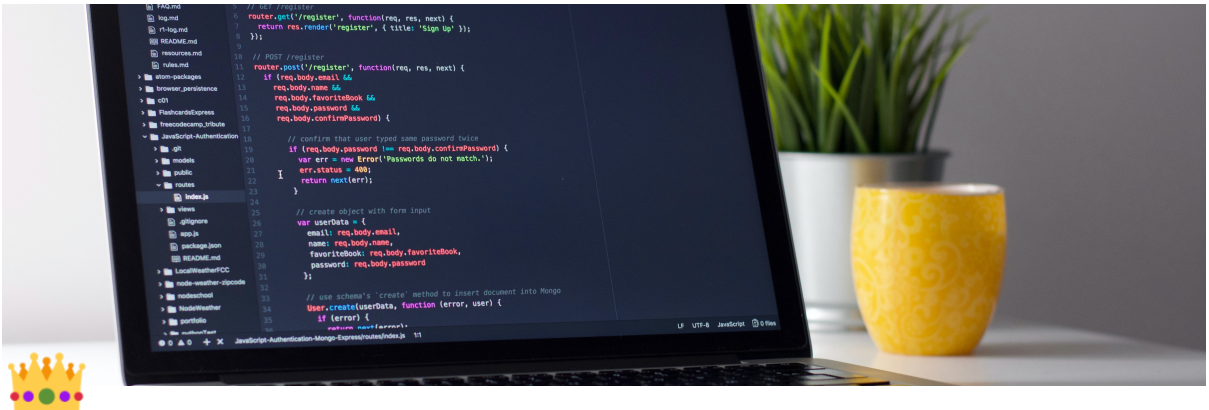


Lecture 05: Command-line Environment

- Job Control
  - Killing a process
  - Pausing and backgrounding processes
  - Terminal Multiplexers
  - Aliases
  - Dotfiles
  - Remote Machines
    - Executing commands
    - SSH Keys
    - Key generation
    - Key based authentication
    - Copying files over SSH
    - Port Forwarding
    - SSH Configuration
    - Miscellaneous
- Shells & Frameworks

Job Control

Killing a process

Your shell is using a UNIX communication mechanism called a **signal** to communicate information to the process. When a process receives a signal it stops its execution, deals with the signal and potentially changes the flow of execution based on the information that the signal delivered.

Typing `Ctrl-C` this prompts the shell to deliver a `SIGINT` signal to the process.

Typing `Ctrl-\` this prompts the shell to deliver a `SIGQUIT` signal to the process.

A more generic signal for asking a process to exit gracefully is the `SIGTERM` signal. To send this signal we can use the `kill` command, with the syntax `kill -TERM <PID>`.

Pausing and backgrounding processes

Typing `Ctrl-Z` will prompt the shell to send a `SIGTSTP` signal, short for Terminal Stop. We can then continue the paused job in the **foreground** or in the **background** using `fg` or `bg`, respectively.

> 💡 Note that backgrounded processes are still children processes of your terminal and will die if you close the terminal (this will send yet another signal, SIGHUP)

The `jobs` command lists the unfinished jobs associated with the current terminal session. To refer to the last backgrounded job you can use the `$!` special parameter.

The `&` suffix in a command will run the command in the background, giving you the prompt back, although it will still use the shell's STDOUT. The `nohup` command prevents from being borted automatically when you log out or exit the shell. (`nohup` stands for "no hangup" and ignores `SIGHUP` signal).

`nohup` command:
1. The output will be printed in the terminal
2. The program ignores `SIGINT` signal (`Ctrl+c`)
3. Send the `SIGHUP` signal (close the session) and the program shuts down

`&` comand:
1. The output will be saved in `nohup.out`
2. Send `SIGINT` signal and the program shuts down
3. Send the `SIGHUP` signal (close the session) and the program continues running

```
$ sleep 1000
^Z
[1]  + 18653 suspended  sleep 1000

$ nohup sleep 2000 &
[2] 18745
appending output to nohup.out

$ jobs
[1]  + suspended  sleep 1000
[2]  - running    nohup sleep 2000

$ bg %1
[1]  - 18653 continued  sleep 1000

$ jobs
[1]  - running    sleep 1000
[2]  + running    nohup sleep 2000

$ kill -STOP %1
[1]  + 18653 suspended (signal)  sleep 1000

$ jobs
[1]  + suspended (signal)  sleep 1000
[2]  - running    nohup sleep 2000

$ kill -SIGHUP %1
[1]  + 18653 hangup     sleep 1000

$ jobs
[2]  + running    nohup sleep 2000

$ kill -SIGHUP %2

$ jobs
[2]  + running    nohup sleep 2000

$ kill %2
[2]  + 18745 terminated  nohup sleep 2000

$ jobs
```

Terminal Multiplexers

Terminal multiplexers like `tmux` allow you to multiplex terminal windows using panes and tabs so you can interact with multiple shell sessions.

Terminal multiplexers let you detach a current terminal session and **reattach** at some point later in time

- **Sessions** - a session is an independent workspace with one or more windows
  - `tmux` starts a new session.
  - `tmux new -s NAME` starts it with that name.

- `tmux ls` lists the current sessions
- Within `tmux` typing `<C-b> d` detaches the current session (`tmux detach`)
- `tmux a` attaches the last session. You can use `-t` flag to specify which (`tmux a -t NAME`)
- `tmux switch -t NAME` switches to specified session
- `tmux kill-session -t <NAME/ID>` kills a specified session
- `tmux rename-session -t OLD NEW` renames a specified session

- **Windows** - Equivalent to tabs in editors or browsers, they are visually separate parts of the same session
  - `<C-b> c` Creates a new window. To close it you can just terminate the shells doing `<C-d>`.
  - `<C-b> N` Go to the *N* th window. Note they are numbered
  - `<C-b> p` Goes to the previous window
  - `<C-b> n` Goes to the next window
  - `<C-b> ,` Rename the current window
  - `<C-b> w` List current windows
- **Panes** - Like vim splits, panes let you have multiple shells in the same visual display.
  - `<C-b> "` Split the current pane horizontally
  - `<C-b> %` Split the current pane vertically
  - `<C-b> <direction>` Move to the pane in the specified *direction*. Direction here means arrow keys.
  - `<C-b> z` Toggle zoom for the current pane
  - `<C-b> [` Start scrollback. You can then press `<space>` to start a selection and `<enter>` to copy that selection.
  - `<C-b> <space>` Cycle through pane arrangements.

Ref: A Quick and Easy Guide to tmux, Terminal Multiplexers

Aliases

A shell alias is a short form for another command that your shell will replace automatically for you.

```
alias alias_name="command_to_alias arg1 arg2"
```

Note that aliases do not persist shell sessions by default. To make an alias persistent you need to include it in shell startup files, like `.bashrc` or `.zshrc`.

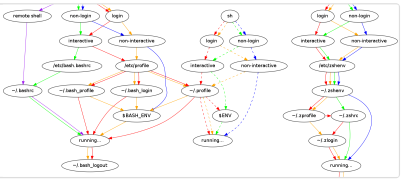
Dotfiles

Many programs are configured using plain-text files known as dotfiles (because the file names begin with a `.`, e.g. `~/.vimrc`, so that they are hidden in the directory listing `ls` by default).

Shell startup scripts

If you're a regular shell user, you've almost certainly got a .bash_profile or .bashrc script in your home folder, which usually contains various tweaks, such as setting environment variables (adding that directory to$PATH), telling your shell to do clever things (like set -o noclobber) and adding various

https://blog.flowblok.id.au/2013-02/shell-startup-scripts.html



Some other examples of tools that can be configured through dotfiles are:

- `bash` - `~/.bashrc`, `~/.bash_profile`
- `git` - `~/.gitconfig`
- `vim` - `~/.vimrc` and the `~/.vim` folder

- `ssh` - `~/.ssh/config`
- `tmux` - `~/.tmux.conf`

How should you organize your dotfiles? They should be in their own folder, under version control, and **symlinked** into place using a script. This has the benefits of:

- **Easy installation**: if you log in to a new machine, applying your customizations will only take a minute.
- **Portability**: your tools will work the same way everywhere.
- **Synchronization**: you can update your dotfiles anywhere and keep them all in sync.
- **Change tracking**: you're probably going to be maintaining your dotfiles for your entire programming career, and version history is nice to have for long-lived projects.

mathiasbynens/dotfiles

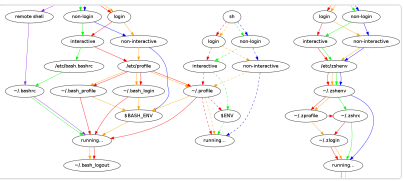
Warning: If you want to give these dotfiles a try, you should first fork this repository, review the code, and remove things you don't want or need. Don't blindly use my settings unless you know what that entails. Use at your own risk! You can clone the repository wherever you want.

https://github.com/mathiasbynens/dotfiles

Shell startup scripts

If you're a regular shell user, you've almost certainly got a .bash_profile or .bashrc script in your home folder, which usually contains various tweaks, such as setting environment variables (adding that directory to$PATH), telling your shell to do clever things (like set -o noclobber) and adding various

https://blog.flowblok.id.au/2013-02/shell-startup-scripts.html



Remote Machines

To `ssh` into a server you execute a command as follows

```
ssh foo@bar.mit.edu
```

Executing commands

An often overlooked feature of `ssh` is the ability to run commands directly. `ssh foobar@server ls` will execute `ls` in the home folder of foobar. It works with pipes, so `ssh foobar@server ls | grep PATTERN` will grep locally the remote output of ls and ls | ssh foobar@server grep PATTERN will grep remotely the local output of `ls`.

SSH Keys

Key-based authentication exploits public-key cryptography to prove to the server that the client owns the secret private key without revealing the key. This way you do not need to reenter your password every time. Nevertheless, the private key (often `~/.ssh/id_rsa` and more recently `~/.ssh/id_ed25519`) is effectively your password, so treat it like so.

Key generation

To generate a pair you can run `ssh-keygen`.

```
ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519
```

You should choose a passphrase, to avoid someone who gets hold of your private key to access authorized servers. Use `ssh-agent` or `gpg-agent` so you do not have to type your passphrase every time.

If you have ever configured pushing to GitHub using SSH keys, then you have probably done the steps outlined here and have a valid key pair already. To check if you have a passphrase and validate it you can run `ssh-keygen -y -f /path/to/key`.

Key based authentication

`ssh` will look into `.ssh/authorized_keys` to determine which clients it should let in. To copy a public key over you can use:

```
cat .ssh/id_ed25519.pub | ssh foobar@remote 'cat >> ~/.ssh/authorized_keys'
```

A simpler solution can be achieved with `ssh-copy-id` where available:

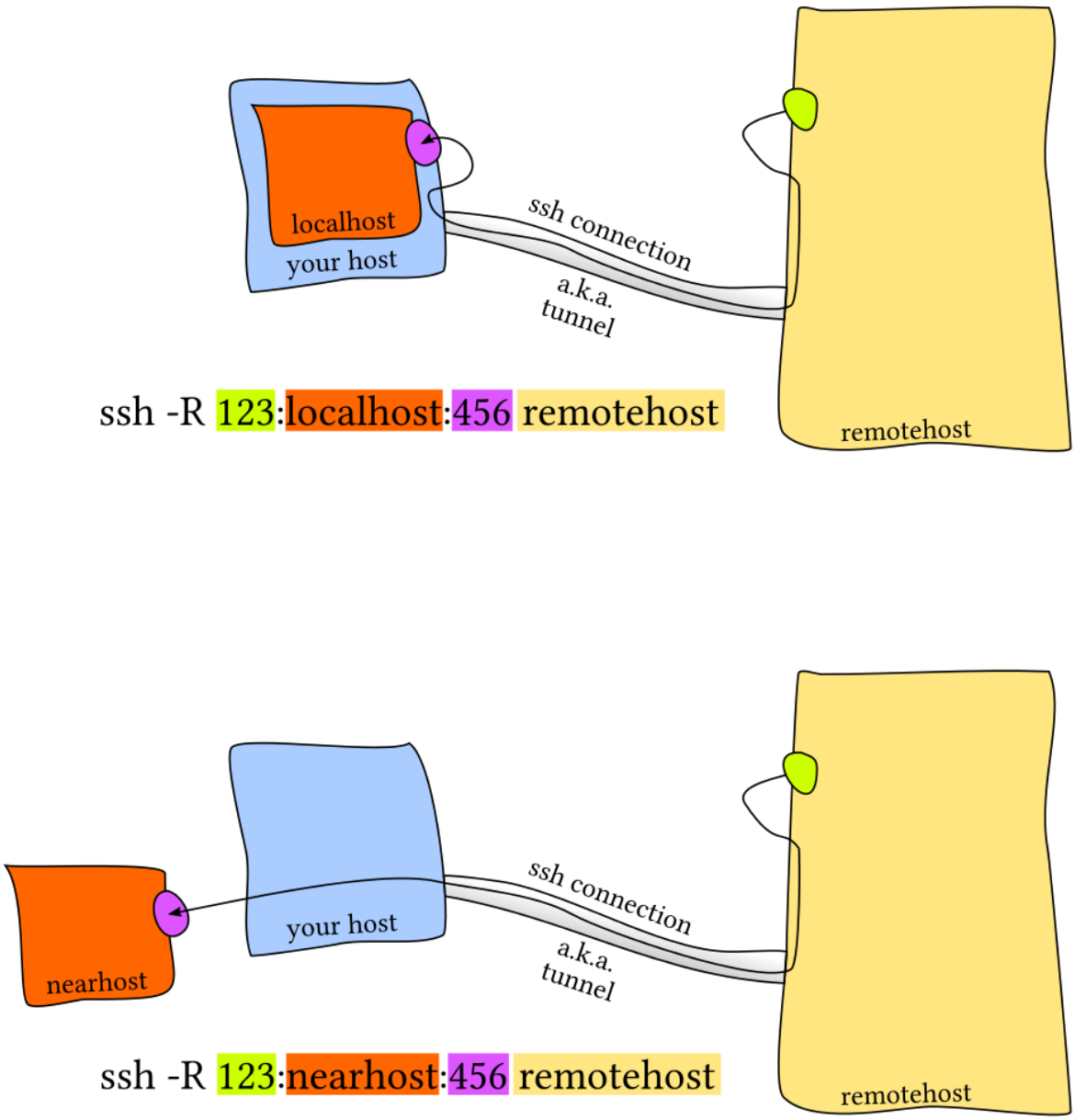
```
ssh-copy-id -i .ssh/id_ed25519.pub foobar@remote
```

Copying files over SSH

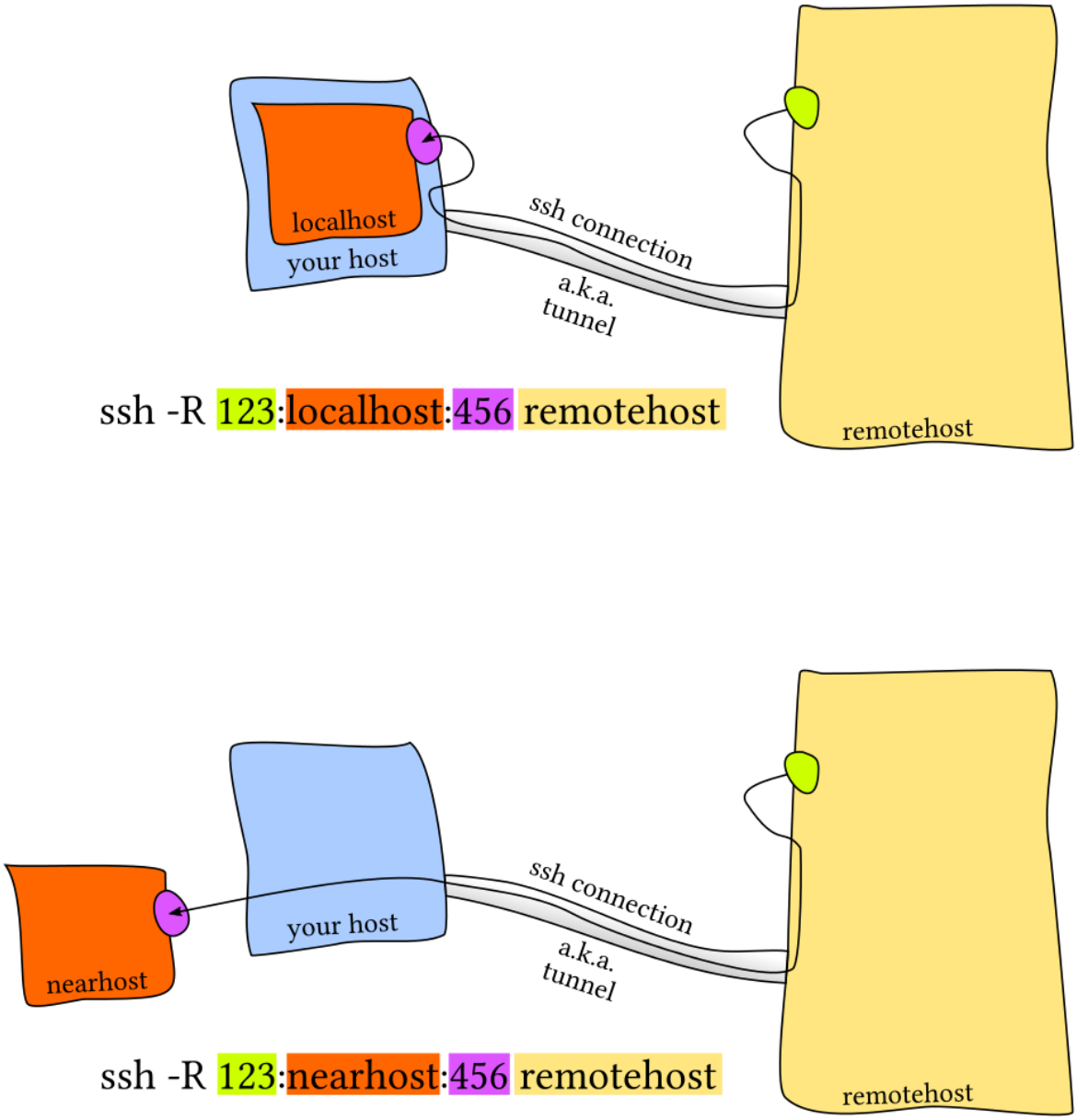
There are many ways to copy files over ssh:

- `ssh+tee`, the simplest is to use `ssh` command execution and STDIN input by doing `cat localfile | ssh remote_server tee serverfile`. Recall that `tee` writes the output from STDIN into a file.
- `scp` when copying large amounts of files/directories, the secure copy `scp` command is more convenient since it can easily recurse over paths. The syntax is `scp path/to/local_file remote_host:path/to/remote_file`
- `rsync` improves upon `scp` by detecting identical files in local and remote, and preventing copying them again. It also provides more fine grained control over symlinks, permissions and has extra features like the `--partial` flag that can resume from a previously interrupted copy. `rsync` has a similar syntax to `scp`.

Port Forwarding



Local Port Forwarding



Remote Port Forwarding

What's ssh port forwarding and what's the difference between ssh local and remote port forwarding

local: -L Specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side. ssh -L sourcePort:forwardToHost:onPort connectToHost means: connect with ssh to connectToHost, and forward all connection attempts to the local sourcePort to port onPort on the machine called forwardToHost,

https://unix.stackexchange.com/questions/115897/whats-ssh-port-forwarding-and-whats-the-difference-between-ssh-local-and-remot

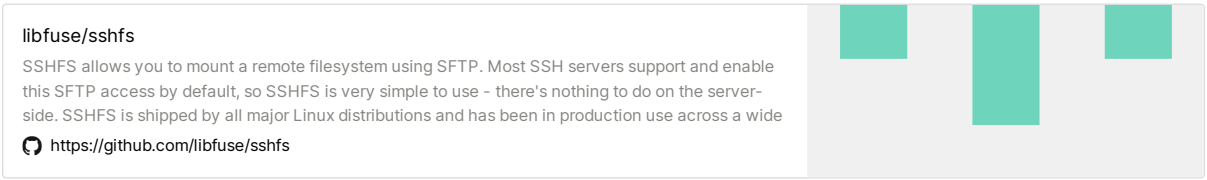
SSH Configuration

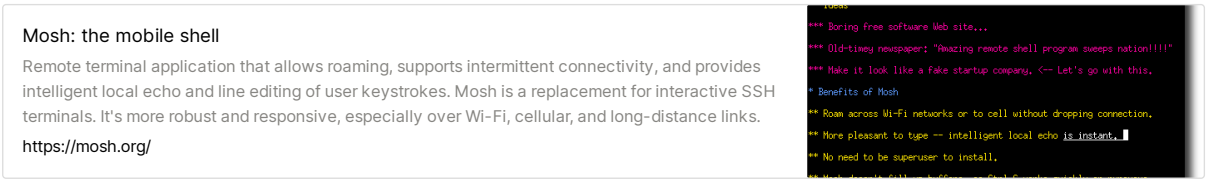
```
Host vm
    User foobar
    HostName 172.16.174.141
    Port 2222
    IdentityFile ~/.ssh/id_ed25519
    LocalForward 9999 localhost:8888

# Configs can also take wildcards
Host *.mit.edu
    User foobaz
```

An additional advantage of using the `~/.ssh/config` file over aliases is that other programs like `scp`, `rsync`, `mosh`, `&c` are able to read it as well and convert the settings into the corresponding flags.

Server side configuration is usually specified in `/etc/ssh/sshd_config`. Here you can make changes like disabling password authentication, changing ssh ports, enabling X11 forwarding, &c. You can specify config settings on a per user basis.

Miscellaneous





Shells & Frameworks

The `zsh` shell is a superset of `bash` and provides many convenient features out of the box such as:

- Smarter globbing, `**`
- Inline globbing/wildcard expansion
- Spelling correction
- Better tab completion/selection
- Path expansion (`cd /u/lo/b` will expand as `/usr/local/bin`)

Frameworks can improve your shell as well. Some popular general frameworks are prezto or oh-my-zsh, and smaller ones that focus on specific features such as zsh-syntax-highlighting or zsh-history-substring-search. Shells like fish include many of these user-friendly features by default. Some of these features include: