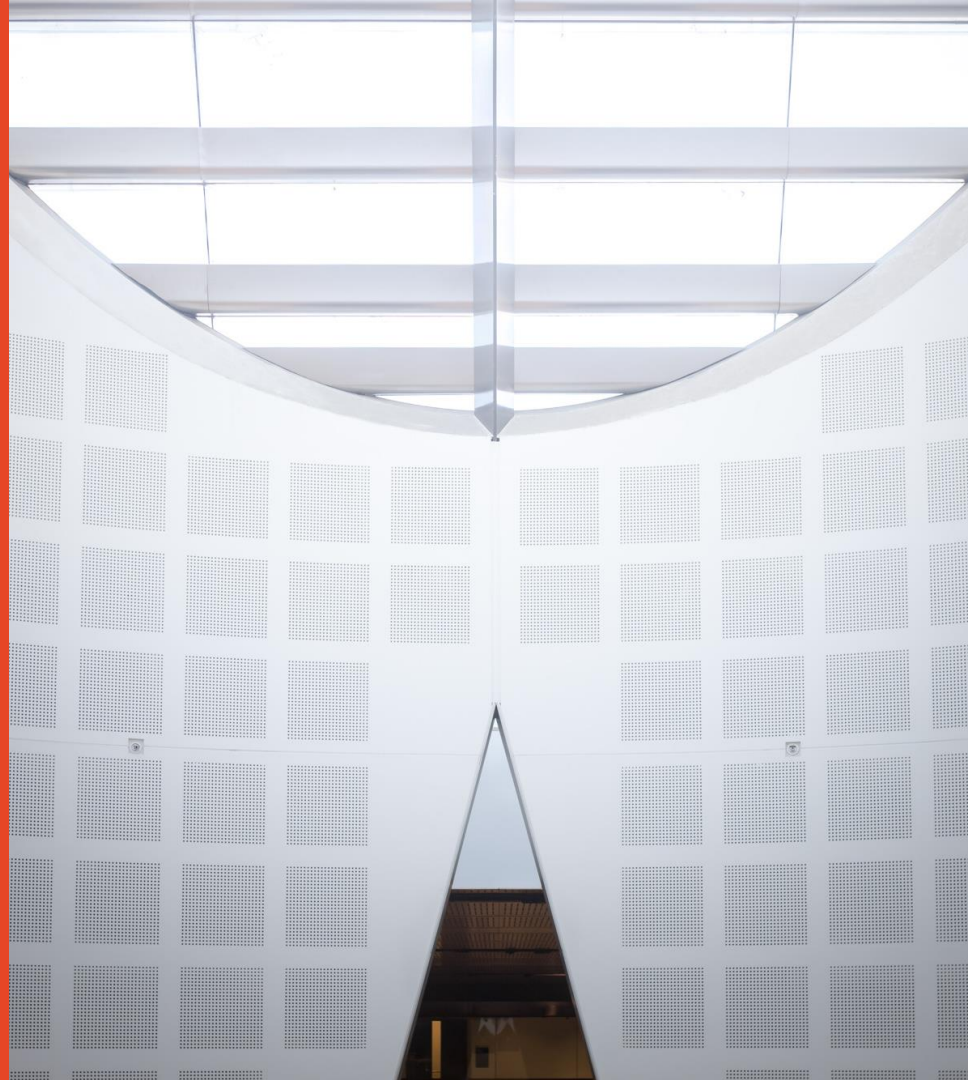


Agile Software Development Practices SOF2412 / COMP9412

Tools and Technologies for
Controlling Artefacts

Dr. Basem Suleiman

School of Information Technologies



Agenda

- Working with Remote Repository
- GitHub
- Distributed Git
 - Remote Branches
 - Collaboration – Workflows

Remote (Hosted) Repository

- A remote repository is generally a *simple repository* – the contents of your project's .git directory and nothing else
- When you really need to work with remote repository?
- **One person project**
 - Local repository should suffice – includes working directory
 - Track changes and history of development as individual
- **Team-based (collaboration) projects**
 - Remote repo team members (collaborators) can access anytime
 - More reliable common repo (rather own local repo)
 - All team members can push and pull
 - Need to have some coordination and permission control

Remote Repository – Running Own Server

- Hosting our code/projects on your own server
 - Configure which protocols your server to communicate with
 - Typical server set-ups using the configured protocols

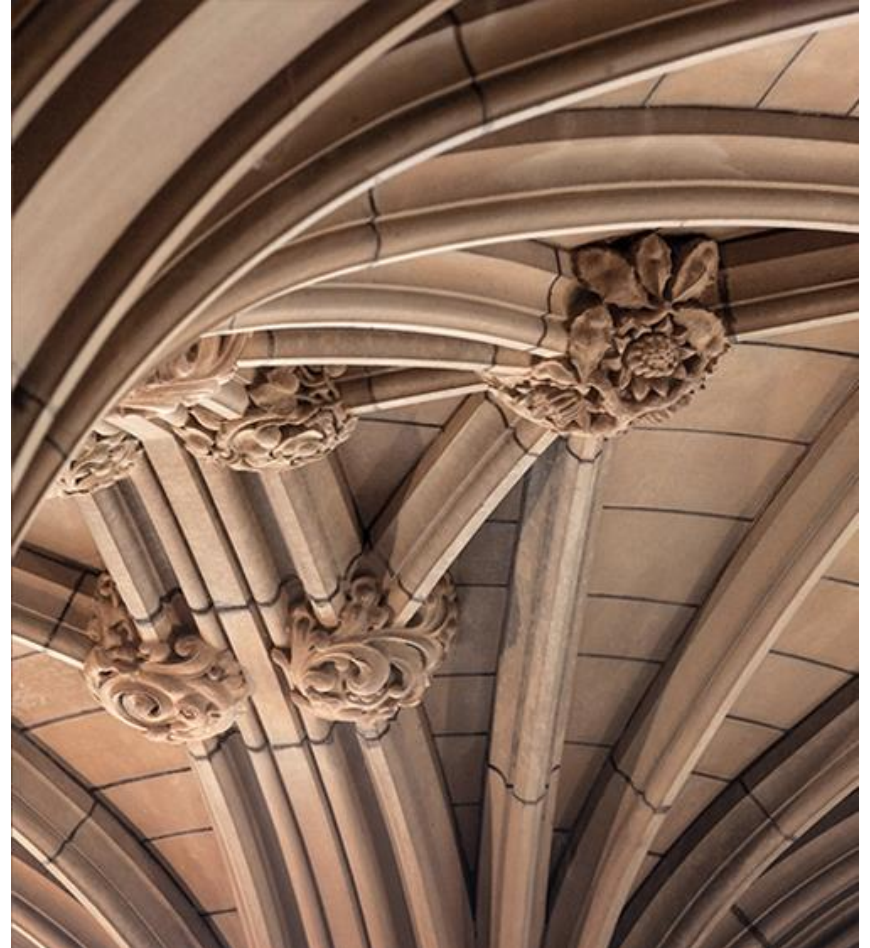
Protocol	Pros	Cons
File system	simple, support access control	public share is difficult to setup
SSH	easy to setup (most systems provide ssh tools), fast (compress data), support authenticated write access	no anonymous access (even read access)
HTTP	unlikely to be blocked	Can become difficult to setup
Git	Fastest protocol, allow anonymous public access	Difficult to setup, lack of authentication, use non standard port (9418) which can be blocked

Remote Repository – Running Hosted Server

- Set-up your project/code directory on a hosted server (Git server)
 - No major concerns about security or privacy
 - Avoid the hassle of setting and maintaining your own server
- Many hosting services including *GitHub, GitLab, BitBucket, Mercurial*
 - Not Git itself but a hosting service for Git repos
 - Host your own projects and open it up for collaboration
 - You can create organization, teams and repos
 - Web-based and desktop/command-line interactions
 - **Public Repos/projects:** provides all GitHub functionality with free accounts but all projects are public (everyone has read access)
 - **Private Repos/projects:** protected access and full control but it is not free with some hosting services

Hosted Git Service

GitHub



Hosted Servers – GitHub

- There are large number of Git hosting options
 - We will focus on Github as it is the largest Git host
- Create one-user (personal) account
- Public and private repos

GitHub – Organizations

- GitHub allows groups of people to collaborate across many projects at the same time in organizations account via **organization** account
 - Group of people with shared ownership of projects
- Organization's members can have owners or member roles:
 - **Owner:** have complete administrative access to the organization (often a few people in the organization should be assigned as owner roles)
 - **Member:** default role for everyone else
- Owners can manage members' access to the organization's repos. and projects with fine-grained permission controls
 - Create your own organization
 - Understand and carefully manage members access to your organization
- Can add collaborators from outside of the organization (consultant) to have access to one or more organization repos. without bring a member of the organization

GitHub – Organization Access Control

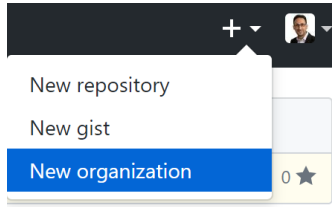
Organization action	Owners	Members
Invite people to join the organization	X	
Edit and cancel invitations to join the organization	X	
Remove members from the organization	X	
Reinstate former members to the organization	X	
Add and remove people from all teams	X	
Promote organization members to <i>team maintainer</i>	X	
Add collaborators to all repositories	X	
Access the organization audit log	X	
Delete all teams	X	
Delete the organization account, including all repositories	X	

Organization action	Owners	Members
Create teams	X	X
See all organization members and teams	X	X
@mention any visible team	X	X
Can be made a <i>team maintainer</i>	X	X
Transfer repositories	X	
View a project board and add or reorganize its cards and columns	X	X
Create or delete a project board and edit its description	X	X
Automate actions for project boards	X	X
View and post private team discussions to all teams (see " About team discussions " for details)	X	
Edit and delete team discussions in all teams (for more information, see " Managing disruptive comments ")	X	

– Examples of access permissions for organization's owners and members

<https://help.github.com/enterprise/2.13/user/articles/permission-levels-for-an-organization/>

Github – Creating Organization



Secure | <https://github.sydney.edu.au/organizations/new>

Enterprise Search GitHub Pull requests Issues Explore

Sign up your team

✓ Completed
Create personal account

Step 2:
Create organization

Step 3:
Add members

Create an organization account

Organization name

✓

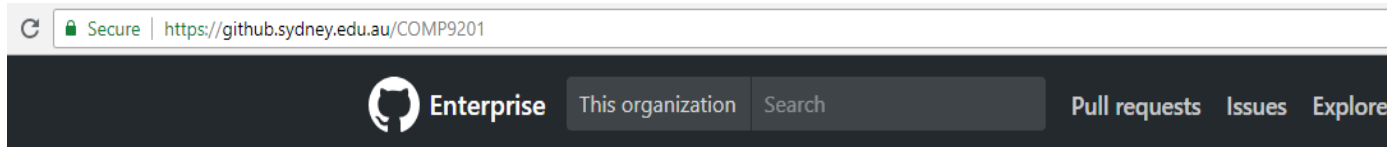
This will be your organization name on <https://github.sydney.edu.au/COMP9201>.

Contact email

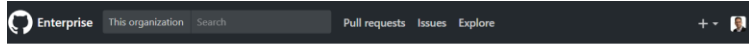
Create organization

Organization accounts allow your team to plan, build, review, and ship software — all while tracking bugs and discussing ideas.

Organizational accounts have a namespace where all their projects exist



GitHub – Add Members to Organization



Add organization members

✓ Completed
Create personal account

👤 Step 2:
Create organization

📋 Step 3:
Add members

Search by username, full name or email address

hoso5448

hoso5448 Hamzah Bin Osop

Farnaz Farid

Finish

Organization members

✓ See all repositories ⓘ

✓ Create repositories

✓ Organize into teams

✓ Review code

✓ Communicate via @mentions

As an organization owner, you'll have complete access to all of the organization's repositories and have control of what members have access using fine-grained permissions.

- Note: when you create a new repo you can create them under your personal account or under any of the organizations that you're owner in

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

Repository name

COMP9201

/

WebStoreApp

Choose another owner

bsul6138

✓ COMP9201

INFO3220-Object-Oriented-Design

PROFESSIONAL-Software-Engineering

SOFT2201-Software-Constr-and-Design-1

SOFT2412-Agile-Software-Development

Need inspiration? How about

choose who can commit.

repository.

GitHub Organization – Manage Repos

The screenshot shows the GitHub organization page for COMP9201. The top navigation bar includes 'Enterprise', 'This organization', 'Search', 'Pull requests', 'Issues', and 'Explore'. The main header displays the organization name 'COMP9201' and a message: 'This organization has no repositories.' with a 'Create a new repository' button. Below this, there are tabs for 'Repositories', 'People', 'Teams', 'Projects', and 'Settings'. The 'People' tab is active, showing two members: 'bsul6138 Basem Fathi Suleiman' and 'ffar6831 Farnaz Farid'. On the left, there is a list of repositories under the 'COMP9201' organization, including 'Front-end', 'Designs', 'Back-end', and 'WebStoreApp', each with a 'Private' label and a description.

The screenshot shows the GitHub repository page for 'COMP9201 / WebStoreApp'. The top navigation bar includes 'Enterprise', 'This repository', 'Search', 'Pull requests', 'Issues', and 'Explore'. The repository name 'COMP9201 / WebStoreApp' is displayed with a 'Private' label. Below the repository name, there are tabs for 'Code', 'Issues', 'Pull requests', 'Projects', 'Wiki', 'Insights', and 'Settings'. The 'Settings' tab is active, showing the 'Teams' section. The 'Teams' section lists three teams: 'FrontEndDeve' (Front-end development team, 2 members, Read permission), 'BackEndDeve' (Back-end Development Team, 2 members, Admin permission), and 'Designers' (Web Application Designers, 1 member, Read permission). There is a '+ Create new team' button and an 'Add a team: Select team' dropdown.

GitHub Organization – Manage People

COMP9201

Repositories 0 People 3 Teams 0 Projects 0 Settings

Find a member...

Members Outside collaborators Add member

Select all	2FA	Role	0 teams
Basem Fathi Suleiman bsul6138	2FA X	Private Owner	0 teams
Farnaz Farid ffar6831	2FA X	Private Member	0 teams
Hamzah Bin Osop hoso5448	2FA X	Private Member	0 teams

- Manage
- Change role...
- Convert to outside collaborator
- Remove from organization

Enterprise This organization Search Pull requests Issues Explore

COMP9201

Repositories 4 People 3 Teams 3 Projects 0 Settings

ffar6831 has access to 3 repositories

Find a repository they have access to...

Repository	Access Level	Manage access
COMP9201/WebStoreApp	Read on this repository	Manage access
COMP9201/Designs	Write on this repository	Manage access
COMP9201/Front-end	Admin on this repository	Manage access

ffar6831 Farnaz Farid

Role: Member

3 repositories

1 team




Membership private

Two-factor security disabled

Convert to outside collaborator

Remove from organization

GitHub Organization – Manage Teams

Find a team...	Import teams	New team
Select all		
Visibility Members		
BackEndDeve Back-end Development Team	 2 members	0 teams
Designers Web Application Designers	 1 member	0 teams
FrontEndDeve Front-end development team	 2 members	0 teams

You may have 3 repos; Designs, Front-end and Back-end. You want FrontEndDeve to work on the Front-end and Designs repos, Designers team to work on Designs repo and BackEndDeve to work on Back-end repo

COMP9201 / BackEndDeve

Discussions

Members 2

Teams 0

Repositories 2

Find a repository...

Add repository

Select all		
COMP9201/Back-end	Private	Admin
updated 21 minutes ago		
COMP9201/WebStoreApp	Private	Admin
updated an hour ago		

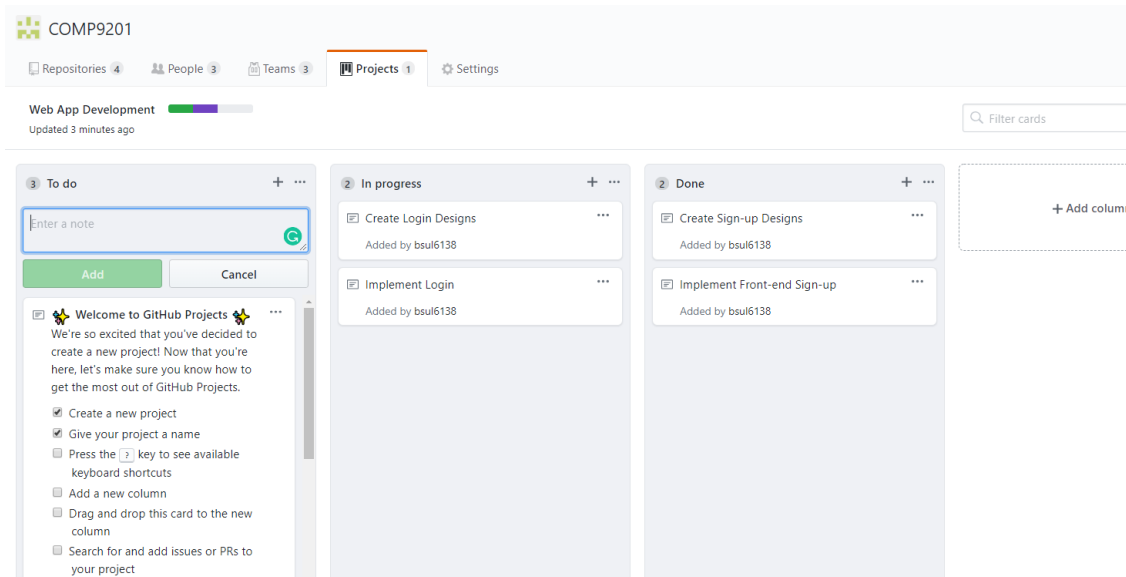
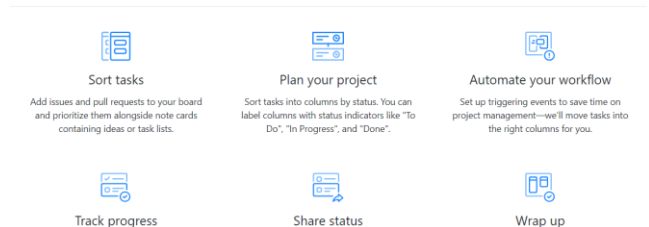
GitHub Organization – Manage Projects



Organize your issues with project boards

Did you know you can manage projects in the same place you keep your code? Set up a project board on GitHub to streamline and automate your workflow.

[Learn More](#) [Create a project](#)



GitHub Organization – Audit Log

- Audit log records all events that have happened at the organization level, who did them and where in the world they were done

The screenshot shows the GitHub interface for the organization COMP9201. The top navigation bar includes links for Repositories (4), People (3), Teams (3), Projects (1), and Settings. The left sidebar lists various settings categories: Organization settings (Profile, Member privileges, Security, Audit log, Hooks, Installed GitHub Apps, Repository topics, Projects, Teams) and Developer settings (OAuth Apps, GitHub Apps). The 'Audit log' section is active, displaying a list of recent events. Each event entry includes a user profile icon, a title (e.g., 'project.create', 'team.add_repository'), a description of the action, and the location (e.g., 'Australia') and time (e.g., 'an hour ago').

COMP9201

Repositories 4 People 3 Teams 3 Projects 1 Settings

Organization settings

- Profile
- Member privileges
- Security
- Audit log
- Hooks
- Installed GitHub Apps
- Repository topics
- Projects
- Teams

Developer settings

- OAuth Apps
- GitHub Apps

Audit log

Filters Search audit logs

Recent events

- bsul6138 – project.create**
Created project [Web App Development](#) in COMP9201
Australia | an hour ago
- bsul6138 – team.add_repository**
Gave COMP9201/frontenddev access to COMP9201/Designs
Australia | an hour ago
- bsul6138 – team.add_repository**
Gave COMP9201/frontenddev access to COMP9201/WebStoreApp
Australia | an hour ago
- bsul6138 – team.add_repository**
Gave COMP9201/designers access to COMP9201/WebStoreApp
Australia | an hour ago
- bsul6138 – team.add_repository**
Gave COMP9201/backenddev access to COMP9201/WebStoreApp
Australia | an hour ago

Distributed Git

Remote Branches



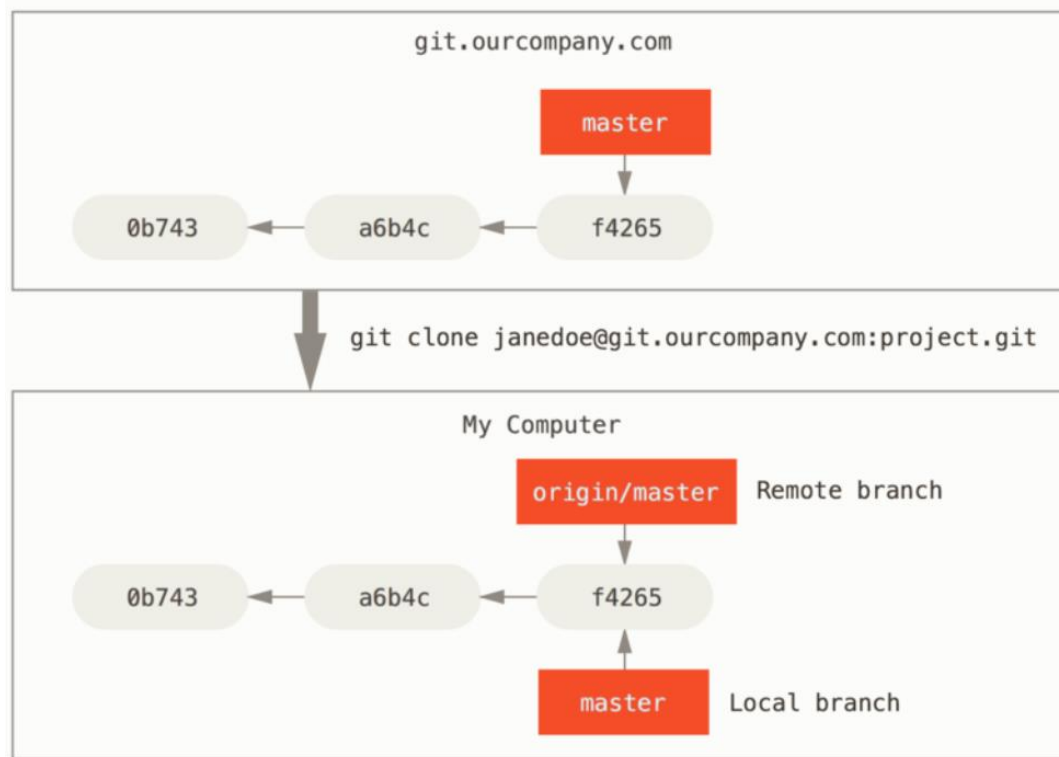
Remote Branches

- **Remote references:** references (pointers) in your repos. (branches, tags, etc)
 - To get full list of remote references use `git ls-remote [remote]`
 - To get list of remote branches use `git remote show [remote]`
- **Remote-tracking branches:** are references to the state of remote branches
 - They are local references that cannot move; Git moves them for you to make sure they accurately represent the state of the remote repo (like bookmarks)
 - Take the form `<remote> / <branch>`
 - E.g., check the `origin/master` branch if you want to see the master branch on your origin remote look like

Remote-Tracking Branches – Example

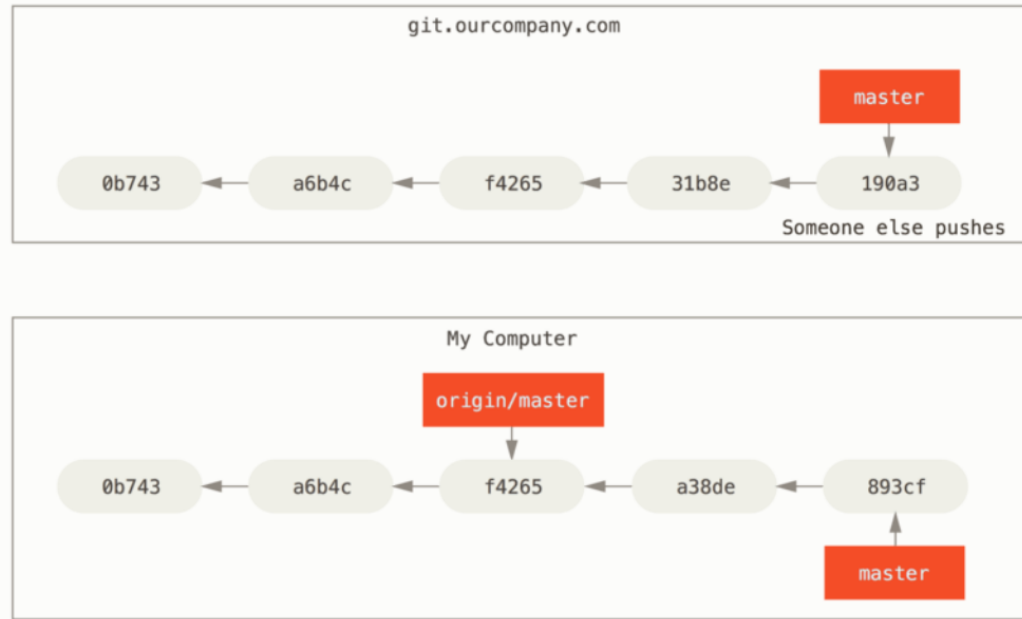
- You have a Git server on your network (git.ourcompany.com)
- If you clone from this, Git's clone command automatically:
 - Names it **origin**
 - Pulls down all its data
 - Creates a pointer to where its master branch is
 - Names it origin/master locally on your PC
 - Set your own local master branch starting at the same place as origin's master branch
- **Note:** origin is not special; origin is the default name for a remote when you run git clone (like “master” – default name for a starting branch when you run git init)
 - Run *git clone -o MyBranch* then MyBranch/master is your default remote branch

Remote-Tracking Branches – Clone Remote Repo



Local and Remote Branches

- Imagine you do some work on your local branch, while another developer pushes updates to the master branch of `git.company.com`?

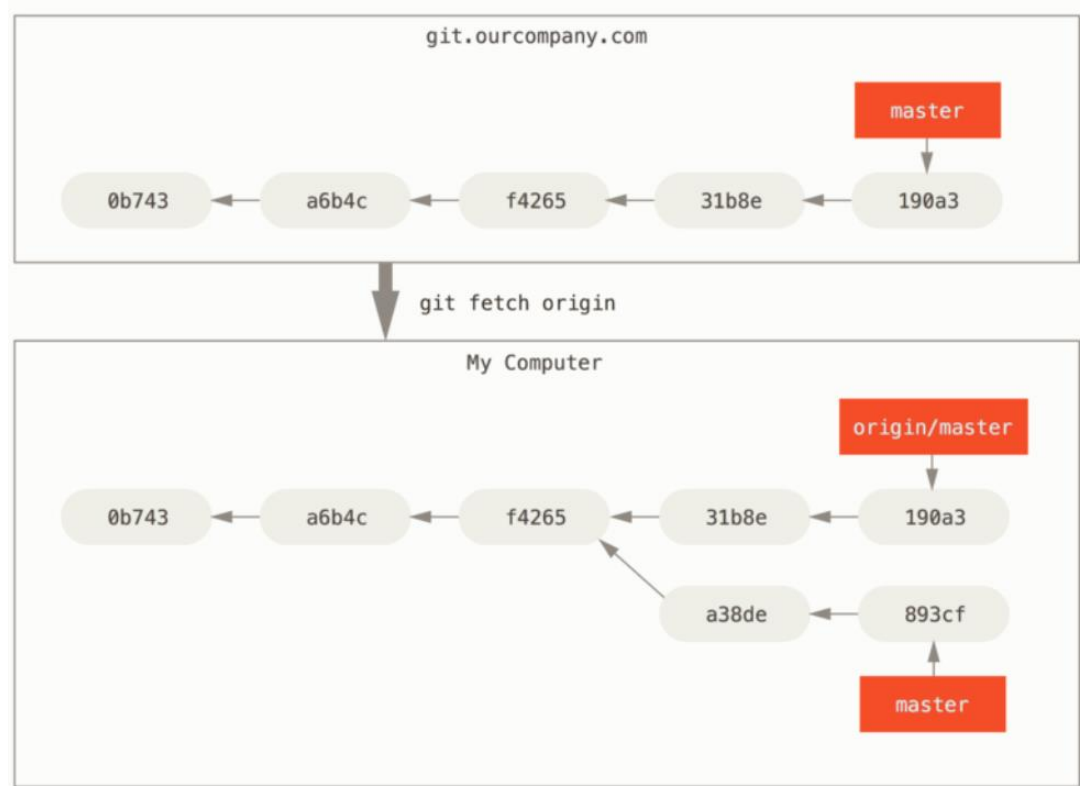


Local and Remote Branches – Synchronization

To sync. Your work run

Git fetch origin

Looks up which server origin, fetches data you do not have yet and update your local repo – moving your origin/master pointer to its new (up-to-date) position



Remote-Tracking Branches – Pushing

- If you want to share a branch with the world, you need to explicitly push it up to a remote that you have write access to
- E.g., to share *serverfix* branch to work on with others,

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
* [new branch]      serverfix -> serverfix
```

“Take my serverfix local branch and push it to update the remote’s serverfix branch.”

`git push origin serverfix:serverfix`

“Take my serverfix and make it the remote’s serverfix

Remote-Tracking Branches – Pushing

- When a collaborator fetches from the server they will get a reference to where the server's version of serverfix is under the remote branch `origin/serverfix`

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
* [new branch]      serverfix    -> origin/serverfix
```

Here, the collaborator will not have a new serverfix branch — they only have an `origin/serverfix` pointer that they can't modify

Remote-Tracking Branches – Merge/Base

- To merge this work into your working branch, you can run `git merge origin/serverfix`
- To work on your own serverfix branch you can base it off your remote-tracking 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

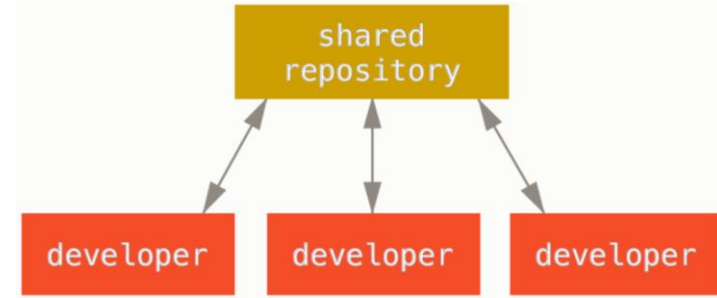
Distributed Git

Distributed workflows



Centralized VCSs

- *Single collaboration model* (centralized workflow)
 - Every developer is a node working on a central shared repo. and sync. to it



Scenario: two developers using Git; Joe and Sarah clone from the shared repo. and both make changes to some files locally.

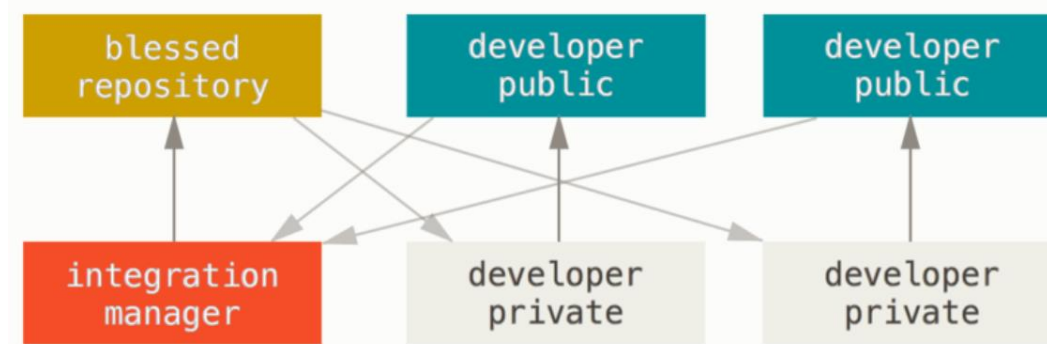
- What happens when Joe pushes his changes to the repo. first?
- What happens when Sarah pushes her changes after Joe?
 - The server will reject the changes.
 - Sarah must first fetch the Joe's changes from the server and merge it locally before pushing the merged changes
- If centralized workflow suits your company/team, create a single repo. and give every team member **push access** (Git will prevent overwriting content)
- Not limited to small teams; Git branching allows hundreds of developers to work on a single project through many branches simultaneously

Distributed VCS

- In Git, every developer can be both
 - Node: can contribute code to other repos.
 - Shared repo.: maintain a public repo. on which others can base their work and which they can contribute to
- This model opens a wide range of workflow possibilities for projects/teams
- In the next, we will cover some common designs and discuss pros and cons of each

Integration-Manager Model

- Often includes a canonical repo. that represents the “official” project
 - Each developer has write access to their own public repo. and read access to everyone else’s
 - Developers make their own public clone of the project and push their changes to it
 - Then they inform the maintainer of the main project pull their changes
 - The maintainer add developer’s repo as a remote, test changes locally merge them into the branch and push back to the their repo



Integration-Manager – Process Workflow

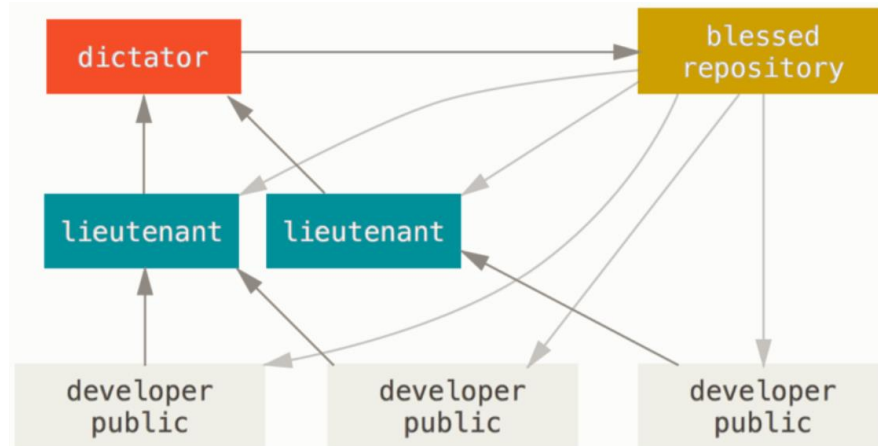
1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an email asking them to pull changes.
5. The maintainer adds the contributor's repository as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

Integration-Manager – Pros

- Very common workflow in hosted servers such as GitHub and GitLab
- Easy to fork a project and push your changes into your fork for everyone to see
- Developers can continue to work on their repos. while the maintainer of the main repo. can pull their changes at anytime
- Contributors do not have to wait for the project to incorporate their changes
 - each can work on their pace

Dictator and Lieutenants Workflow

- Variation of multiple-repository workflow suited for huge projects (hundreds of collaborators), famous example, Linux Kernel
 - **Lieutenants** various integration managers are in charge of certain parts of the repo.
 - **Benevolent dictator** All Lieutenants have one integration manager
 - The benevolent dictator pushes from his directory to a reference repository from which all the collaborators need to pull



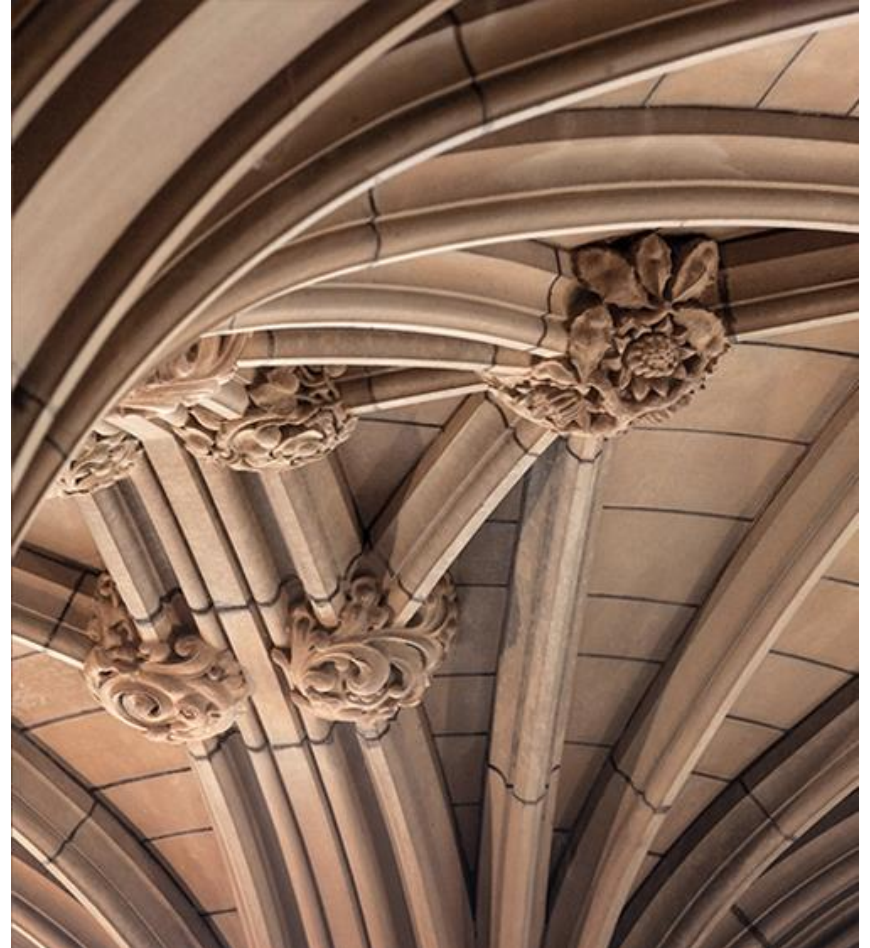
Dictator and Lieutenants - Process Workflow

- Regular developers work on their topic branch and rebase their work on top of master. The master branch is that of the reference repository to which the dictator pushes
- Lieutenants merge the developers' topic branches into their master branch.
- The dictator merges the lieutenants' master branches into the dictator's master branch.
- Finally, the dictator pushes that master branch to the reference repository so the other developers can rebase on it.

Dictator and Lieutenants – Use

- This kind of workflow is useful for very big projects or in in highly hierarchical environments
- It allows project leader (the dictator) to delegate much of the work and collect large subsets of the code at multiple points before integrating them

Contributing to a Project



Contributing to a Project (1)

- Teams can contribute to a Git project in various ways (as Git is flexible)
- A number of factors affect how one can contribute effectively to a project
- **Active contributor count:** how many users are actively contributing code to this project, and how often?
 - Small projects: e.g., 2-3 developers with a few commits a day
 - Large projects: probably hundreds/thousands of developers with hundreds /thousands of commits each day
 - What is the relationship between number of developers and commits? Commits and potential conflict/merge issues?
 -

Contributing to a Project (2)

- **Project workflow:** what project workflow is utilized?
 - Centralized with equal write access to main code-line?
 - Does the project have a maintainer or integration manager who check all commits?
 - Is a lieutenant system in place and do you have to submit your work to them first?
- **Commit access:** how the type of commit access would affect the contribution workflow?
 - Do you have write access?
 - If not, is there a policy on how the contributed work is accepted?
 - How much work a developer may contribute at a time? and how often

Contributing to a Project – Commit Guidelines

- Working with Git and collaborating with others could be more effective when adopting and following good guidelines for commits
- **No whitespace errors:** run `git diff --check` before commits to identify and list possible whitespace errors
- **Commit logically separate changeset:** do not work on many different issues in your code and submit them as one commit!
- **Use quality commit messages:** a concise description of the change followed by a blank line then a detailed explanation
- Git has a full guide for commits described in [Git source code](#)

Contributing to a Private Small Project (1)

- Private (private repo, not open source) project with few developers all have push access to the repo
- Centralized workflow with offline committing and simple branching and merging
- Scenario: 2 developers working on a shared repo.
 - John clones the repo., make a change and commits locally
 - Jessica clones the repo., make a change and commits locally
 - Jessica pushes her work to the server, and this should work fine
 - shortly afterwards, John makes some changes, commits them to his local repository, and tries to push them to the same server
 - John's push fails because of Jessica's earlier push of her changes

Contributing to a Private Small Project (2)

```
# John's Machine
$ git clone john@github:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'remove invalid default value'
[master 738ee87] remove invalid default value
1 files changed, 1 insertions(+), 1 deletions(-)
```

1

```
# Jessica's Machine
$ git clone jessica@github:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
1 files changed, 1 insertions(+), 0 deletions(-)
```

2

```
# Jessica's Machine
$ git push origin master
...
To jessica@github:simplegit.git
1edee6b..fbff5bc master -> master
```

3

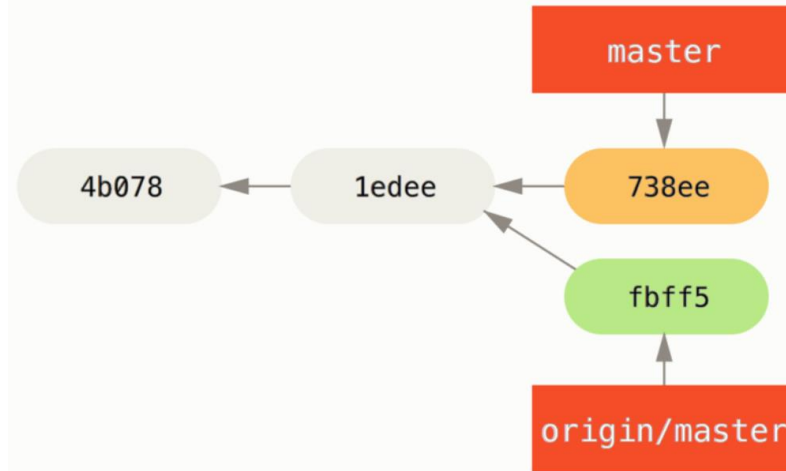
```
# John's Machine
$ git push origin master
To john@github:simplegit.git
! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@github:simplegit.git'
```

4

Contributing to a Private Small Project (3)

- John fetches Jessica's work (this only *fetches* Jessica's upstream work, it does not yet merge it into John's work)

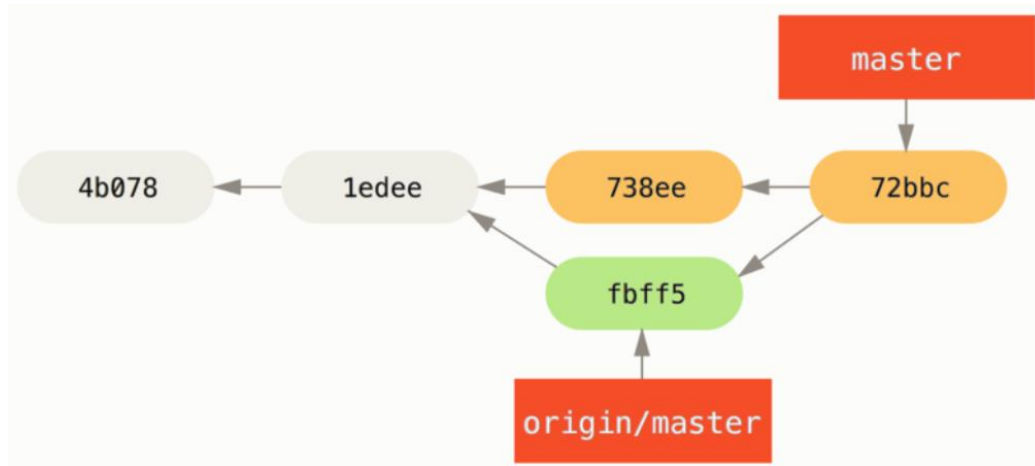
```
$ git fetch origin
...
From john@github:simplegit
+ 049d078...fbff5bc master    -> origin/master
```



Contributing to Private Small Project (4)

- Now John can merge Jessica's work that he fetched into his own local work:

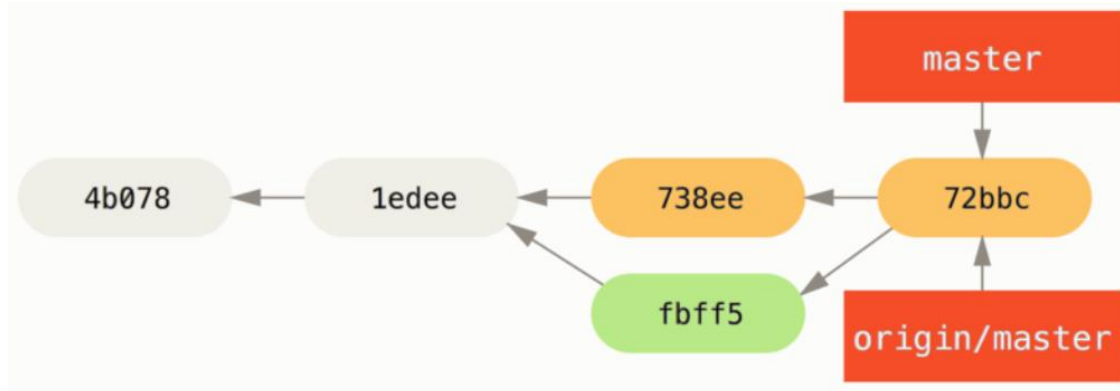
```
$ git merge origin/master
Merge made by the 'recursive' strategy.
TODO | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```



Contributing to Private Small Project (5)

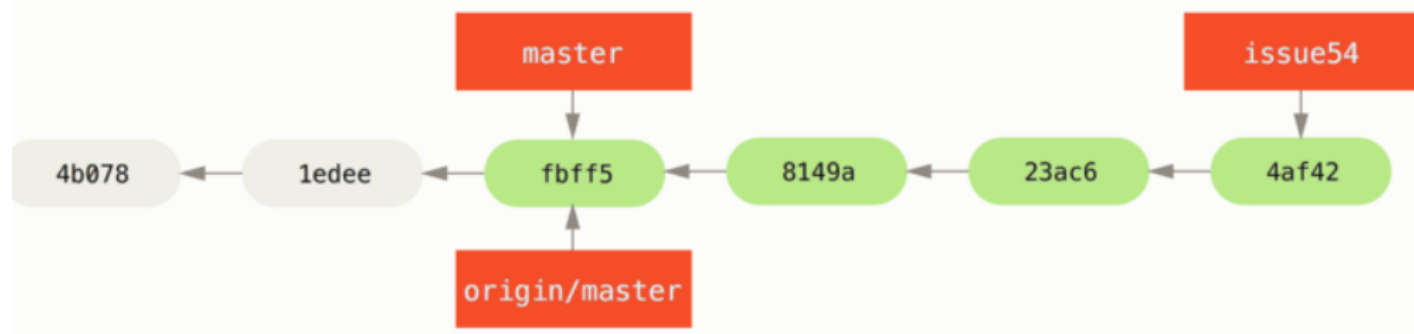
- Now, John might want to test this new code to make sure none of Jessica's work affects any of his and, he can finally push the new merged work up to the server (as long as everything seems fine):

```
$ git push origin master
...
To john@githost:simplegit.git
    fbff5bc..72bbc59  master -> master
```



Contributing to Private Small Project (6)

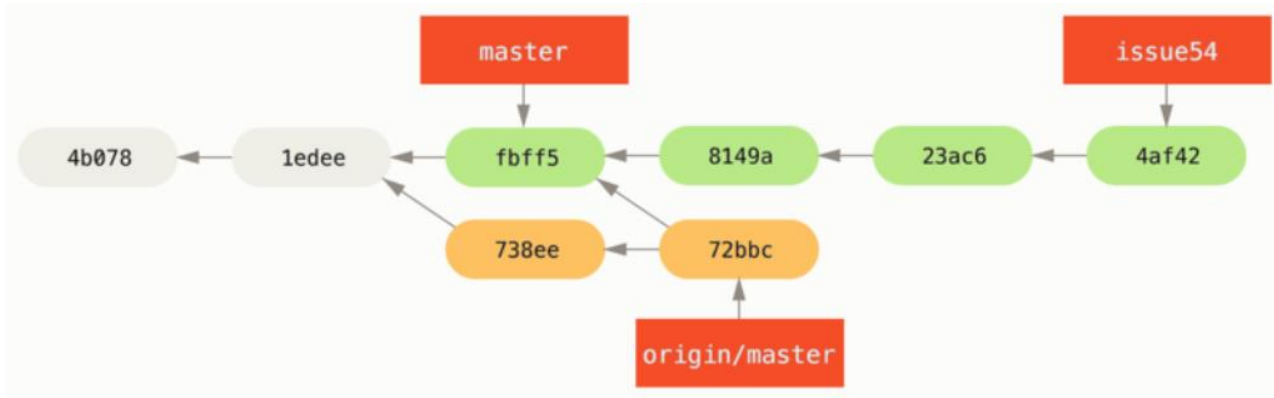
Meanwhile, Jessica created a new topic branch *issue54*, and made three commits to that branch. She hasn't fetched John's changes yet, so her commit history looks like this



Contributing to Private Small Project (7)

Suddenly, Jessica learns that John has pushed some new work to the server and she wants to take a look at it, so she can fetch all new content from the server that she does not yet have with:

```
# Jessica's Machine
$ git fetch origin
...
From jessica@github:simplegit
 fbff5bc..72bbc59  master    -> origin/master
```



Contributing to Private Small Project (8)

Jessica thinks her topic branch is ready, but she wants to know what part of John's fetched work she has to merge into her work so that she can push

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    remove invalid default value
```

The **issue54..origin/master** syntax is a **log filter** that asks Git to display only those commits that are on the latter branch (i.e., origin/master) that are not on the first branch (i.e., case issue54)

The output tells there is a single commit that John has made that Jessica has not merged into her local work.

If she merges origin/master, that is the single commit that will modify her local work.

Contributing to Private Small Project (9)

Now, Jessica can merge her topic work into her master branch, merge John's work (origin/master) into her master branch, and then push back to the server again

First Jessica switches back to her master branch in preparation for integrating all of her work on issue54 topic branch:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Jessica can merge either origin/master or issue54 first — they're both upstream, so the order doesn't matter

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README          |      1 +
 lib/simplegit.rb |      6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

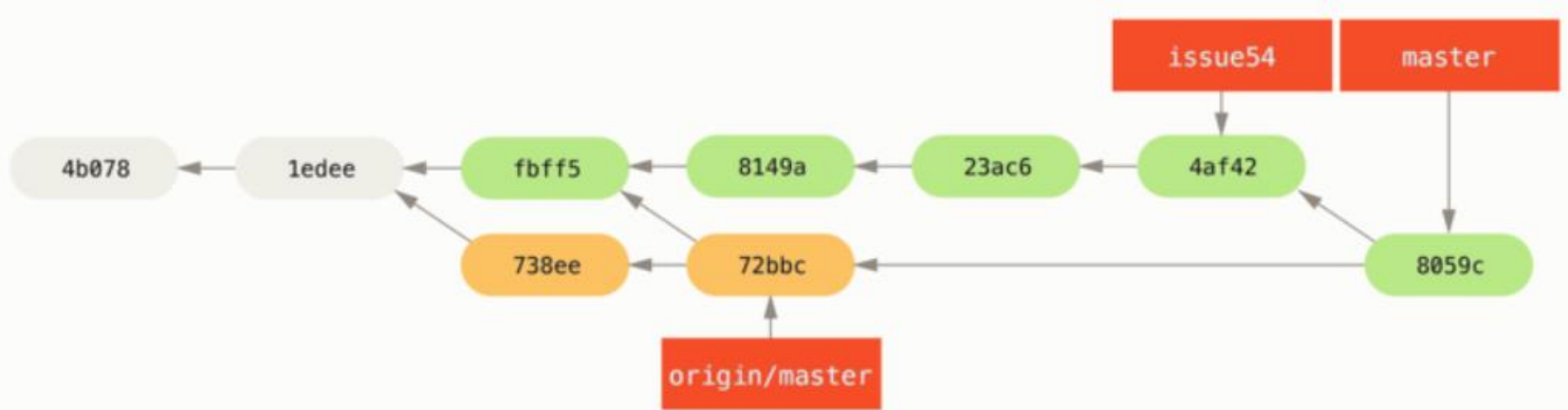
No problems occur; as you can see it was a simple fast-forward merge.

Contributing to Private Small Project (10)

Jessica now completes the local merging process by merging John's earlier fetched work that is sitting in the origin/master branch:

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb | 2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Everything merges cleanly, and Jessica's history now looks like this:

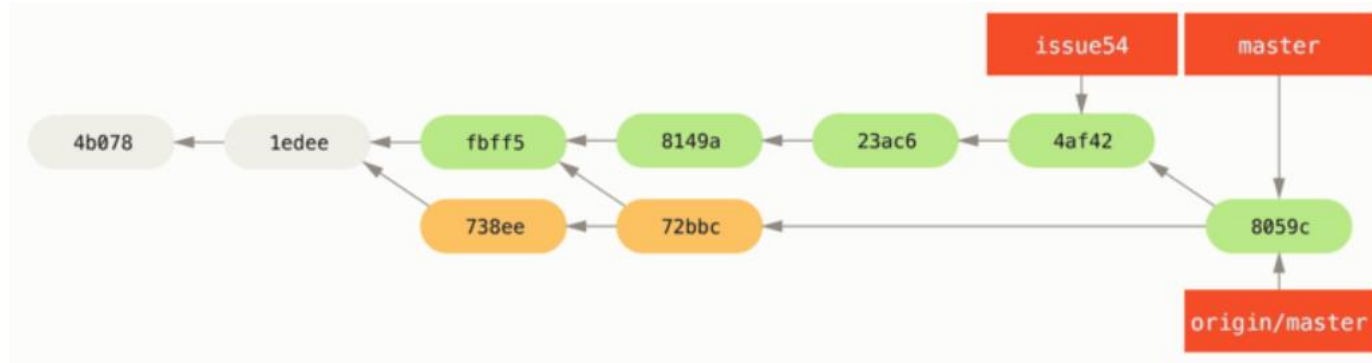


Contributing to Private Small Project (11)

Now origin/master is reachable from Jessica's master branch, so she should be able to successfully push (assuming John hasn't pushed even more changes in the meantime):

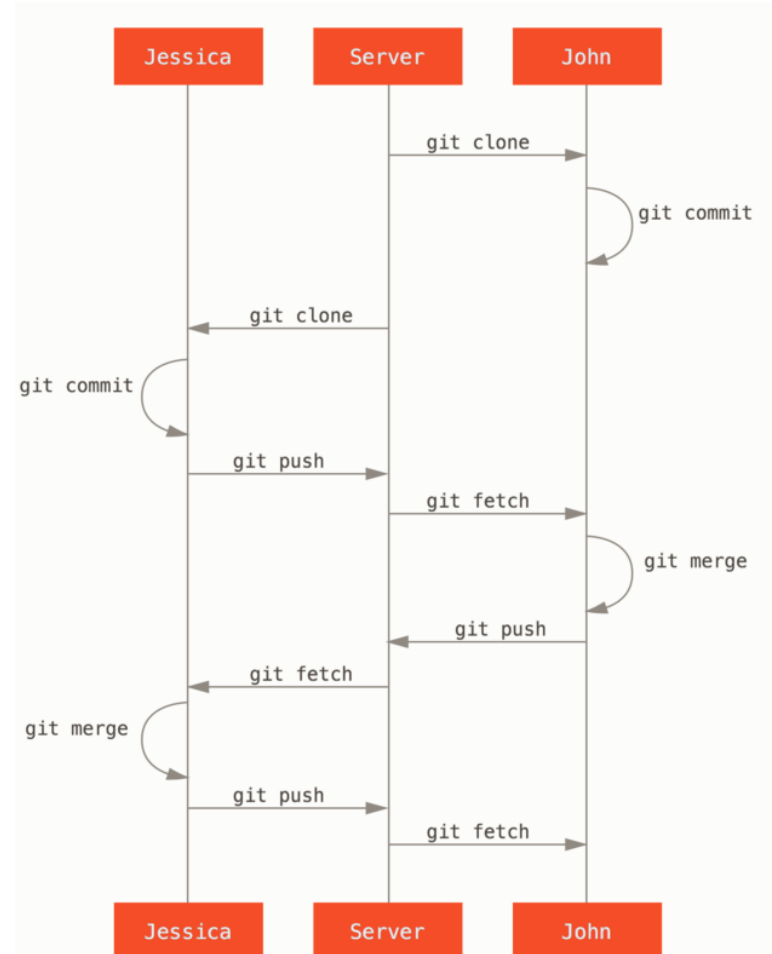
```
$ git push origin master
...
To jessica@github:simplegit.git
72bbc59..8059c15 master -> master
```

Jessica and John has committed a few times and merged each other's work successfully.

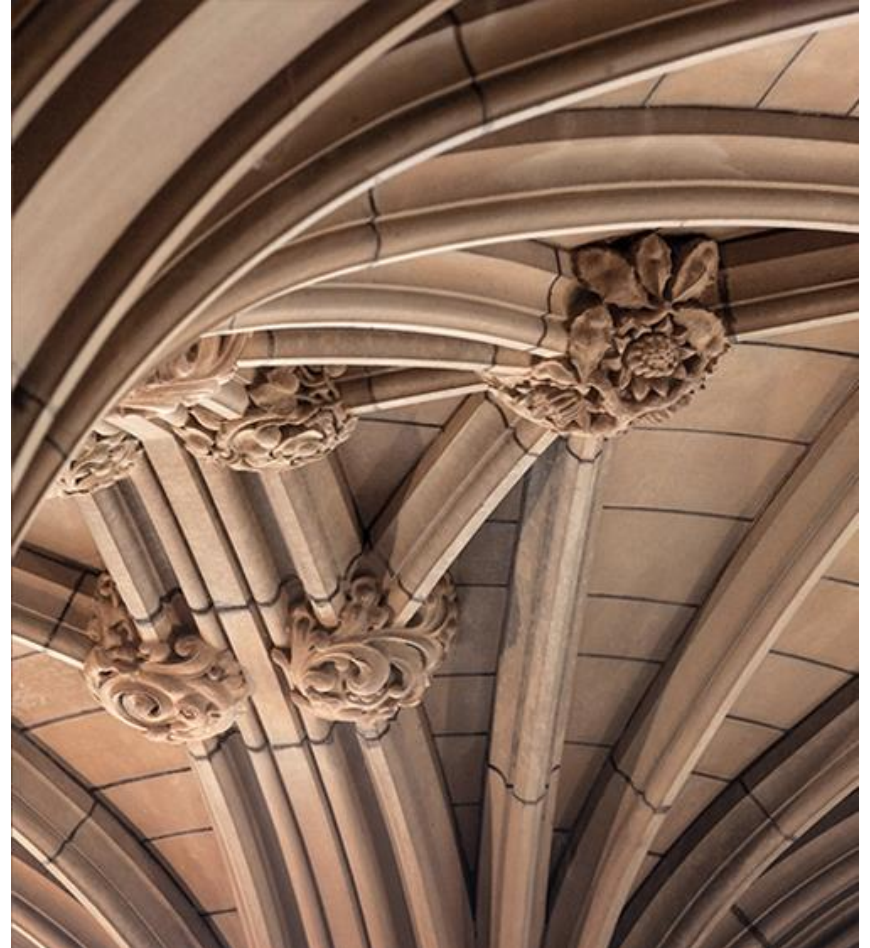


Contributing to Private Small Project (12)

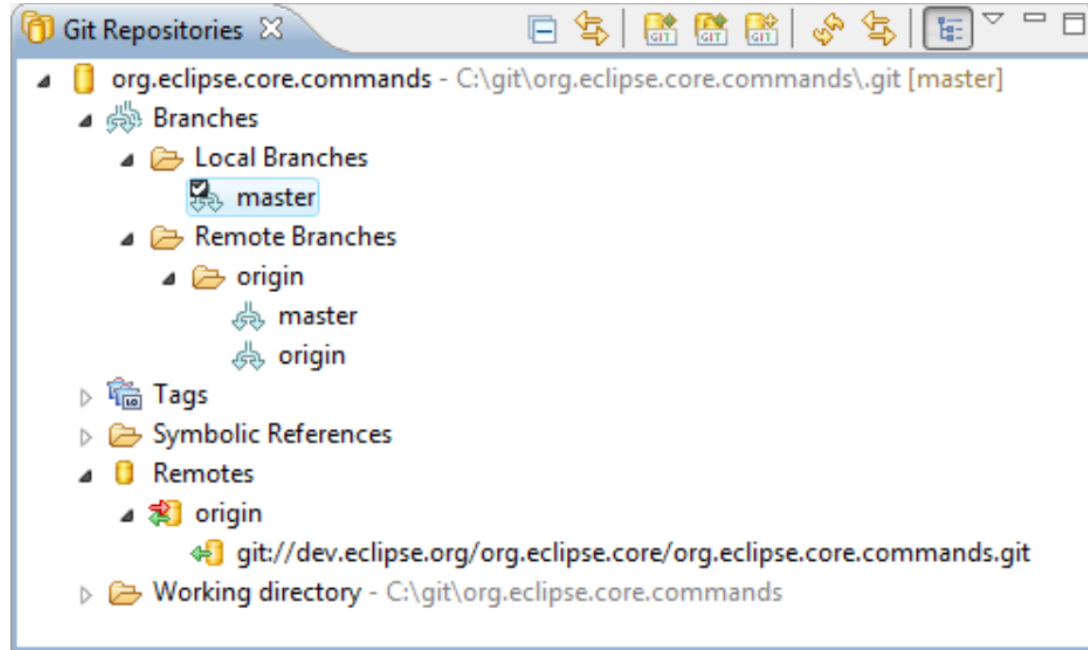
General sequence of events for a simple multiple-developer Git workflow



Git in Development Environments

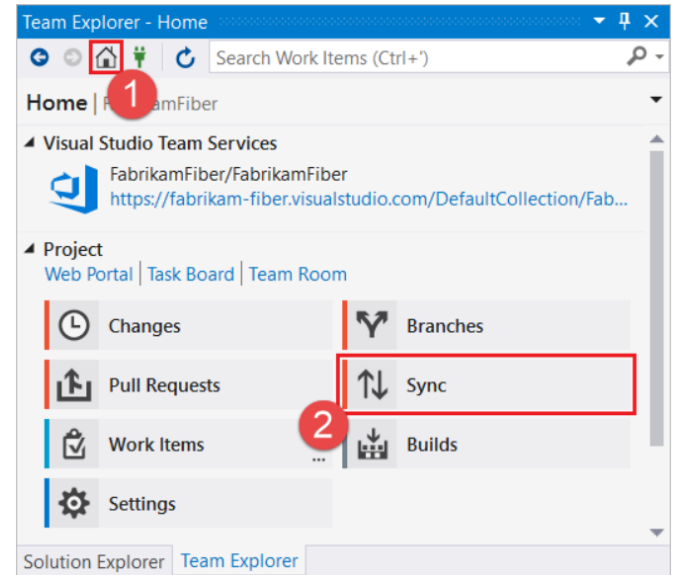
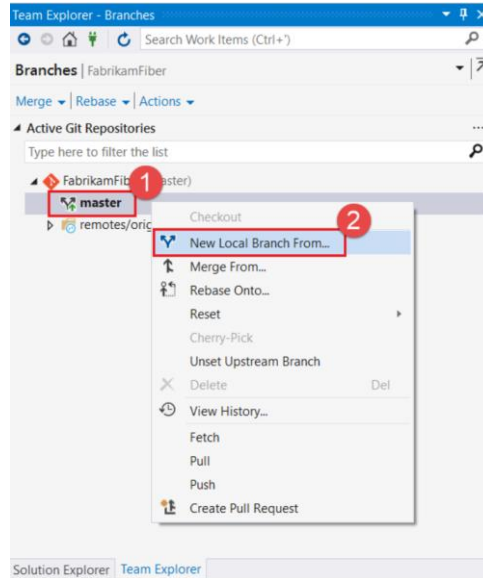
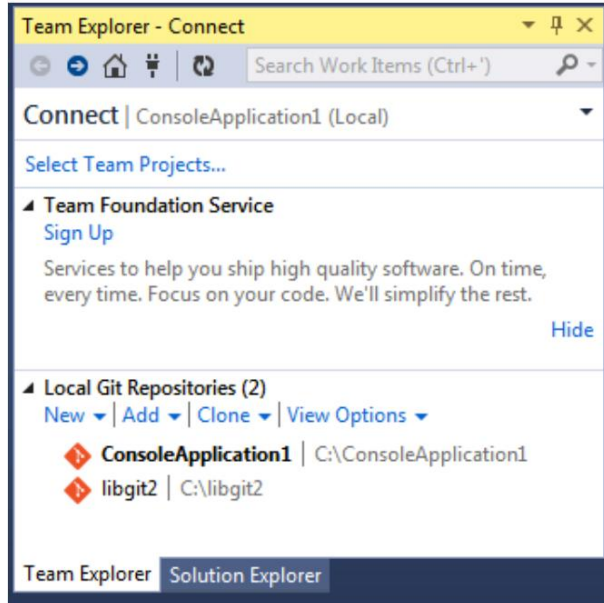


Eclipse Plugin – Egit



<https://www.eclipse.org/egit/>

Git in Visual Studio



<https://docs.microsoft.com/en-us/vsts/repos/git/gitquickstart?view=vsts&tabs=visual-studio>

References

- Scott Chacon. 2014. Pro Git (2nd ed.) Apress
 - Free online book – download from <https://git-scm.com/book/en/v2>
- Additional Resources – Paper
 - H-Christian Estler, Martin Nordio, Carlo A. Furia and Bertrand Meyer: *Awareness and Merge Conflicts in Distributed Software Development*, in proceedings of ICGSE 2014, 9th International Conference on Global Software Engineering, Shanghai, 18-21 August 2014, IEEE Computer Society Press (best paper award),
 - http://se.ethz.ch/~meyer/publications/empirical/awareness_icgse14.pdf

Contributing to a Project – Private Small Project (2)

- John clones the repo., make a change and commits locally
- Jessica clones the repo., make a change and commits locally
- Jessica punches her work to the server, and this should work fine
- shortly afterwards, John makes some changes, commits them to his local repository, and tries to push them to the same server
- John's push fails because of Jessica's earlier push of her changes