CitiusTech

Git

Version Control System

- Also known as **Software Control Management** or **Source Code Management – SCM**

- What is it?
  - A category of software tools that helps in recording changes made to files by keeping a track of modifications done to the code so that it can be recalled later if required

- Why is it required?
  - Maintaining multiple versions of source code file(s) manually is a complex activity
  - Changes made by every developer needs to be **tracked**
    - Who made the change?
    - When was the change made?
    - What change(s) were/was made?
  - Accidental overwriting of code should not happen
  - Sharing code with peer developers so that developers can work in a collaborative manner
    - Promotes parallel development
  - Access control as to who can read/modify code

Version Control System – Core Concepts

- Working Directory / Workspace
  - Where developers create/modify files
  - Here VCS is not applicable, it's used only for storing files

- Repository
  - Stores files, version history & metadata
  - VCS is applicable here

- Some kind of client utility is required for the developer(s) to store versions of modified files into the repository

- Repository contains one or more **commits**
  - A commit essentially a revision to a file/files which git stores along with some metadata
  - Each commit has some kind of information associated with it along with a unique commit id

- Process of sending files from working directory to repository is called a **COMMIT** or **CHECK IN**

- Process of getting the required file version from repository to working directory is called a **PULL** or **CHECK OUT**

Version Control System – Centralized / Client-Server

- Only one central repository
- All the source code is stored in the central repository
- Scales to very large codebases
- Ability to lock files exclusively to handle complex merge scenarios
- Best used for large integrated codebases
- Examples: Subversion (SVN), CVS, Perforce, TFS, etc....
- Limitations:
  - Every developer is required to remain connected with the repository to commit and pull changes
  - Single point of failure since it's centralized
  - All check-in and checkout is always done on the centralized server which forces developers to be always connected to the repository
  - Since all operations are performed on a network, performance is slow

Version Control System – Distributed

- Repository is distributed → each developer has his/her own copy of the repository locally in addition to a main repository
- Cross platform
- Since all check-ins and check outs are performed on the local repository, there is a fast performance
- Developers need not be continuously connected to the main repository
- Single point of failure risk is eliminated
- Complete repository with portable history
- Pull requests for code review
- Process of sending commits from local repo to remote repo is **PUSH**
- Process of getting a commit from remote repo to local repo is **PULL**
- Examples: Git, Mercurial, etc.

Version Control System – Distributed

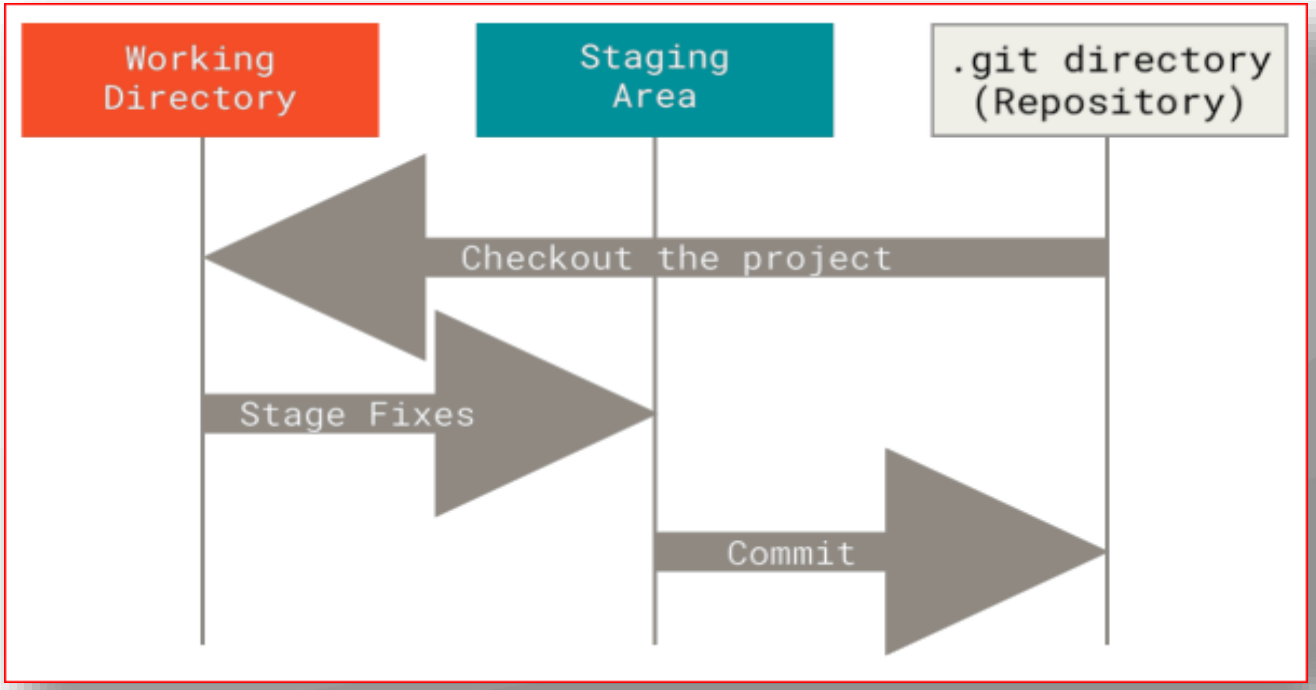
- Best used when
  - Modular codebases not too big in size
  - Highly distributed teams who need to work offline
  - Need to work on various platforms

Introducing Git

- A DVCS Developed by **Linus Torvalds** in 2005

- How does Git work?
  - Git thinks of its data more like a series of **snapshots** of a miniature filesystem
  - With Git, every time you commit, or save the state of your project, Git takes a picture of what all your files look like at that moment and stores a **reference to that snapshot**
  - **Git thinks about its data more like a stream of snapshots**

  - The Three States



Introducing Git

- The working tree is a single checkout of one version of the project
  - These files are pulled out of the compressed database in the Git directory and placed on disk for modification

- The staging area stores information about what will go into the next commit
  - Also known as **index** in Git terminology

- The Git directory is where Git stores the metadata and object database the project
  - This is what is copied when cloning a repository from remote

Basic Git Workflow

- You modify files in your working tree
- Selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area
- Do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory
- If a particular version of a file is in the Git directory, it's considered **committed**
- If it has been modified and was added to the staging area, it is **staged**
- If it was changed since it was checked out but has not been staged, it is **modified**

Installing Git - Windows

https://git-scm.com/download/win

Configuring Git – DO IT

- Verify Installation

```
C:\Users\karthikeyanj>git --version
git version 2.32.0.windows.1
```

- Set username and email which Git uses when committing changes so that it can identify who made the changes

```
C:\impact-ui-batch2-git>git config --global user.name "Karthikeyan"

C:\impact-ui-batch2-git>git config --global user.email "karthik1409mct@gmail.com"
```

  - **--global** option applies to all projects
  - For a specific project, omit the **–global** option

- To see the settings, use the **git config** command

```
C:\impact-ui-batch2-git>git config user.name
Karthikeyan

C:\impact-ui-batch2-git>git config user.email
karthik1409mct@gmail.com
```

Getting a Git Repository

- A Git repo represents a virtual storage of a project
- Each project in Git is referenced as a repo
- Can be done in 2 ways:
  - Take a local directory that is currently not under version control, and turn it into a Git repository *(initialize a Git repository)*
  - **Clone** an existing Git repository from elsewhere

```
$ git clone https://github.com/libgit2/libgit2
```

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Initializing a Git Repository locally – DO IT

- A Git repo represents a virtual storage of a project
- Each project in Git is referenced as a repo
- To initialize a project or a project that already exists, get into the folder & use the **git init** command
  - This tells Git to version control the all files and sub-directories in the project
  - Adds a hidden folder named **.git** which contains metadata used by Git internally to version control the project

```
C:\impact-ui-batch2-git>git init
Initialized empty Git repository in C:/impact-ui-batch2-git/.git/

C:\impact-ui-batch2-git>dir .git
 Volume in drive C is Windows
 Volume Serial Number is E49D-EFDF

 Directory of C:\impact-ui-batch2-git\.git

                                130 config
                                 73 description
                                 23 HEAD
                    <DIR>          hooks
                    <DIR>          info
                    <DIR>          objects
                    <DIR>          refs
               3 File(s)            226 bytes
               4 Dir(s)  846,214,193,152 bytes free
```

Code Hosting Providers

- Allows a developer to access a Git repository from anywhere, any device & pull down a local copy of the repo

- Several companies specialize of this type of product offering → web-based Git repository hosting services with additional features
  - GitHub
  - Bitbucket
  - GitLab
  - Azure Repos → part of Azure DevOps offerings
  - AWS Code Commit → part of AWS DevOps offerings
  - Others….

- Developers can push changes from their local Git repositories to an online hosting service
  - Allows collaboration with other peer developers
  - Allows contributions to me made to the repository from other developers
  - Integrate collaborations easily with multiple developers

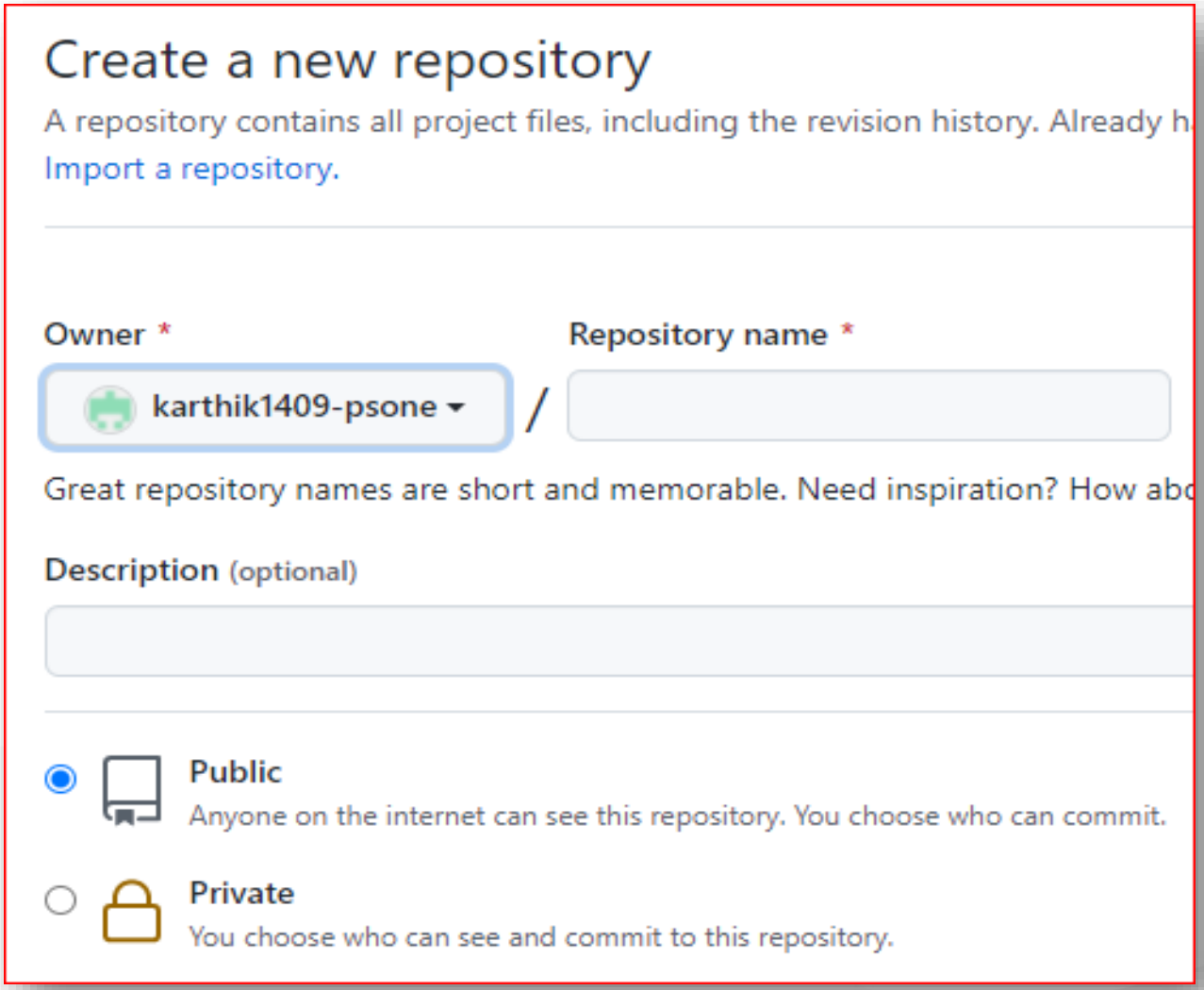
- **To start → choose a provider and create an account**

Creating an account on Github and signing in – DO IT

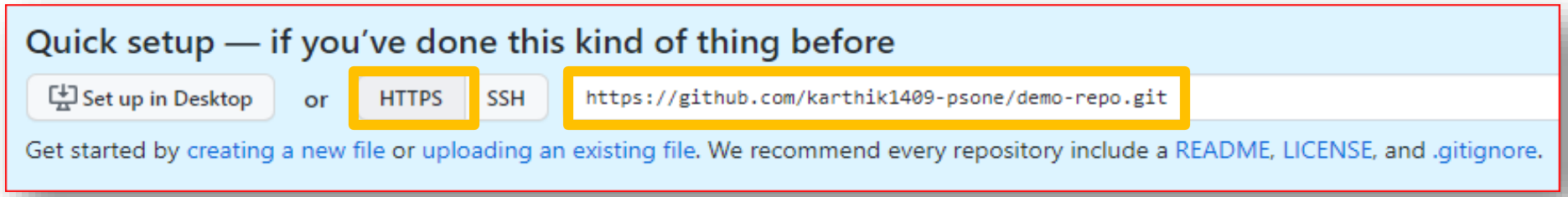
https://github.com/signup

https://github.com/login

Creating a public repo on Github – DO IT



Copy the URL of the repo for PUSH and PULL – DO IT

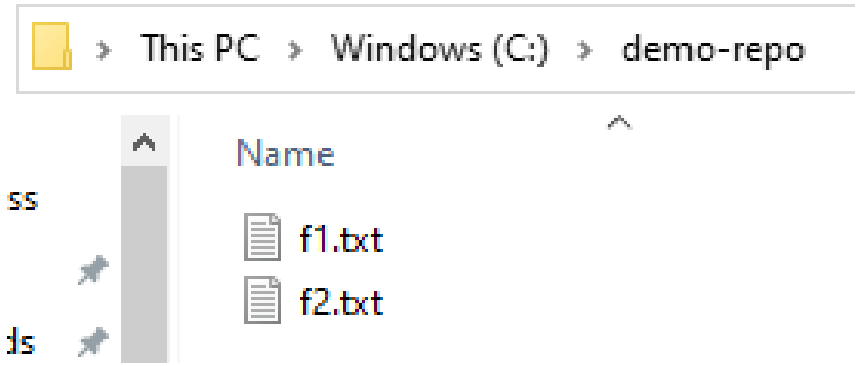


First Commit to local repo – DO IT

- Initialize an empty Git repo on the local system
- Check status → **git status**
- Create some files inside it
- Check status → **git status**
- Add to staging → **git add** command
- Check status → **git status**
- Commit → **git commit –m *"<<commit message>>"*** command
- Check status → **git status**

First Commit to local repo – 1

- Create a folder with 2 files in it



First Commit to local repo – 2

- Initialize an empty Git repository

```
C:\demo-repo>git init
Initialized empty Git repository in C:/demo-repo/.git/
```

- Check the status

```
C:\demo-repo>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        f1.txt
        f2.txt

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

First Commit to local repo – 3

- Add the files to staging & check status

```
C:\demo-repo>git add .

C:\demo-repo>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   f1.txt
        new file:   f2.txt
```

First Commit to local repo – 4

- Commit the changes and check status

```
C:\demo-repo>git commit -m "First Commit"
[master (root-commit) bba29fb] First Commit
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 f1.txt
 create mode 100644 f2.txt

C:\demo-repo>git status
On branch master
nothing to commit, working tree clean
```

Connect local repo to remote repo on Github – DO IT

- Create a **link between the local and remote Git repo** which allows pushing and pulling changes between the two
  - **git remote add origin *<<URL of remote Git repo>>***

    OR
  - **git remote add *<<short name for repo>> <<URL of remote Git repo>>***

```
C:\mygitrepos>git init
Initialized empty Git repository in C:/mygitrepos/.git/

C:\mygitrepos>git remote add r1 https://github.com/karthik1409-psone/repo1.git

C:\mygitrepos>git remote add r2 https://github.com/karthik1409-psone/repo2.git

C:\mygitrepos>git remote
r1
r2

C:\mygitrepos>git remote -v
r1      https://github.com/karthik1409-psone/repo1.git (fetch)
r1      https://github.com/karthik1409-psone/repo1.git (push)
r2      https://github.com/karthik1409-psone/repo2.git (fetch)
r2      https://github.com/karthik1409-psone/repo2.git (push)
```

- Push the local commits to the remote repo
  - **git push *<<repo short name>> <<branch name>>***

- Verify the remote repo to ensure that the local repo contents have been pushed to the remote repo to be hosted and tracked by Github

Connect local repo to remote repo on Github – 1

- Add the remote repo as an origin and push the changes

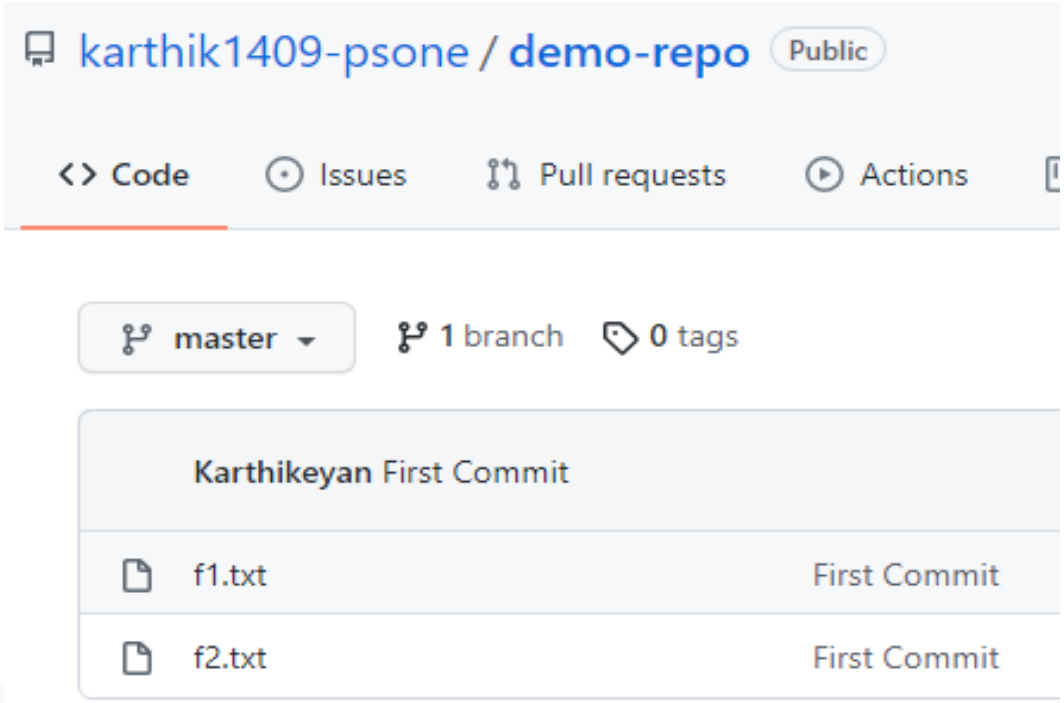
```
C:\demo-repo>git remote add origin https://github.com/karthik1409-psone/demo-repo.git

C:\demo-repo>git push origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 217 bytes | 217.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/karthik1409-psone/demo-repo.git
 * [new branch]      master -> master

C:\demo-repo>git status
On branch master
nothing to commit, working tree clean
```

Connect local repo to remote repo on Github – 2

- Verify the remote repo



Sneak Preview of Git "BRANCHES”

- A branch in Git is a lightweight movable pointer to the project at a specific point in time

- By default a branch named **master** is created automatically when the repo is initialized
  - When the repos are linked, the local master branch is equal to the remote master branch

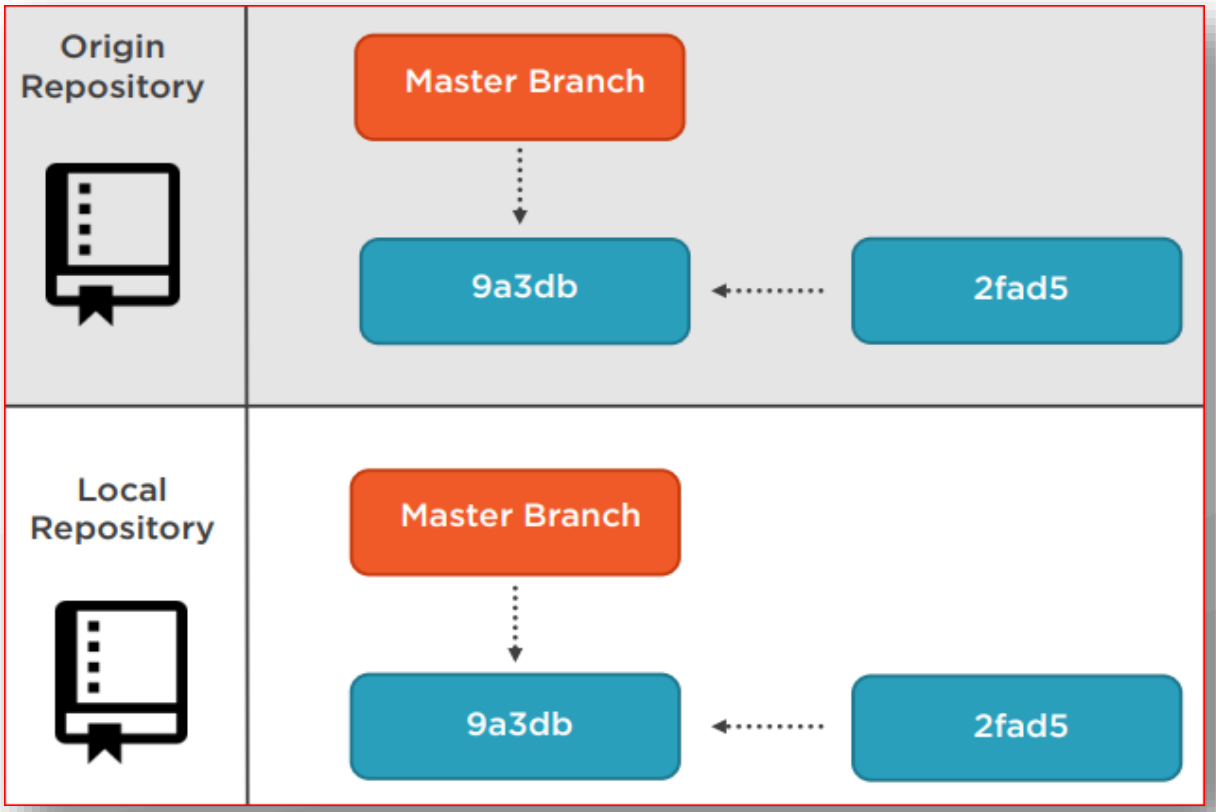


Image courtesy: Pluralsight

Sneak Preview of Git "BRANCHES”

- A commit is always made on a **branch**
- A branch is a pointer to a specific commit
- Each commit updates the pointer to point to the latest commit made

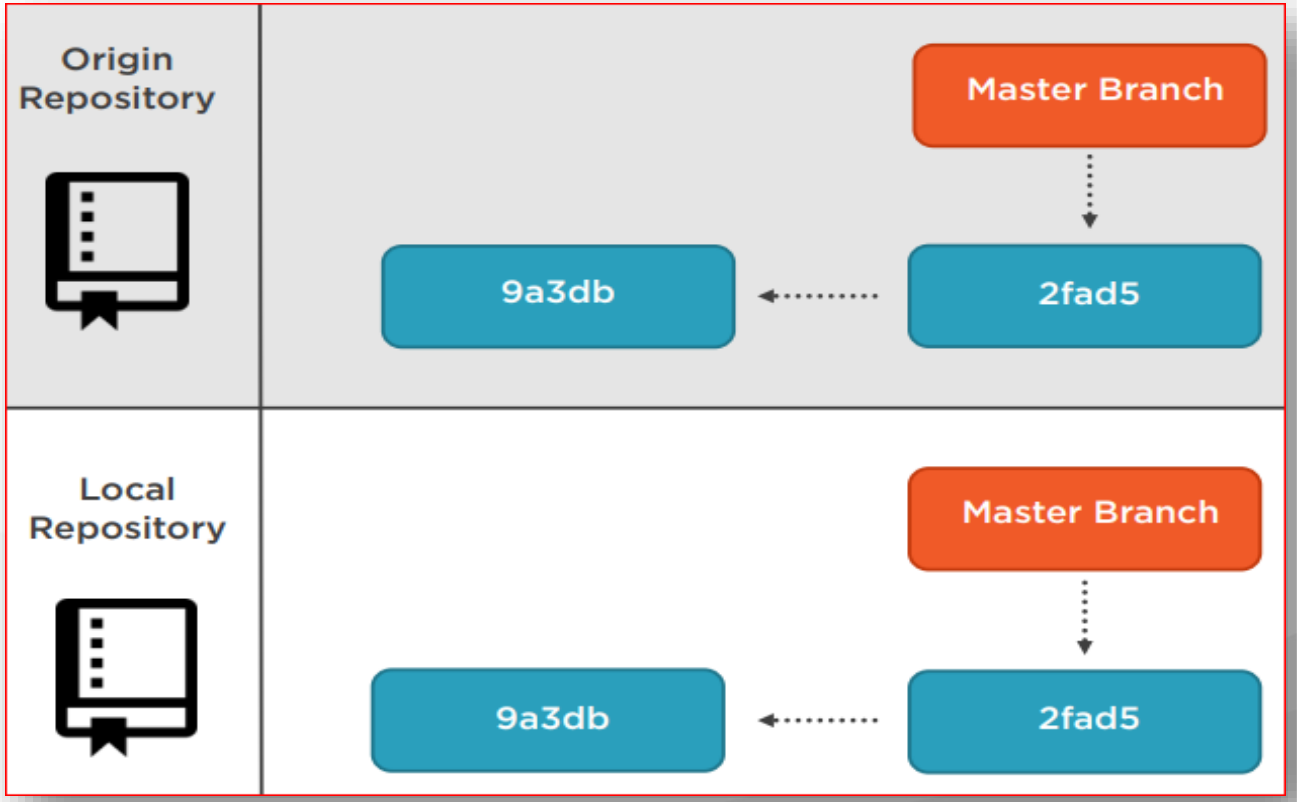


Image courtesy: Pluralsight

Track a new file – DO IT

- Create a new file
- Check the status
- Add it to staging
- Check the status

Track a new file – 1

- Create a new file and check status

```
C:\demo-repo>git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        f3.txt

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

- Add to staging and check the status

```
C:\demo-repo>git add .

C:\demo-repo>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   f3.txt
```

Stage a modified file – DO IT

- Open an existing file
- Modify it and save the changes
- Check the status
- Add to staging
- Check the status again
- Reopen the file
- Make some more change & save it
- Check status again
- Add to staging again
- Check status again

Stage a modified file – 1

- Makes changes to an existing file and save the changes. Check the status

```
C:\demo-repo>echo "Change1 in f1.txt" >> f1.txt

C:\demo-repo>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   f3.txt
        new file:   file1.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   f1.txt
```

- Add the file to staging and check the status

```
C:\demo-repo>git add .

C:\demo-repo>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   f1.txt
        new file:   f3.txt
        new file:   file1.txt
```

Stage a modified file – 2

- Make some more changes to the staged file and check the status

```
C:\demo-repo>echo "Change2 in f1.txt" >> f1.txt

C:\demo-repo>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   f1.txt
        new file:   f3.txt
        new file:   file1.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   f1.txt
```

Stage a modified file – 3

- Add to staging and check the status again

```
C:\demo-repo>git add .

C:\demo-repo>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   f1.txt
        new file:   f3.txt
        new file:   file1.txt
```

Short Status – DO IT

- Can be used to see which file is in which stage

git status -s | **git status --short**

Staged	Modified	File Name
M		File 1
M	M	File 2
A		File 3
?	?	File 4

M = Modified

A = New file added to staging area

?? = New file untracked by Git

Image courtesy: Pluralsight

Short Status – 1

- Modify an already staged file and check the short status

```
C:\demo-repo>echo "file1.txt which is staged is modified again" >> file1.txt

C:\demo-repo>git status -s
M  f1.txt
A  f3.txt
AM file1.txt
```

- Commit the changes and check the status again

```
C:\demo-repo>git commit -m "Second commit"
[master 3333c55] Second commit
 3 files changed, 3 insertions(+)
 create mode 100644 f3.txt
 create mode 100644 file1.txt

C:\demo-repo>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt

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

Pushing to the remote origin & viewing history – DO IT

- All changes committed are pushed to the local repo by default
- Use the **git push origin *<<branchname>>*** to push commits to the remote repo
- Use the **git log** command to view a history of all commits in a reverse chronological order
  - The last commit made is shown first
- Use the command **git log -*<<number>>*** to display specific number of commits
- Use the command **git log --oneline** command to view all commits in a more compressed form
- There are many more options to view and search the git logs
  - Example: search by author, by a commit message, date range, etc.

Pushing to the remote origin & viewing history – 1

- Use the **git log** command to view a history of all commits in a reverse chronological order
- Use the command **git log -*<<number>>*** to display specific number of commits
- Use the command **git log --oneline** command to view all commits in a more compressed form

```
C:\demo-repo>git log
commit 3333c551f4eb937de491df9f50fa21c93aed17ec (HEAD -> master)
Author: Karthikeyan <you@example.com>
Date:   Sat Sep 11 07:31:10 2021 +0000

    Second commit

commit bba29fb783ad0fddbceffb1fa1fc83c8309ef2c9 (origin/master)
Author: Karthikeyan <you@example.com>
Date:   Sat Sep 11 06:57:32 2021 +0000

    First Commit

C:\demo-repo>git log -1
commit 3333c551f4eb937de491df9f50fa21c93aed17ec (HEAD -> master)
Author: Karthikeyan <you@example.com>
Date:   Sat Sep 11 07:31:10 2021 +0000

    Second commit

C:\demo-repo>git log --oneline
3333c55 (HEAD -> master) Second commit
bba29fb (origin/master) First Commit
```

Branching and Merging

- What is a Branch?
  - A lightweight movable pointer to a commit
  - Default branch name is **master**
  - Each time a commit is made, the master branch pointer moves forward automatically
  - Branches are isolated from each other

- Why is it required?
  - Enables multiple parallel development workflows

- Developers can commit changes to a branch and then **merge** the changes with the master branch

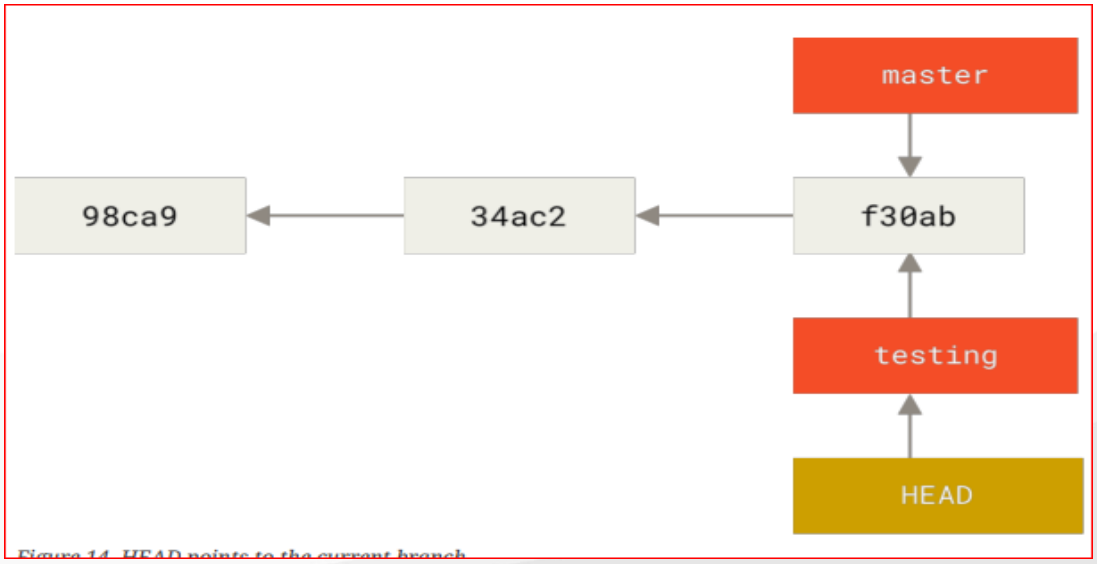
- Creating a new branch simply creates a new pointer

- Git maintains a special pointer named **HEAD** which simply points to a specific branch
  - This is how Git knows the current branch on which a developer is currently working

Branching and Merging

Before creation of a new branch

After creation of a new branch



Visualize how branching works

http://git-school.github.io/visualizing-git/

Working with branches

- Create a new branch

```
git branch testing
```

- Switch to a branch

```
git checkout testing
```

- Switch back to the default **master** branch

```
git checkout master
```

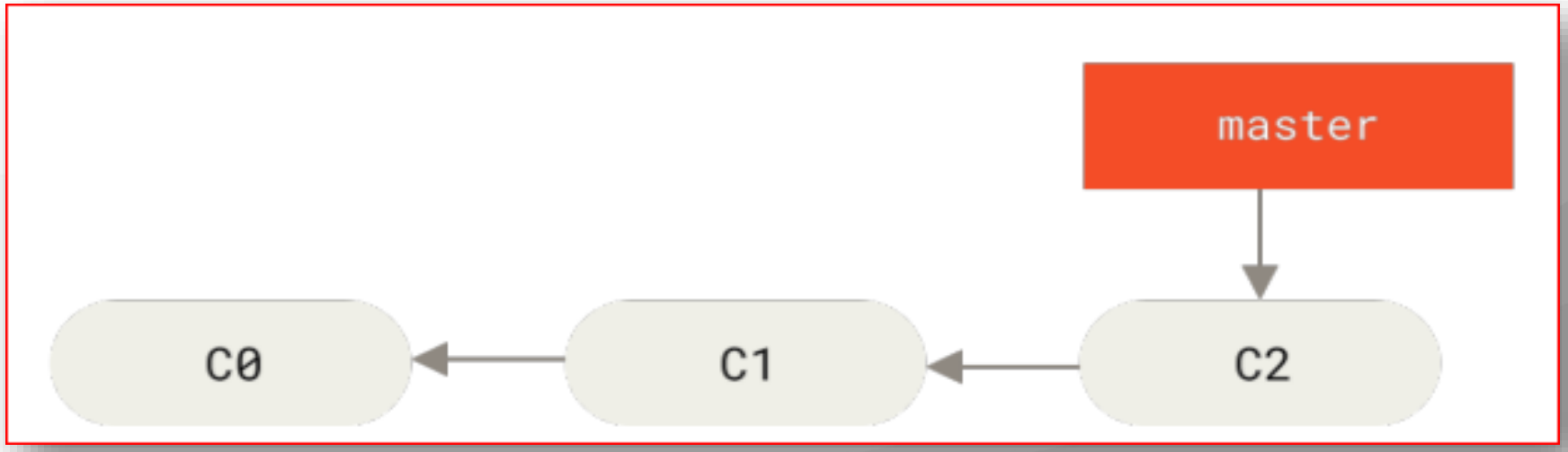
- When switching branches in Git, files in the working directory will change
- When switched to an older branch, the working directory will be reverted to look like it did the last time when a commit was made on that branch

A basic branching workflow

- Do some work on a website
- Create a branch for a new user story
- Do some work in that branch
- **Some another issue is critical and there is a need for a hotfix**
- Switch to the production branch
- Create a branch to add the hotfix
- After it’s tested, merge the hotfix branch, and push to production
- Switch back to the original user story and continue working

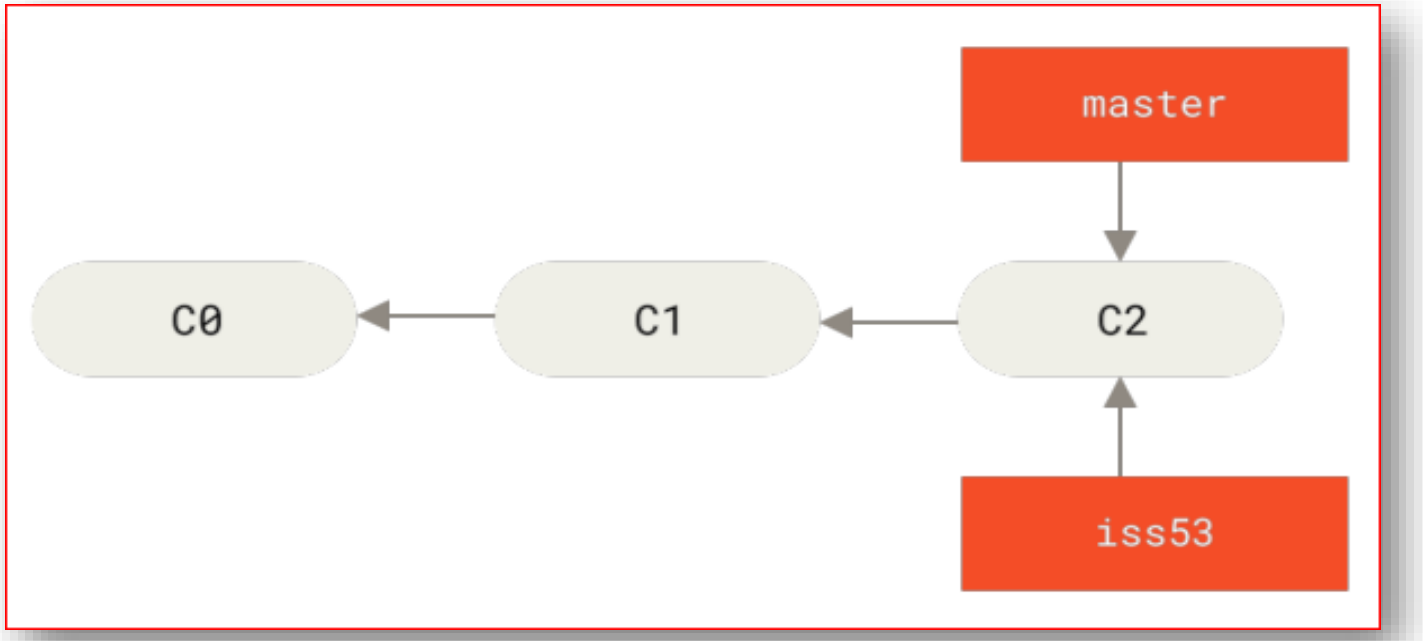
Basic Branching – DO IT (1)

- Create a new remote repo
- Create a new folder and initialize an empty Git repo
- Add the remote repo
- Create two text files in the local folder
- Perform some commits



Basic Branching – DO IT (2)

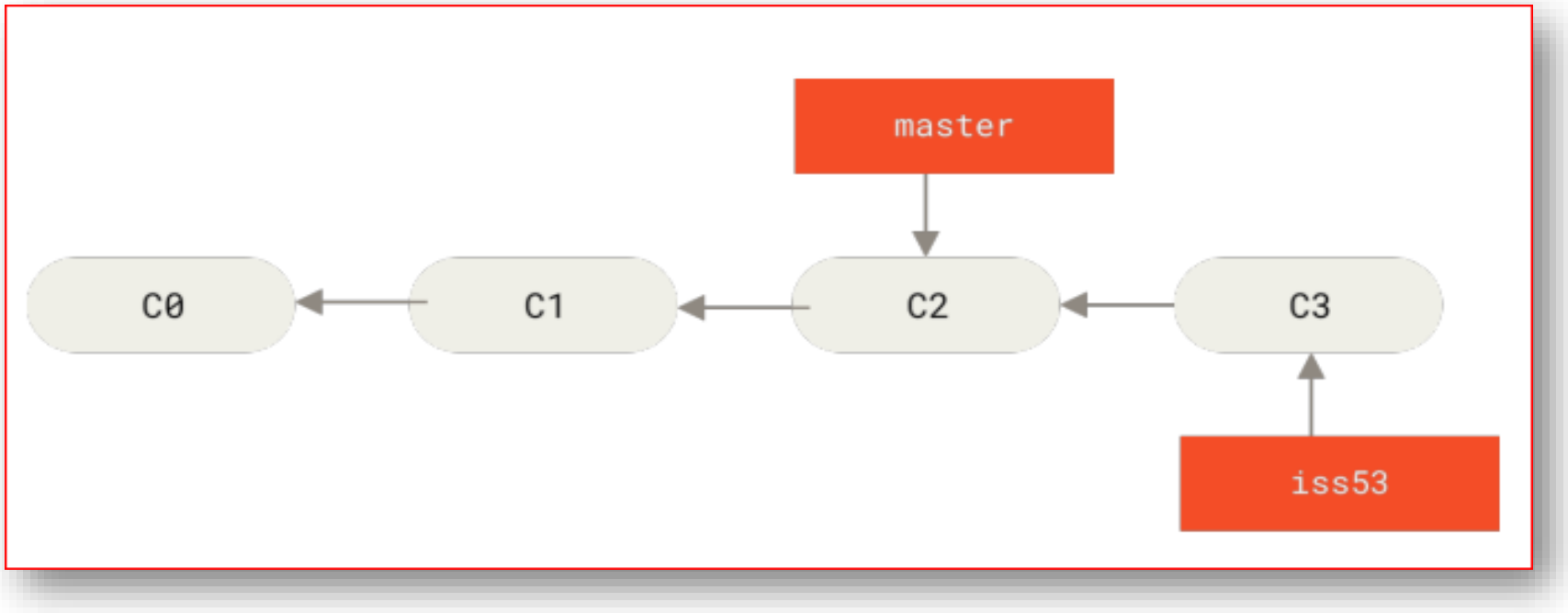
- Create a new branch for a new user story and switch to it




```
C:\my-website>git checkout -b newuserstory
Switched to a new branch 'newuserstory'
```

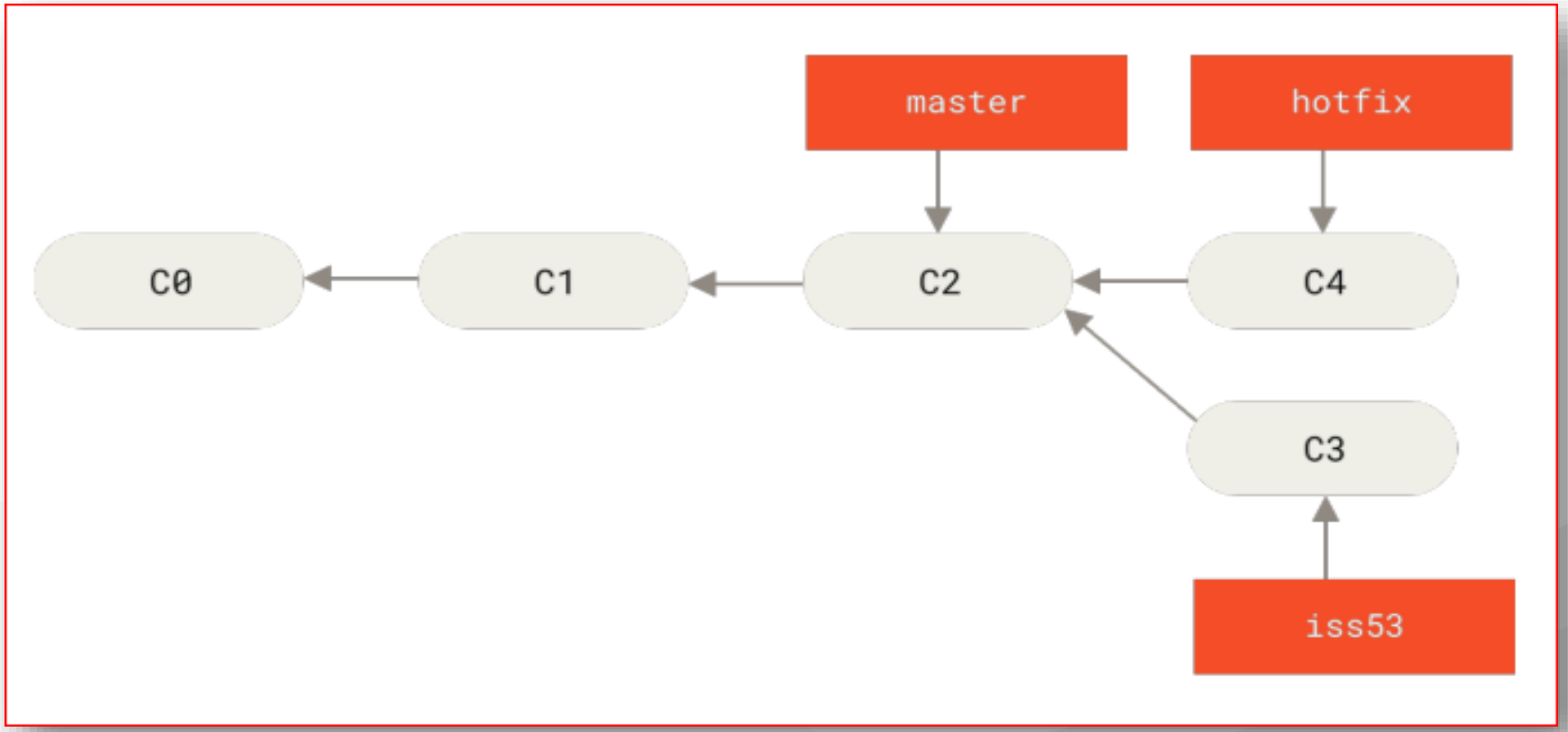
Basic Branching – DO IT (3)

- Perform some commits in this new branch



Basic Branching – DO IT (4)

- Switch back to master branch and create a new branch to work on the hotfix
- Perform some commits on the new hotfix branch



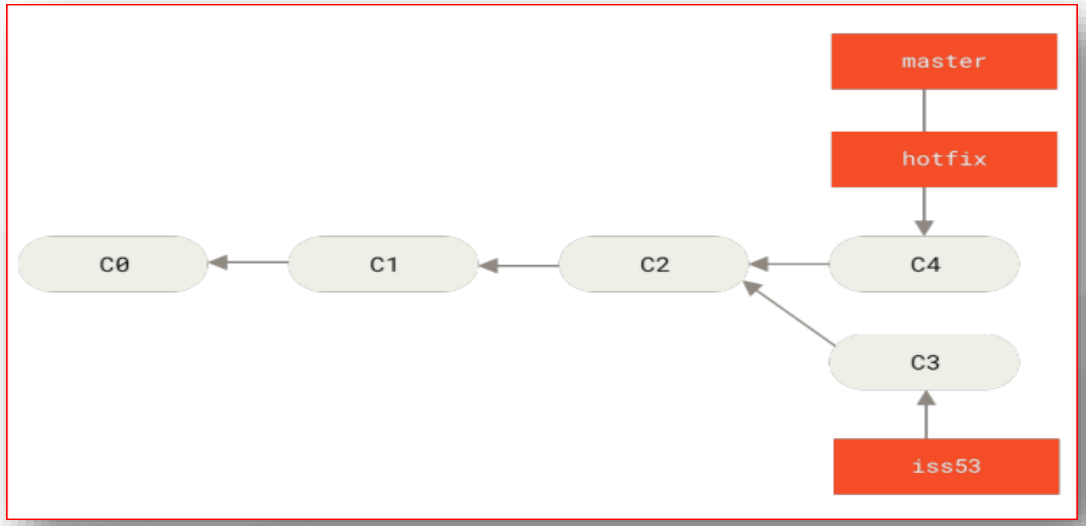
Basic Branching – DO IT (5)

- After testing the hotfix, merge the hotfix branch, switch back to the master branch and merge the hotfix branch with the master branch

```
C:\my-website>git merge hotfix
Updating 705fe1f..ec1821d
Fast-forward
 f1.txt | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)
```

When you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a "fast forward"

- The change is now in the snapshot of the commit pointed to by the master branch, and the fix can now be deployed



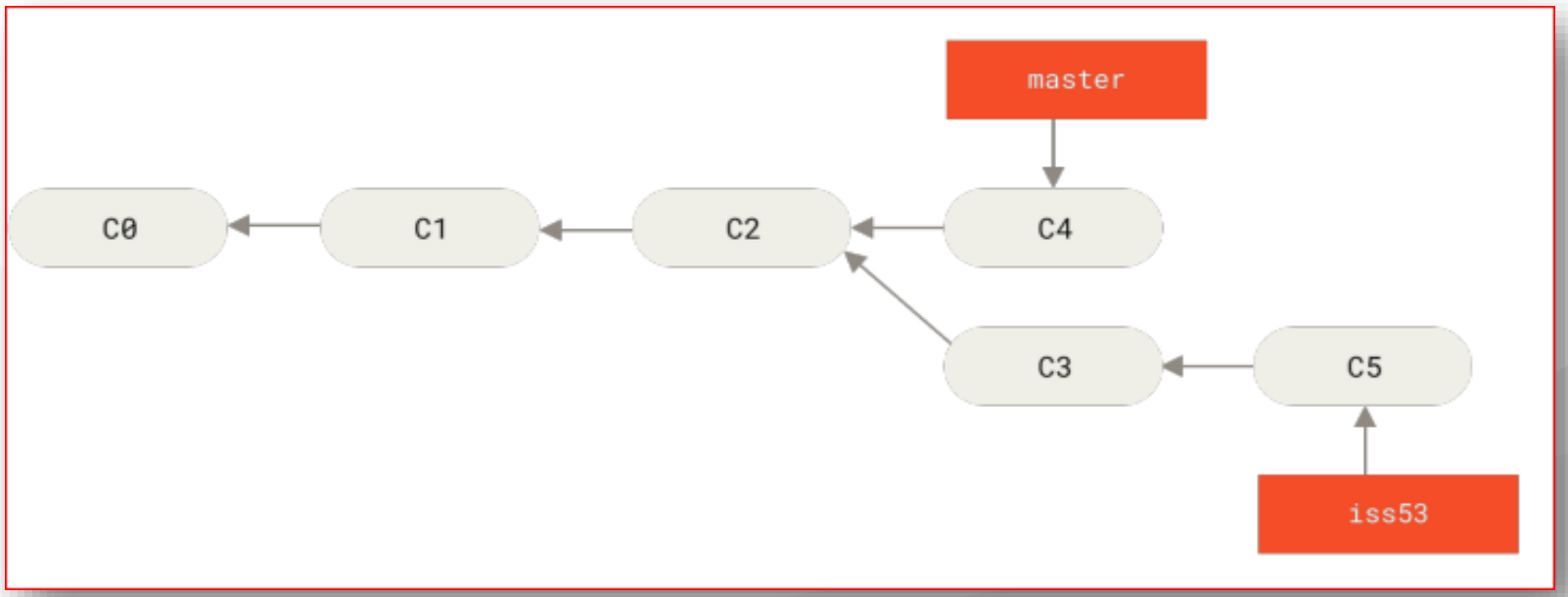
Basic Branching – DO IT (6)

- Delete the hotfix branch and continue working on the new user story branch which was created earlier

```
git branch -d hotfix
```

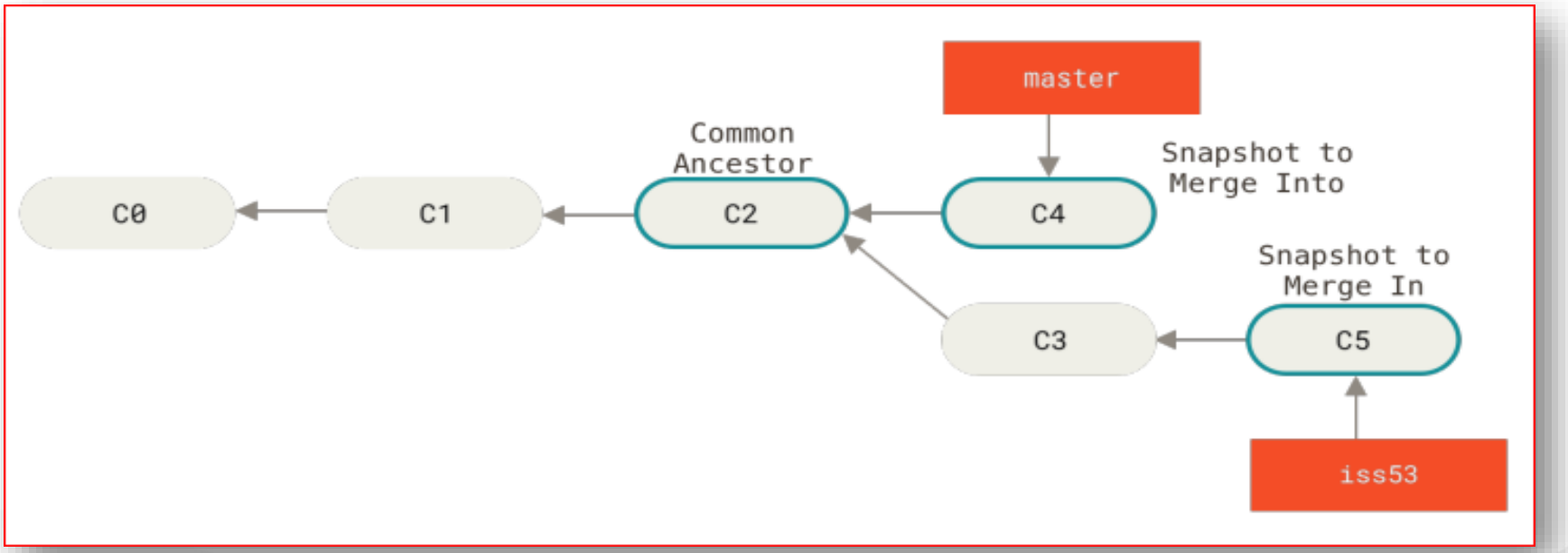
- **git checkout newuserstory**

- Perform one more commit in the new user story



Basic Branching – DO IT (7)

- Now merge the changes made in the user story with the master branch
- Switch to the master branch first and then use the **git merge newuserstory** command



- Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work
  - In this case, Git does a simple **three-way merge**, using the two snapshots pointed to by the branch tips and the common ancestor of the two

Basic Branching – DO IT (8)

- Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it

- This is known as a **merge commit,** which essentially has more than one parent