>>> network.toCode()

Collaborative Workflows with Git and GitHub

Workshop

Agenda

- Introduction to Git & Version Control Systems
- Working with a Local Git Repo
  - Demo + Hands-on Labs!
- Collaborating with GitHub
  - Demo + Hands-on Labs!
- Collaborating on GitHub Repos with Travis-CI
  - Demo + Challenge Lab!

Version Control Systems

Why Version Control?

- Collaboration
- Storing Versions
- Restoring Previous Versions
- Understanding What Happened

Tools: **Git**, Subversion (SVN), Mercurial (HG), CVS, Fossil, Perforce

Git Overview

- Command-line utility created by Linus Torvalds in 2005
- Original purpose was to support multiple collaborators working on Linux Kernel development
- Git is a **distributed** version control system
  - peer-to-peer interaction vs. client-to-server
  - version control = historical change tracking
- Intent
  - Provide version control (core functionality)
  - Foster better code collaboration
  - Reduce mistakes (bugs!)

Git Use Cases

Developers use Git to:

- Work on different versions of code
- See the difference between two or more versions of code
- Review the history of code
- Store code in a shared repository
- Experiment with a new feature without interfering with working code

Git Use Cases - Network Automation

Network Engineers can use Git for:

- Network Device Configs
- Playbooks
- Scripts
- Variables Files
- Any file that would benefit from being version controlled!

Collaboration Platforms

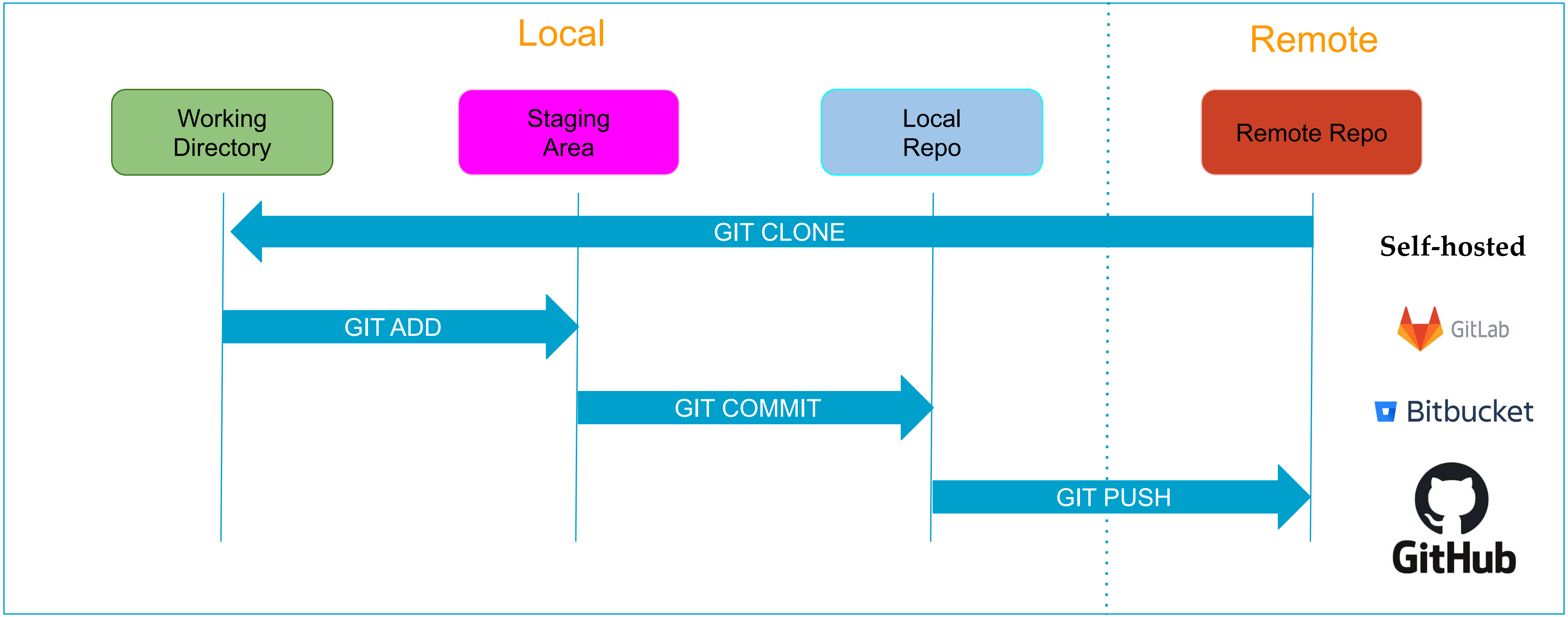
- Cloud-based or private on-premises storage for your code
- Common features:
  - Web UI
  - Viewing code differences between versions
  - Merging together code from different feature branches
  - Code review capabilities
  - Notifications, Discussion Boards
  - Support multiple version control systems
  - Issue and Project tracking

GitLab

Bitbucket

GitHub

Git Architecture

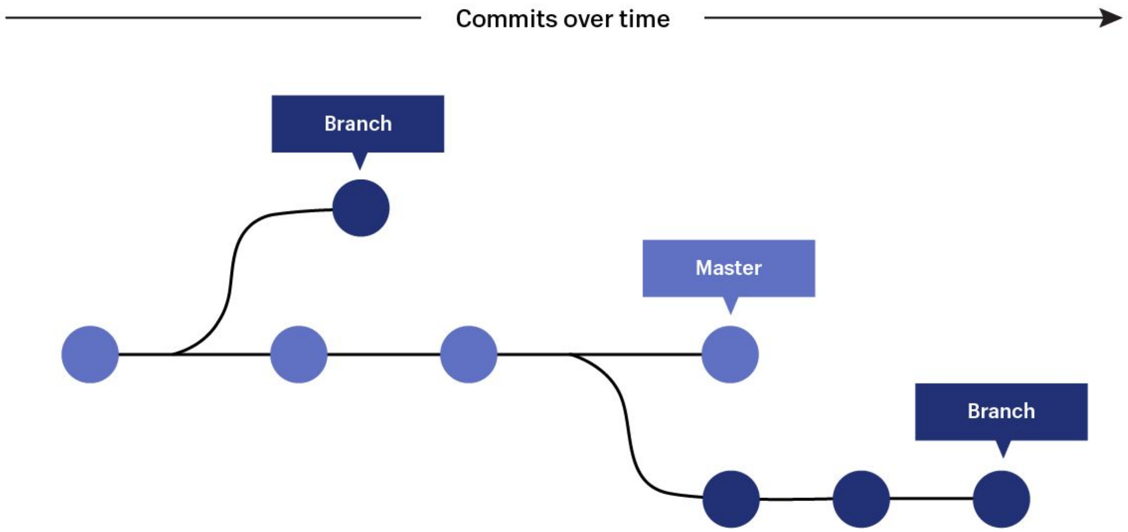


>>> network.toCode()

DEMO TIME!

Git Branches

- **Repository (repo)** - A project or collection of work being managed by git
- **Branch** - A particular series of changes to content within the repo
  - Branches can diverge and merge, like paths through a forest
  - The default branch is typically named master by convention
  - Branches may be temporary or permanent



Git Branches

- Commits allow us to move forward and backward in time
- Branches **exist in parallel** to one another
  - Each branch has its own series of commits
- Branches are "**checked out**" to the current workspace
- Disruptive development can be confined to a particular branch without sacrificing the stability of another
  - Example: **master** versus **develop** branches
  - Example: **bugfix** and **feature-add** branches

>>> network .toCode()

DEMO TIME!

Fork or Clone?

Clone Use-Cases

- you just want to copy, use, or explore the project
- you have write access to the repo and therefore, can push back up directly

Fork Use-Cases

- you want to contribute to a project you don't have write access to
- standard to fork'n'pull vs. clone/push

Introduction to CI/CD and Travis-CI

- **Travis CI** is Software-as-a-Service (**SaaS**) providing a cloud-based **continuous integration (CI)** server.
  - It integrates with code hosted on GitHub, Bitbucket, GitLab or Assembla
  - The work it performs (the pipeline) is defined in a YAML file (.travis.yml)
- **CI** is the practice of merging code changes frequently and leveraging automated testing - this builds trust in the system and healthier code.
  - "Code" can be anything - Ansible playbooks, data models, device configs etc.
- CI is often paired with **CD** - i.e. Continuous Deployment / Delivery.
  - A pipeline can test but also deploy your "code" to a staging environment, to production, package on PyPi, Ansible collection on Galaxy etc.
- Other platforms: Jenkins, Circle-CI, GitHub Actions, Gitlab-CI, Drone CI etc.

Travis CI - Terminology

- **phase** - a bunch of steps executed sequentially inside of a job
  - install phase -> script phase -> (optional) deploy phase
- **job** - the automated process that clones your repository into a virtualized environment and works through *phases* to run tests, compile code etc.
  - Success or failure is dependent on the return code of the phases
- **build** - a group of jobs that are run sequentially
  - test your playbooks with ansible 2.9 and 2.10
  - test your code with python 3.7 and 3.8
- **stage** - you can group jobs for certain parallel tasks (testing)
  - separate tests for ansible 2.9 and 2.10
  - single collection release to Galaxy for both versions if tests succeed

Travis CI - Example

https://github.com/networktocode/codelint/blob/master/.travis.yml

```
---
language: "python"
python:
 - "3.6"

install:
 - "pip install yamllint black pylama"

script:
 - "find . -name '*.yml' | xargs yamllint -s"
 - "black -v --check ."
 - "pylama -i E501 ."
```

>>> network .toCode()

DEMO TIME!

The Challenge

- **Lab 04 is a Challenge Lab!**
- You have to fix all the errors in the https://github.com/networktocode/codelint/ repository
  - There are Python syntax and formatting errors
  - There are YAML syntax errors
  - There are Ansible specific errors
- Submit a Pull Request with your fixes and see that the Travis Build is Green (Passed) in your PR.