Local Customization Management - Why Git isn't just for developers

Table of Contents

Git is a wonderful tool that the Evergreen developers use to keep track of the changes in Evergreen and OpenSRF. It allows for multiple developers to work on the same area at the same time and in many cases figures out most of what was intended automatically later.

And that power isn't just useful to developers. Git is a Version Control System, in that it keeps track of multiple versions of files. New versions of files such as, say, the customizations being made to an Evergreen install for branding are just as valid a thing to track as the main code itself. By using the same tools that the developers are using you also get many of the same benefits the developers do. As the developers change things around your changes Git can, in many cases, re-apply your changes without having to ask you. Only when two different changes have touched the same lines will you have to figure things out for yourself.

As an added bonus, Git keeps older copies of changes as well, so you can go back to a previous version later, perhaps to see how something had been done previously, or because something broke and you need to go back to a known working version.

1. Git Basics

1.1. Quick Git glossary

Repository
: A collection of files and changes to those files managed by Git. Git is a distributed version control system, unlikely CVS or Subversion, and is designed to make it easy to copy a Git repository and push or pull changes between repositories.

Branch
: A sequence of changes recorded in a Git repository. A Git repository can have many branches, which can be used to organize lines of development and customization. Managing a Git repository ultimately boils down to deciding when to create and merge branches.

Commit
: A discrete set of changes to one or more files recorded in a Git repository. A commit

also has a description, an author, and date attributes.

Patch
: A Git commit expressed as a "diff" file.

1.2. Installing Git

The first task in using Git to install Evergreen and manage customizations is to install Git itself. For this you should rely on your package manager, such as apt or yum, but take note that the package you want is likely named "git-core" instead of just "git".

Installing Git examples

```
# Debian/Ubuntu
sudo apt-get install git-core
# Fedora/Red Hat
sudo yum install git-core
```

Once installed you should have access to the git command.

1.3. Configuring user information

Git stores author information with commits, but has horrible default settings. Thus it is highly recommended that you set your name and email address before committing anything with git.

You can set your name and email address for all repositories with two commands.

```
git config --global user.name "Firstname Lastname"
git config --global user.email "your_email@youremail.com"
```

This is important because it indicates who made a change when multiple people work on things, and if you wish to share your work later you will want to make sure your name is on it.

1.4. Cloning a repository

Next up is cloning the Evergreen repository. Git provides a clone function for this purpose that copies an entire remote repository to the local machine. While you can clone from any mirror you want, we recommend always cloning from the original (barring local mirrors thereof, of course).

```
# Clone the Evergreen repository
git clone git://git.evergreen-ils.org/Evergreen.git
```

This will create an "Evergreen" folder, in which you will find the "master" branch checked out.

1.5. Checking out branches

Assuming you are in a git directory (say, the Evergreen folder created by cloning above) you can "check out" new branches. These branches can be "local" or "remote", and you can create local branches from remote ones.

Checking out a local branch is as simple as using the checkout command on the local branch.

```
# Checkout a local branch
git checkout local_branch
```

To check a remote branch out as a local one you need to specify both the local name to use and the remote branch. This is also fairly easy, and only needs to be done once for each local branch you create.

```
# Create a local copy of a remote branch
# Note that this doesn't involve network traffic, and can be done offline
git checkout -b local_name remote_branch
```

Technically, this runs "git branch" in the background before checking the resulting branch out. For more information see "git checkout --help" and "git branch --help".

Note that remote branches are named based on the remote's name and the branch name. By default the remote created by the clone command is "origin" and thus all remote branches will be named "origin/branchname".

As a shortcut, if you are checking out a remote branch from your origin remote and wish to use the same name you can use the first method of checking out. Thus, to check out the 2.2 working branch you would actually checkout origin/rel_2_2.

Example checkouts

```
# Switch to local branch rel_2_2, or create from origin if non-existent
git checkout rel_2_2
# Create a copy of rel_2_2 named rel_2_2_local
git checkout -b rel_2_2_local origin/rel_2_2
# Switch to a previously created branch named local_customizations
git checkout local_customizations
```

2. Installing from a Git clone

For the most part installing from a Git clone of Evergreen or OpenSRF works the same way as installing from a tarball, just with a couple of extra steps tacked on.

First off you need to install and run the tools that build the configure script, and after running "make install" for the first time you need to install the dojo toolkit.

The best source of instructions for handling this is the README file itself, in the "Developer Instructions" section. By checking there you will be able to keep up with changes in dojo versions as well as additional needed packages.

The other potentially significant issue with installing from Git is translations, in that they are not built in the checked out branch. For a development system or one that only caters to en-US users this may not be a problem, but for everyone else building updated translations is important.

A more complete translation building document can be found at Internationalization (I18N), Localization (L10N), and Globalization (G11N), but the basics are as follows. Note that you would ideally only do this just before installing.

```
# Install the translation build tools with your package manager. This only needs to be done once. On Debian/Ubuntu:
sudo apt-get install translate-toolkit python-dev python-levenshtein python-polib python-setuptools python-simplejson python-lxml
# Change to the build/i18n folder
cd build/i18n
# Build all locales
make install_all_locales
# Or, be picky (also useful when testing specific locales)
make newpot # install_all_locales does this automatically
make LOCALE=fr-CA install
```

Once that is completed the various translation files present in the tarball releases will be generated and ready to go, and should be copied into place as appropriate.

3. Adapting existing customizations to git

Once you can install from git you will want to get your existing customizations into it. If you don't have any yet then welcome to the community!

To start getting your customizations into git you should check out the release tag for the tarball you used. These will generally be in the form of "origin/tags/rel_?*?", for example "origin/tags/rel_2_1_0" for the original 2.1 release. While you can simply use the rel_?*? branch name I recommend using something more distinctive. In fact, for reasons I will go into in a moment, I recommend having multiple branches.

Once you have the branch checked out you can simply copy your edited tarball files over the branch. Git will keep track of which files actually changed and which ended up

identical. At this point you could store all of the resulting changes as one big commit, but that can be an unmaintainable mess. Instead I recommend splitting things up across commits, and even across branches.

Splitting things across well-named commits will make it easier to find changes later when looking to review them, makes those changes easier to remove if desired, and simplifies sharing those changes with the community if desired. In addition, when moving your changes to later versions of Evergreen you will be dealing with smaller chunks in the event something you changed conflicts with a change made in Evergreen.

Splitting commits across branches can keep changes grouped logically and simplifies having multiple people providing customizations down the road. For example, you may want to keep staff client changes separate from OPAC changes. If you have multiple skins for different library OPACs it is probably a good idea to keep them in a single branch each.

Luckily Git provides us with ways to take only certain changes when making commits.

3.1. Finding changed files

The first step in committing your changes is noting what files have been changed. Git provides us with the "git status" command, which shows us all modified files, as well as new files or folders. Modifications can be "staged" to be included in the next commit, or "unstaged" wherein they will not be added to the new commit. New, or "untracked," files and folders can be staged just like modified ones, however, and are treated as if the previous version was an empty file.

The same file can be both "staged" and "unstaged" when some of the changes made to it are being committed and others are not.

3.2. Staging changes

You can see the full set of unstaged changes in a file with the "git diff" command. Without any arguments it will show you all unstaged changes (but not untracked files), with arguments you can limit it to specific files or folders.

If you want all of the changes in a specific file you can add it to the commit with the "git add" command. If you want to be more picky you can use the -p option to trigger interactive selection of changes to stage. This is useful when two somewhat unrelated changes have happened in the same file.

You can add as many changes as you want to a given commit.

If you pass in a directory instead of a filename everything in the directory will be added, or prompted about with -p active.

3.3. Committing the changes

Once you have staged your changes you need to commit them. This is accomplished with the "git commit" command. By default it will launch an editor to ask for a message, and I recommend ensuring that the message is descriptive of the intent of the change.

For example, if you are changing a color in a CSS file you could simply label the commit as "Change CSS file", but that won't tell you much later. Instead you should go with something more descriptive, taking advantage of the summary and full message. Note that the full message doesn't have to be very long.

Example commit message

```
Change Default Background Color

Our logo doesn't work well on a white background, so change to a light grey.
```

In the example above the "Change Default Background Color" will show up in summary views of commits and gives a basic idea of what was being changed. The full message

explains the reasoning behind the change, letting yourself and others know why the change was made later. Note that if the subject is longer than around 50 characters it may be truncated in some lists.

Oh, and just to be clear, the blank line is important as it splits the subject (the first line) from the rest of the commit message. This is thus a bad idea:

Example bad commit message

```
Change Default Background Color
Our logo doesn't work well on a white background, so change to a light grey.
```

You can commit frequently as you stage files or combine larger groups of changes into single commits. A decent goal should be related changes all making it into the same commit, with minimal, if any, unrelated pieces in any given commit.

Example Stage/Commit Command Set

```
# Find changed files
git status
# Check changed file
git diff Open-ILS/web/opac/skin/default/css/layout.css
# Stage all changes in that file
git add Open-ILS/web/opac/skin/default/css/layout.css
# Check second changed file
git diff Open-ILS/web/opac/skin/default/xml/footer.xml
# Only stage some changes
git add -p Open-ILS/web/opac/skin/default/xml/footer.xml
# Commit the staged changes
git commit
```

3.4. Switching branches

Once you have collected all of the commits for a given broad topic, say OPAC customization, you may wish to switch branches to split off changes to other parts of the system. The most reliable way to accomplish this is to stash your current changes, checkout the new branch, and then pop the changes off of the stash.

Changing branches for storing additional changes

```
# Stash the existing changes
git stash
# Checkout the new branch
git checkout -b staff_client_changes origin/tags/rel_2_1_0
# Pop the changes off of the stash
git stash pop
```

Once you have done this you can return to staging and committing changes.

3.5. Discarding changes

In the event there are changes you no longer want, or are unrelated to your own work, you can discard them with the "git checkout" command, limited to a specific file. Just like the add command you can use -p to interactively choose changes to discard.

```
# Discard changes to file
git checkout Open-ILS/web/opac/skin/default/css/layout.css
# Selectively discard changes to file
git checkout -p Open-ILS/web/opac/skin/default/css/layout.css
```

If you pass in a directory all changed files within it will have the changes within them discarded, or prompted about if you have -p active.

4. New Changes

Future changes you wish to make can be built on existing branches or in new ones. Simply

edit files as desired, stage the changes, and commit them.

5. Installing Evergreen with your changes

Once you have your changes in place you will want to install them. For this purpose I recommend making an "install branch" that you have merged your changes into. This branch will consolidate your various change branches into a single complete whole from which you can perform your install.

Once you have created your branch you will want to merge each customization branch into place. This is accomplished with the "git merge" command. It merges whatever branch(es) you specify into the current one.

```
# Merge Client Customizations
git merge client_customizations
# And OPAC Customizations
git merge opac_customizations
# And two library skins at the same time
git merge library1_skin library2_skin
```

6. Upgrading Evergreen

The real power in keeping your changes in git is being able to easily upgrade to later releases of Evergreen. Whether you are upgrading to the next security release or to another major version you can use git to get you there.

Either way, the first thing you want to do is update your copy of the server's repository. This can be done with the fetch command.

```
# The --all parameter says to update all remotes.
# You can specify just one instead of --all if you want.
git fetch --all
```

6.1. Minor upgrades

When staying in the same major version you can use the "rebase" feature of git to re-apply your changes to the later code.

Unfortunately, moving from one tagged release to another means we will need to use an interactive rebase, so that we can remove the version numbering commit.

```
# Interactive Rebase
git rebase -i origin/tags/rel_2_1_1
```

One of the commits, hopefully the first one, will say something about version numbers, perhaps stating "Bumping version numbers" or similar. You will want to delete that line, then save and close the editor.

With any luck things will go smoothly and you will have had to do nothing special. If things don't go smoothly then git will stop where things have gone wrong so that you can fix them. At that point you would treat things like any other changes you have made, but resolving the conflicts between two sets of changes to the same areas. Also, instead of "git commit" you will want to use "git rebase --continue".

Once you have rebased all of your customization branches you can proceed to making your install branch and installing Evergreen.

Another tactic is to instead rebase your changes onto the in progress branch releases are built off of, say origin/rel_2_1. In that case you can usually merge without concern across updates until something conflicts or you want to be looking at more up to date code. Only then would you go back and rebase the branch again.

6.2. Major upgrades

Major upgrades are a different matter. In this case you should build a new branch for each customization branch you have, say with the new version added to the end.

Once you have your new branch you will want to cherry-pick each commit from the old one onto the new one. To do that you will need the commit hashes for the commits you care about, and for that you can use git log.

```
# Show commits on a branch, summaries and hashes only
git log --pretty=oneline client_customizations
```

Once you have your list of commits you want to apply them from oldest to newest with git cherry-pick. If there are conflicts you would resolve them just as with a rebase, but in this case you return to using "git commit" when you are done staging the resolved conflicts.

```
# Cherry-pick - You can use a small piece of the front of the hash, or the entire thing
# And no, neither of these are likely hashes
git cherry-pick ffffffff
git cherry-pick eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
# In the event of a failure, go and correct things. Then commit it:
git commit
```

A rebase is basically a more automated version of the cherry-pick steps just described.

Once you have your new branches you can then build a new install branch based on the new version.

7. Sharing with the community

Sometimes you have fixed a bug or built a new feature that you then want to share with the community. If you have been creating decent commit messages and keeping your changes split up fairly well then there are really only two more things to take into account when it comes to preparing your work, and then you have to actually share it.

7.1. Preparing a branch to share

The first is that you will need to sign-off on your changes, to indicate that you are permitted to submit the patch and is treated as signing the Developers Certificate of Origin (DCO).

The second is that you will want to split off your relevant commits into a branch of their own, so that only the changes you want to submit are in the branch.

Luckily, the process of building the branch provides us with a wonderful opportunity to apply the sign-off. As you cherry-pick the relevant commits into place on the new branch you can add the -s parameter to sign-off on the them automatically.

```
# Cherry-pick with sign-off
git cherry-pick -s ffffffff
git cherry-pick -s eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
```

In the event that you need to update the commit message on one of the commits, to add more detail or correct typos or such, you can also amend the commit afterwards to edit the commit message.

```
# Amend the commit, either to add more changes or to tweak the commit message
# To add more changes, stage them just like you would for a normal commit
git commit --amend
```

7.2. Sharing the work

Once you have your branch you still need to make it visible to the world. I highly recommend reading the Git page on the dokuwiki, specifically the "Getting commit access to working repositories" and "Quick start for Evergreen contributors" sections. In general, however, your goal is basically:

1. Find a public location to push to, such as the working repos. In the latter case, make sure you submitted your SSH key.
2. Add the public location to your git checkout with the "git remote" command.
3. Push your branch to the remote you are using.
4. Publish the location of your branch, ideally via Launchpad.

8. Other uses of Git

There are many other things you can use Git for. The key is to remember that Git helps track changes, and that any folder can be made into a Git repo simply by running "git init" while inside of it. Even those inside of other Git repos.

> The following line is likely groan-worthy. At least. You have been warned.

Yes, that means you can put Git in your Git so you can change while you change.

Potential other uses of Git include:

- Managing configuration files. By pushing changes into a Git branch you can track who changed what, push updates more easily across multiple servers, and make it easier to undo changes that turn out to be problematic.
- Swapping out files. By adding a Git repository to a folder on your web server you can swap out files quickly and easily by committing files to and checking out different branches. One such use I saw recently was to temporarily swap out CSS files for a weekend.
- Moving files around. With remotes you can use Git to move changes, on a push and a pull basis, across multiple machines. Regardless of what those changes are.
- Backing up files. Remote repositories can hold copies of your changes that you can easily get at later, though you should ensure any passwords are kept in private locations and for sanity reasons **DO NOT BACK YOUR DATABASE UP WITH GIT**.

9. Going beyond Git 101

Once you're comfortable using Git to manage your Evergreen customizations, there are a variety of additional resources available to make Git even more useful.

9.1. Repository management

One of the things about Git repositories is that you are unlikely to have just one. Over time, you're going to accrete a number of repositories, and it can be handy to have a tool to easily create repositories on a central server and grant access for users to pull or push from them.

Examples of Git repository mangement tools are:

1. Gitolite
2. Gitosis

9.2. Additional reading

1. Git Community Book
2. Pro Git book by Scott Chacon

Last updated 2012-05-03 16:09:12 EDT