Web Developer Exercise

Web developers in particular stand to benefit from the command line, because there's a wide swath of well-established tools for working with servers and web applications. Below, we'll cover some common applications of command line tools and hopefully give you some interesting ideas for programs that you can use every day in your work.

Downloading Files, Unpacking Archives

If you've come this far, then you've already used the `wget` command to download a file from the web via the command line. Downloading, unpacking, and moving files can be very useful operations when administering web applications. Often, we need to pull down a module or plugin and unzip it to add it to our application, or we want to download images and other media assets hosted elsewhere.

Let's walk through a simple example of downloading a Drupal module. Drupal, if you're not familiar, is an open source content management system with a large library of community-supported modules hosted on https://www.drupal.org. Let's download the Biblio module:

```
> wget https://ftp.drupal.org/files/projects/biblio-7.x-1.0-rc7.tar.gz
```

First, `wget` prints out a bunch of HTTP information, then a download progress bar appears, and finally we should have a "biblio-7.x-1.0-rc7.tar.gz" file sitting in this directory. Note that `wget` also tells you how large and what file type you're downloading ("Length: 337860 (330K) [application/octet-stream]"). But what is a ".tar.gz" file? It's a compressed archive of a directory structure, the same thing you could create in Windows or Mac by secondary-clicking a folder and choosing to compress it. We can unpack the archive with the `tar` command:

```
> tar -xvf biblio-7.x-1.0-rc7.tar.gz
```

"Whoa there, what could '-xvf' possibly mean?" I hear you ask. The `tar` command has a lot of options & remembering them all is tough. Even experienced command line users commonly need to look up all the flags they want to use for some of these programs. Here's a brief summary of each one:

- -x means "extract" so we're telling `tar` that we want to *unpack* an archive, as opposed to creating one
- -v means "verbose" which is why `tar` printed out every file name in the package
- -f means "read the archive from this file" so it is actually attached to the file name argument we pass afterwards

A cleaner but more laborious way of writing the above `tar` command would be:

```
> tar -x -v -f biblio-7.x-1.0-rc7.tar.gz
```

At the end of this `tar`, we have a new "Biblio" folder full of files. If we truly were adding a module to Drupal, we would move this to the appropriate folder underneath the application's route (e.g. typically in sites/all/modules for Drupal 6 & 7). We can do this by running `mv biblio /drupal/sites/all/modules` where "drupal" is the folder we've installed the CMS in.

Now we know how to download and move around a tarball, but what if we want to *create* one, for instance so its easier for us to later download a set of text files over a shaky hotel WiFi network? There are only a few new flags to learn:

```
> tar -czf bundle.tar.gz instructions.md urls.txt
```

This will create (`-c`) a compressed (`-z` here, think z for "zip") archive named "bundle.tar.gz" (the argument following the `-f` flag) in the current directory, adding the files we list at the end of the command to it (in this case, a couple text files we've provided). Try bundling up all the Biblio module files again.

With `wget`, we can also download *a list of URLs*, which might be convenient in certain situations. Try running this command:

```
> wget --input-file=urls.txt
```

A whole bunch of messages will print. Take a look at the urls.txt file & use `ls` to see what happened. The `--input-file`flag lets us download a list of URLs from a text file, in this case a set of JavaScript files for parsing ISBN numbers. But you can see how this would have many more applications, like pulling down a series of pages on a site or bulk downloading media assets.

For practice, trying packing all the JavaScript files we just downloaded into a single .tar.gz archive. Hint: "*.js" is a handy way to the shell's wildcard replacement to refer to *all* .js files in the current directory.

One last thing to note: we've learned a few `wget` flags here, but they're all rather long to type. I prefer to start with the long form of these flags because they're explicit; it's pretty clear what "input file" will refer to, whereas the letter "i" is a little enigmatic. But there are shortcuts available for every `wget` option, here are a few:

- -h for --help (usage information)
- -i for --input-file (read URLs from a file)
- -O for --output-document (specify the downloaded file name)
- -q for --quiet (no progress bar, fewer printed messages)

HTTP Headers

If you manage a lot of servers and web applications, you probably know about the importance of HTTP headers. Headers can affect all sorts of things, from how quickly your content loads to how a browser interprets the data you send to it. There are a few major headers you'll probably always want to check:

- Caching headers, such as `Cache-Control`, can impact performance immensely by letting the browser know when it can use its internal cache instead of requesting fresh assets over HTTP
- The Content-Type header, which is set to a resource's MIME type, is trivial…until it's not & everything breaks. For instance, newer MIME types like "text/cache-manifest" (HTML5 appcache) and "application/x-web-app-manifest+json" (Mozilla's web app manifest) often aren't recognized automatically by servers & client applications won't recognize them without the appropriate Content-Type header
- Security can be slightly improved by oscuring the server version in the `Server` header & removing application-specific headers like `X-Generator: Drupal` & `X-Powered-By: PHP/5.1.2`

OK, so we get the picture, headers can be important. And sure, you can ivestigate them in your browser using the developer's tools. But sometimes looking at HTTP headers in the terminal is actually more convenient, for a variety of reasons, plus since our shell commands can be combined into larger scripts we might develop an automated way to check headers across *numerous* sites and apps which just wouldn't be feasible with a browser GUI.

How can we check HTTP headers? `curl` is our friend here, a program that stands for "Client URL library". `curl` can do all sorts of HTTP wizardry but we only need one command:

```
> curl --head http://example.com
```

This sends a HEAD (not a GET) HTTP request, which compliant servers respond to with *only* their headers and no body content (e.g. no actual HTML file). That makes it easily the quickest way to check headers. The syntax of `curl` is `curl [option flags] [URL]` and there are numerous options; take a quick peak at `man curl` but don't get overwhelmed, we're

not going to use them all.

Take a moment to look at the output of `curl --head` on your own servers. Do you see anything interesting or odd? You can look for the issues outlined in the list above; caching headers, appropriate content types (hint: try something other than an HTML page, request a PDF document or image file), exposed server & framework versions.

Next, let's think about something a little tricker: what if we want to view the headers for a particular page, but that page lies behind a redirect? To make things a bit clearer, I was testing `curl` by looking at the wordpress.com headers:

```
> curl --head http://wordpress.com
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Tue, 01 Mar 2016 00:35:59 GMT
Content-Type: text/html
Content-Length: 178
Connection: keep-alive
Location: https://wordpress.com/
X-ac: 1.sjc
```

As you can see from the 301 Moved Permanently response code, this site has been redirected! So the headers we're looking at aren't all that informative. Now, we could just send another manual request to the URL listed in the `location`header, but `curl` also has a built-in option for following redirects, no matter how many. Try the below:

```
> curl --head --location http://wordpress.com
```

What happens? How many HTTP requests did `curl` send & where did you end up? While following redirects is minimally useful in this example, it can *really* help when troubleshooting odd website issues, like redirects through proxy servers, URL resolvers, or strangely architected vendor sites. Being able to actually follow a stream of redirects, while seeing the headers at each step, can be edifying.

Do you have any redirects on your institution's website? Try following them! If you can't think of one, try a URL that's been sent through a link shortener (e.g. copy any URL off of Twitter).

Finally, let's do something a bit more sophisticated. We want to write a command line script which will test a series of URLs & return their HTTP status codes. We can run this script after a major software upgrade to see that all our web pages are still rendering properly. Create a file ending in ".sh", either in the Nitrous editor or a command line editor like `nano` or `vim`, and write something akin to the following:

```
#!/usr/bin/env zsh
URL_ONE='http://example.com'
URL_TWO='http://dp.la'
URL_THREE='http://lol.cat'

status_check () {
    echo "Testing URL ${1}"
    curl --head --silent ${1} | grep 'HTTP/'
}

status_check "${URL_ONE}"
status_check "${URL_TWO}"
status_check "${URL_THREE}"
```

There are a few things going on in this script, which I'll explain only briefly:

- the first line gibberish tells the operating system how to run the script
- the next three lines *initialize variables* to a few different values
- then we *define a function* named "status_check"

    - the `${1}` inside it refers to the first parameter passed to the function
    - the `| grep` part means we're piping ("|") the `curl` response through `grep`, a pattern-matching tool that we've told to look for text matching the pattern "HTTP/"
- then we execute the function three times, passing it a different URL each time

Writing that `status_check` function saved us a lot of redundant typing. There are much more elegant ways to write such a script, but this is a good start without being too complicated. But we still need to make the script *executable* by running the following before we can use it:

```
> chmod +x name-of-script.sh
```

That essentially changes the file's permissions such that we can run it without telling the operating system what program needs to interpret our code every time. We invoke a script simply by typing its full path into our terminal and hitting return, as if it was a command. But typing the full path `/home/nitrous/c4l16-cli-workshop-master/webdev-exercise/name-of-script.sh` every time we want to run the script is…tiring. I mean, I'm exhausted just from typing it into these instructions. Whew.

Instead, let's use the `.` which refers to the parent of our current directory, it's much shorter! Now we just type:

```
./name-of-script.sh
```

Bingo! What prints out when you run this? Try giving a few different URLs as tests. What would you change if you wanted the script to follow redirects? What would you change if you only wanted to look for certain HTTP status codes in the script's output, say 400-level or 500-level errors? Try to insert a URL that 404s and print out *only its* HTTP status code instead of all status' beginning with "HTTP/" as our original script does.

How would you actually go about fixing improper headers that you've found? That's a topic for another workshop! But suffice to say you could research the configuration options of your particular server software (Apache, nginx, IIS, etc.) or pass along the information to your friendly local sys admin.

SSH & SCP

If there's one tool every web developer who deals with servers should know, it's `ssh` or the "secure shell". `ssh` allows you to connect to and control remote servers. Server operating systems often don't include a GUI, like Windows or OS X, so the command line is literally the only way to perform most operations.

Unfortunately, setting up a server for folks to `ssh` into from a Nitrous account is a bit complicated and beyond the scope of this workshop. I'll introduce a bit of the syntax but we won't be able to play around much. The basic look of SSH is:

```
> ssh [user]@[domain]
```

That will connect you to a server; if you leave off the username & "@" portion, your shell username will be used (e.g. on Nitrous it's "nitrous", run `whoami` to find out your username). You'll be prompted for your password and then dropped into a shell session *on the server*. Pretty cool, right? You can run all the commands we've learned so far—inspecting the file system, moving things around, downloading from the web—on the server. It's also worth noting that, instead of connecting and creating a new shell session, and you can simply provide the text of a shell command for the server to run:

```
> ssh demo@example.com 'ls -l'
```

In this example, the user "demo" asks the example.com server to run the `ls` command with the `-l` flag; once "demo" has typed in their password, they should see a verbose directory listing of their home folder *on the server* printed to their local

terminal screen. This can be very handy if you just want to run one quick command.

Next, any discussion of `ssh` warrants a mention of `scp`, which (as you might've guessed) is just our friend the `cp` copy command but done over a SSH connection. The `scp` syntax looks like:

```
> scp [user]@[starting server]:[file]  [user]@[destination server]:[destination file]
```

That copies a file from one server to another; realistically, we omit the entire user and server string off the first parameter because we'll use a file on our own local system. So for instance

```
> scp urls.txt demo@example.com:~/docs/list.txt
```

will copy the "urls.txt" file sitting in this folder and put it inside demo's home (~) folder, inside a directory named "docs", and rename it to "list.txt". Because copying a lot of files one at a time is too time-consuming, `scp` has a handy "recursive" flag `-r`:

```
> scp -r . demo@example.com:~/docs
```

The dot `.` here refers to the current directory. A lot of the commands we've talked about, including `cp` and `rm`, also come with "recursive" `-r` flags. It is a common design pattern worth looking for in command line tools.

While we don't have a server for you to play with, I'll list a few exercises below. Feel free to continue to the `rsync` section, which you can do without connecting to a remote server.

- set up a free SSH account for http://sdf.org/
- try to `ssh` into sdf.org without using the link on the website
- if you have access to your own server, try to `ssh` in & `scp` files to it
    - upgrade from logging in with your password to logging in with a cryptographic key, `ssh-copy-id` can really help here
    - try setting up a config file of SSH aliases in ~/.ssh/config

Unfortunately, it looks like sdf.org doesn't have `scp` installed so we can't practice that without access to another server.

RSYNC

`rsync` is `scp`'s big sibling; it looks the same but is a lot more powerful, with many more options. `rsync` doesn't just naïvely copy files into place, it uses compression and duplication checks to ensure two separate directories mirror each other with as little data transfer as possible. Luckily for us, it also works *locally*—you can sync two folders on the same machine. Let's practice some with the "source" and "destination" folders inside this directory.

I'm going to go over several of the `rsync` settings, then leave it to you to determine how to achieve a set of tasks provided below. Here is the general syntax of the program:

```
> rsync [user]@[domain]:[files] [user]@[another domain]:files
```

Look familiar? It follows the structure of `scp`. Try doing a simple `rsync` between the "source" and "destination" folders. We get an error, "skipping directory source". Try copying *all the files inside* "*source*" with `rsync source/* destination`. What differences do you see between the two afterwards? Really investigate—go inside each and use `ls -al` (verbose directory listing) to see the permissions, files, and folders present in each. It's apparent that we haven't really "synched" them as many differences are present. We need to use what sets `rsync` apart—its long list of powerful options, which we specify as flags listed before the source/destination locations.

- `-z` or `--compress` uses *compression* to transfer fewer bytes
- `-v` or `--verbose` increases the *verbosity* of the program's output
- `-r` or `--recursive` makes the command *recursive*, meaning that it'll transfer subdirectories and their contents as well
- `-c` or `--checksum` will use checksums to determine if files differ before transferring them
- `-l` or `--links` copies *symbolic links* (think of: shortcuts, aliases) too
- `-p` or `--perms` preserves *permissions* between the source and destination
- `-n` or `--dry-run` shows us what a given command would do without actually modifying anything
- `--delete` *deletes* extraneous files in the destination that are no longer present in the source
- `--exclude=*.txt` *excludes* files ending in ".txt" or any other provided pattern, while
- `--exclude-from=file.txt` *excludes* files matching any pattern listed in file.txt

Whew, that's a lot of options! Luckily, there's one mecha-Voltron *archive mode* option which enables several things at once:

```
> rsync -av source destination
```

Yes, the `-a` alone *implies* a bunch of other options, most of which are sensible defaults we always want to use: recursive copying, maintaining permissions/ownership/symbolic links, etc.

So, without further ado, here are some tasks for you. Try accomplishing them with `rsync` and then, if you feel like torturing yourself, try using `cp` to achieve the same (to get a sense of why a tool like `rsync` is more suitable for complex copy operations). Make sure you are a) always using the quickest, most optimal transfer (i.e. use compression) and b) using "verbose" mode to see what's going on.

- first, wipe out the "destination" directory & remake an empty one with `rm -rf destination; mkdir destination` (the semicolon just separates two commands on the same line)
- copy all the files & directories in the "source" directory into the destination directory (such that "source/code.sh" becomes "destination/code.sh", etc.)
- change a file in the source directory (doesn't matter which one) & then sync them again
  - how many files did `rsync` transfer?
  - how much faster was the `rsync` than the first?
- repeat the last two operations, but use the "Biblio" module we download earlier instead of "source"
- copy all the files & directories in the "source" directory into the destination directory *except for the "secret" file*
- copy all the files & directories in the "source" directory into the destination directory *except for the "secret" file and one other file of your choice* (use an exclude list file)
- delete a file (other than "secret") from the "source" directory, then use `rsync` to sync "destination" such that the corresponding file is deleted as well
- wipe out the "destination" directory again, then try to sync the "source" folder *inside* of it (e.g. such that source and destination/source are identical). How did you have to change the `rsync` command?

That's it, hopefully you've gotten some good `rsync` practice in. It's an incredibly useful command for creating backups or maintaining parity between development, staging, and production web servers. If you do have `ssh` access, I recommend trying to set up a basic `rsync` script to help you with a tedious copying task you perform periodically.

Explore even more!

Well, if you've made it this far, you're probably pretty good at the command line. There's not a whole lot else we can add, but there are a few interesting tools we can present for you to either learn or spend some time configuring.

Does your library use the **Drupal** CMS? Take a look at setting up Drush: http://www.drush.org/en/master/

Drush is the "Drupal shell" and it allows you to perform common administrative tasks from the command line. You can also set up aliases using a configuration file to run command on external servers. If you're able to `ssh` into your own servers, give setting up `drush` a try, and if not take a look at the documentation and see if it'll be useful to you.

There's a similar project for the **Wordpress** CMS, Wordpress-cli: http://wp-cli.org/

Same thing here; try setting this up on your servers if you have access, or reading through the documentation if you don't.