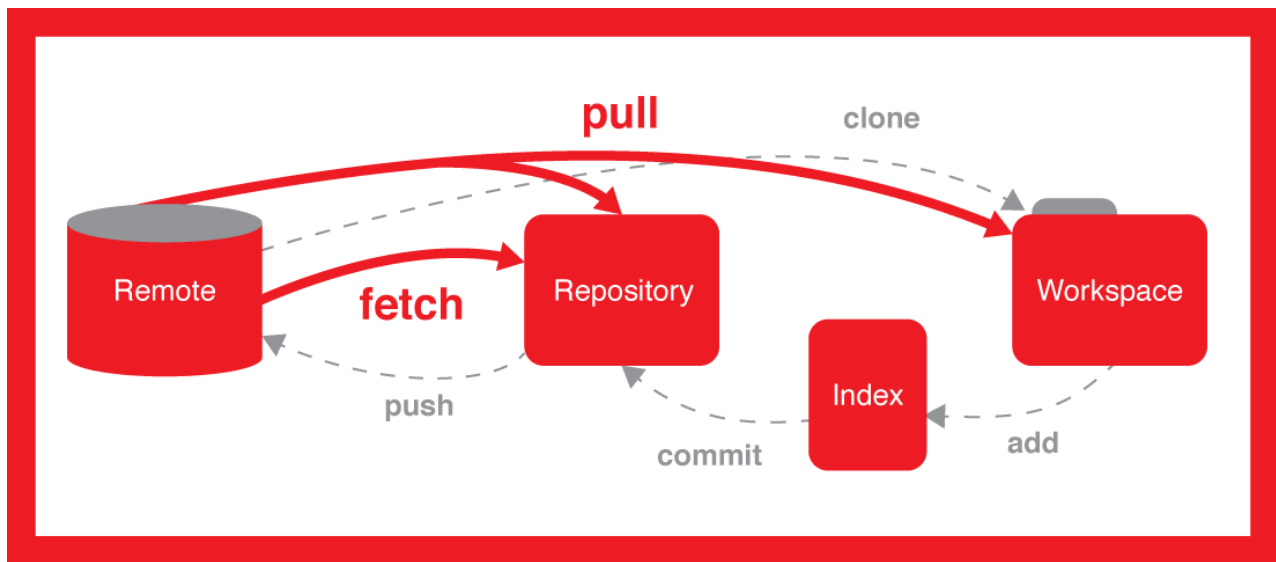


Lecture 2 - Git, GitHub and more BASH



Introduction

When a developer is working on a project which involves writing code, it is critical to ensure that all changes are tracked for both a historical context and the ability to rewind should development go south. Being aggressive with your Version Control System helps keep the introduction of bugs to a minimum and allows for an easy transition back to a stable state without potentially losing hours of work.

A DevOps engineer can yield a great deal of power and provide a safe environment for developers to work in when a well-thought-out version control strategy is in place. We'll be looking at some commonly used strategies, the problems they solve and the puzzle piece version control fills when developing a robust build pipeline to ensure deliverables are shipped with little risk.

Before looking at the strategies, we must have a good understanding of what version control is, why we want to use it, how it's used, along with an overview of the version control system we'll be using – Git. It'll be fundamental to ensure we

are comfortable with basic commands as we will be automating the process and we're also going to look at Microsoft's recently acquired platform, GitHub.

Why Version Control?

Have you ever seen files that look like this?

- KalidAzadResumeOct2014.docd
- KalidAzadResumeMar2015.doc
- instacalc-logo3.png
- instacalc-logo4.png
- logo-old.png

If an end-user is given the ability to mutate the contents of a file, chances are the file is susceptible to numerous state changes over a given period of time. A Version Control System allows a user to freely make modifications or changes with the ability to record the date/time and state of the change. Imagine having to work on a single file with a team of other developers. Things could take a turn for the worse rather quickly if severe coordination was not implemented amongst all developers. Without version control, you'd pretty much need to be telepathic to know who's working on what and ensure that no toes are getting stepped on when making your changes. Since we don't currently have telepathy, we must use a Version Control System to communicate amongst a team of developers when working on a project.

Version Control Systems (VCS) allow a team to work on the same project, modify the same files and work collaboratively amongst small to large groups when managing changes. The problems solved by a VCS may not be apparent at first, but after making file changes either independently or amongst a team, you may be surprised how easily headaches can accrue when VCS is not used.

Some of the benefits VCS brings:

1. **Enables a safe workflow** - A developer can commit code in small and easily testable increments. If any commit causes a problem... it can easily be reset to a stable state.
2. **Enables experimentation without fear** – If you'd like to try a change that is at risk for the introduction of unintended behavior, a developer can simply revert their work in progress back to a clean copy.
3. **Rolling Back** – A “roll back” or “rolling back” allows a developer to recover older versions which may not have that bug which was not caught before reaching production. With VCS, you can roll back to a working build with ease.
4. **Isolating Problems** - Sometimes problems can arise and it may be difficult to isolate where the problem stems from. VCS provide the luxury of traveling back in time to old code commits. Rolling back to a previous commit allows a developer to isolate and identify when the problem was introduced. This is really helpful as finding the source of the problem often times is 90% of the work.
5. **Enables CI/CD** - As we will learn throughout this course, CI/CD is crucial for scaling up a development team and automating builds, testing and deployment. VCS is a fundamental enabler for these advanced techniques.

How is a Version Control System used?

We know that a VCS is used to track our changes over time. We also know that we do this to ensure that if we mess up, we can easily revert to the most recently working change or the change our heart desires.

Before we see how version control is used; we must learn the lingo to understand what concepts are available for us to leverage when using a VCS. You'll commonly find most VCS's use the following concepts though terminology may differ depending on which VCS you choose.

The VCS used in this course will be commonly using the following terminology:

Basic concepts:

- **Repository (repo):** The database storing the files.
- **Server:** The computer storing the repo.
- **Client:** The computer connecting to the repo.
- **Working Set/Working Copy:** Your local directory of files, where you make changes.

Basic Actions:

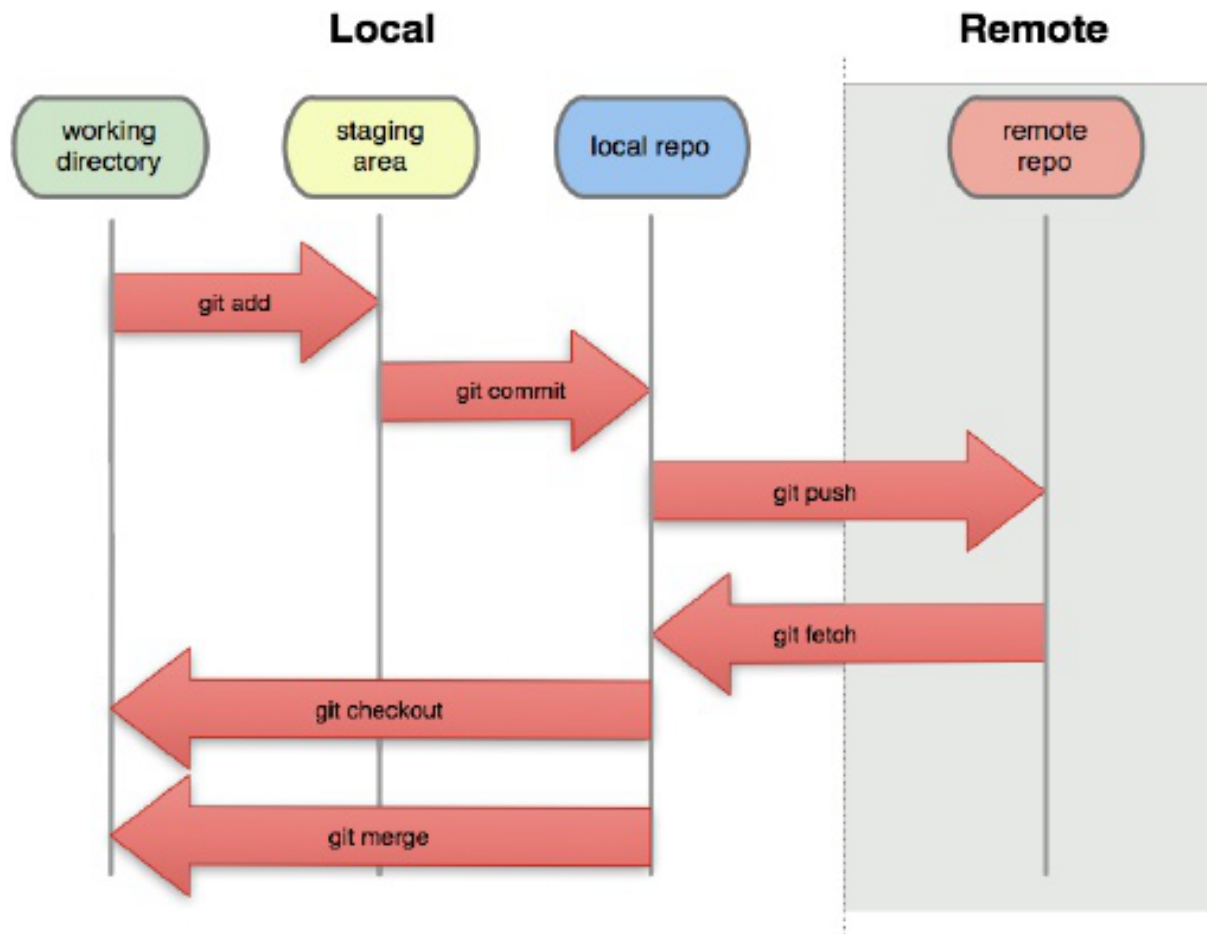
- **Add:** Put a file into the repo for the first time, i.e. begin tracking it with Version Control.
- **Revision:** What version a file is on (v1, v2, v3, etc.).
- **Head:** The latest revision in the repo.
- **Clone:** Download a remote repository
- **Check out:** Download a file from the repo.
- **Commit:** Upload a file to the repository (if it has changed). The file gets a new revision number, and people can “check out” the latest one.
- **Commit Message:** A short message describing what was changed.
- **Changelog/History:** A list of changes made to a file since it was created.
- **Pull/Fetch:** Fetch your files with the latest from the repository. This lets you grab the latest revisions of all files.
- **Revert:** Throw away your local changes and reload the latest version from the repository.

Advanced Actions:

- **Branch:** Create a separate copy of a file/folder for private use (bug fixing, testing, etc). Branch is both a verb (“branch the code”) and a noun (“Which branch is it in?”).
- **Diff/Change/Delta:** Finding the differences between two files. Useful for seeing what changed between revisions.
- **Merge (or patch):** Apply the changes from one file to another, to bring it up-to-date. For example, you can merge features from one branch into another.
- **Conflict:** When pending changes to a file contradict each other (both changes cannot be applied).
- **Resolve:** Fixing the changes that contradict each other and checking in the correct version.
- **Fork:** Forking or cloning your repository is an alternative to branching. Where a branch is an internal copy of a repository, a fork is an external copy: a separate repo that can eventually be merged back to the forked repo.

Since we have some of the terminology fresh in our minds, let’s look at one quick scenario:

Alice **adds** a file (list.txt) to the **repository**. She **checks it out**, makes a change (puts “milk” on the list), and checks it back in with a **commit message** (“Added required item.”). The next morning, Bob **fetches** his local working set and sees the latest revision of list.txt, which contains “milk”. He can browse the **changelog** or **diff** to see that Alice put “milk” the day before.



How does all this fit into our build pipeline?

Now that we've had a light introduction to VCS, let's see how all this tie's in to DevOps, specifically how it'll be applicable for this course. For this course our chosen VCS is Git and we'll be using GitHub for our remote repository. The remote repository is where our app will live and be hosted thanks to GitHub's Pages service.

Git was created by Linus Torvalds (creator of the first Linux kernel) and released in 2005. Git was designed to help developers working with Torvalds on the Linux kernel to help track changes during development. It was designed specifically for

the coordination of working amongst other developers when making file changes and can certainly be used to track changes in any set of files.

Getting to know our first tool (Git) and learning about what role it will play in our pipeline will help us not only understand how VCS is used but also, how it can trigger deployment changes for our web application. In an ideal scenario, if a developer wants to have their changes deployed, we know that first the source code change must make its way through the pipeline. Using Git, we're able to coordinate and track these efforts rather intelligently.

Since we know that our application will not only be hosted on GitHub but also that GitHub can be used as a centralized repository for our code to live, let's take a quick dive into GitHub and the power it offers a developer.

GitHub Overview

A few years after the creation of Git, 2008 to be exact – GitHub was created to offer Git users the ability to have remote repositories to store source files and keep a historical context of all changes made. Aside from the features offered by Git, GitHub also came equipped with other collaborative additions to help build a community of open source development.

Some of the most noted features GitHub offers which have become integral to developers are things like issue tracking, documentation, wikis, web interfaces to manage pull requests and overall a better visualization of what Git is doing. In 2018 Microsoft purchased GitHub at a tune of \$7.5 billion USD. GitHub offers plans for free, profession, and enterprise accounts. Most commonly, GitHub is used for their notorious unlimited free public repositories. If a team or an independent developer wants a “private repository”, meaning that the source code is not publicly available on the Internet and only accessible by those given permission to access the repo.

One of the features that we'll be leveraging from GitHub is GitHub Pages. GitHub Pages offers developers the ability to turn a repository into a self-hosted domain by accessing the page. To be specific, the style of hosting is known as a **static web hosting service**. This allows users to use GitHub as a platform to upload things like blogs, project documentation, books or a Single Page Application (SPA), like we'll be doing.

In 2018, GitHub surpassed 100 million repositories. 10 years prior when GitHub was founded, there were 33,000 repos. This should provide some context as to the rapid adoption in the developer community for not only Git but also GitHub which accounts for about 31 million developers around the world. When you're entering the market, Git knowledge is considered an asset to have. If you find a job that is not using Git, no worries as conceptually VCS's are very similar for the most part. So, if a prospective employer sees that you frequently use Git or have foundational knowledge of it, it's enough to transition effectively to their other VCS of choice.

More BASH

After last week's lab, you should have some basic BASH skills acquired which has provided you with the power of traversing and creating directories. You should also be familiar with the `home` directory and how to get there along with finding out where your command line is currently at.

In this week's lab we'll be unpacking some more BASH concepts, techniques and tools for the command line. We'll be starting things off with a protocol commonly used to access servers, Secure Shell better known as SSH. When we want to push a local change from our dev machine to our remote repository on GitHub, the protocol that does the heavy lifting for us is SSH.

SSH allows a client to access the contents of a remote server. It provides the ability for one to remotely log in and potentially have access to the file contents of the server. It's one of the most common and safe ways to administer remote servers. The safety mechanisms in place for SSH are made possible by establishing a cryptographically secure connection between the client and the server. Once connection is established for both the client and the remote server, each side can pass commands and output back and forth.

We won't be taking a deep dive into encryption and the underlying mechanisms for the protocol, we're only going to be focusing on what you need to know in the context of using SSH with GitHub.

When connecting to a **remote** GitHub **repo**, the connection is established using SSH which relies on asymmetrical encryption. Asymmetrical encryption involves two associated keys. One of these keys is known as the **private key**, while the other is called the **public key**.

The public key can be freely shared with any party. It is associated with its paired key, but **the private key cannot be derived from the public key.** The mathematical relationship between the public key and the private key allows the public key to encrypt messages that can only be decrypted by the private key.

Warning:

Never ever, ever, share your private key or leave it in a place others may stumble upon. Keeping the private key private is the only way to keep your communication safe. The private key should be kept entirely secret and should never be shared with another party. This is a key requirement for the public key paradigm to work.

The private key is the only component capable of decrypting messages that were encrypted using the associated public key. By virtue of this fact, any entity capable of decrypting these messages has demonstrated that they are in control of the private key.

SSH key pairs are used to authenticate a client to a server. The client creates a key pair and then uploads the public key to any remote server it wishes to access. This is placed in a file called **authorized_keys** within the `~/.ssh` directory in the user account's `home` directory on the remote server.

Let's take a break from SSH for a moment and talk about accessing text files on the command line. This next section is going to be very helpful for you to have a good grasp of, and in our labs we'll heavily be using text editors on the command line. You may be wondering, why the heck one would want to text edit anything on the command line when there is a plethora of very nice editors across all operating systems.

The reason for this, is sometimes you may find yourself in a position where you're dealing with strictly a command line environment. No desktop environment, task bars, docks or anything graphical other than what you see in your terminal. When faced with situations like this, it's critical for a user to know how to edit text files.

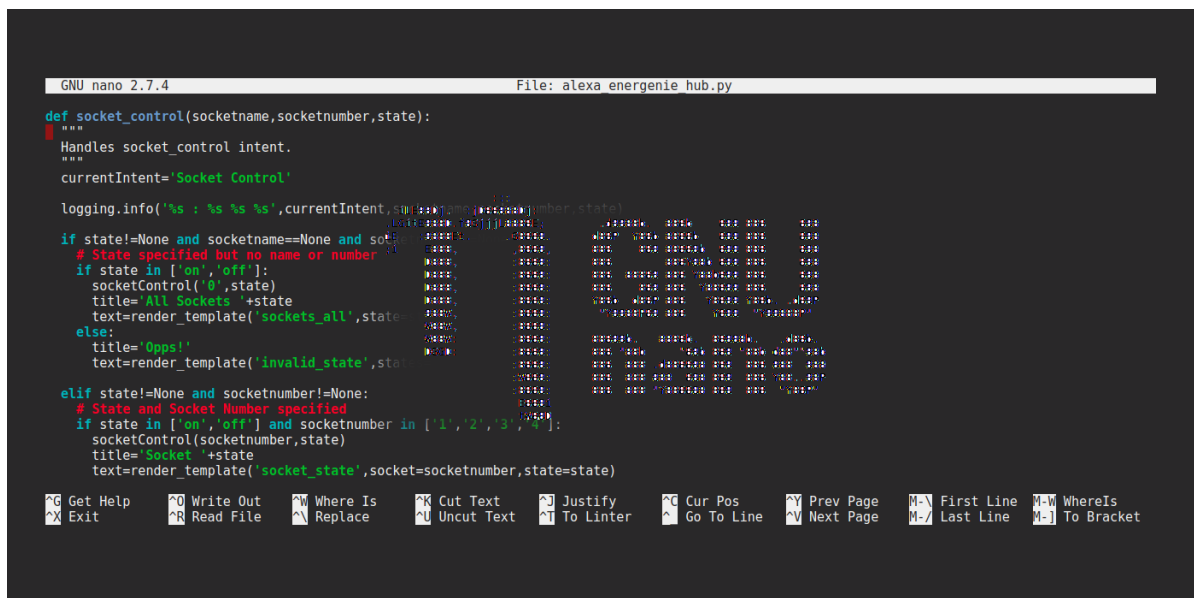
Editing Text Files from Your Terminal

At some point in your career, you are going to run into an instance where a source file must be edited through the command line. This file could be something like a configuration file in the `~/.ssh` directory, a BASH script, to take a note, or really any given task you can think of involving a file suitable for a text editor.

On the command line, there typically is two ways of doing this natively on the system. Chances are you'll hear of two different text editors on a Linux platform **nano** and **vi** or **vim**.

All three of the previously mentioned text-based editors are offered for Linux systems when the need arises. Nano is considerably the easiest one to navigate with, vi is often times referenced as an acronym for Virtual Insanity and vim is vi but improved. It has more bells and whistles. For those looking for a quick & easy solution to edit a file on the command line, nano would be the recommendation but if a user wants a more robust alternative, vim would be the editor of choice.

In our labs we'll be exploring all three and also show examples on how we can install vim as it usually doesn't come with a standard Linux install. Learning how to install vim in our lab will allow us to also explore how we can install other third-party applications in a Linux environment.



```
GNU nano 2.7.4 File: alexa_energenie_hub.py

def socket_control(socketname,socketnumber,state):
    """
    Handles socket_control intent.
    """
    currentIntent='Socket Control'

    logging.info('%s : %s %s %s',currentIntent,socketname,socketnumber,state)

    if state!=None and socketname==None and socketnumber!=None:
        # State specified but no name or number
        if state in ['on','off']:
            socketControl('0',state)
            title='All Sockets '+state
            text=render_template('sockets_all',state)
        else:
            title='Oops!'
            text=render_template('invalid_state',state)
    elif state!=None and socketnumber!=None:
        # State and Socket Number specified
        if state in ['on','off'] and socketnumber in ['1','2','3','4']:
            socketControl(socketnumber,state)
            title='Socket '+state
            text=render_template('socket_state',socket=socketnumber,state=state)

~ Get Help  ~ Write Out  ~ Where Is  ~ Cut Text  ~ Justify  ~ Cur Pos  ~ Prev Page  ~ First Line  ~ WhereIs
~ Exit      ~ Read File  ~ Replace  ~ Uncut Text ~ To Linter ~ Go To Line ~ Next Page  ~ Last Line  ~ To Bracket
```

Above you'll see a quick example of nano in use.

Below you'll see an example of the capability's vim offers.

```
1 // TAKE A FUNCTION LIKE THIS:
2
3 var greeterFunc = function(name) {
4   return console.log('Hello ', name);
5 };
6
7 // AND AUTOMATICALLY TRANSFORM TO ES6
8 // ARROW FUNCTION:
9
10 const greeterFunc = (name) => {
11   return console.log('Hello ', name);
12 };
13
```

NORMAL 4,+ <ample.js 0c 13 lines [javascript.jsx]

Knowing a bit about SSH and the ability to edit text files from the command line will be combined in our lab to produce something of value for your use.