SLT Dev Ops

This is a draft for some things we can change to help our programmers work better. Specifically, these changes will help programmers learn to write better code, make our development cycles safer and more stable, and make our software more robust and flexible.

These tools will help:

- Git
- MAMP (for old PHP sites)
- Laravel (for new sites)

In addition to these tools, we will need some simple policies in place on how we *use* these tools.

Git

What is Git?

Git is a computer program. It's a simple binary like `diff` or `grep` or `find`. Git is not a web site. *GitHub* is a web site, and it uses Git to work. GitHub lets programmers share their Git repositories. We can use Git independent of GitHub.

Installing Git is as easy as installing any other package. Git is extremely well maintained, and has legendary documentation.

What does one do with Git?

Git lets you maintain control of different versions of your programs. It also lets you collaborate with multiple programmers and still keep your code from getting messy. It lets you see every change you've made to your software and when. (And much more!)

Having multiple versions of software lets us keep a nice, stable version meant for production. We can have another version for the development environment which, once the code has been proven to be stable and safe, can be merged easily into the production version. (Manually copying files is not easy because we miss things. Running `git merge` and then something like `make deploy-production` is.)

We don't always have multiple programmers on the same project, but when we do, it can get a little hairy. Git lets multiple developers write code without stepping on each other's toes.

Being able to see what changes were made when will *greatly* mitigate emergencies like the one experienced with the *Foundations of Leadership* application. We solved the problem by using `CMD-z` to roll-back changes until it worked. Obviously, it was extremely fortunate that such history was preserved. Git makes this kind of roll-back easy, safe, and ubiquitous.

The master plan

This is how we would put these changes into effect:

1. Select a server to be our repository.

This could be one of our development servers, or better yet, a dedicated server.

1. Setup the repository.

Either we just create a directory for each project, or we install [GitLab][https://about.gitlab.com] on the server, which would provide a nice front-end for managing our repositories. GitLab is like a self-hosted, private version of GitHub. Basically, it's a collection of managed repositories. GitLab has issue tracking features which would give us Job Tracker-like ticket tracking on projects, minus the headache that Job Tracker is.

GitLab integrates with Slack which all the developers are now using. This would let us know if a bug/issue gets assigned to one of us. Note: this is not intended to replace face-to-face interactions, but merely to help us keep a nice TODO list of what needs doing.

GitLab has a lot of functionality for automated testing and deployment. I haven't used it personally, but after looking at some of what it can do, it looks *awesome*! It's also a little overwhelming, but we don't necessarily have to use all of its features. It would give us a nice place to put documentation.

1. Make it bullet-proof with backups.

Just as we have been backing up our production and development servers, we can back up our repository server. The repositories themselves function as mini, incremental backups, but having full-on backups will protect us in the case of hardware failure.

For building new projects, this is the flow we would follow:

1. Create a new repository on the repository server.

(Optionally using GitLab.)

1. Clone the repository anywhere

Usually this means cloning onto a programmer's desktop machine.

1. Make build scripts to automatically build the database.

This functionality comes out-of-the box with Laravel. We could also write a simple PHP script to connect to and setup the database tables automatically.

1. Make build scripts to deploy the code to an environment of our selection.

Laravel makes deploying to different environments extremely easy.

With a Makefile, targets like `deploy-dev` and `deploy-prod` can make deployment painless.

Why?

- Git makes debugging easy
- Collaboration between multiple programmers is a breeze
- Git makes experimenting much safer
- Git will help us maintain consistency between dev/prod environments
- Git will get knowledge out of the heads of the programmers and into records others can access

Git makes debugging easy

With Git, we can find the last working version of a piece of code, see the exact lines of code that were changed, and spot bugs quickly and confidently. See the example under Fixing a bug.

Collaboration between multiple programmers is a breeze

We don't often work in groups on sites, but Git makes managing this very convenient when we do. See the Usage example for a little more details on how this is done.

Git makes experimenting much safer

We have backups, but those are made every night. What happens when we want to try out some drastic changes to code that might break everything if it doesn't work out? Well, in the past, we've made a copy of the directory. That however can get very messy. Plus, if there are *some* things that you liked about your changes, you have to merge those changes in by hand, effectively doubling the amount of work you do. Git manages this for you. You can (and should) create topic *branches* (virtual copies of the project) that can be merged (fully or partially) back into the `master` branch once things are stable.

Git will help us maintain consistency between dev/prod environments

The Git repository will become the single source of "truth"—both the development and production environments will be based off of what is in the repository. Basically, in the repository we have multiple branches: `master`, `development`, and `some_feature`:

```
A---B master (production)
     \
      E---F---G v1 (development)
           \
            H---I some_feature (developer branch)
```

We can merge the `some_feature` branch (name doesn't matter—only the `master` branch name is fixed) developer branch into the `v1` development branch whenever:

```
A---B master
     \
      E---F---G---J-------M v1
           \     /       /
            H---I---K---L---N some_feature
```

Commit `M` now is effectively the most up-to-date bleeding-edge but semi-stable version of the project. Meanwhile, commit `N` holds some changes that the developer has been working on. Once we're ready to push to production, we merge with master:

```
A---B-----------------------O master
     \                     /
      E---F---G---J-------M v1
           \     /       /
            H---I---K---L---N some_feature
```

Note: simply merging into master will not push something to production. Use FileZilla or `rsync` to move into production environments. Ideally, use some sort of build script like `make` to push changes.

Git will get knowledge out of the heads of the programmers and into records others can access

With Git we have a step-by-step history of the development process. While this is not a replacement for good documentation, it does provide massive insight into what a programmer was thinking as they developed a program. When picking up an old program, a programmer can read the log and see how the program evolved over time.

By moving the source of the code off of the production/development environments and into a repository forces us to maintain build scripts. Having build scripts means we have to put all the knowledge of migrating a site into a script. The next time we need to migrate a site from one server to another, it is as simple as changing the URL we deploy to. **This** would take a huge load off the sysadmins and would eliminate a lot of guess-work.

What sort of policies should we adopt?

- Code reviews
- Branch `master` is always deployable

Code reviews

Before anyone merges into `master` (or we could also say before anyone merges into development) at least one other programmer has to review their code. That would encourage programmers to write clean code, and would help us catch bugs early on.

Branch `master` is always deployable

The idea is that the `master` branch *always* contains clean, working code. When we want to test something out, we make a new branch. That way, if we ever need to restore the production environment, we simply pull the copy from the master branch. No hassle.

Concerns

This section addresses some concerns with using Git.

Memory

Q: How much memory does a Git repository use during its lifetime?

Answer: Not a lot. I cloned the official Drupal repository:

```
$ time git clone https://github.com/drupal/drupal.git
Cloning into 'drupal'...
remote: Counting objects: 589968, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 589968 (delta 154), reused 192 (delta 84), pack-reused 589611
Receiving objects: 100% (589968/589968), 176.49 MiB | 11.98 MiB/s, done.
Resolving deltas: 100% (402738/402738), done.
Checking connectivity... done.
Checking out files: 100% (11318/11318), done.

real    0m31.898s
user    0m24.627s
sys 0m8.564s
```

Here's the memory usage on the file:

```
$ du -h -d 1 drupal
194M    drupal/.git
 73M    drupal/core
4.0K    drupal/modules
4.0K    drupal/profiles
 56K    drupal/sites
4.0K    drupal/themes
267M    drupal
```

Yes, the git repository (`drupal/.git`) adds 194 Megabytes, but that is literally the *entire history* of the drupal project. (33,480 commits from 42 contributors over 17 years.) We won't be doing anything nearly that big. My supply tracker/print jobs project by comparison looks like:

```
$ du -h -d 1 tracker
8.3M    tracker/.git
152K    tracker/app
 24K    tracker/bootstrap
 60K    tracker/config
124K    tracker/database
 20K    tracker/doc
138M    tracker/node_modules
6.4M    tracker/public
 72K    tracker/resources
 20K    tracker/routes
3.3M    tracker/storage
 84K    tracker/tests
 40M    tracker/vendor
197M    tracker
```

My repository is only 8.3 megabytes. Our projects will look more like this than the drupal project.

Conclusion: even if we were to have a massive project running over several decades, repository size is still extremely reasonable.

Usage example

Setting up a Git repository is super easy. Let's say we designate a place on `ayeaye` as our central repository. This is all we would need to do on ayeaye:

```
$ mkdir repos
```

Done! So far we only have a directory. What happens when we want to make a new web site? Well, we just do this:

```
$ cd repos; mkdir new_site; cd new_site
$ git init --bare
Initialized empty Git repository in /home/programmer/repos/new_site/
```

Now we have a repository for the site. This will function as the "single source of truth" for the site.

Let's say a programmer wants to start working on the site. On their desktop computer, they *clone* the repository:

```
$ cd Projects
$ git clone programmer@ayeaye.byu.edu:repos/new_site/
Cloning into 'new_site'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
```

Now we can made some edits, add files, etc. Once we are done, we first use `git add` to specify what files we want to *commit*, then we use `git commit` to finalize the changes:

```
$ git add index.html js/basics.js
$ git commit -m "added basic headers, etc. on landing page"
[master (root-commit) b7c2e7a] added basic headers, etc. on landing page
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 index.html
create mode 100644 js/basics.js
```

We can do this `git add` and `git commit` process as often as we want. Ideally, this is done after every atomic change: adding a function, a new `<div>` tag, etc. Let's say we add a new function in `basics.js`. We can see what changes we need to commit like so:

```
$ git status

Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

    modified:   js/basics.js

no changes added to commit (use "git add" and/or "git commit -a")
```

That shows us that some modifications have been made to `js/basics.js`. We can see *exactly* what lines of code changed by using the `git diff` command:

```
$ git diff js/basics.js

diff --git a/js/basics.js b/js/basics.js
index e69de29..45bfc50 100644
--- a/js/basics.js
+++ b/js/basics.js
@@ -0,0 +1,3 @@
+function factorial (n) {
+    return n == 0 ? 1 : n * factorial(n - 1);
+}
```

We can see that we added a function called "factorial". We can then commit these changes like so:

```
$ git add js/basics.js
$ git commit -m "added the factorial function in basics.js"
```

We can see a history of all changes by running `git log`:

```
$ git log

commit e7e0d15ab2c6951d5c8efbdc4860d9fee361a34a
Author: Ashton Wiersdorf <ashton.wiersdorf@mailblock.net>
Date:   Fri Jun 2 13:35:12 2017 -0600

    added the factorial function in basics.js

commit b7c2e7a4b155e07eacf2794f63254e3d93d7f8b5
Author: Ashton Wiersdorf <ashton.wiersdorf@mailblock.net>
Date:   Fri Jun 2 13:27:00 2017 -0600

    initial commit
```

If we run `git log --stat` we can see exactly what files changed with each commit:

```
$ git log --stat
commit e7e0d15ab2c6951d5c8efbdc4860d9fee361a34a
Author: Ashton Wiersdorf <ashton.wiersdorf@mailblock.net>
Date:   Fri Jun 2 13:35:12 2017 -0600

    added the factorial function in basics.js

js/basics.js | 3 +++
1 file changed, 3 insertions(+)

commit b7c2e7a4b155e07eacf2794f63254e3d93d7f8b5
Author: Ashton Wiersdorf <ashton.wiersdorf@mailblock.net>
Date:   Fri Jun 2 13:27:00 2017 -0600

    initial commit

index.html   | 0
js/basics.js | 0
2 files changed, 0 insertions(+), 0 deletions(-)
```

Once we're ready to make our changes available to everyone, we can run `git push`:

```
$ git push
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (8/8), 679 bytes | 0 bytes/s, done.
Total 8 (delta 0), reused 0 (delta 0)
To programmer@ayeaye.byu.edu:repos/new_site/
 * [new branch]      master -> master
```

We'd do a few things differently in practice, but that's the basic idea.

Now anyone who wants to see the changes can run `git clone programmer@ayeaye.byu.edu:repos/new_site/` and get a copy to work with. Once they are done they too can run `git push`. (In reality, if we're using a setup like this, they'd run `git pull` first to get any changes that anyone else has made, then apply their changes with `git merge` or `git rebase`. Once that is done, they'd run `git push`.)

Fixing a bug

Okay, let's say that some changes sometime in the last week have introduced a bug. Here's the log:

```
$ git log
commit cd7b4cc9073a512b52ade5da3dcf8c340604dde6
Author: Ashton Wiersdorf <ashton.wiersdorf@mailblock.net>
Date:   Fri Jun 2 14:20:05 2017 -0600

    said 'hello, world' on the landing page

commit e4b53b1a5b0ca79fad9ff629f8acbcdc5a5daefe
Author: Hector Bitbucket <hector@bitbucket.com>
Date:   Fri Jun 2 14:18:50 2017 -0600

    added a shout function

commit 42f607a97400e279217a687ded94eee496a5132b
Author: Ashton Wiersdorf <ashton.wiersdorf@mailblock.net>
Date:   Fri Jun 2 14:13:28 2017 -0600

    added a fibbinaci (sp?) function

commit e7e0d15ab2c6951d5c8efbdc4860d9fee361a34a
(...)
```

We figure out that the problem is in the file `js/basics.js`. We can see what commits affected this file by running `git log js/basics.js`.

Hector Bitbucket has always been a bit sloppy with his code. We can see what he changed in his commit by running `git diff 42f607a9 e4b53b1a`: (those strings are the commit tags from the "added a fibbinaci function" and the "added a shout function" commits respectively—just pull off the first few characters from the full commit hash)

```
$ git diff 42f607a9 e4b53b1a
diff --git a/js/basics.js b/js/basics.js
index 4b5b992..efd6edf 100644
--- a/js/basics.js
+++ b/js/basics.js
@@ -5,3 +5,7 @@ function factorial(n) {
 function fib(n) {
     return n == 0 || n == 1 ? 1 : fib(n-1) + fib(n-2);
 }
+
+function shout() {
+    alert('hey!)
+}
```

Hhmmm… looks like he didn't close a single quote in the `shout` function, nor did he end the statement with a semi-colon. We can now fix those changes and see if it works. If it does, we commit the change and push.

Also, we know now to get on Hector's case for committing bad code.