Workflow Disaster Prevention

This workbook is a summary of how to avoid (or mitigate) some of the more common problems solo developers, and small developments teams, run into.

License

Table of Contents

What Version of the File Is On My Server?....2
What Was I Thinking When I Wrote This?....3
Intro to Using Version Control....5
*&^! That's Wrong....12
Untested Code (Eventually) Breaks Stuff....19
My Client Changed Their Mind ... Again....21
I Changed Something, And Stuff Broke....23
My Computer Died ... Now What?....24
Two Clients Want Something Similar...Leverage Your Work....25
My [Collaborator] Overwrote My Work....26
About Design To Theme....28

What Version of the File Is On My Server?

The easiest way to deal with this is to have a very strict policy of how files on the server got there in the first place. It is most common to simply download a copy of a file before editing and uploading it to the server. Easy enough … if you only have one person working on the files.

If we are using version control, we can easily ask "what was I thinking when I changed this code"? with the following command (-1 can be modified to show any number of commits):

```
$ git log -1
```

You can also look at the revision log for a single file:

```
$ git log -1 <filename>
```

If you want to know what actually changed in the file, instead of just reading the commit messages, you can include the modifier -p which will include the lines that were changed as part of the commit:

```
$ git log -p <filename>
```

OR, if you want to write a shorter command, you can use "show" instead of "log":

```
$ git show <filename>
```

If your files are not under version control, you can still compare two existing copies of the file. Place a copy of the two files you want to compare into the same directory, and issue the following command:

```
$ diff --side-by-side --suppress-common-lines <file_1> <file_2>
```

What Was I Thinking When I Wrote This?

When you write new code, you should add documentation as close as possible to the code. This means adding comments to your PHP, HTML and CSS files. When I first started writing code over a decade ago, I had to comment everything, including simple "for" loops. As my confidence grew, I found I needed to add fewer comments to my code to explain the syntax and started writing only comments about the *intention* of what the code was supposed to do.

CSS comments use the following structure:

```
/** comment goes here **/
```

The comments can stretch over multiple lines.

HTML uses the following structure:

```
<!-- comment goes here -->
```

PHP uses the following structure:

```
# single line comment
/* multi-line comment */
```

Javascript uses the following structure:

```
# single line comment
/* multi-line comment */
```

In addition to your in-code documentation, you should also use version control commit messages to indicate the desired effect of the change in your code. A commit message is

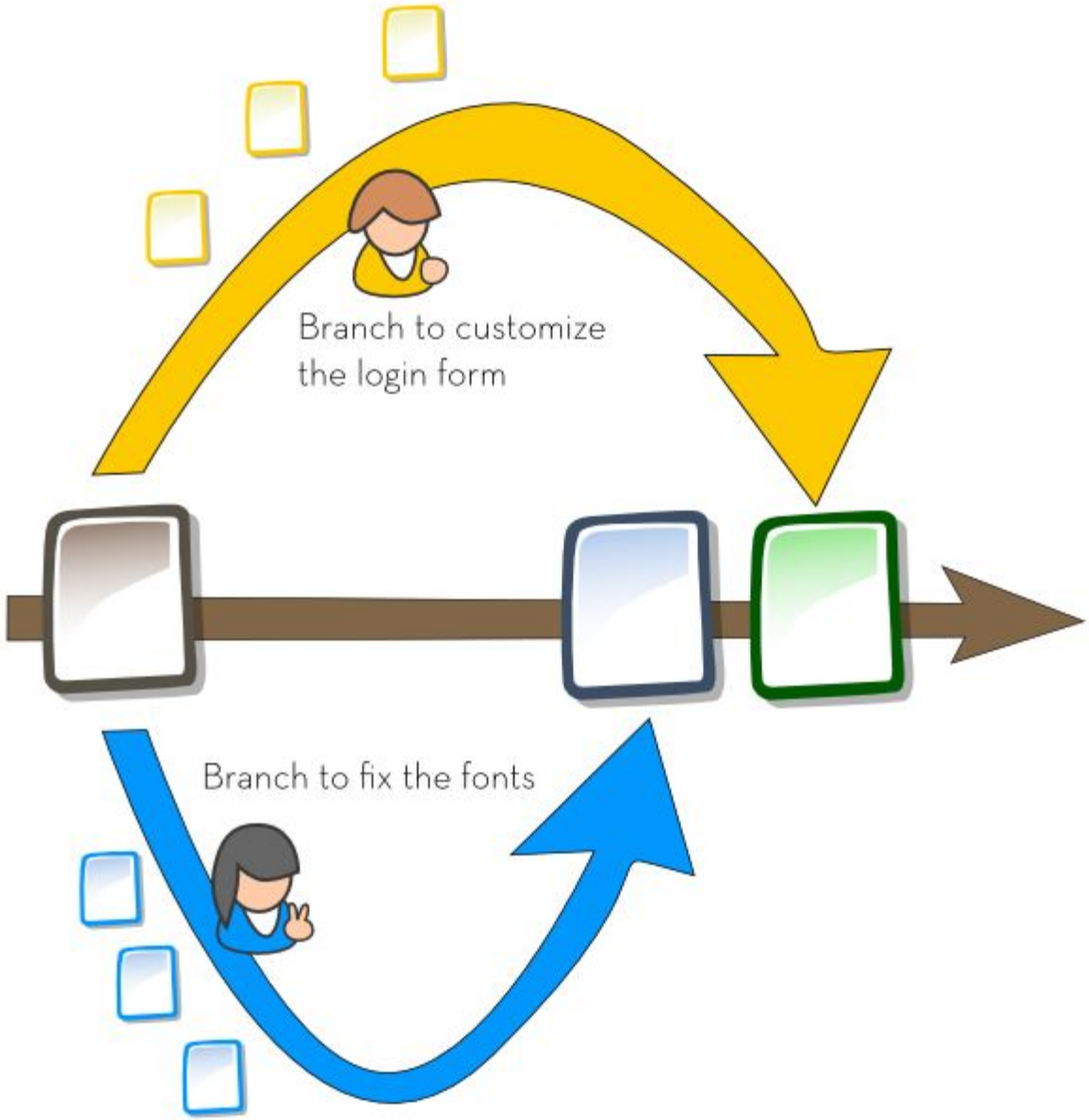
added as follows:

```
$ git commit -m “A terse description of the change.”
```

Intro to Using Version Control

With distributed version control we see more use of branches to separate the code tasks we're working on.

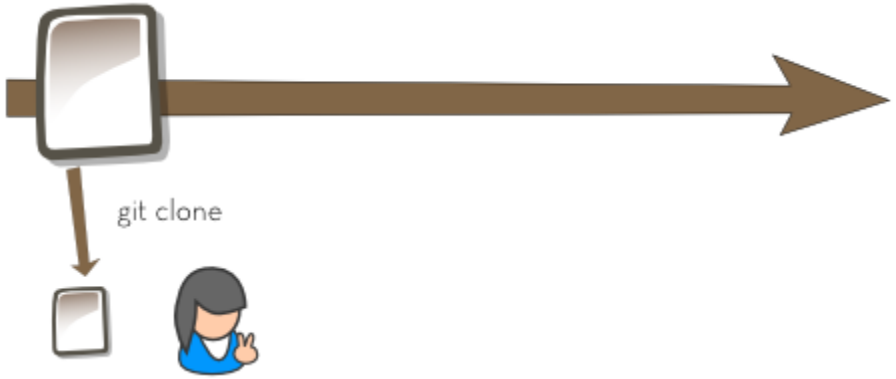
For example: you have eight things that you need to work on for a project. These pieces are discrete enough that they don't rely on one another. e.g. font sizes, and the formatting of forms on the login page. Small teams may want to work on separate *branches* for each of these tasks. This allows another teammate to pick up only that sliver of work to test.



Clone: Download a Copy of the Code Repository

Create a copy of the main project files on your main workstation:

```
$ git clone http://urltogitreponsitory.com/team/project_name.git
```



List all current local branches for this project:

```
$ git branch
```

Look for the *. This is the branch you are currently working on.

Branch: Create a Named Sandbox for Your New Code Feature

Create a named branch that will contain the work for this issue (for example, *global_fonts*):

```
$ git branch global_fonts
```

List all current local branches for this project, your new branch should be listed:

```
$ git branch
```

Switch to the fonts branch (I think of the command “checkout” as if my friend was saying, “this is awesome! check it out!”):

```
$ git checkout global_fonts
```

Confirm you are working on the correct branch (look for the *):

```
$ git branch
```

Add + Commit: Save Your Changes to the Local Repository

Begin editing the files in the branch, adding new ones if necessary. Each time you make a collection of changes you'd like to be able to "undo", make a commit to your local branch.

```
$ git add --all
$ git commit -m "A terse message about the changes you made."
```

Merge: Locally Combine Your Changes With the Work of Others

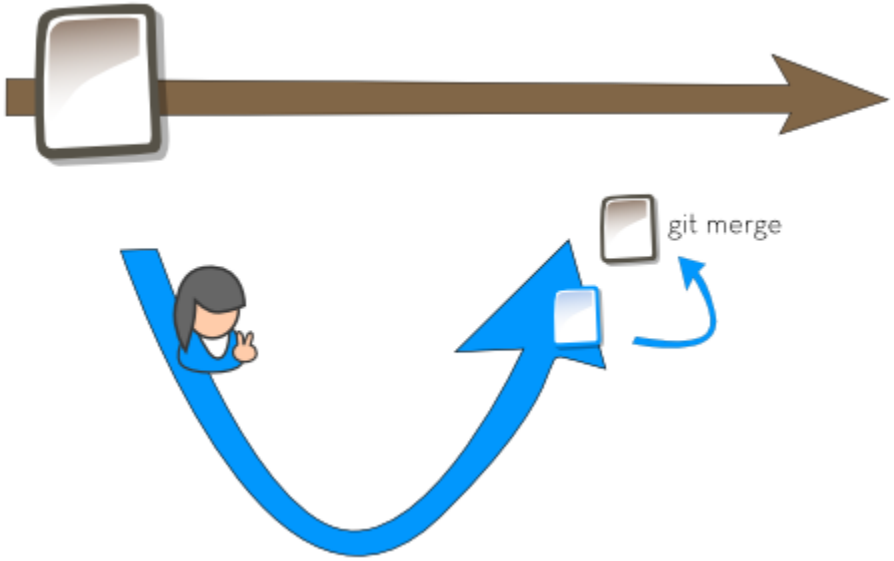
Ready to share your changes?

Update your local copy of the main code base:

```
$ git checkout master
$ git pull
```

Then merge your changes into your local copy of the master branch of the repository:

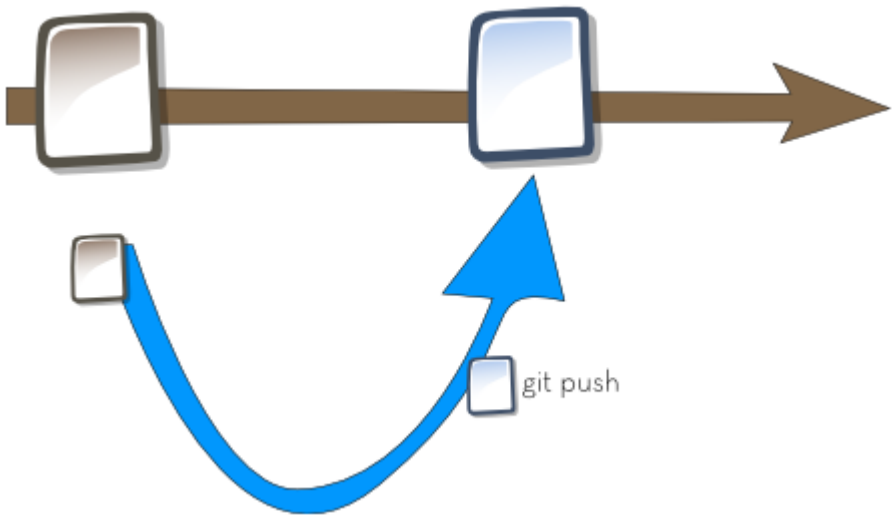
```
$ git merge global_fonts
```



Push: Share Your Changes With Others

Push your changes to the central code base (origin == centralized repo; master == my branch; remote_branch_name == the name of the branch in the centralized repo):

```
$ git push origin master:<remote_branch_name>
```



This will only work if you started by cloning a remote repository. If you need to connect the repository to your local work, you can use the command:

```
$ git remote add origin <server>
$ git push origin master:<remote_branch_name>
```

Add Milestones With Git Tags

At any point in time you can add “meta” commit information by tagging a specific commit. Tags are easily located *if* you don’t add too many of them.

To review existing tags, issue the command (without any modifiers or parameters):

```
$ git tag
```

To create a tag on a branch, issue the following command:

```
$ git tag <the_name_of_the_tag>
```

If you would like to add a message to the tag, you can annotate it with the following command:

```
$ git tag <the_name_of_the_tag> -a ‘Notes about this tag’
```

*&^! That's Wrong

Determine how much of your work is wrong. Is it a mistake since you last made a commit? Did you make a whole bunch of mini commits, and now you want to throw them out your work and get a fresh copy of the file? Or do you want to go backwards in time because a client changed their mind?

I Need to Backup One Step

Everything was going well, and then something wasn't right. You want to backup to the last commit. You don't want to save any of your work. You don't care that it might be right and it might just be a caching issue, you just want everything to back one step and clear the slate.

```
$ git checkout .
```

Or you can revert just one file, not the entire repository:

```
$ git checkout -- <filename>
```

The Work I'm Doing Right Now is Wrong, I Need a Fresh Copy of One File

First check to see that your file is listed as having been changed (if it's not listed here, you've already committed your changes, or perhaps the file isn't even being monitored by Git).

```
$ git status
```

You should get a report like the following:

```
# On branch master
# 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:   one.file.txt
#     modified:   two.file.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

At this point the changes to NEITHER file will be saved. To commit the changes to only one.file.txt, add it to the list of files to commit changes to:

```
$ git add one.file.txt
```

Check the status again:

```
$ git status
# On branch master
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     modified:   one.file.txt
#
# 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:   two.file.txt
#
```

```
$ git commit -m “save only the changes to the first file”
```

Now we see the status as follows:

```
# On branch master
# 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:   two.file.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

At this point you still want to destroy the changes you’ve made to the second file (two.file.txt).

The status message tells us how to discard the changes (marked in bold, above).

```
$ git checkout -- two.file.txt
```

Our changes have been destroyed, and a fresh copy is available for you. Magic, right?

I've Committed My Mistake

Did you notice too late that you'd made a mistake? No problem! In fact, this is the best way to make a mistake. It's far less risky to "commit" a mistake and go back in time than it is to grab a fresh copy of the file (which eliminates all traces of any work that you have done). At least when you commit a mistake you can prove you worked on something, whereas grabbing a new copy of the file (using the previous method) is the equivalent of saying "the dog ate my homework". Yeah. Sure.

The first step is to find the point that you want to go back to. This information is stored in Git's log. You can access the history with the following command:

```
$ git log
```

We've already used this command. The output is something like this:

```
commit 55e5206f9a11e986ad4525892cb46484c36cda59
Author: Emma Jane Hogbin <emma@hicktech.com>
Date:   Thu Nov 22 16:29:25 2012 -0500

    5th group of edits

commit 49fc9bf473c6cf2b3d84e55055b94571c4d328c8
Author: Emma Jane Hogbin <emma@hicktech.com>
Date:   Thu Nov 22 16:28:42 2012 -0500

    4th group of edits
(etc)
```

To go back in time you need to know the ***SHA***, which appears after the word "commit". A SHA

is a Secure Hash Algorithm. It represents a unique identifier for each commit you make. It is not sequential.

There is a long version of the commit number, and a short version. You can access the short version by using the --pretty-format modifier on the log command. For example:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```

Displays the following output the following (the first line is not displayed, it is for reference only):

```
SHA     - Author,           Time Ago    : commit message
55e5206 - Emma Jane Hogbin, 2 hours ago : 5th group of edits
49fc9bf - Emma Jane Hogbin, 2 hours ago : 4th group of edits
06004e9 - Emma Jane Hogbin, 2 hours ago : 3rd group of commits
22cea18 - Emma Jane Hogbin, 2 hours ago : saving changes to the first file
only
de5afb5 - Emma Jane Hogbin, 2 hours ago : adding two more empty files
6e287ae - Emma Jane Hogbin, 2 hours ago : initial commit of four empty files
```

Note: this is sorted reverse chronologically (newest at the top). If you prefer to have the newest information at the bottom (closest to the command prompt), use the following:

```
$ git log --pretty=format:"%h - %an, %ar : %s" --reverse
```

You need to locate the SHA for the last time things were “right” so that you can go back to that place. Once you’ve located the SHA, you have several options on how to proceed:

- Observe the past: Look around, without making changes.
- Start Over: Start a new branch in the past, abandoning (but not deleting) the current branch.
- Erase history (very dangerous).

- Personal time travel: remember history, but start fresh from a point in the past.

Option 1: Look Around Without Making Changes

The following commands will allow you to look back at the past without making changes.

```
$ git log
```

Find the SHA for the commit you'd like to return to, and then go back in time:

```
$ git checkout <SHA>
```

Look around at that time. You're currently in limbo and not associated with any branch. Issue the command git branch to see what this looks like.

To go back to the "present" issue the following command:

```
$ git checkout <current_branch_name>
```

Option 2: Start a New Branch in the Past

The following commands will allow you to step back in time. Although you've gone back in time, no one else will even be aware that there is a possible future in another branch. IF someone has merged their code with your "abandoned" future, there will be problems. Proceed with caution.

```
$ git log
```

Find the SHA for the commit you'd like to return to, and then go back in time:

```
$ git checkout -b new_branch_name <SHA>
```

Now there are two branches, one is abandoned (formerly "today") and the other starts at SHA and has no knowledge of the work you had done between then and now. You do not need to checkout the new branch, you are already using it.

Option 3: Erase History

Very DANGEROUS as you might not be able to merge with others who've checked out versions of your work during the period that you're deleting.

```
$ git reset --hard <SHA>
```

Your "undo" work will never be recoverable. Ever. There are no backups and no take-back-sies.

Option 4: Personal Time Travel

In my opinion, this is the best option. It retains the complete history of what you've done, but folds time onto itself and brings the past up to the present as well as having it in the past. In this scenario, you are changing all files (represented by the . in the command checkout).

```
$ git log
$ git checkout <SHA> -- .
$ git commit -a -m "Save the 'back in time' checkout"
```

You now have the same branch with two copies of <SHA>, but with two different commit messages. All of your work to date on this project has been saved into the same branch.

Untested Code (Eventually) Breaks Stuff

Maybe you've been really lucky and none of your code has ever had a mistake in it and you're not really sure why you'd want to test stuff before putting it on your live server. Someday that lucky streak is going to catch up with you and you're going to have a colossal mess on your hands. (Even if I'm projecting, why risk it?)

If you haven't had a test environment before now, here's how to set one up.

Short version: use the Drupal module Demo (http://drupal.org/project/demo) to replicate a site entirely in a test environment.

Longer version:

1. Create a test environment for your Drupal site. There are many appropriate tools to create the test environment including Vagrant, WebEnabled or Acquia's Developer Desktop.
2. Install Drupal in your test environment.
3. With Drush, download all modules used in the live environment. If there are custom modules, upload them to the test environment.
4. Using the Drupal module Backup and Migrate, create a snapshot of your live server.
5. On the test server, install the Drupal module Backup and Migrate and then "restore" the live server data to your test environment.

6. Implement the changes you'd like to test on the test server.
7. Check to see if stuff breaks.
8. If nothing breaks, ensure your backups are up-to-date on your live server, and replicate the changes you made to your test server in step 6 on your live server.

Note: while it is a smart idea to work with "real" data in your test server, it is not a smart idea to use Backup and Migrate to bring all data back to the live server. Export configuration settings using the Drupal module Features (http://drupal.org/project/features) where possible to move functionality from the test environment to the live server. If there are not many working parts (e.g. it's just a View you're updating), you may not need to use the Features module.

My Client Changed Their Mind … Again

You need a giant undo button to be able to roll back your code to a previous state. (We've already covered this to a certain extent.) This time we're not rolling back to an arbitrary point, we're rolling back to a specific milestone. You can add the milestones using tags (highlights on the news reel), or branches (completely different channels).

List the existing tags on this branch (project) in alphabetical order:

```
$ git tag
```

To add a tag (with notes) to your project, use the following command:

```
$ git tag -a <tag_name> -m 'Notes about this new tag'
```

Drupal uses tags to denote different version within a whole-number series. e.g. 7.3.x --> 7.3.1 vs. 7.3.2. So for a given module the tags might be:

```
7.3.1
7.3.2
7.3.3
```

Whereas I like to use words for my client projects. They might look something like this:

```
phase1_design_blue
phase1_design_orange
```

The command "show" which we used previously also works with tags. If we need to retrieve the SHA for a specific tag, we can issue the command as follows:

```
$ git show <tag_name>
```

We can now travel back in time (without deleting anything, just in case they change their mind again) as we did in the previous disaster scenario where our work was wrong.

```
$ git show <tag_name>
```

Obtain the SHA for the tag we want to revert to.

```
$ git checkout <SHA> -- .
$ git commit -a -m “Reverting back to the <tag_name> of the project at the client’s request.”
```

I Changed Something, And Stuff Broke

When you're saving your changes into your version control system, use the smallest possible commits. Every time you check your work in your browser, you should definitely be committing your work. What if the browser was caching something? Or Drupal was caching something? Make your undo button as useful as possible by having lots of tiny commits instead of just one giant commit at the end of the working day.

Within each change that you make, do only one thing at a time. For example: if you're working on the formatting of the login screen: change only the fonts, or the colours at a time. Do not change the fonts AND the colours in a single commit.

My Computer Died ... Now What?

You've got backups, right? No biggie. Just download your backups and restore your system. Yeah, you might lose a day's worth of work (and the time to download your backups), but it's not so bad...

At a minimum something like Dropbox, Wuala, or Ubuntu One can be configured to periodically save your work to the cloud. I also use JungleDisk to backup my workstation nightly to an AmazonS3 bucket.

On my server I also have my backups as follows: database dumps from Drupal to system files. Machine image backups from my hosting provider. Drupal database backups into NodeSquirrel. You can never be too careful.

Check your backup files at least once a month to make sure your automated scripts are working properly.

Two Clients Want Something Similar...Leverage Your Work

Before you go sharing client work between clients, make sure you have permission. Who owns the copyright for your work? If you're in a work-for-hire situation, the *client* might own your work. Assuming all the copyright stuff is out of the way... Think about how Drupal divides its modules to get the right granularity for your repositories. You want just enough, but not too much in a single repository.

1. Create a central repository for similar client work.
2. To work on a client project, clone the repository and create a new local branch.
3. Do stuff for your client.
4. If it makes sense, merge the client enhancements with the original master branch, and push your changes back to the central repository.

The step-by-step on how to clone, branch, merge and push with git are all covered in the first section of this document.

My [Collaborator] Overwrote My Work

This is no longer possible now that you're all using version control! When you attempt to merge your work back into the central repository, Git will try to elegantly combine your changes with your collaborator's. If the exact same spot in a file has already been changed, you will have a **conflict** that needs to be **resolved**.

When a conflict occurs, Git will keep a copy of all the offending code in your local branch. In a text editor, open the file(s) with the conflicts and locate the conflict area(s). They will be identified like so:

```
<<<<<<< HEAD
Your code
=======
Their code
>>>>>>> master
```

To resolve the conflict, complete the following steps.

1. Edit the file to remove the code you don't want to have in the newest version of the software.
2. Delete the lines beginning with <<<, === and >>>.
3. Save the file.
4. `$ git add <filename>`
5. `$ git commit -m "Resolving conflict message"`

6. Resume the work you were doing when you first attempted to merge the two branches. Typically this is a code push back to the central repository.

Depending on the scale of your project, it's probably smart to only have one central repository whose code can pushed to the live server. This will prevent collaborators from making unintended changes.

Random Stuff

A list of random commands I know about, which are interesting, but not necessarily best practices.

Extract A Specific Version of a File

In the log you were able to see a number of revisions, if you'd like to "extract" a previous version of a file, you can use the following command:

```
$ git show <branch>~<#_of_commits_ago>:<filename> > <exportedFilename>
```

About Design To Theme

Emma Jane Hogbin is the founder of a great little Drupal consulting and training agency. She makes theming Drupal easier, faster and more profitable.

- **Drupal Site Building Consulting**: We're great at saving you money. There are thousands of Drupal modules out there that will get you from idea to finished Web site faster and with higher profits. With a Site Building Consultation we can help you choose (install and configure) the *best* modules for your next project. If you're tired of handing over all your profits to your programmer we need to talk.
- **Support for Small Businesses and Designers**: Did you get in over your head a little bit with a project? We can help you get unstuck with gentle technical support that will make you feel smart and wonderful and capable of taking back control of your Drupal project.
- **Drupal Training**: Drupal site building and theming training sessions are available on-line. Check the web for a list of upcoming workshops. Custom training is also available.

Accolades

"Emma is an amazing teacher." — Betty

"Thank you for sharing your experience through e-books in addition to seminars and presentations. It's really helpful to have short, **easy-to-use examples to learn from as well as refer back to** while trying to develop good Drupal theming skills." — Spence

"Emma Jane worked with me on a dramatically ambitious Drupal project a couple of years ago. She managed to keep my overactive imagination in check so we focused on realistic goals and milestones, all the while **making me feel like I could get my hands dirty** in the project. She was timely, proficient, and a joy to work with." — Kim Werker, founder of CrochetMe.com

"Emma Jane combines the perfect amount of predictability and spontaneity...her technique has **sparked my excitement** about developments in technology and has inspired me to engage in new projects. She presents herself as very approachable and always answers questions thoroughly, making sure that the user feels comfortable and at ease." — Jorge Castro, External Developer Relations, Canonical Ltd.

"Taking your course is one of the **best investments I have made**." — Louise