Modern Version Control: Creating an Efficient Development Ecosystem

Nic Bertino
Santa Clara Law Webmaster
500 El Camino Real
Santa Clara, CA 95050
1 (408) 551-3000 Ext 6137
nbertino@scu.edu

ABSTRACT

In 2011, Santa Clara University School of Law Technology and Academic Computing (LTAC) identified that its version control system could greatly benefit from the use of modern source control management software. Source code for high value projects such as the Santa Clara Law website, were previously held in a Subversion (SVN) repository in a client-server model, providing version control and redundancy. Because of the resource footprint associated with SVN, only projects with high importance could be setup with version control. As more web-based applications were introduced, the need for a more efficient revision control system arose.

Git, a highly efficient decentralized version control system (DVCS), was selected after evaluating similar technologies. This change transformed the entire development process, making the development cycle more streamlined and with greater flexibility. In the early use of Git, LTAC also discovered its use as a deployment tool, increasing redundancy on servers and reducing overhead usually associated with revision control. It also serves as the vital link between LTAC's issue tracking system, Redmine, and the development team. The introduction of Redmine has helped LTAC monitor website issues, manage projects, and continually review changes to the code base.

LTAC has created a development ecosystem that provides redundancy and accountability using open source products that carry no cost. Git has significant performance gains over SVN, making its integration and use less frustrating and distracting for developers. Redmine gives developers and customers the opportunity to organize, track, and resolve issues. The flexibility of the technology used means that any project, from a content management system to a one-off script, can benefit from source control without large costs or long deployment times.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – *Version control*

I.7.1 [Document and Text Processing]: Document and Text Editing – *Version control*

General Terms

Management

SIGUCCS'12, October 15–19, 2012, Memphis, Tennessee, USA.

Keywords

Git, Redmine, Issue Tracking, Source Control, Decentralized Version Control, Automated Deployment

1. INTRODUCTION

The Santa Clara University Law School' Law Technology and Academic Computing (LTAC) department supports Law faculty, staff, and students through a variety of means. One area of responsibility is the deployment of web applications and appliances, with the main Law website being the high value application that serves as Santa Clara Law's front-facing identity on the internet.

Deployed applications often require customization or modification of source code. These enhancements may introduce issues, and source control management allows the development team to quickly locate changes to the source that may have introduced the bug. In other uses, a new developer working on an application may need to become familiar with the development history. This change history is crucial to quickly understanding the life of an application.

The "safety net" that source control management software provides should not impede on the workflow of the developer; it should also allow anyone working on the code to quickly acquire, view, and contribute to its history. LTAC's legacy SVN system served as a source control management solution, but its centralized nature and sheer size made it a candidate for replacement. In moving to Git, LTAC realized workflow efficiency improvements coupled with fewer systems resources.

It should be noted that version control is not a replacement for backup systems. It is best used as in addition to disaster recovery initiatives. Version control should not be used on non-project or system files under any circumstances. Backups of these files should be made using rsync or similar.

2. REVISION CONTROL CONCEPTS

Revision control operates on the premise that files are watched for changes. These changes are stored in a repository. Exact changes, additions, and subtractions (deltas) are captured and stored in the repository, documenting the history of the project. Multiple file modifications are grouped by commits, which are generally associated with a bug or feature. In team situations with two or more developers working on the same code, revision control can become a powerful tool by assisting with the merging of two persons' work. Developers can branch off of working states of the code and easily merge their changes if successful or delete them if needed.

Similar to a ghost image of a computer or a snapshot for a virtual machine, revision control allows a developer the ability to restore a file or group of files to a previous state easily. This redundancy

allows developers to experiment with code and easily revert to a working state in the event that their code creates more issues.

2.1 Centralized Version Control

Centralized version control focuses on a client-server approach to revisions. All actions are generally performed against the server, meaning a checkout, check-in, or history review would be downloaded from a server hosting the repository. This approach, used by SVN and CVS, taxes network resources, along with creating a major point of failure in the event that there is an issue with the central server.

2.2 Decentralized Version Control Systems

DVCS operate on the principle that all developers have a full and complete local copy of the repository, and changes are transmitted via incremental patches. This methodology creates natural redundancy, as every person that clones the repository has full access to the project files and their history. Version control operations, such as commits, viewing file history or differences, are significantly faster as the repository is kept locally.

DVCS methodology isn't just for programmers; Apple OSX Lion introduced Versions[1], which automatically creates and stores snapshots of files as a user is working. The need for version control originated from software development, but the fundamental benefits can be used by anyone creating and modifying files on a computer system.

2.3 SCM Evolution

Currently, many free and proprietary options exist for version control. Local-only options such as Source Code Control System (SCCS)[2] and its successor Revision Control System (RCS)[3] were the first distributions of source code management software. Concurrent Versions System (CVS) was introduced in 1990[4] as the first client-server technology, allowing developers to collaborate on common source code by checking out a codebase, making changes, and checking the changes back in.

SVN was created in 2000[5] as another client-server application with the intent of drastically improving upon CVS. In the proprietary realm, BitKeeper[6] was launched in 1998 as a distributed version control software (DVCS) marketed toward medium and large businesses.

In 2002, BitKeeper was controversially selected for the development of the Linux kernel[7]. Due to issues with the licensing model of BitKeeper[8], Git was created in 2005 to provide an open source, distributed option for work on the Linux kernel. Two other open source applications also emerged that year, with Mercurial and Bazaar seeing initial releases.

3. CONVERSION TO GIT AND DVCS

The qualification of a new source control management tool consisted of an evaluation of the available DVCS programs. LTAC elected Git for its widespread use, update cycle, documentation, ease of use, and flexibility. Git is used by many high profile open source projects, including the Linux kernel, Ruby on Rails, Perl, and Android. Resources for the conversion from SVN to Git were also abundant. The objective of the transition was to migrate the Santa Clara Law website, and to specifically reduce the repository size and integrate production, development, and staging servers into the DVCS environment.

3.1 Planning

Planning the conversion to Git required careful consideration of the migration of previous data, setting up a branching model, and managing repository permissions. Due to the centralized nature of SVN, migrating the existing revision history proved difficult. Ultimately, it was decided that the last stable version would be checked in as the "master" branch. The legacy SVN codebase would remain in place for additional reference as needed.

LTAC designed a branching model that worked off of the master branch. Each server in the environment had its own branch; the production server had a "prod" branch, development had "dev", and the local "alpha" environments had local branches. The servers would sync only with their branch, meaning a change made on the developer's workstation could be merged with the development branch and synced to the server for testing prior to a rollout to production. Changes to the website were now handled by Git without the use of FTP (see Figure 1).

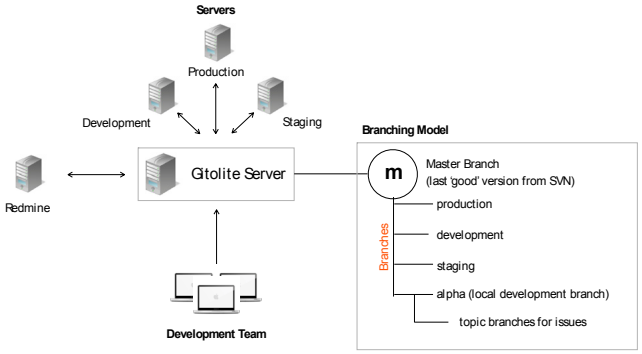

Figure 1 Branch model and deployment planning

The "trunk" from SVN was exported into Git, and had an initial size of 4.06 gigabytes (GB). Ignoring non-codebase related files, particularly binary files, such as MP3, PDF, and FLV files that had been generated by users, reduced the repository size significantly. After finalizing the files that should not be indexed by version control, the repository size was 334 megabytes (MB). With only the most critical code files and all user-generated files ignored, the repository size was further reduced to 9.8 megabytes (MB) (see Figure 2).

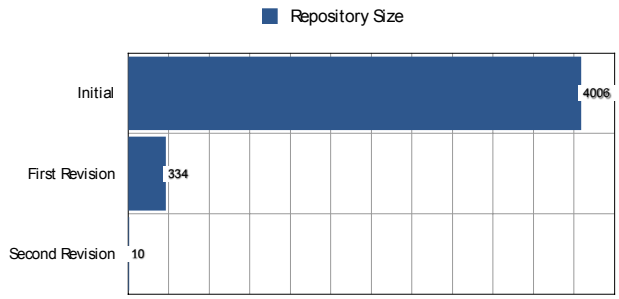

Figure 2 Repository size in megabytes after adjustments to Git ignore file.

3.2 Performance Gains

The smaller repository size resulted in faster implementation and deployment of the website. Setting up a clone of the development server now just required using Git to synchronize the "dev" branch. The total size of the SVN server at the time of retirement was approximately 14 gigabytes (GB); as of writing, the remote Git server is 409 megabytes (MB). Reliance on the local network is also minimized.

The nature of the application also allowed further performance gains with regards to version control. Because LTAC uses a content management system that publishes static pages daily, those published files did not need to be indexed. Instead, a "hook" was created that published the files when the repository was first cloned to the server or workstation. Redundancy on those files is handled by a database, so tracking revisions proved to be superfluous.

A perceived disadvantage to distributed systems is repository size on the initial clone of the repository. Because the entire commit history is copied, the repository size can grow quickly and eventually eclipse the size of the working directory. Taking actions early to eliminate files that do not need to be tracked or utilizing submodules can reduce the overhead of the repository. Some application frameworks, such as Ruby on Rails, have premade ignore files that automatically disregard unnecessary files from the Git repository.

3.3 Access Control Using Gitolite

While the setup of remote repositories is trivial, there is no inherent access control to restrict groups or teams from accessing or committing to those repositories. To secure and manage LTAC's repositories, the open source software Gitolite was elected. Gitolite uses SSH and scripts to securely manage access to repositories. Combined with CentOS, a fully secured remote repository was deployed.

The remote repository serves as a router for code changes, and a finalization of a patch or bug fix. Code changes should be tested on the developer's local machine before pushing to the remote repository; doing so keeps experimental or non-tested code from entering the permanent project history. Once code is pushed to the remote repository, amending or changing it can cause issues for servers and team members.

Because interactions with Gitolite are done through SSH, an intimate understanding of the technology is necessary. Authentication is handled through the use of public keys, which are a pair of unique files. One file resides on the developer's machine, while the other is transmitted to the Gitolite server. Developers and servers that interact with Gitolite need to generate these keys in order to access remote repositories. LTAC has identified public keys as a much more secure method of access than password-based authentication, so while there may be an initial learning curve with SSH and public keys, the security of this method definitely warrants the time it may take to learn the protocol.

3.4 Git as a Web Deployment Tool

The automation of branch synchronization has made Git a web deployment tool. Using shell scripts called on repository events such as new commits, servers are synchronized with the latest commits automatically, with the exception of the production server. Due to the sensitivity of automatically pushing new code in a production environment, synchronization on the production server is handled manually. The flexibility of Git allows multiple approaches to web deployment.

Gitolite acts on new code and triggers actions accordingly. Using very simple scripts and SSH, servers are updated and relevant participants are notified of changes via email. Other external services that use the source history are updated with the new changes as well. These triggers, known as hooks, can be executed from any copy of the repository.

3.5 Revisions and Lessons Learned

Because Git was developed for Linux, the integration for Windows/Windows Server is not as seamless as it is on Linux/UNIX based systems. A BASH command line emulator is used, creating a more setup steps for automation and some performance loss. Windows Server is being phased out of LTAC's server environment, so these hurdles will be lessened in the future.

The Git ignore file was revised multiple times as unnecessary files were identified. Tracking only the most critical development files and providing a build step or submodule is the key to keeping repository sizes low and deployment times fast. Transactions against the repository, both remote and local, benefit from lower overhead as well. For instance, Git scans each file in the working directory for changes. If the working directory has tens of thousands of files, this operation can be delayed significantly, even with modern technology such as solid state disks.

Initially, a nightly automated commit was in place to take periodic snapshots of LTAC's content management system. After approximately six months of deployment, the development history was inundated with daily commits that did not provide additional redundancy. Furthermore, attempts to consolidate the commits and tidy up the development history created more issues. Once commits are pushed to the remote server, they cannot be modified without consequences for the development team. This practice was discontinued after six months of deployment.

4. ECOSYSTEM ENHANCEMENT

With Git in place, LTAC analyzed the overall process of handling issues that required codebase changes. These issues would often originate from users, who would experience a bug or need functionality added, and email these issues to the Webmaster. The Webmaster would then create a fix, commit it to the codebase, and push it to the live server, followed by an email to the user informing them that the issue had been addressed. As more issues were submitted, the need for tracking software specifically for bugs and issues was realized.

4.1 Issue Tracking Software

Issue tracking software allows developers, managers, and users to create, fix, and monitor software. Much like ticket systems for help desks, issue trackers allow management, accountability, and detailed history of interactions. With issue tracking software, requests are reported to a central location where all interested stakeholders can easily manage them with detailed information about progress. A user of the software can submit an issue and follow its progress. A developer can easily see all of the issues that are assigned to them. Managers can see the workload or time spent by their staff on issues related to the software, along with the ability to change priority of issues.

4.2 Redmine Deployment

After a careful review process, the open source software Redmine was selected for issue management. Redmine is an open source Ruby on Rails-based web application with comprehensive version control integration and extensibility via plugins. Native support for LDAP authentication was another key consideration when selecting Redmine. A turnkey image of Redmine is also available, making deployment into LTAC's virtual environment simple.

After electing Redmine, LTAC was able to offer enterprise-level issue tracking. Because of Redmine's strong integration with version control, issues were now associated with commits, allowing developers to close issues without ever logging in to Redmine. Email notifications are another feature of the software,

allowing all involved parties to stay informed during the life of the ticket.

4.3 The Life of an Issue

With Redmine deployed and configured with internal email services, LTAC had the opportunity to revisit the issue creation process. Redmine has the ability to create issues by email; to take advantage of this feature, the Law website's error logging object was modified to submit errors to Redmine via email and create a new ticket. Before, these emails would be emailed directly to the Webmaster, who would have to create a new issue manually. With the issue tracker in place, the Webmaster receives an email that from Redmine indicating that a new issue was created.

Users can also email issues directly to the issue tracking software to take advantage of the automatic issue creation. Once their issue is created, the developer can address the issue by creating a hotfix on their local machine. These changes are tested thoroughly on the local machine before being committed with a reference to the ticket such as "Fixes #34" to the remote repository. Gitolite, recognizing a new commit, synchronizes its development history with Redmine by issuing a command over SSH. Redmine then reads the repository history and scans for certain keywords such as "fixes" or "references" and associates those commits with their parent issues. In the case of the commit having the "fixes" keyword, the issue is closed automatically and an email is dispatched to the user and developer.

Because the code revision is associated with the issue, Redmine complements the version control system by providing context, and if needed, extensive documentation of the problem. The exact changes, including the precise deltas of each file modified, exist forever with the issue. If the change causes a new bug or issue, the code can be referenced quickly and refactored.

5. MOVING FORWARD

5.1 Project Management with Redmine

In addition to its ability to track issues, Redmine also contains extensive project management tools to assist with the lifecycle of projects. In pre-project development, tasks can be setup with timelines that can be easily visualized in the form of a calendar or Gantt chart. Versions and milestone releases can also be used for products or deployments that adhere to strict lifecycle expectations. Time entry is also available to monitor cost and investment.

Redmine also has collaborative features, like a wiki and file uploads. As a front-facing application, the publication of issues and documentation in the form of a wiki can help users self-serve if they are experiencing an issue that has been documented previously. Redmine also offers granular control over access to its modules; internal tickets and features can require explicit access rights, restricting public access.

5.2 Use of Git on Open Source Projects

Git has proved to be extremely useful in the customization of open source projects. Traditionally, customizations to core elements in open source projects have caused issues with future upgrades. With version control, the effects of customization on the ability to upgrade in the future can be significantly lowered. For example, the blogging platform WordPress can be setup and deployed as an unmodified installation. By committing this untouched and working base to the repository, the plain state of the install is preserved. Customizations can then be made on branches of the working installation. In the event of an upgrade, the project can be temporarily reverted to its original state and the upgrade or patch applied. The customizations can then be merged on top of the upgraded base and tested for issues. Should the customizations create issues with the upgrade, "line by line" merges can be made to quickly identify the offending code.

6. ACKNOWLEDGMENTS

The author sends his thanks to Ed Mananquil, Systems Administrator at Santa Clara Law, for his wisdom and assistance in realizing this project.

7. REFERENCES

[1] APPLE, INC. 2012. Apple - Auto Save and Versions - Every edit and rewrite. Saved. http://www.apple.com/macosx/whats-new/auto-save.html

[2] berliOS. 2012. SCCS - The POSIX standard Source Code Control System. http://sccs.berlios.de/

[3] GNU. 2012. GNU RCS - GNU Project - Free Software Foundation (FSF). http://www.gnu.org/software/rcs/

[4] GNU. 1995. CCVS Revision Log http://cvs.savannah.gnu.org/viewvc/ccvs/NEWS?revision=1.1&root=cvs&view=markup

[5] Collins-Sussman, Ben; Brian W. Fitzpatrick; C. Michael Pilato. 2011. *Version Control with Subversion (for Subversion 1.7)*. http://svnbook.red-bean.com/en/1.7/svn.intro.whatis.html#svn.intro.history

[6] LKML. 1998. LKML: (Larry McVoy): A solution for growing pains. https://lkml.org/lkml/1998/9/30/122

[7] Jeremy Andrews. 2005. Feature: No More Free BitKeeper. http://kerneltrap.org/node/4966

[8] Linus Torvalds. 2005. 'Re: Kernel SCM saga..' – MARC http://marc.info/?l=linux-kernel&m=111288700902396