Using Git for Centralized and Distributed Version Control Workflows

11 March, 2016

Presenter:
Brian Vanderwende

CISL Computational & Information Systems Laboratory

NCAR
NATIONAL CENTER FOR ATMOSPHERIC RESEARCH

Testing Git on your machine

- Remote users: https://goo.gl/qPiRdX
- Survey of operating systems in use
- Open a terminal (or git bash) and type **git**
  - Does it not work for anyone?

Day 1 - Introduction to Git

1. Version control evolution through to Git
2. Basic features
    a. Repositories, branches, commits, staging, working directory
3. Feature development
    a. Fast-forward merging
4. Using remote repositories
5. Git workflows
    a. Local, centralized, distributed
6. Enhancing your Git experience

A very brief history of version control

- Before **version control systems**, developers would track changes using named folders
  - **If a folder is misnamed or overwritten, you are out of luck!**
- Developers began creating local database programs to store old versions of projects, with only one “checked out” version at a time
  - **Did not allow for easy collaboration**
- **Centralized VCS** solved problem by storing project history on a server. How to manage conflicts?
  - CVCS prevents users from overriding others’ work. Conflicts must be managed manually.

So CVCS is great, but there are drawbacks...

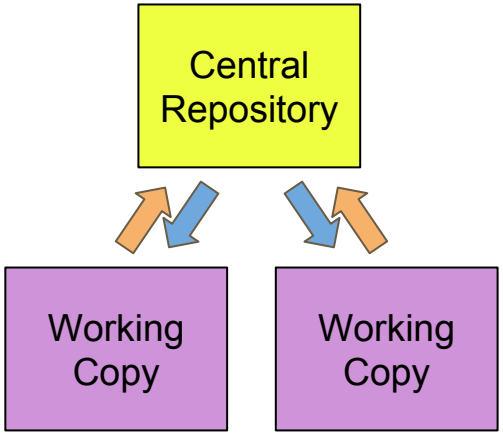
- Since the repository is centralized, a server crash or corrupted database can kill the only copy
- If you don't have a connection to the repo, you can't commit changes (e.g., on an airplane)
- There can be namespace issues with branches
- Can have "access" politics... who is worthy of contributing to the repository?

Enter the distributed version control system!

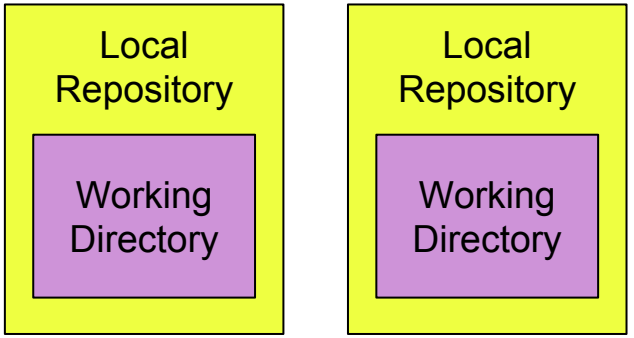
In a DVCS, everyone has their own repository

- All developers have a local copy of the entire project
- Everyone can work at their own pace, and merge with the “official” repo when convenient

CVCS



DVCS



We will discuss how to collaborate using DVCS soon…

Git was created around 2005 to manage version control for the Linux kernel

- Since it was designed for large open-source projects, it is built for speed
- Originated as a command-line program but now many visual interfaces exist as well
- Supports many workflows - more on this later
- Many web-based repository hosts are available

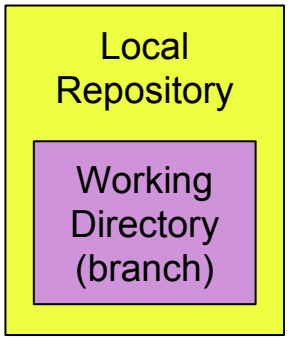
Migration from SVN to Git involves some mental transitions

As we explore Git, do the following:

Think about how a Git workflow/feature compares and contrasts to any SVN equivalent. We will discuss together in two weeks.

Unlike SVN, branches are central to using Git

- Since every user has a local repository, their working directory is part of **a branch**
- Git keeps track of project history at the branch level, not for individual files
- Developers update a repository branch by **committing** a new **snapshot** of the files **staged** in the branch

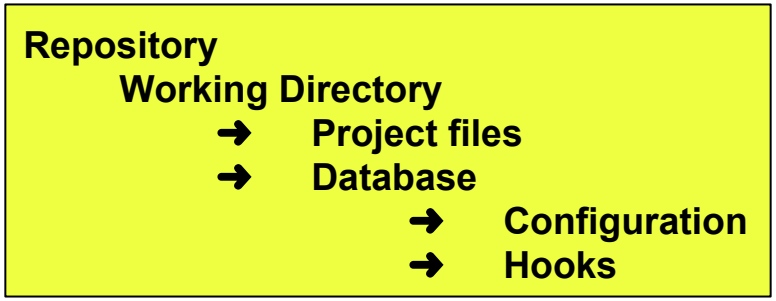


Basic Git workflow in developer's local repository

- Initiate the local repository
  - Create and configure a new repository OR
  - Clone an existing local or remote repository
- Add the initial project state to the local **master branch**
  - Files are modified in **working directory**, then **staged** and then **committed** to the branch history
- Develop new features and/or bug fixes in **topic branches**
- Merge the **topic branch** into the **master branch**

Let's create a new local repository

- All Git commands are prefaced by ***git***
- Navigate to desired path on filesystem, create a folder to contain the repository, and run ***git init***
  - The directory you created is now the repository's **working directory**, from which you will edit project files
  - The init command creates a new hidden directory **.git** within the working directory, which contains database



Configuring the repository for a new user

- All Git users have global account settings
- You can view and edit pertinent ones by:

  git config [--global] user.name [<"First Last">]
  git config [--global] user.email [<address@here.edu>]

- If you leave out the **--global** option, you can configure account settings on a repository basis

Alternatively, you can clone an existing repository and create your own local copy

- So that we are all working on the the same project (i.e. collaborating), let's clone a repository
- Please **clone** a remote repository I've provided using:

  git clone
  https://github.com/vanderwb/roster_site.git
  <local-copy-path>

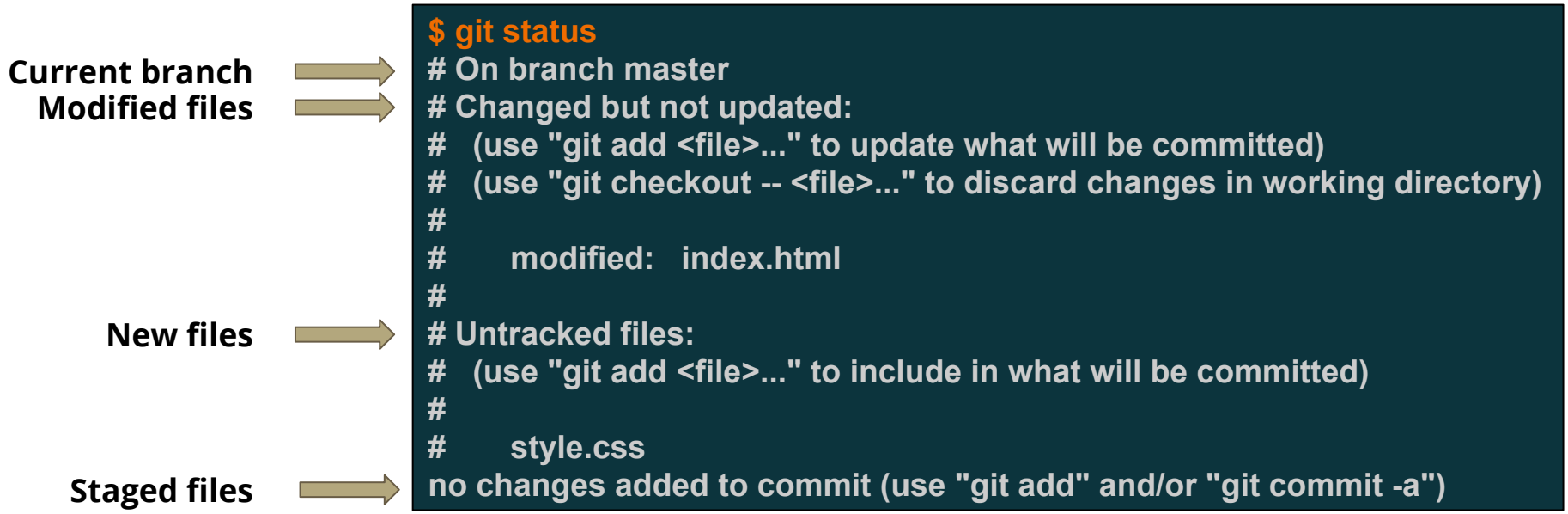
- There are three files in the project so far:
  - roster.html - a listing of attendees to the workshop
  - example.html - a template page for bio information
  - style.css - cascading style-sheet formatting

As with (most) VCS, you make changes to project files within your working directory...

Keeping track of changes in the working directory

- How do we track files we have modified or created?

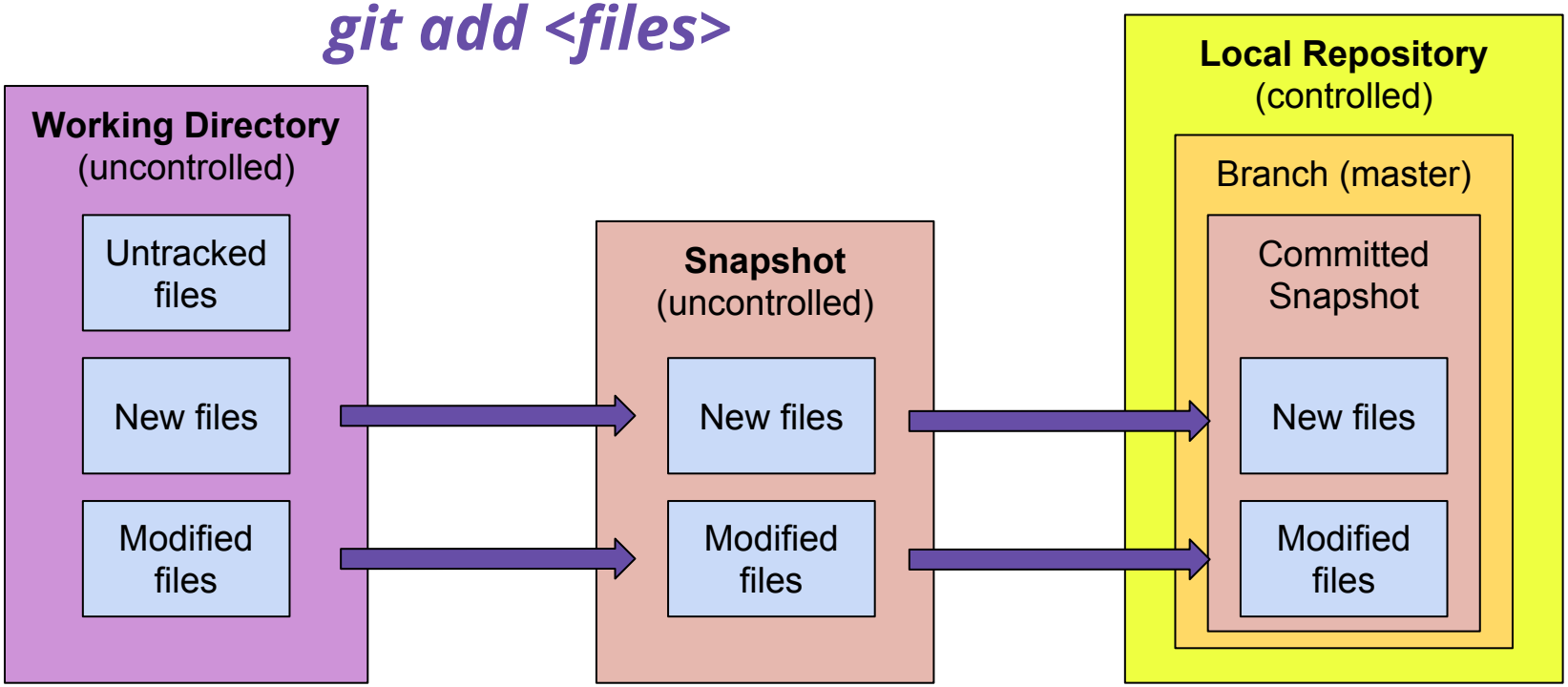
  git status



To commit development to our local repository, we must stage changes in a branch snapshot

To **stage** files:

git add <files>



To **commit** a snapshot:

git commit -m "<message>"

[Remember] *Staged* files are not version controlled!

Anatomy of a Git commit

- A **commit** is simply a **snapshot** of the *entire* branch at one particular time, including any files **staged** from the **working directory**
- Commits are identified by **SHA-1 checksums**, 40-digit hexadecimal hash codes
  - In practice, the first ***seven*** digits are used to specify commits
  - Using the checksums, Git can detect accidental or intentional corruption of a commit
- The most recent commit in a branch is referred to as the **HEAD** commit

Making commits human-readable with tags

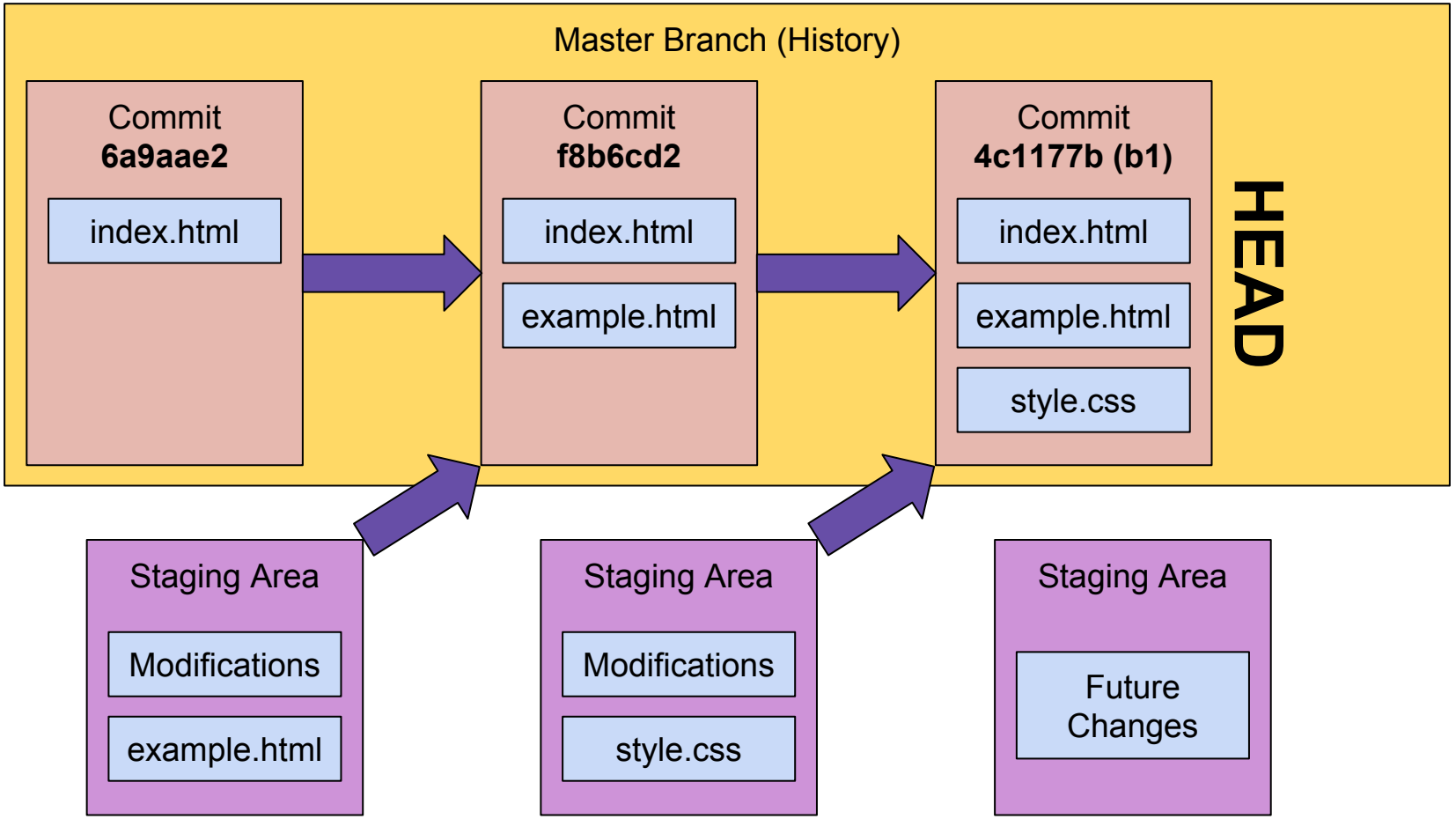
- Unfortunately, **checksums** are both unwieldy and less informative compared to SVN revision numbers
  - commit **f8b6cd2**2c82a43ff750d36b29e270bd27660f2ff
  - commit **6a9aae2**3a6e12c8ce9eb75e58c10359c69895867
- Checksums are not self-documenting for history
- To solve this issue, we can tag the latest commit using:

  git tag -a <name> [-m "<description>"]

  git show <name>

- Tags have limitations though... we'll cover them later

Branches contain the history of all committed snapshots, which contain all existent project files



Log is a powerful command that provides some or all of the commit history of a branch

git log [--oneline] [-n <N>]

```
$ git log
commit 4c1177bd94f312fe616c594e0064241ad684880a
Author: Brian Vanderwende <vanderwb@ucar.edu>
Date:   Thu Mar 10 13:44:40 2016 -0700

    Added CSS formatting to pages

commit f8b6cd22c82a43ff750d36b29e270bd27660f2ff
Author: Brian Vanderwende <vanderwb@ucar.edu>
Date:   Thu Mar 10 12:01:32 2016 -0700

    Updated roster to include example

commit 6a9aae23a6e12c8ce9eb75e58c10359c69895867
Author: Brian Vanderwende <vanderwb@ucar.edu>
Date:   Thu Mar 10 11:02:47 2016 -0700

    Initialized repository with index page
```

```
$ git log --oneline -n 2
4c1177b Added CSS formatting to pages
f8b6cd2 Updated roster to include example
```

Since branches are easy to create in Git, they are used for most forms of development

- **Branches** offer an error-proof method for development
- While a project may have a complex history, each branch has only a linear history
- Only one branch may be **checked out** at a time - meaning the working directory contains one commit from one specific branch
- View all branches using the branch command!

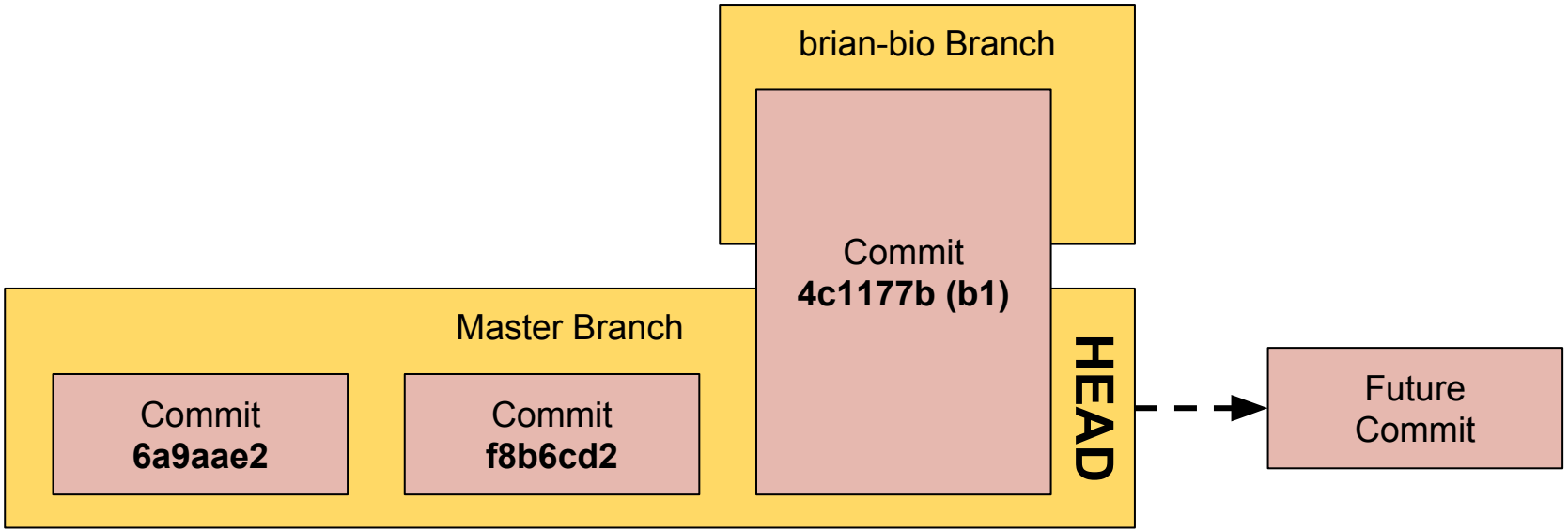
  git branch [-v]

Let's say I want to create a development branch in which I design my bio page...

- Use the branch command with a unique branch name

  git branch <new_branch_name>

  e.g.) ***git branch brian-bio***

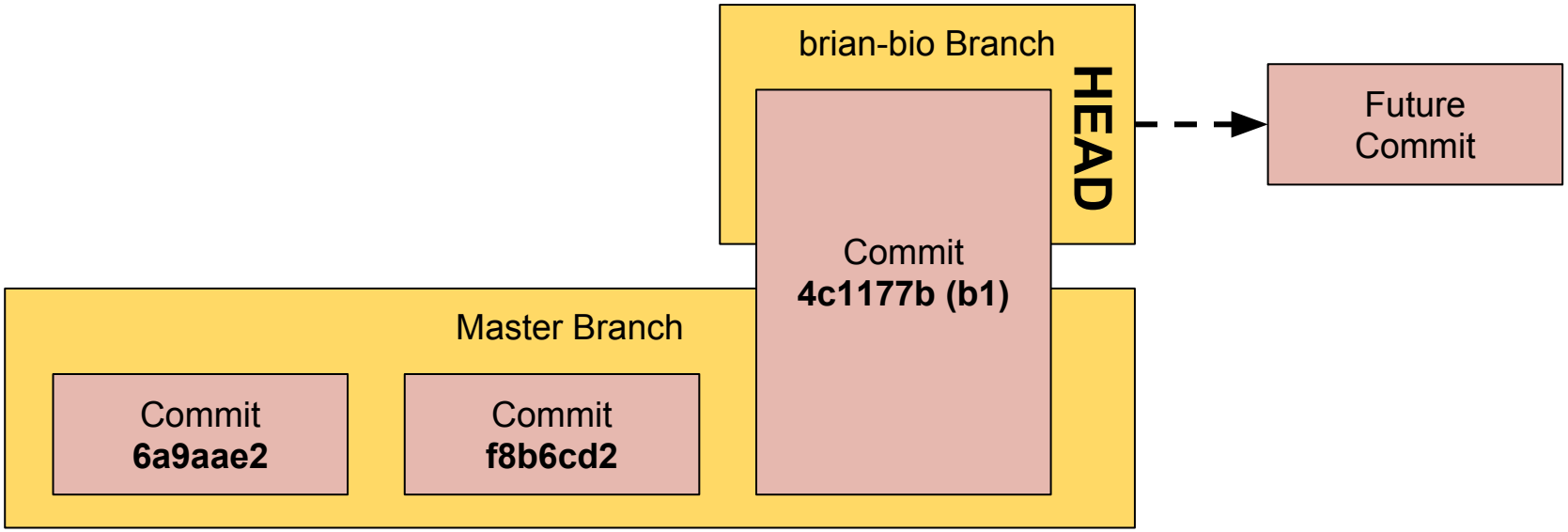


Now that the new branch is created, we need to switch to it to begin development

- We can switch between branches using a **checkout**

  git checkout <branch>

  e.g.) ***git checkout brian-bio***



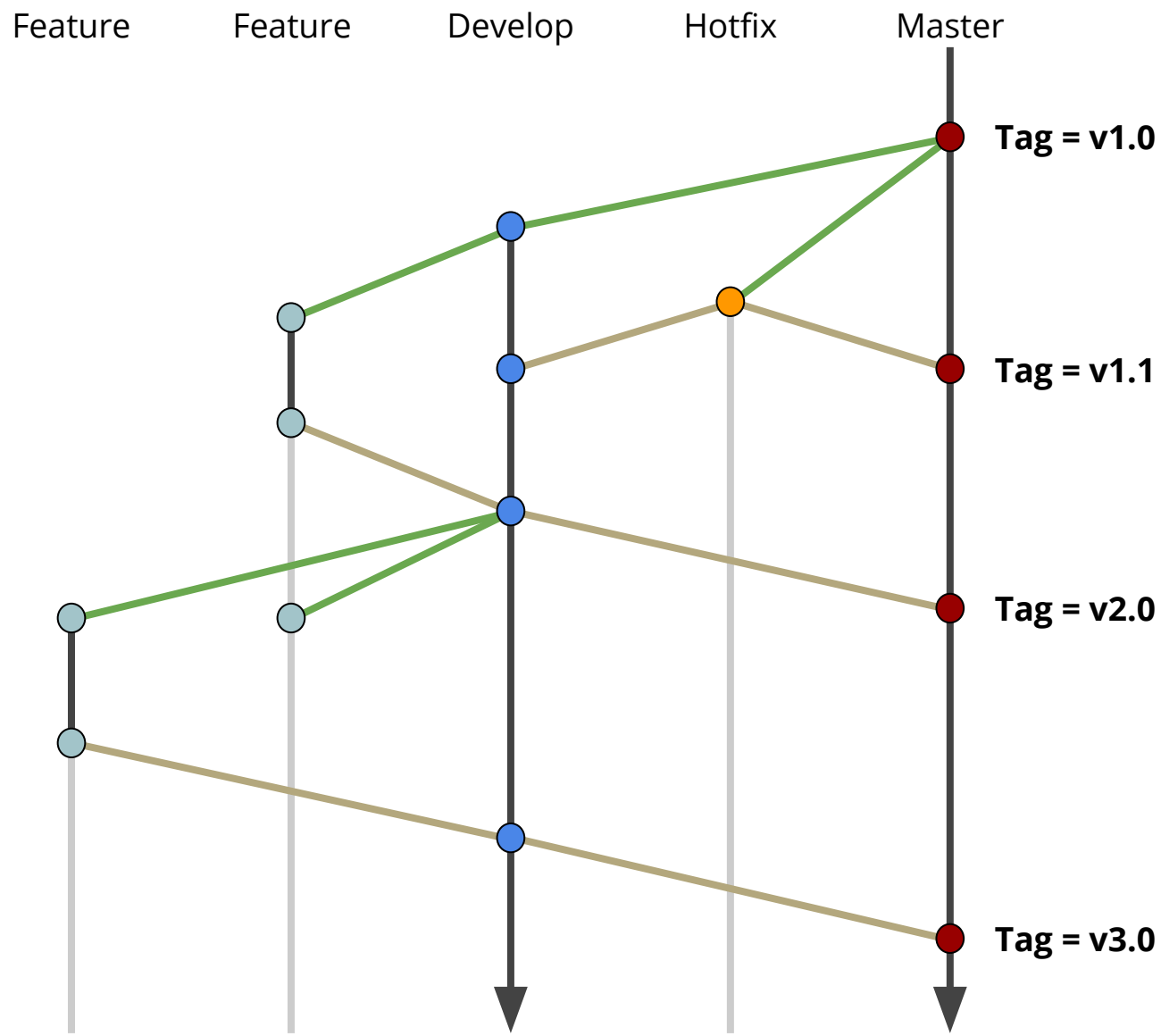
While Git is branch agnostic, it helps to categorize them into specific purposes

1. **Master** branch - the main branch where production-ready code is kept
2. **Develop** branch - the *integration* branch where the latest distributed development is collected
3. **Feature/topic** branches - **fork** off of develop and must **merge** back to develop
    a. Exist until feature is complete or discarded
4. **Hotfix** branches - fork off of master and are used to fix bugs within production code
    a. Should be merged into both master and develop

Let's visualize this branch hierarchy

Branch Operations

- Commit
- Fork
- Merge

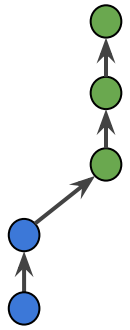


With many interacting branches, merges will be fairly common. How Git handles merges depends on the respective branch histories.

[Scenario] A developer creates a feature branch by forking from the develop branch. After a few commits, she attempts to merge back to develop. There have been no commits in develop since the fork. What happens?

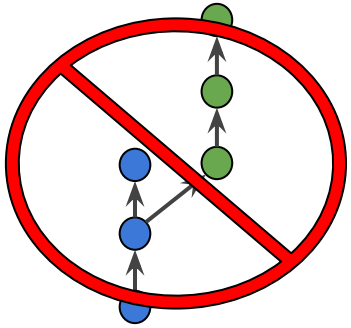
The simplest case is the *fast-forward* merge

- A fast-forward merge occurs when no commits are found in the destination branch after the start of the current branch
  - To put into Git terms, the **HEAD** of the **target** branch is the same commit as the **base** of the **current** branch



Before FF merge

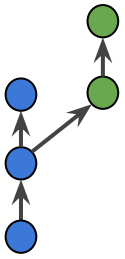
After FF merge



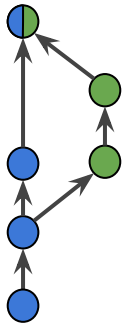
Cannot FF merge

If two branches have diverged, but do not directly conflict, a *three-way merge* is performed

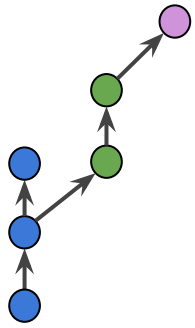
- In a **three-way merge**, two commits are used as parents to a new merge commit
- Repository history will be kept nonlinear, unlike in **fast-forward merges**
  - To some, this is desirable behavior, and so an option exists to disable FF merges altogether



Cannot FF Merge



Three-way merge



What merges are possible here?

Before merging branches, it is a good idea to see what changes (commits) have been made

- The logs can be filtered to see unique commits between two branches:

  git log <branch1>..<branch2> --stat

```
$ git branch
  master
* new-page

$ git log master..HEAD --stat
commit 233392580a9499397318366734f3ab9de0192eae
Author: Brian Vanderwende <vanderwb@ucar.edu>
Date:   Thu Mar 10 23:34:13 2016 -0700

    Added stub for Javascript login system

 login_system.js |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
```

Initiating a merge is simple

- First, checkout the branch you wish to merge commits to (the target branch)
- Then, from the target, run the following command:

  git merge [-no-ff] <source_branch>

- After a merge, if the source branch is redundant (e.g., a **feature branch**), it can be deleted as follows:

  git branch -d <source_branch>

[Scenario] What if two branches diverge with changes that directly conflict? How do we merge?

Some rules of thumb for branches

1. **Create** a new branch for each major feature addition to your project
2. **Do not create** a branch if you can't come up with a specific, succinct name for it
3. If you are collaborating with others, **do not create** branches for each developer. Remember that:
   a. **Repositories are for people**
   b. **Branches are for development**

For more information, check out:

https://git-scm.com/doc

http://rypress.com/tutorials/git/index

http://nvie.com/posts/a-successful-git-branching-model/

http://ariya.ofilabs.com/2013/09/fast-forward-git-merge.html

My contact information:

Brian Vanderwende
CISL Consulting Services Group
ML-55L (x2442)
vanderwb@ucar.edu