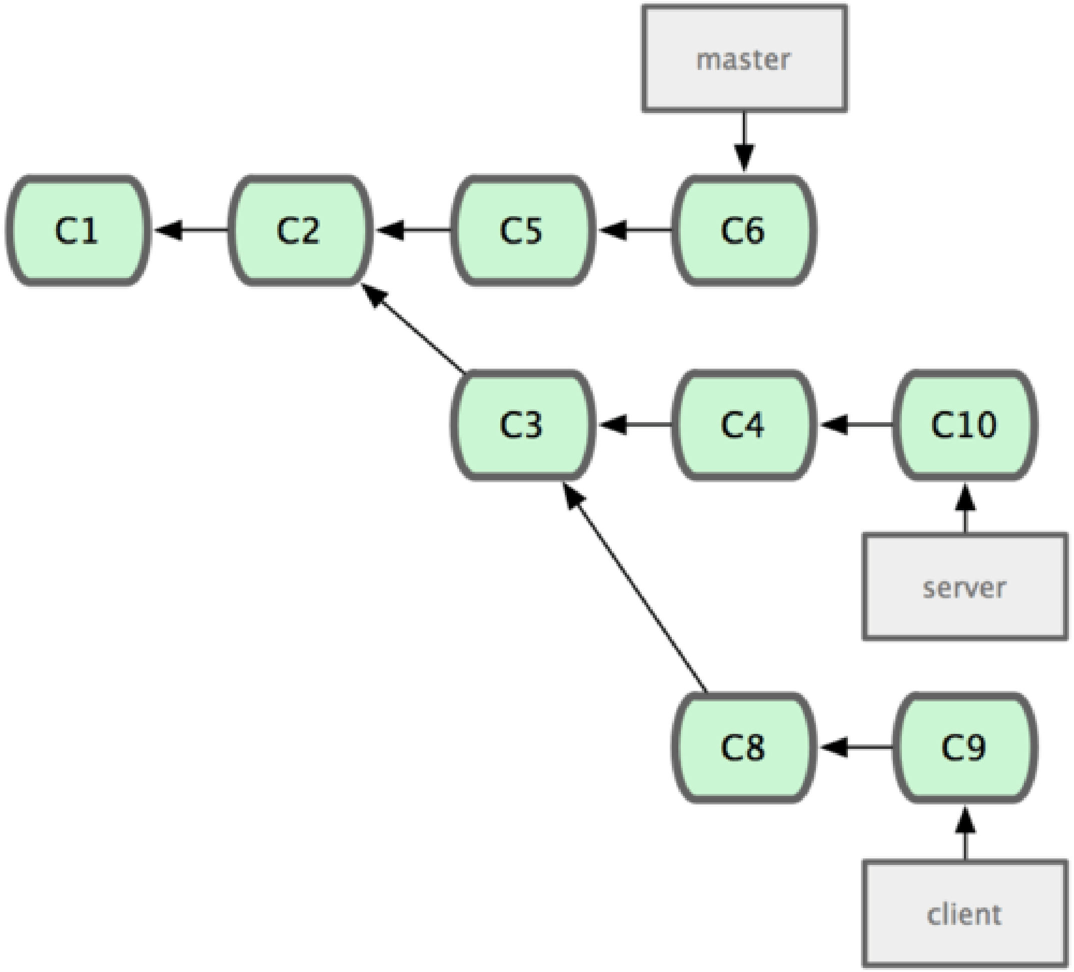


Rebasing

- At this point, you can go back to the master branch and do a fast-forward merge



More complex rebasing



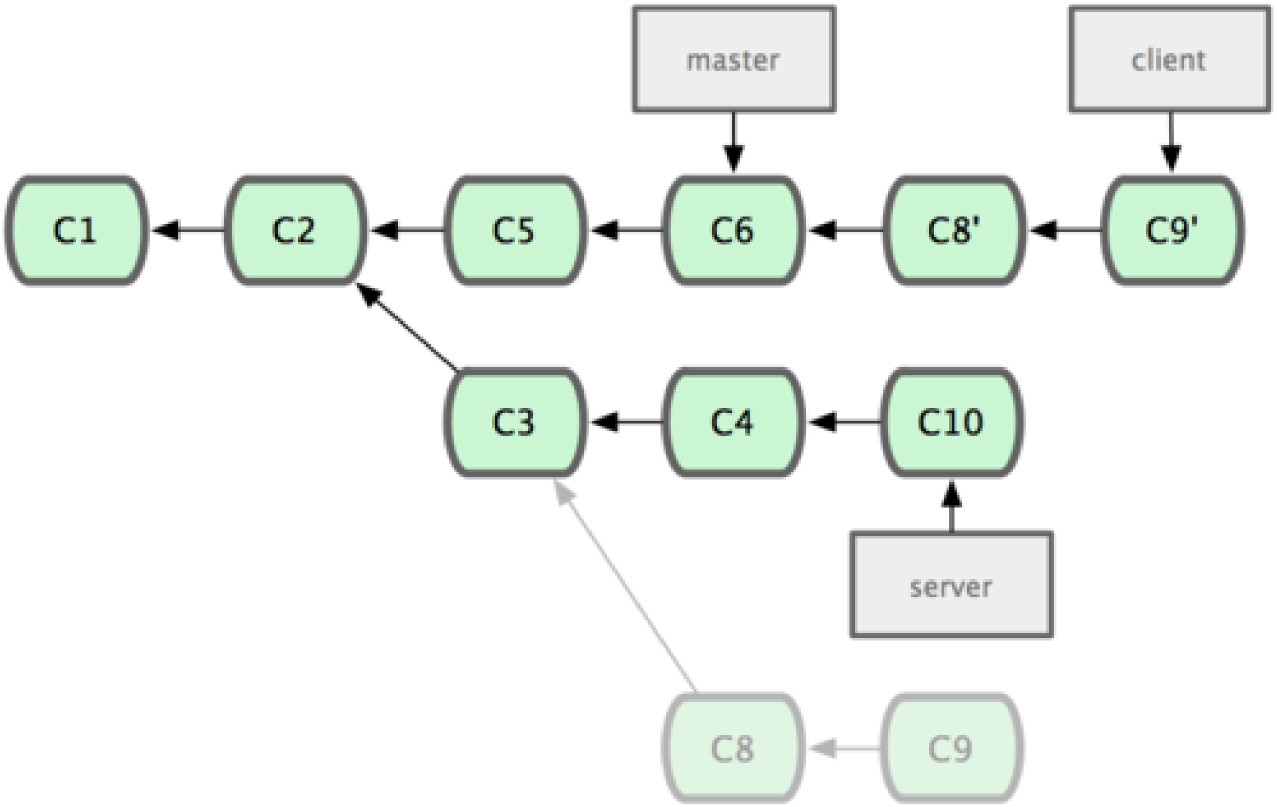
More complex rebasing

- Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it's tested further.
- You can take the changes on client that aren't on server (C8 and C9) and replay them on your master branch by using the --onto option of git rebase:

```
$ git rebase --onto master server client
```

- This basically says, "Check out the client branch, figure out the patches from the common ancestor of the `client` and `server` branches, and then replay them onto `master`." It's a bit complex, but the result is pretty cool.

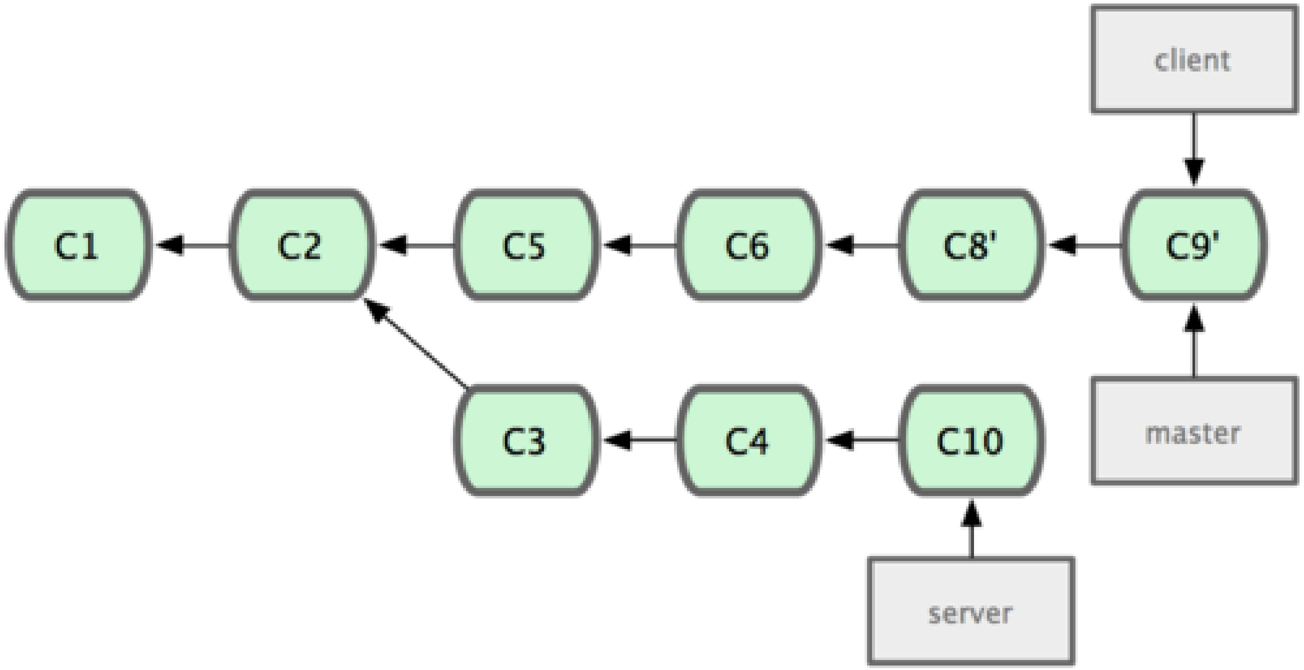
More complex rebasing



More complex rebasing

- Now you can fast-forward your master branch:

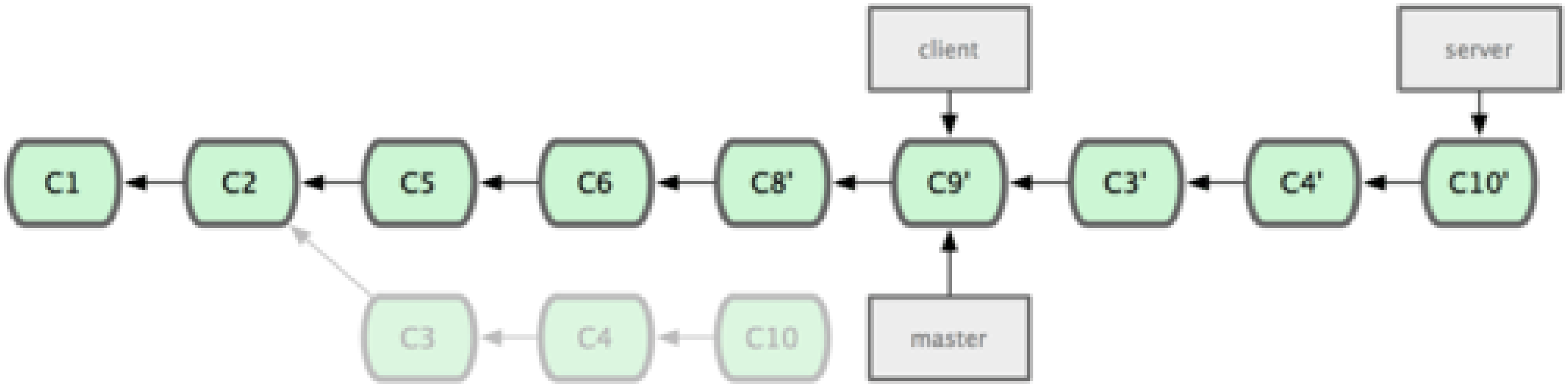
```
$ git checkout master
$ git merge client
```



More complex rebasing

- Let’s say you decide to pull in your server branch as well.
- You can rebase the server branch onto the master branch without having to check it out first by running `git rebase [basebranch] [topicbranch]`

```
$ git rebase master server
```



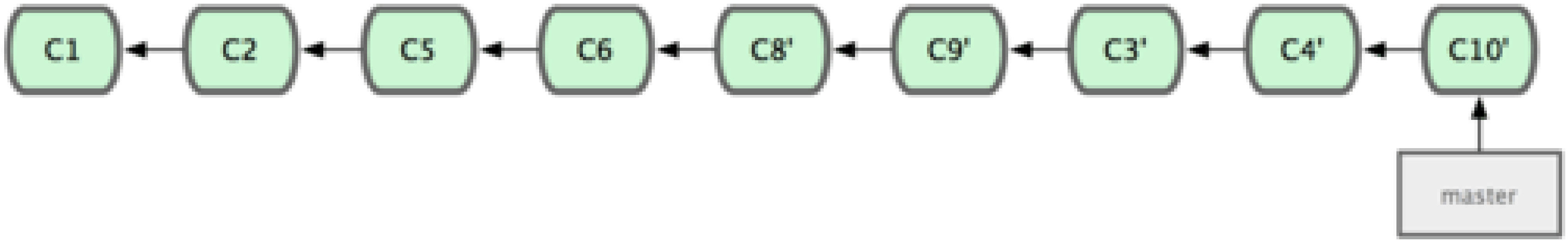
More complex rebasing

- Then, you can fast-forward the base branch (master):

```
$ git checkout master
$ git merge server
```

- You can remove the client and server branches because all the work is integrated and you don't need them anymore:

```
$ git branch -d client
$ git branch -d server
```

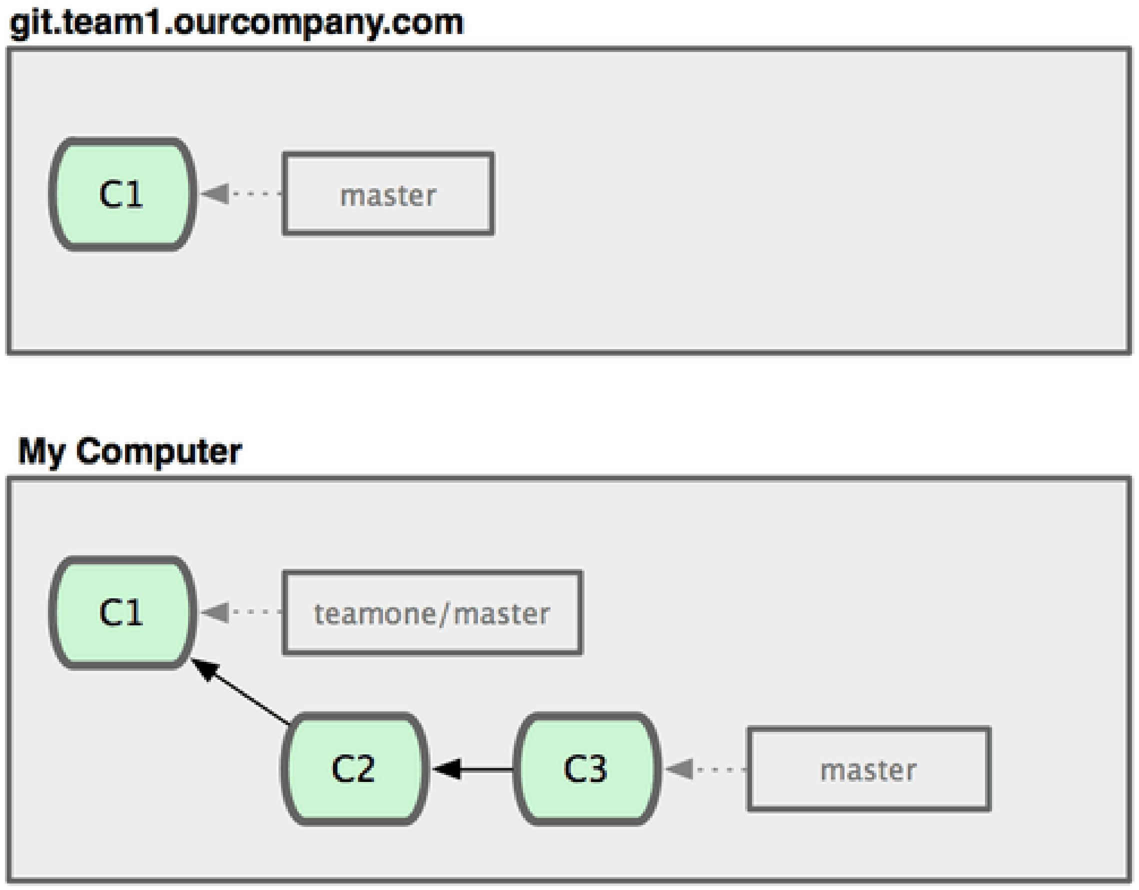


The perils of rebasing

- **Do not rebase commits that you have pushed to a public repository**.
- When you rebase stuff, you’re abandoning existing commits and creating new ones that are similar but different.
- If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with git rebase and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

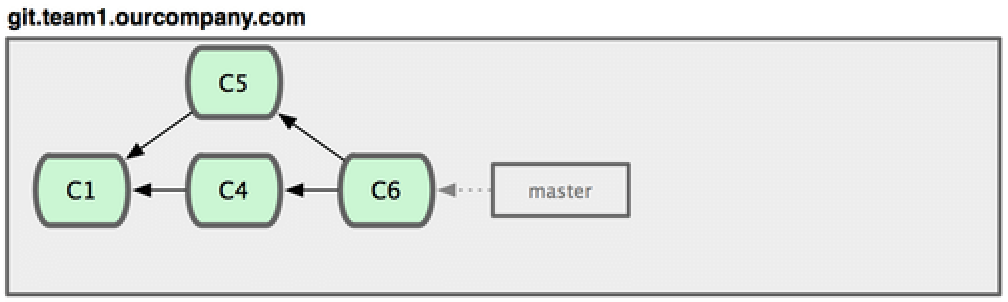
The perils of rebasing: an example

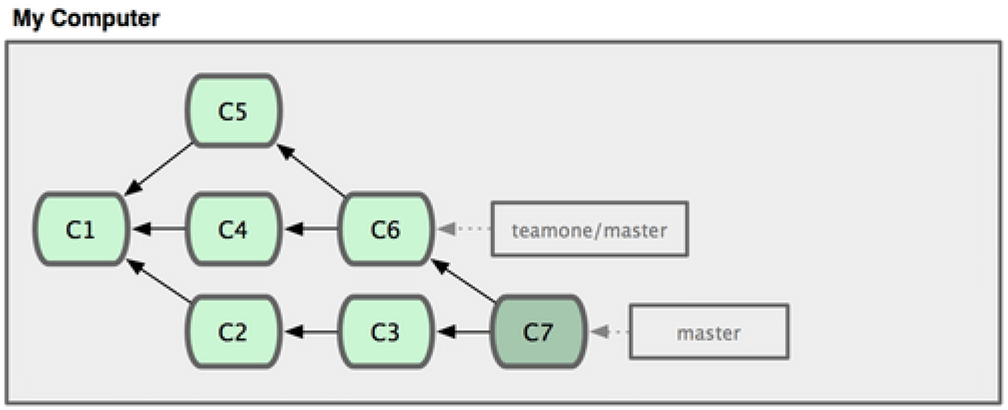
- Let’s suppose to have the following



The perils of rebasing: an example

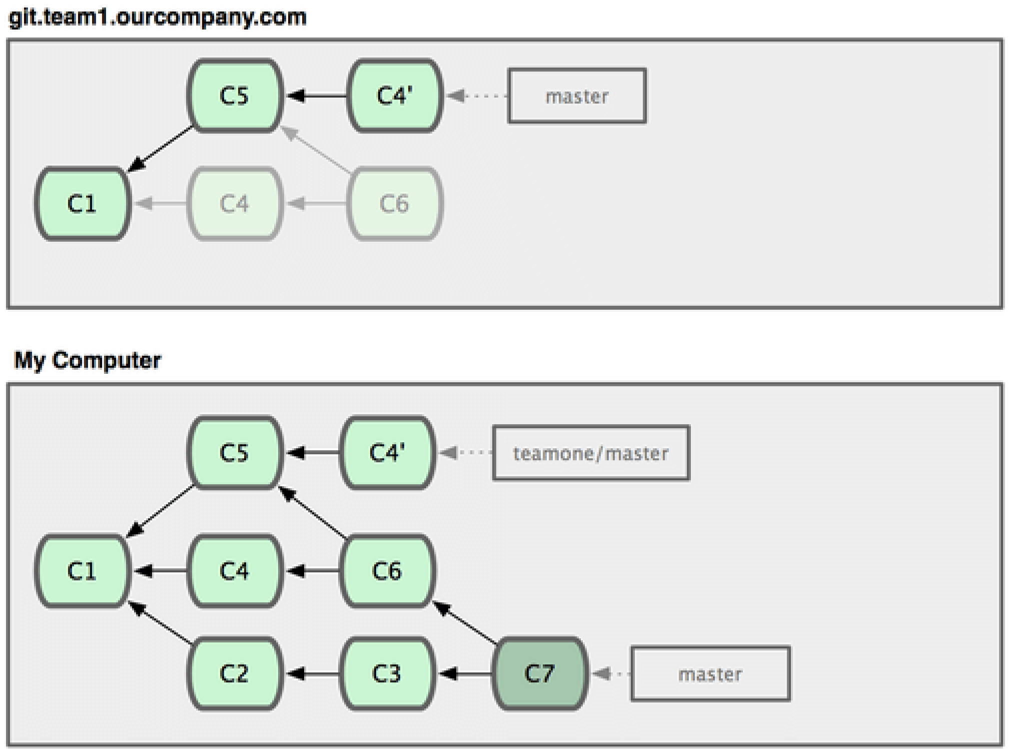
- Now, someone else does more work that includes a merge, and pushes that work to the central server.
- You fetch them and merge the new remote branch into your work, making your history look something like





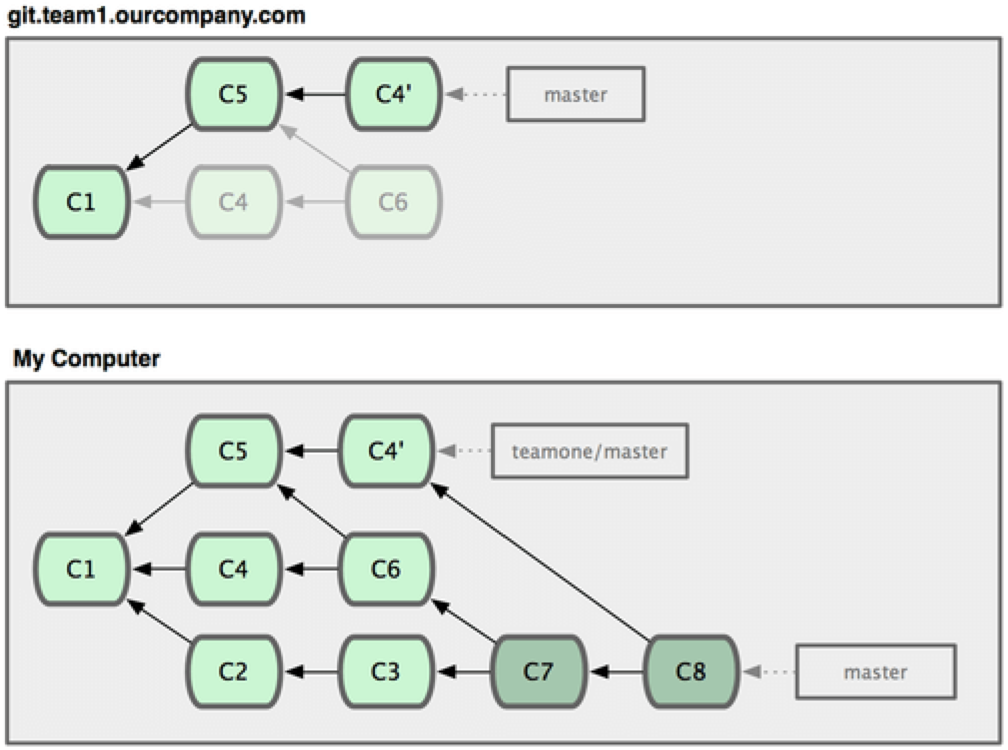
The perils of rebasing: an example

- Next, the person who pushed the merged work decides to go back and rebase their work instead.
- You then fetch from that server, bringing down the new commits.



The perils of rebasing: an example

- At this point, you have to merge this work in again, even though you’ve already done so.
- Rebasing changes the SHA-1 hashes of these commits so to Git they look like new commits, when in fact you already have the C4 work in your history.



Rebase vs. merge

- Which one is better?
- First point of view: repository’s commit history is a **record of what actually happened**.
  - It’s a historical document, valuable in its own right, and shouldn’t be tampered with. From this angle, changing the commit history is almost blasphemous; you’re lying about what actually transpired. So what if there was a messy series of merge commits? That’s how it happened, and the repository should preserve that for posterity.
- Second point of view: the commit history is the **story of how your project was made**.
  - You wouldn’t publish the first draft of a book, and the manual for how to maintain your software deserves careful editing. This is the camp that uses tools like rebase and filter-branch to tell the story in the way that’s best for future readers.
- Answering to the question is not that simple.
- **Rebase local changes** you’ve made but haven’t shared yet before you push them in order to clean up your story
- But **never rebase anything you’ve pushed somewhere**.

Git on the server

- Git can use four major network protocols to transfer data:
  - Local
  - Secure Shell (SSH)
  - Git
  - HTTP/HTTPS

Git on the server: local protocol

- The remote repository is in another directory on disk
- It can be a shared filesystem (shared directory, NFS, etc)

```
$ git clone /opt/git/project.git
```

- PROS:
  - File-based repositories are simple and they use existing file permissions and network access
  - Nice option for quickly grabbing work from someone else's working repository
- CONS
  - Shared access is generally more difficult to set up and reach from multiple locations than basic network access
  - This isn't necessarily the fastest option if you're using a shared mount of some kind

Git on the server: SSH protocol

- Probably the most common transport protocol for Git is SSH
- SSH is also the only network-based protocol that you can easily read from and write to

```
$ git clone ssh://user@server/project.git
```

- PROS:
  - Authenticated write access to your repository over a network
  - SSH is relatively easy to set up — SSH daemons are commonplace
  - Access over SSH is secure — all data transfer is encrypted and authenticated
  - SSH is efficient, making the data as compact as possible before transferring it
- CONS
  - No anonymous access allowed (not ideal for open source projects)

Git on the server: Git protocol

- This is a special daemon that comes packaged with Git; it listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication

- PROS:
  - The Git protocol is the fastest transfer protocol available
  - Same data-transfer mechanism as the SSH protocol but without the encryption and authentication overhead
- CONS
  - Lack of authentication: generally, you’ll pair it with SSH access for the few developers who have push (write) access and have everyone else use git:// for read-only access
  - Difficult to set up

Git on the server: HTTP/S protocol

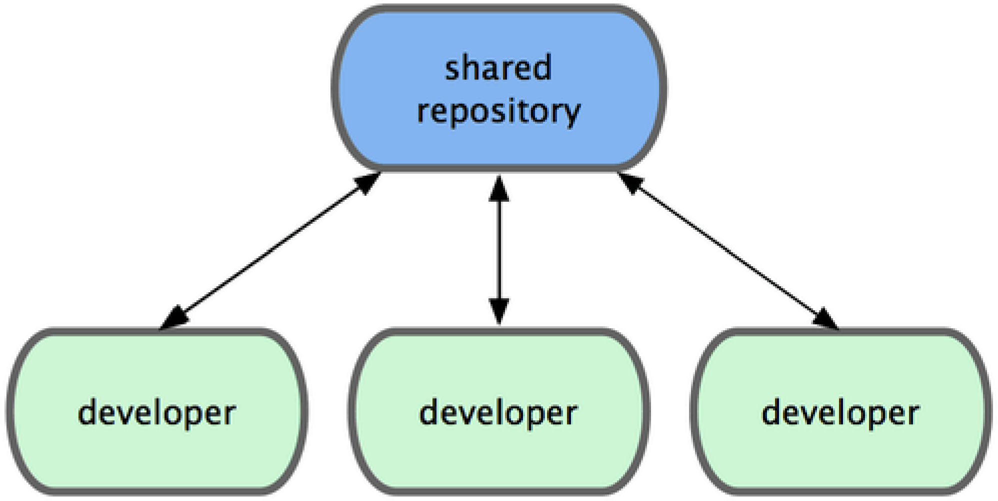
- The beauty of the HTTP or HTTPS protocol is the simplicity of setting it up
- Uses the web server: anyone who can access the web server under which you put the repository can also clone your repository

```
$ git clone http://example.com/gitproject.git
```

- PROS:
  - Easy to set up and no resource intensive on the server
  - You can also serve your repositories over HTTPS, which means you can encrypt the content transfer
  - HTTP is such a commonly used protocol that corporate firewalls are often set up to allow traffic through this port
- CONS
  - Inefficient for the client: it takes a lot longer to clone or fetch from the repository, and you often have a lot more network overhead and transfer volume over HTTP than with any of the other network protocols

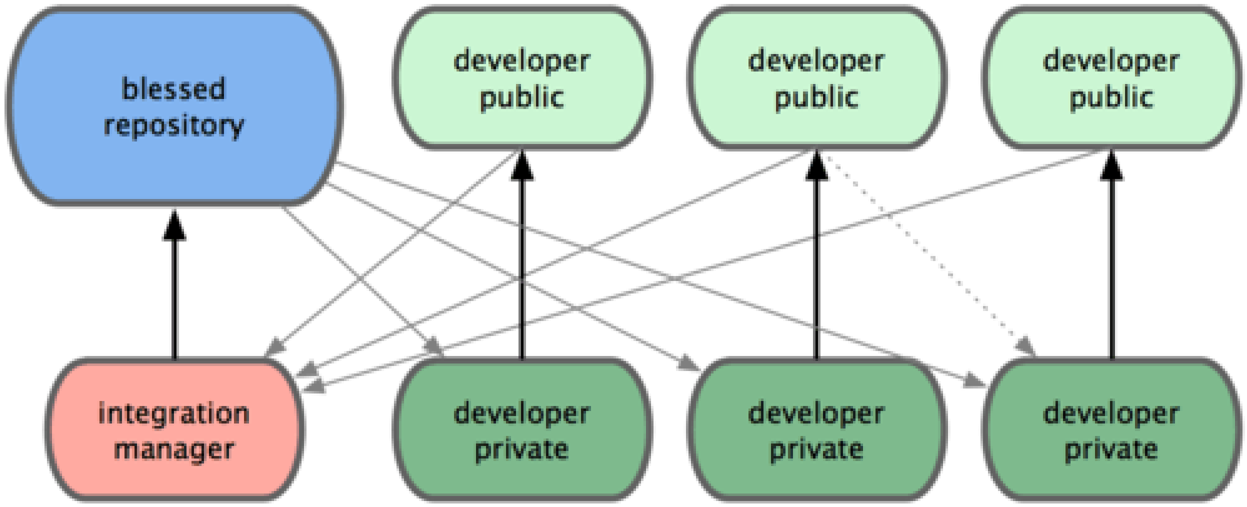
Distributed workflows

- Centralized Workflow
- Small team or already comfortable with a centralized workflow in your company or team



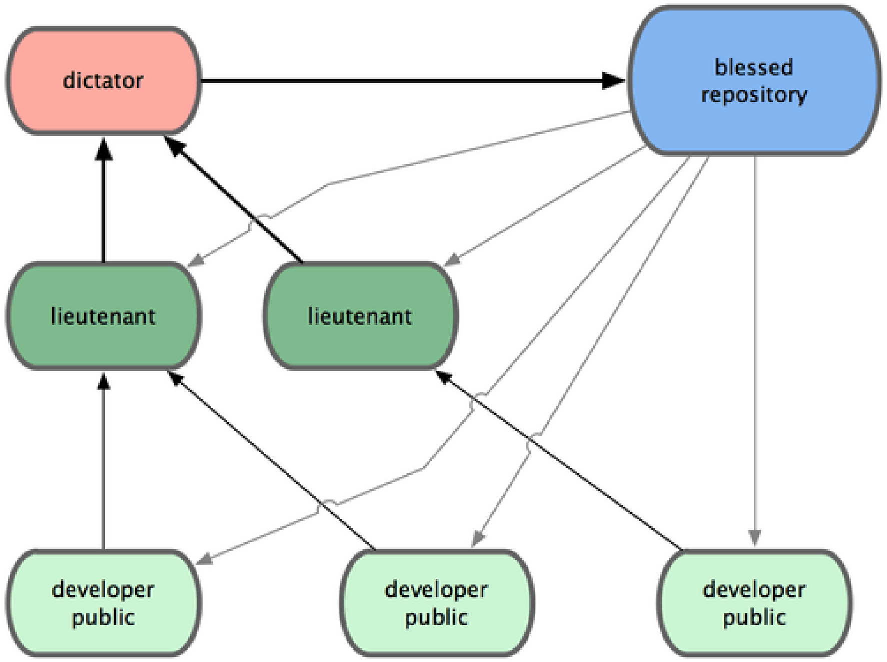
Distributed workflows

- Integration-Manager Workflow
- One of the main advantages of this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time



Distributed workflows

- Dictator and Lieutenants Workflow
- This kind of workflow isn’t common but can be useful in very big projects or in highly hierarchical environments, as it allows the project leader (the dictator) to delegate much of the work and collect large subsets of code at multiple points before integrating them



Stashing

- Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else.
- The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later.
- The answer to this issue is the `git stash` command.
- Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time.

Stashing

- Go into your project and start working on a couple of files and possibly stage one of the changes. If you run git status, you can see your dirty state:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

Stashing

- Now you want to switch branches, but you don’t want to commit what you’ve been working on yet; so you’ll stash the changes.
- To push a new stash onto your stack, run git stash:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

- Your working directory is clean:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Stashing

- At this point, you can easily switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use git stash list:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

- You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`.
- If you want to apply one of the older stashes, you can specify it by naming it, like this: `git stash apply stash@{2}`.

Stashing

- The changes to your files were reapplied, but the file you staged before wasn't restaged.
- To do that, you must run the `git stash apply` command with a `--index` option to tell the command to try to reapply the staged changes.
- The apply option only tries to apply the stashed work — you continue to have it on your stack. To remove it, you can run `git stash drop` with the name of the stash to remove.
- You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

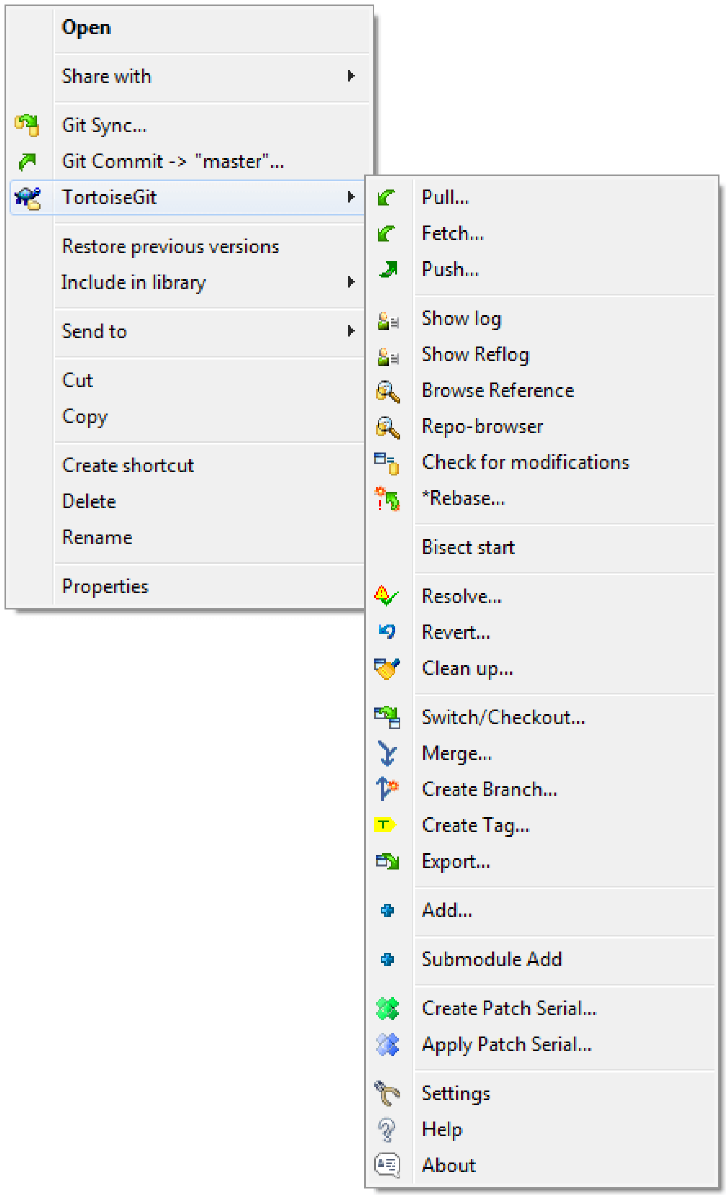
Creating a branch from stash

- If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work.
- If the apply tries to modify a file that you’ve since modified, you’ll get a merge conflict and will have to try to resolve it.
- If you want an easier way to test the stashed changes again, you can run `git stash branch`, which creates a new branch for you, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully.

TortoiseGIT

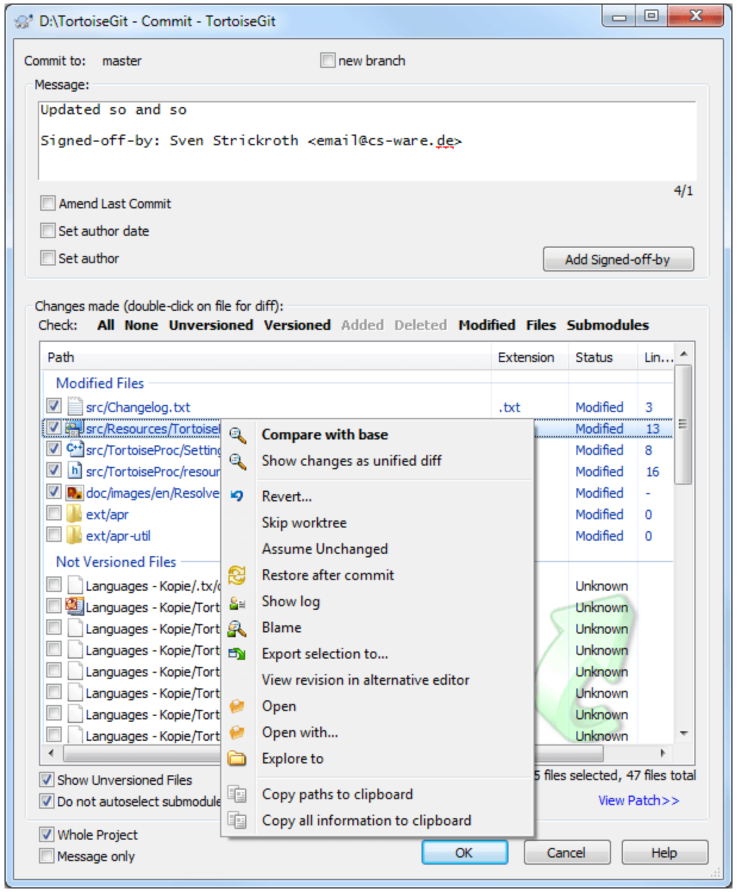
- Download and install TortoiseGIT
- https://code.google.com/p/tortoisegit/
- TortoiseGit is a Windows Shell Interface to Git and based on TortoiseSVN. It's open source and can fully be build with freely available software.
- TortoiseGit supports you by regular tasks, such as committing, showing logs, diffing two versions, creating branches and tags, creating patches and so on.
- You're welcome to contribute to this project (help on coding, documentation, Translation, testing preview releases or helping other users on the mailing lists is really appreciated).

Context menu



Commit Dialog

- Support spell checking and autocomplete for filenames (and class, variable, method, and interface names)


D:\TortoiseGit - Commit - TortoiseGit

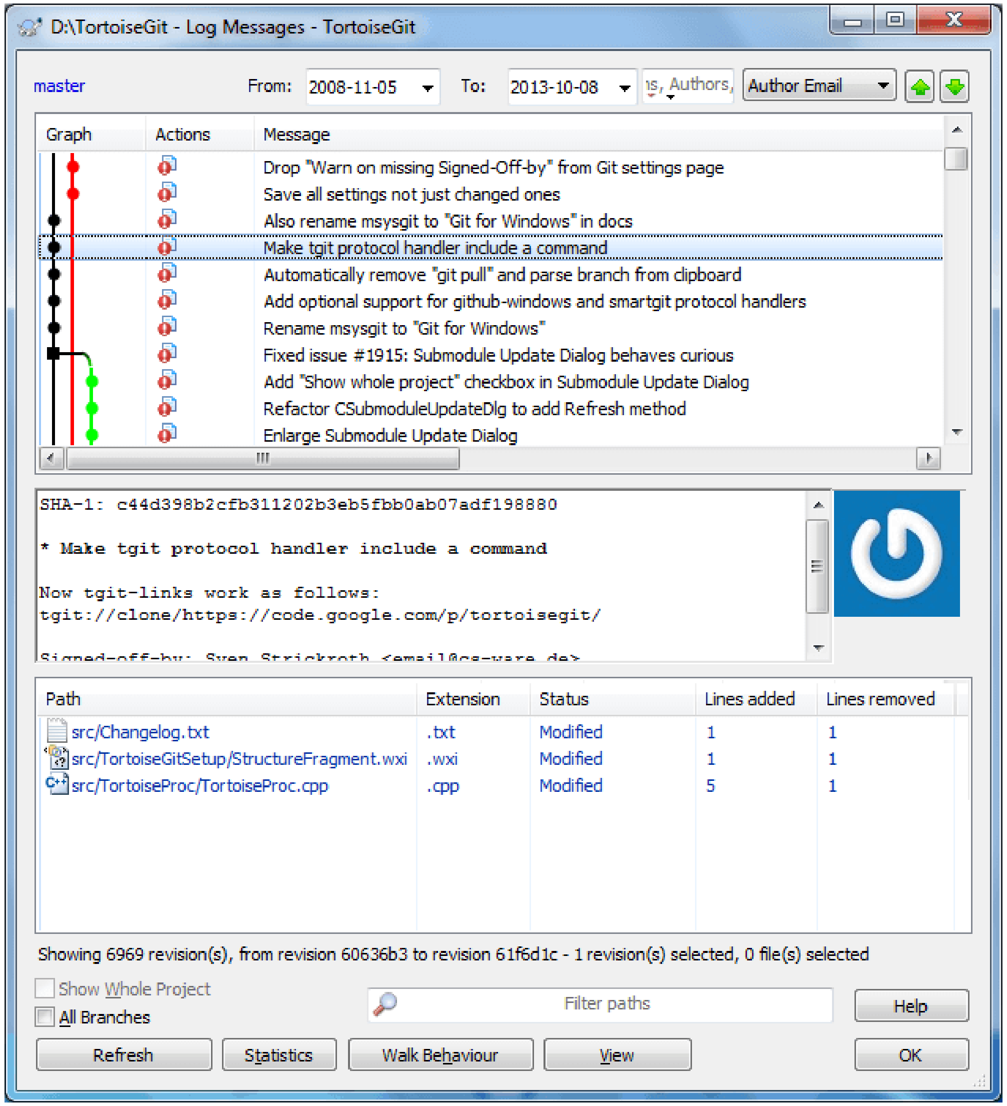
Commit to: master

Message:

```
Updated so and so

Signed-off-by: Sven Strickroth <email@cs-ware.de>
```

4/1

Log dialog



Graph	Actions	Message
		Drop "Warn on missing Signed-Off-by" from Git settings page
		Save all settings not just changed ones
		Also rename msysgit to "Git for Windows" in docs
		Make tgit protocol handler include a command
		Automatically remove "git pull" and parse branch from clipboard
		Add optional support for github-windows and smartgit protocol handlers
		Rename msysgit to "Git for Windows"
		Fixed issue #1915: Submodule Update Dialog behaves curious
		Add "Show whole project" checkbox in Submodule Update Dialog
		Refactor CSubmoduleUpdateDlg to add Refresh method
		Enlarge Submodule Update Dialog

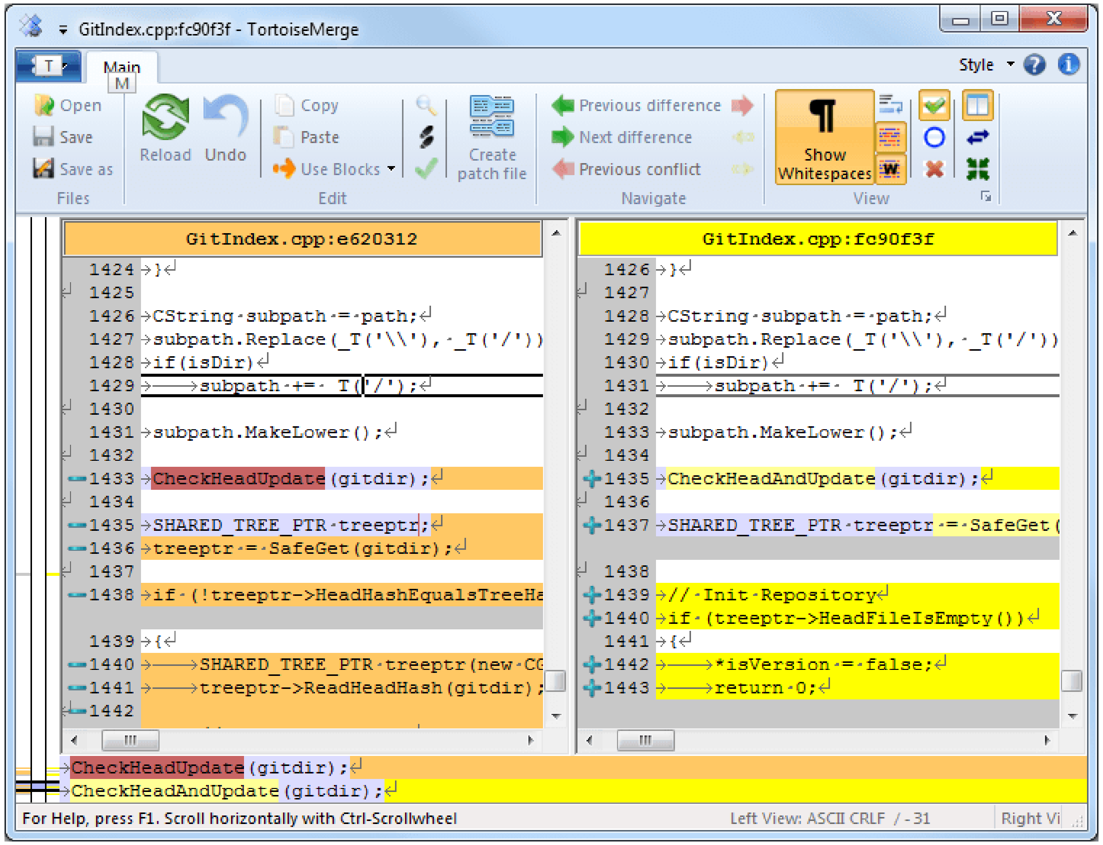
```
SHA-1: c44d398b2cfb311202b3eb5fbb0ab07adf198880

* Make tgit protocol handler include a command

Now tgit-links work as follows:
tgit://clone/https://code.google.com/p/tortoisegit/

Signed-off-by: Sven Strickroth <email@cs-ware.de>
```

Path	Extension	Status	Lines added	Lines removed
src/Changelog.txt	.txt	Modified	1	1
src/TortoiseGitSetup/StructureFragment.wxi	.wxi	Modified	1	1
src/TortoiseProc/TortoiseProc.cpp	.cpp	Modified	5	1

Showing 6969 revision(s), from revision 60636b3 to revision 61f6d1c - 1 revision(s) selected, 0 file(s) selected

TortoiseGitMerge Support Git Patch file

- TortoiseGitMerge can open Git patch files directly, you review them and apply to your working copy



GitIndex.cpp:e620312

```
}

CString subpath = path;
subpath.Replace(_T('\\'), _T('/'))
if(isDir)
        subpath += _T('/');

subpath.MakeLower();

CheckHeadUpdate(gitdir);

SHARED_TREE_PTR treeptr;
treeptr = SafeGet(gitdir);

if (!treeptr->HeadHashEqualsTreeHa

{
        SHARED_TREE_PTR treeptr(new CG
        treeptr->ReadHeadHash(gitdir);
```

GitIndex.cpp:fc90f3f

```
}

CString subpath = path;
subpath.Replace(_T('\\'), _T('/'))
if(isDir)
        subpath += _T('/');

subpath.MakeLower();

CheckHeadAndUpdate(gitdir);

SHARED_TREE_PTR treeptr = SafeGet(

// Init Repository
if (treeptr->HeadFileIsEmpty())
{
        *isVersion = false;
        return 0;
```

```
CheckHeadUpdate(gitdir);
CheckHeadAndUpdate(gitdir);
```

For Help, press F1. Scroll horizontally with Ctrl-Scrollwheel   Left View: ASCII CRLF / -31   Right Vi