Paper SAS6501-2016

Preparing for a SAS® 9.4 Deployment and Managing That Deployment Throughout Its Lifecycle

Alec Fernandez and Leigh Fernandez, SAS Institute Inc.

ABSTRACT

SAS® 9.4 allows for extensive customization of configuration settings. These settings are changed as new products are added into a deployment and upgrades to existing products are deployed into the SAS® infrastructure. The ability to track which configuration settings change during the addition of a specific product or the installation of a particular platform maintenance release can be very useful. Often, customers run a SAS deployment step and wonder what exactly changed and in which files. The use of version control systems is becoming increasingly popular for tracking configuration settings. This paper demonstrates how to use Git, a tremendously popular, open-source, version-control system to manage, track, audit, and revert changes to your SAS configuration infrastructure. Using Git, you can quickly list which files were changed by the addition of a product or maintenance package and inspect the differences. You can then revert to the previous settings if that becomes desirable.

INTRODUCTION

Throughout its history, SAS has allowed you tremendous flexibility in managing operational settings. Everything about SAS is configurable. Line size, page size, virtual memory size, paper size, and hundreds of other settings control the functioning of SAS® Foundation. Different languages and locale-specific settings for currencies and formatting of dates require tremendous flexibility for storing and representing information in many encodings. The extremely important and rapidly growing area of data security requires myriad settings to control the encryption algorithms used for transporting and storing data. Distributed applications on the Internet that interface with browsers on distant desktops provide still more areas where flexibility and configuration are required in order to support different authentication methods and security protocols.

The need to allow for all this flexibility requires that specific site and user settings be chosen for any given deployment of SAS. Often, even different individual users of a single deployment want different settings. For example, John prefers black text on a white background, whereas Mary prefers the opposite.

All of these configuration settings and preferences must be stored so that they can be “remembered” and become the current setting the next time that SAS is invoked. The number, location, and formats of these storage locations vary considerably, depending on the application in question.

SAS reads settings from files such as sasv9.cfg or autoexec.sas. And there are wrapper scripts that allow for inclusion of site-specific environment variables (for example, workspaceserver_usermods.sh). But the SAS suite of software products includes applications other than SAS, such as web servers, database servers, web application servers, and the like. Each of these third-party applications has its own set of configuration files. Then there are configuration files for controlling the operating system, PAM authentication, LDAP, web-based authentication, and so on. Many of these configuration files can be organized into hierarchies to allow for one group’s individual preferences to override the company-wide default.

All of this adds the necessary flexibility, but also adds complexity.

Furthermore, the need to introduce new settings or to change the values of existing settings becomes necessary when new versions of software products are released. For example, the maximum allowed number of simultaneous connections needs to increase as new features are added in the latest release of SAS. Memory settings need adjustment as new optimizations are implemented.

All of this flexibility combines to require the need for managing a large number of configuration files in a large number of directories both in the SASHome directory and in the configuration—also known as “Lev1”—directory.

Since many of these configuration files require updates during the process of installing a new maintenance version of SAS, a systematic approach for managing the different versions of these files is quite useful.

Git is an immensely popular, revolutionary, new, distributed version-control system. It allows for an easy way to track changes to files over time. It is free and open source. It is available from the Git website (https://git-scm.com/) and has extensive documentation and video training materials.

In this paper I offer insight into the ways in which changes to the collection of SAS configuration files and the configuration files associated with required third-party ancillary products can be versioned, tracked, and managed using Git.

TYPICAL USE CASE

1. When you first deploy SAS, it is delivered in an initial, default state. For some products, this default state is sufficient for meaningful work to begin right away. For more complicated deployments, additional setup work is required before you can even begin to use the products. It is these more complicated deployments that would most benefit from using Git. The following steps illustrate a typical deployment workflow: Deploy SAS® Enterprise BI Server using SAS® Deployment Wizard. The SASHome directory and the Lev1 directory contain many configuration files with the default settings.
2. Change the default LINESIZE= and PAGESIZE= options in your sasv9.cfg file to better suit your business preferences.
3. Add many LIBNAME statements and other statements to various autoexec.sas files.
4. Set up security (for example,, Integrated Windows Authentication, client certificates, FIPS, firewalls, SiteMinder, WebSEAL, reverse proxy servers, and so on). This requires changes to many XML and conf files.
5. Add customizations to other config files in support of specific individuals or departments. For example, you might need to lengthen application time-out values because the accounting department must download currency conversion tables, which takes too long, so the application logs you out.

GETTING STARTED USING GIT

Unless you have had prior experience using Git, now would be a good time to familiarize yourself with the basics. Because Git has become so popular, there are a tremendous number of training resources available online, free of charge. You can start by navigating to https://git-scm.com/documentation.

Before you start reading, it's safe to make a few simplifying assumptions. This will help prevent getting overwhelmed by all the powerful features available in Git that won't apply to this particular use case. Remember that Git was designed to allow a large number of developers, spread around the globe, to jointly collaborate on programming projects. For this reason, Git is based around the idea of synchronizing to remote repositories on the Internet.

Since configuration changes are only relevant to the machine on which the configuration exists, and because we don't need to share configuration settings with other team members on other machines around the world, you have no need for remote repositories (at least not for any of the use cases that are being presented in this paper).

Therefore, you do not need to bother with concepts such as Git on the Server or GitHub. Everything that you will be doing is local to the file system in which you deployed SAS. In addition, the concept of cloning a repository will not be particularly useful at first and can also be safely ignored. Trust me, there will be plenty of time to go back and deepen your knowledge of Git once you have mastered the basics.

STEP BY STEP: USING GIT TO TRACK CONFIG CHANGES

Step 1. Install Git

For Windows, Linux, and Solaris based SAS deployments, installation packages are available at https://git-scm.com/downloads.

Simply download the appropriate package and follow the instructions. You need to make sure the version of Git you are using is 1.8.3.1 or higher because required enhancements to the .gitignore feature are in this version.

For other operating systems it will take a little bit of web searching, but due to its immense popularity, you will find abundant resources for installing Git on any directory-based operating system.

Step 2. Initializing the Config Directory (LevX) with Git

One of the appealing features of Git is that it is quite lightweight and unobtrusive. You can begin tracking changes to all the files in a directory with one simple command: `git init`. When you issue this command, Git creates a subdirectory in the current directory named .git. Everything that Git needs to track these changes will be stored in this .git subdirectory. If after tracking changes with Git you decide to abandon the effort, simply delete the .git directory and you are no worse for wear. Git will not alter files outside the .git directory. Note however, that all Git versioning data will be lost if you delete the .git directory, so do not delete it if you intend to ever return to using Git in the future. Other than using some disk space, this directory will not interfere with anything.

Screen Display 1. Initializing a Directory for Use with Git

```
ettlax47> pwd
/install/SASServer/config/Lev1
ettlax47> git init
Initialized empty Git repository in /data/install/SASServer/config/Lev1/.git/
```

Screen Display 1. Initializing a Directory for Use with Git

Once you execute the `git init` command, that's it! You've done it. You are now ready to start tracking changes. You must admit that it's pretty simple so far.

Step 3. Excluding Unnecessary Files Using .gitignore

You are now ready to begin tracking changes in files. The question becomes: Which files do I need to track?

By default, Git tracks changes to every single file in the directory. In our use case, a SAS LevX directory, this is unnecessary and would waste a great deal of disk space. Consider that there are many files that you do not need to bother tracking. In general, you only need to track files that will contain configuration modifications that cannot be re-created in an automated fashion. This leaves out vast categories of files.

The following types of files should be excluded from version tracking:

- .log files (with a few notable exceptions)
- code that can be reinstalled or re-created by using SAS tools such as SAS® Deployment Manager
    - .so, .exe, and .dll files that do not change after initial installation
    - .jar files (these are inside of the .war or .ear files, which can easily be re-created using build webapps)
    - Configuration management utilities, which are not customizable
- Examples
- Documentation
- Temp files

You tell Git to intentionally ignore files by placing an entry in a special file named .gitignore. This file uses

specific filenames and generic patterns to exclude files from tracking. Documentation can be found here: https://git-scm.com/docs/gitignore

Suggested syntax for a typical LevX/.gitignore file. You should copy and paste these lines into the LevX/.gitignore file prior to executing the `git add` command below.

```
# Exclude general directories that contain files that do not require versioning
**/Backups/
**/data/
**/demo/
**/customer/
**/docs/
**/Documents/
**/example/
**/examples/
**/htdocs/
**/javadoc/
**/lib/
**/libs/
**/log/
**/logs/
#**/Logs/
!**/Logs/Configure/
**/*.log
**/Repository/
**/Repositories/
**/Scripts/
**/temp/
**/Temp/
**/tmp/
**/Utilities/script_templates/
**/work/

#Exclude Development Tester files
**/DeploymentTesterServer/
**/dtest/

#Exclude Metadata Server files
**/MetadataRepositories/
**/MetadataServer/Journal/
**/SASMeta/MetadataServer/rposmgr/

#Exclude SAS Environment Manager files
**/activemq-data/
**/bundles/
**/hq-engine/
**/wrapper/

#Exclude gemfire and activemq files
**/admin/
**/gemfire/instances/
**/gemfire/templates/
**/dtd/

#Exclude web files
**/cacheroot/
**/ROOT/
**/Staging/
**/*.war/
**/Web/Utilities/
**/Web/WebAppServer/SASServer1_1/conf/Catalina/localhost/

#Exclude files by type
*.bin
*.class
*.dll
*.gif
*.html
*.jar
*.jpg
*.so
```

Step 4. Adding Files That You Want to Track to the Git Index

Once you are satisfied that you have excluded all the files and directories with content that need not be tracked and versioned, you are ready to add the initial version of the desirable files to the Git index. The index is a list of files that Git keeps so that it knows which files you are changing at any given point in time. It's is a good concept to understand fully, so I recommend that you read this section of the Git documentation before proceeding: https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository.

Once you understand the concepts of a Git working directory and index, you know that before proceeding you need to load the index with the initial version of all config files. To do this, you use the `git add .` command. This command is sensitive to the directory from which it is entered, so you should ensure that you are in the LevX directory. Since there are quite a few files in a typical Lev directory, this command will take a while to complete. I recommend that you add the `–dry-run` argument to your Git add command to make sure that your .gitignore file is working as you intended. If not, edit the LevX/.gitignore file and repeat the command. Once the list of files looks right, remove the `–dry-run` argument from the command line and Git will do the work.

Once the command completes, you can see what files have been added to the index by using the command `git status`. `git status` lists the files that were added to the index by the `git add` command. Again, you want to keep the list of files you are tracking limited, so look closely at this list. If you see files that are not used to track configuration settings, you should repeat the process as follows:

1. Remove all files from the Git index by executing the `git rm –r --cached *` command.
2. Add or adjust entries in the Lev*X*/.gitignore file.
3. Re-run the "`git add .`" command

Fear not, the `git rm –r –cached *` command does **not** delete files! It simply removes them from the Git index. The files themselves will be left undisturbed. Also, like most Git commands, it is sensitive to the directory from which it is issued. If you want the "*" to match all files in the Lev directory, it must be executed from the Lev directory. If you only want to remove files from a specific subdirectory, issue the `git rm –r –cached *` command from that directory.

Screen Display 2. Using git add and `git` status to `Manage` Files in the Git Index.

```
ettlax47> pwd
/install/SASServer/config/Lev1
ettlax47> git add .
ettlax47> git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   .gitignore
#       new file:   AppData/SASContentServer/Repository/SCSOK
#       new file:   AppData/SASContentServer/Repository/repository.xml
#       new file:   AppData/SASContentServer/Repository/repository/index/_0/_0.cfs
#       new file:   AppData/SASContentServer/Repository/repository/index/_0/cache.inSegmentParents
#       new file:   AppData/SASContentServer/Repository/repository/index/_0/segments.gen
#       new file:   AppData/SASContentServer/Repository/repository/index/_0/segments_1
#       new file:   AppData/SASContentServer/Repository/repository/index/_0/segments_2
#       new file:   AppData/SASContentServer/Repository/repository/index/_1/_0.cfs
```

Screen Display 2. Using git add and git status to Manage Files in the Git Index

At this point, you are probably being tempted to put all files in the Lev dir into the Git repository. We all are. This is not a wise idea and you should resist the urge to do so. Tracking all files in a Lev directory is not the intended purpose of Git. You do not want to attempt to use Git as a backup and recovery system. Other tools are better suited to this task. You want to add entries to the .gitignore file until only those files

that have configuration settings are left in the Git index. Later we will discuss combining and coordinating backups and Git commits to roll back updates. Both will need to be used in unison.

It might take a few iterations before you are comfortable with the contents of your .gitignore file. The good news is that you only have to go through this process once and the example I include above might be all you need.

Step 5. Committing the Initial Contents of Files to Recorded History

Once you have successfully whittled down the contents of the Git index to your liking, you are ready to store the current version of all files in the index for all of eternity. To do this you use this command:

```
git commit –m "Some detailed message here"
```

Complete documentation for the commit command can be found here: https://git-scm.com/docs/git-commit but in summary, the `commit` command stores the current contents of every file in the index. Think of this of making a complete copy of every file in the index into some safe "backup" directory, as a "snapshot" of the current state of things. Once the commit is done, for the rest of eternity you will be able to manage the contents of every file that has been committed in relation to future versions of content for the same file. It is your safety net. You will never lose another change again and if you are conscientious about committing often, you will always be able to safely return to the version you had working "5 minutes ago!"

The commit message is very important, it should clearly denote why you decided to take the snapshot at this particular point in time. When crafting messages, you should think back to when you were looking at old sepia pictures with elderly relatives. Often you turn the picture over hoping for some sort of inscription. Perhaps you were hoping that it would say "This is Theodore Smith, the son of Elijah Smith, on his wedding day. The bride is Doris Hester and her cousin Louise Williams is the flower girl". If every old snapshot were to have this sort of information, our ability to re-create history would be made much easier.

Similarly, Git commits should include similarly detailed messages such as the following examples:

- Initial SAS 9.4M0 deployment with unmodified factory default settings
- Configuration updated per Paul Smith to increase session time-out
- Hotfix HF09734 added to increase query performance
- Apache web server working but is annoying me with too much logging
- Apache web server is almost working but I'm not able to get outside the corporate firewall

Keep in mind that it is extremely unlikely that you will ever kick yourself for excessively long commit messages, or for committing too often, whereas the opposite is often true. Nobody has used Git for long before saying "Dang, if only I could get back to the version of the file I had 3 minutes ago." If you commit early and often, you will not have this regret.

Screen Display 3 Very First Commit That Records Initial State of All Files.

```
ettlax47> pwd
/install/SASServer/config/Lev1
ettlax47> git commit -m "Initial_OOTB_SSL_Web_Server_only"
[master (root-commit) b7899ec] Initial_OOTB_SSL_Web_Server_only
 Committer: QUEST Installation ID <qstinst@ettlax47.unx.sas.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

If the identity used for this commit is wrong, you can fix it with:

    git commit --amend --author='Your Name <you@example.com>'

 2968 files changed, 70821 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
 create mode 100644 AppData/SASContentServer/Repository/SCSOK
 create mode 100644 AppData/SASContentServer/Repository/repository.xml
 create mode 100644 AppData/SASContentServer/Repository/repository/index/_0/_0.cfs
 create mode 100644 AppData/SASContentServer/Repository/repository/index/_0/cache.inSegmentParents
 create mode 100644 AppData/SASContentServer/Repository/repository/index/_0/segments.gen
 create mode 100644 AppData/SASContentServer/Repository/repository/index/_0/segments_1
 create mode 100644 AppData/SASContentServer/Repository/repository/index/_0/segments_2
 create mode 100644 AppData/SASContentServer/Repository/repository/index/_1/_0.cfs
```

Screen Display 3 Very First Commit That Records Initial State of All Files

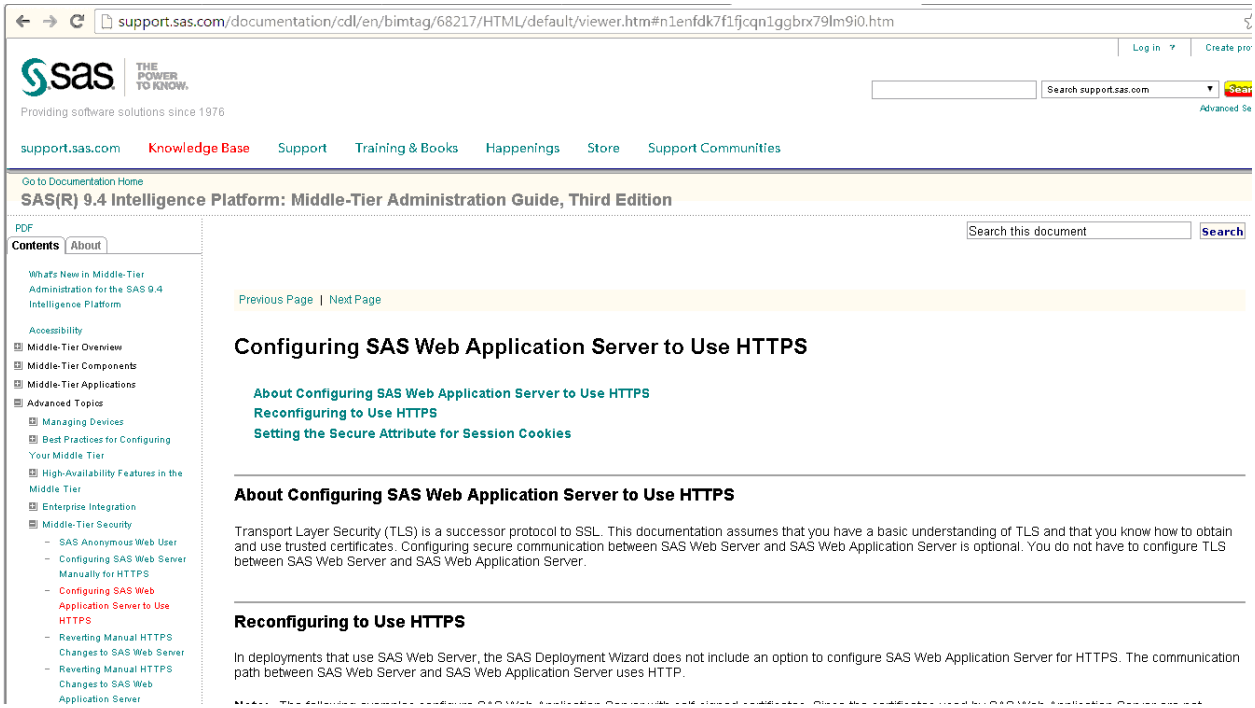
That's it. Mission accomplished. The current state of all files in the index is perfectly safe. If you were to enter the `git status` command now, it would return `nothing to commit, working directory clean`. This means you are now free to fearlessly modify, experiment, corrupt, delete, trash files at will without fear of recrimination. Anything you do can be easily inspected, refined, compared, and undone as you will now see.

Step 6. Making Changes to Config Files without Fear or Remorse

At this point in the process you have SAS products running in an initial, default state. If this were sufficient for all use cases, the job of a SAS administrators would be very boring indeed. Luckily, as you know, nobody is ever happy with default values. This provides countless hours of fun-filled entertainment for SAS administrators and makes us the envy of software types around the world.

Once you finish your install, the process of customizing begins. The example selected for this paper is the addition of a non-default security setup. This process clearly is documented in the *SAS® 9.4 Intelligence Platform: Middle-Tier Administration Guide*. However, as is the case with most issues surrounding security, it is a bit involved. It requires many steps requiring many edits to several configuration files in different directories. If you make any syntactical mistakes while editing these files, it can result in the system becoming non-functional.

Screen Display 4. Documentation for Adding HTTPS to App Servers.



Screen Display 4. Documentation for Adding HTTPS to App Servers

Once you get through the document and complete the listed steps, it would probably be useful to see which files have been changed from their original default versions. Now is when Git becomes quite handy. As you probably anticipated, the command to use is `git status`. It shows which files have been modified since the most recent commit. You probably recall that this commit was our initial one with the default version of the files.

Screen Display 5. git status Showing Files That Have Been Modified

```
ettlax47> pwd
/install/SASServer/config/Lev1
ettlax47> git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   Web/WebAppServer/SASServer1_1/bin/setenv.sh
#       modified:   Web/WebAppServer/SASServer1_1/conf/server.xml
#       modified:   Web/WebAppServer/SASServer1_1/conf/web.xml
#       modified:   Web/WebServer/conf/sas.conf
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Web/WebAppServer/SASServer1_1/bin/setenv.sh.ORIG
#       Web/WebAppServer/SASServer1_1/conf/server.xml.ORIG
#       Web/WebAppServer/SASServer1_1/conf/web.xml.ORIG
#       Web/WebServer/conf/sas.conf.ORIG
no changes added to commit (use "git add" and/or "git commit -a")
```

Screen Display 5. git status Showing Files That Have Been Modified

In the output above, Git is showing which files have been modified. You can rest assured that no other files in the Lev*X* directory have been touched. This is the complete set other than the ones you specifically told Git to ignore by including them in the .gitignore file.

In addition, you can see that there are untracked files. This means that these files were added to the Lev*X* since we last used `git add .` to add files to the index. In this case, as experienced admins are prone to do, a "safe" copy of the original version of the config files was made before beginning the modifications (a simple `cp web.xml web.xml.ORIG`). That way, if something goes wrong, you can put the .ORIG versions back in place and get things working. Since you are now using Git, you no longer need to do this, your original versions were saved earlier when you did the `git commit`.

At this point you have several options:

- If you want to see the modifications you made to each file line by line use `git diff`
- If you want to undo your modification and revert to the prior version of the files, you have two choices. Either of these two choices will put the system back to the same, functional state that it was in prior to any changes being made to the files.
    - If you want to give up on your modifications without saving them, use `git reset --hard`. This overwrites the changes in the working directory with the most recently committed versions the files. It's a simple "undo".
    - If you want to undo the changes but save the changed versions off to the side for future work you can use the `git stash` command. See the document here: https://git-scm.com/docs/git-stash.
- If you are happy with the changes and want to leave the new versions of the files in place and also keep the new versions for all of eternity, repeat the same `git add .` and `git commit -m 'xxx'` workflow that you used earlier. A good message to use with the commit might be "After conversion to two-way ssl is complete for web app server".

Screen Display 6. git `add and` commit `after Manual` SSL .

```
ettlax47> pwd
/install/SASServer/config/Lev1
ettlax47> git add .
ettlax47> git commit -m "After_manual_SSL_changes_webappserver"
[master 679c917] After_manual_SSL_changes_webappserver
 Committer: QUEST Installation ID <qstinst@ettlax47.unx.sas.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

If the identity used for this commit is wrong, you can fix it with:

    git commit --amend --author='Your Name <you@example.com>'

 8 files changed, 4014 insertions(+), 18 deletions(-)
 mode change 100755 => 100644 Web/WebAppServer/SASServer1_1/bin/setenv.sh
 create mode 100755 Web/WebAppServer/SASServer1_1/bin/setenv.sh.ORIG
 create mode 100644 Web/WebAppServer/SASServer1_1/conf/server.xml.ORIG
 create mode 100644 Web/WebAppServer/SASServer1_1/conf/web.xml.ORIG
 create mode 100644 Web/WebServer/conf/sas.conf.ORIG
```

Screen Display 6. git add and commit after Manual SSL Changes

At this point you now have a couple of commits under your belt, so it's time to consider a few more Git commands:

- `git log` this command lists all the commits you have creates to date. Commits are references by a long list of letters and numbers called a SHA
- `git checkout *` this command is used to retrieve files from earlier commits. An * char means retrieve all file from that previous commit.

Step 7. Reverting Manual Changes in Preparation for Applying Maintenance

If you modify a deployment to add SSL connections between the web server and the web application server, this modification must be reverted prior to applying maintenance. If you used Git to version your configuration modifications, this step is made much easier. Simply use the `git log` command to list out all the commits. Use the commit messages to locate the commit object that contains the original versions of the files that were modified when you added the SSL connections. Use the `git checkout <filename>` command to replace the modified versions of the files that are currently in the Lev*X* directory with the original versions that Git stored in the commit object. If you want to revert every file in the commit, use * for the filename argument. Once you have checked out the files, you repeat the `git add .` followed by `git commit -m` workflow to save the state of the files after successfully reverting the modifications. Once you have successfully put the old, original versions of the files back in the working set, you can apply SAS maintenance to your deployment.

Once this is done you can use the `git status` command to observe which files were changed by applying maintenance and which new files got added. Again it's important to note that if a file got deleted and a new one was installed in the same location, Git will show that this is a new file rather than a modification.

SPECIAL CONSIDERATION FOR METADATA SERVER AND DATABASE SERVER FILES.

It is very important to note that Git controls changes made to files by recording the differences in two versions of those files on disk. Changes made to metadata or database servers cann be controlled by Git. These servers cannot be controlled by Git because making copies of the files into which the metadata or database data are persisted is not sufficient. In order to make a database or metadata backup, special procedures must be followed and the servers must be quiesced or shut down so that the files are in a stable state for the backup to be successful. Databases and metadata servers have transactions and caches that they keep in memory. The state of the file on disk is not fully synchronized with the state of the database until you tell the database or metadata server to do that in preparation for a backup.

For this reason, if you are managing a change that involves both configuration files and metadata or database content, you must make backups of the metadata and database servers and be sure that these backups are synchronized with the versions of the configuration files that you commit to the Git repository.

The process of updating to a new maintenance release of SAS is a good example. It involves changes to configuration files, updates to metadata and updates to database content. In this case you should use the SAS Backup Utility to make a system backup after the update is run on each machine. At the same time you should commit the version of the configuration files in the Lev*X directory.*

USING GIT TO MANAGE PRODUCT ADDITIONS

You can use the steps outlined above when managing the addition of new products to an existing SAS deployment. As you probably know, the addition of new products requires that configuration files, metadata, and database entries be expanded with information related to the new products. As noted above, the metadata and database files will need to be coordinated and synchronized with the configuration files, but if you do this, it is possible to revert to the version of a deployment prior to the addition of the new product.

CONCLUSION

Because of the flexibility that SAS affords customers in customizing deployments, there are a large number of configuration files required to store these customized settings. Managing these customizations is made much simpler and safer by using version control systems to track changes. Git is a state-of-art version-control system that allows free, lightweight, fast, and simple version control of all files in a SAS deployment.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Alec Fernandez
SAS Institute Inc
Alec.Fernandez@sas.com

Leigh Fernandez
SAS Institute Inc
Leigh.Fernandez@sas.com