

Paper SAS4197-2020

Git for the SAS® Programmer: Using Source Control to Organize Your Code and Collaborate with Others

Amy Peters, Danny Zimmerman, Grace Whiteis, Joe Flynn, and Stan Polanski, SAS Institute Inc.

ABSTRACT

Are you tired of using elaborate comments in your code or saving multiple copies of your files to manage changes as you make them? Wish you could go back in time to that version of your program that worked? Do you live in terror of clobbering someone else's work? Version control can help, and the front runner in the version control world is Git. Git is a free and open source distributed version control system that you can use on your own or in collaboration with others. It can also be used with a central, shared repository such as GitHub or Bitbucket. Learn Git concepts such as clone, commit, and merge, and how to execute them using the Git interfaces in SAS® Studio and SAS® Enterprise Guide® or in code using SAS® functions.

INTRODUCTION

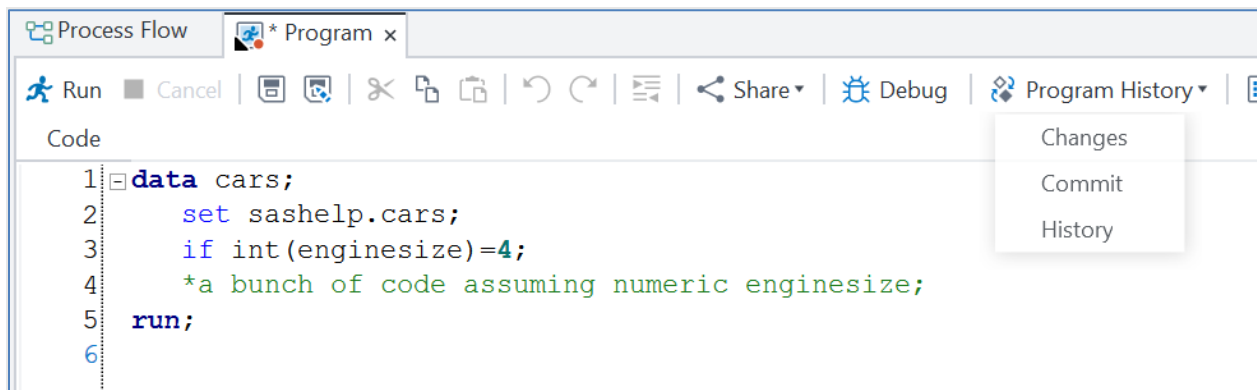
Git allows groups of developers to collaborate on the same files without overwriting each other's work. Git also tracks the history of any file changes, including what has been changed, when the change occurred, and by whom it was made. Known as "version control", this system allows you to review a project's history and revert to a specific version later. Developers can use separate branches to segregate work and control when their content is merged into the common copy. SAS programmers have many options for using Git, including SAS Enterprise Guide features that use Git hidden in the background, SAS functions for executing Git commands, and full-featured Git interfaces in SAS Studio and SAS Enterprise Guide.

SINGLE SAS ENTERPRISE GUIDE CODER – PROGRAM HISTORY

If it's just you, you use SAS Enterprise Guide, and all you need is help managing the versions of your own code, then the simplest solution is to use the program history feature in SAS Enterprise Guide. While this uses Git technology behind the scenes, the complexities of Git are largely hidden, and there's no extra setup needed to use it like there is for other available Git integrations. Program history was introduced with SAS Enterprise Guide 7.1.

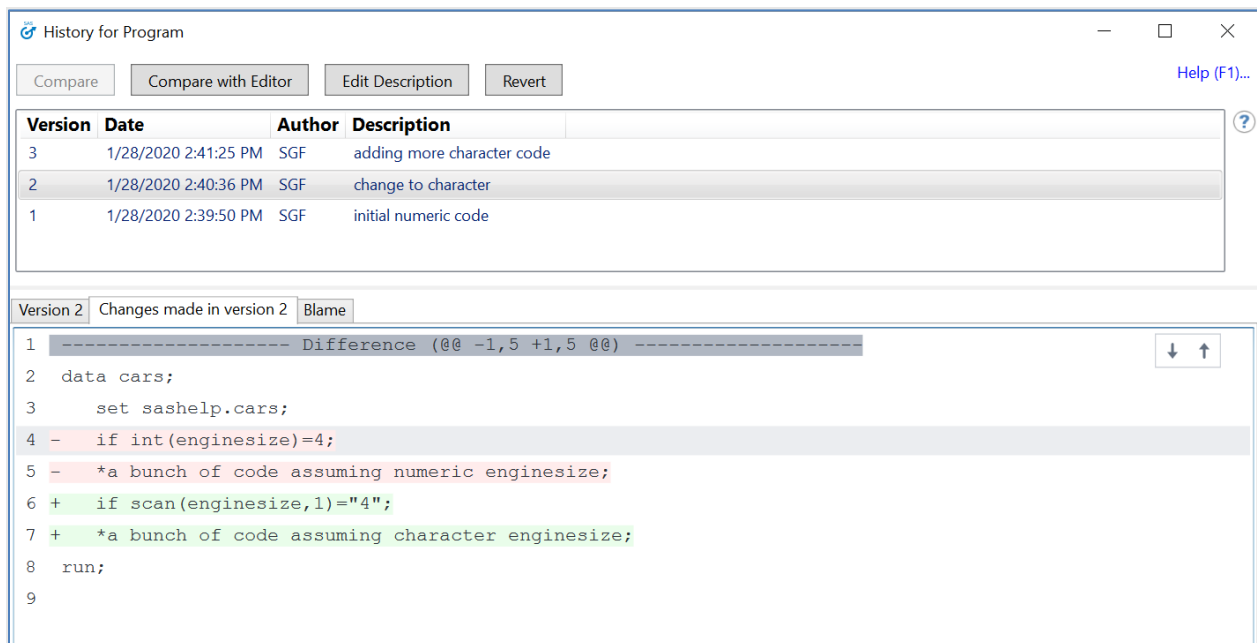
SETTING UP AND USING PROGRAM HISTORY

Make sure program history is enabled by selecting **Tools > Options > Version Control** and clicking **Enable program history for embedded files**. Start a new project, open a new code node, and notice there is now a Program History button on the toolbar. For this example, you are manipulating a dataset in the code below using the assumption that one of the variables is numeric. Since you are not completely sure this is the correct path to go down, commit your current version, so that you have a snapshot of the code that you can go back to later if needed.



Display 1. Committing Code You Might Need Later

You decide the way forward is to treat the variable as character, so you make a bunch of changes under that assumption and commit them. Then you make more changes. But then you realize you have completely messed up, and the original numeric approach was the better one. Select **Program History** > **History** from the toolbar to see all the commits you have made, how they differ from each other, and to revert to a version you were happy with.



Display 2. Viewing Changes and Going Back in Time with Revert

There are more things to explore, such as using a more sophisticated diff tool to see side-by-side comparisons, for example. See "Working with Programs" > "Understanding Program History" in *SAS Enterprise Guide: User's Guide*.

LIMITATIONS

Program history can only be used for programs that are embedded in a SAS Enterprise Guide project. It is not available for stand-alone SAS files. The program history is stored as part of the project, which means you need to have a robust backup process in place. Do not think of program history as a backup; if something happens to the project, the program history will be gone. Program history is also not ideal for multiple users. If you plan to collaborate with others, then using the full Git interface is likely a better choice.

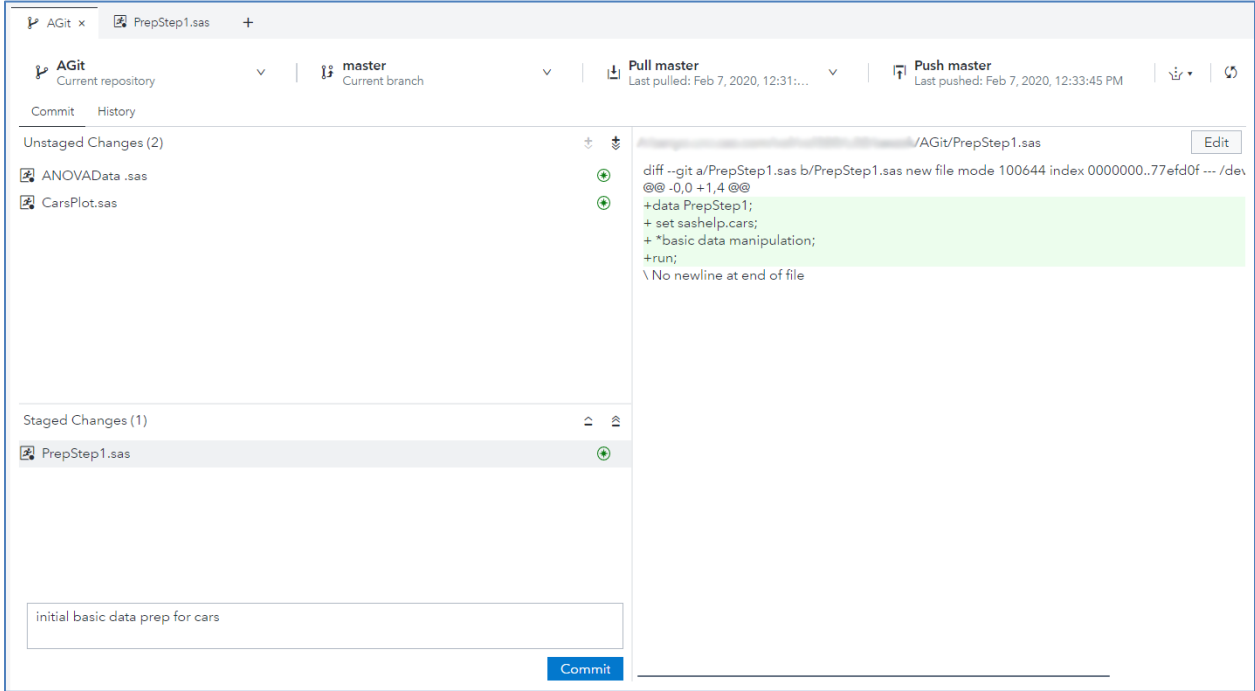
USING THE GIT INTERFACE IN SAS ENTERPRISE GUIDE OR SAS STUDIO TO WORK WITH A REMOTE REPOSITORY

If the program history feature in SAS Enterprise Guide doesn't fit your needs, you can move up to the full Git interface in SAS Enterprise Guide 8.2, SAS Studio 3.8, or SAS Studio 5.2. The interfaces are similar and offer a lot more control over the code management process, but require that you set up a connection to a remote repository. There are also some Git terminology and processes you'll need to learn to be effective. Git terms will be shown below in ***bold/italic*** font style.

First let's work through a scenario where everything is already set up and you are working in a team that is sharing a remote repository. SAS Studio 5.2 is shown, but the process is the same for SAS Studio 3.8 or SAS Enterprise Guide 8.2. Later, we will talk through any differences as well as work through the setup process.

Unlike the previous example, you are not working alone. You are on a project with others. Instead of only working locally, you are working with a ***remote repository*** where you and your co-workers are accessing (and potentially updating) the same files. You have ***cloned*** the remote repository, which means you've taken a copy of all the remote files and their history and put them in the file system you see in SAS Studio (on the SAS server) or SAS Enterprise Guide (in a folder on your local machine). This copy is your ***local repository***, which is just a fancy name for a folder that you can see in SAS Studio or SAS Enterprise Guide and that Git knows about and is tracking. Now as you make and save changes to files in that folder, the changes appear in the Git interface as ***unstaged*** changes.

In this example, you got the project files from the remote repository when you cloned it. You then created a new file PREPSTEP1.SAS. You also added two new files: ANOVADATA.SAS and CARSPLOT.SAS. You're ready to save this version of PREPSTEP1.SAS, so you've moved it from unstaged to staged. The other two files aren't ready for saving, so they remain in the unstaged area.

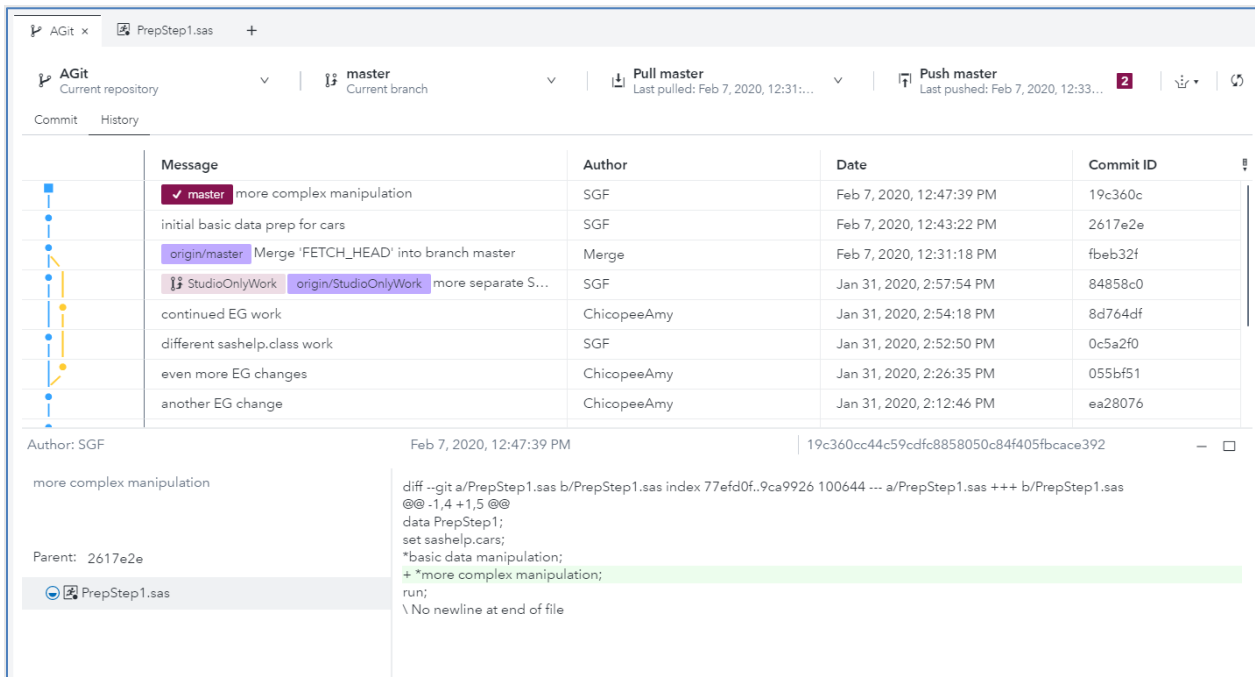


Display 3. Git Interface in SAS Studio – Viewing Staged and Unstaged Changes

A way to think of this scenario is that Git is automatically tracking things as you change them in the local repository (the folder you told Git to track). Your changes are shown as unstaged files. You are in control of what constitutes a saved point, which is known as a **commit**. To put together a commit, you **stage** one or more files and then describe the changes in a comment before pushing the commit button. You must commit to be able to go back in time to a previous commit, which is known as a **reset**. To say it a different way, we have been talking about four levels of operating on files:

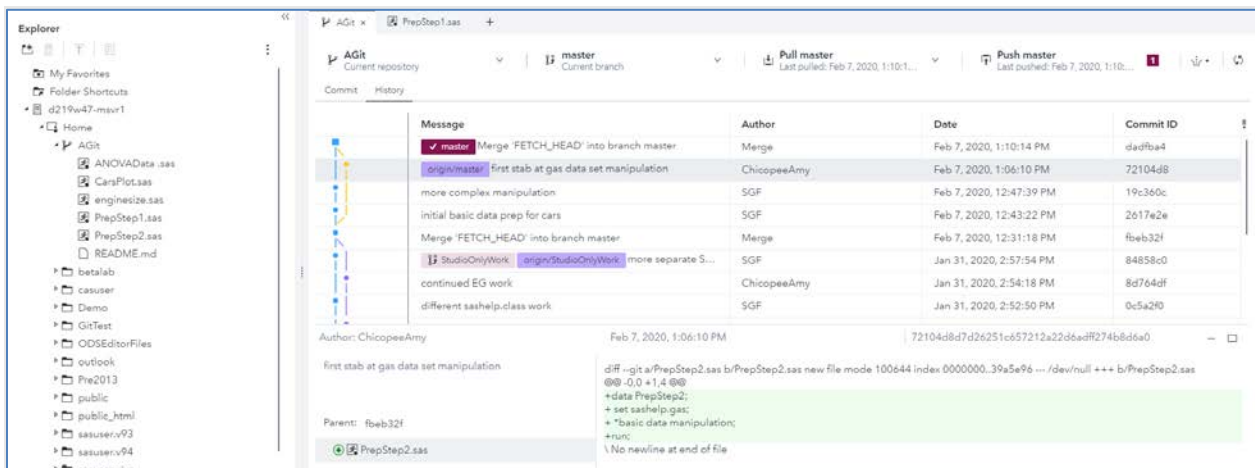
1. Editing an unsaved file – Undo/redo in the editor or using the submission history in the editor are the ways to go back to a prior version, and those are only available within your session. If you close an edited file without saving, everything is gone. You are not using Git until you save a file.
2. Unstaged files – Every time you save a file with changes, Git shows that file in the Unstaged Changes area of the Git Repositories pane, and you can see differences by clicking on the file in the unstaged area. If you want to go back to a past version, you can copy and paste from the differences report, or, if you are in the same session, you can use the editor options in #1 above. If you close without saving, you retain the last saved copy.
3. Staged files – There is really no difference between this and #2 in terms of recovery. All staging does is serve as a waiting area for commits. Most people use staging as a holding area for related changed and saved files, allowing them to work on other files before committing a chunk of work. Some people stage and commit immediately. It's a matter of how you like to organize your work.
4. Committed files – You have decided you are at a level of quality in your code that you want to keep, so you are asking Git to track these changes. Once you have committed, the file changes in that commit become part of the Git **history**. Note that in SAS Studio's file navigation pane, you cannot tell a difference between these four levels of files – it just looks like a file. It's in the Git Repositories pane where the distinction is made.

Going back to our example, you have made more changes to PREPSTEP1.SAS, so you have saved it, staged it, and committed it. You can use the History tab to see what you've done in the past and what is different between this version and the last committed version.



Display 4. Using the History Tab to See Progress and Differences in the Code

You have been using staging and committing to organize your iterations on this file for yourself. Now you are comfortable with the quality and want to make it available to others involved with the project. You do this by invoking a **push** to the remote repository. But before you push, it's a good practice to first do a **pull**. Pulling ensures that you have the most current copy of what's in the remote repository. After doing a pull, the History tab shows that indeed someone else had pushed since your last copy, and a new file is available, PREPSTEP2.SAS.

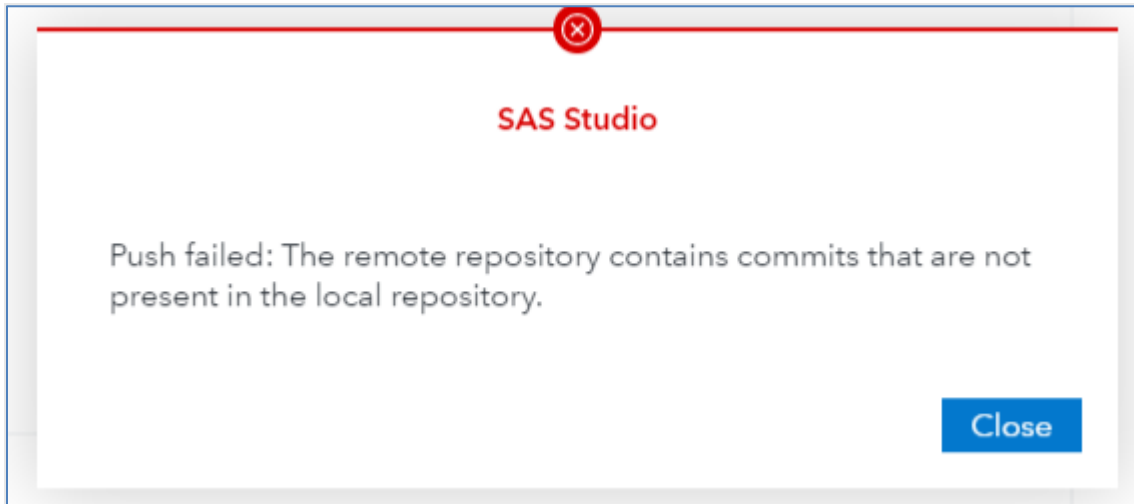


Display 5. Seeing New Content After a Pull

The new file does not conflict with the work you want to push, so you can confidently push your changes.

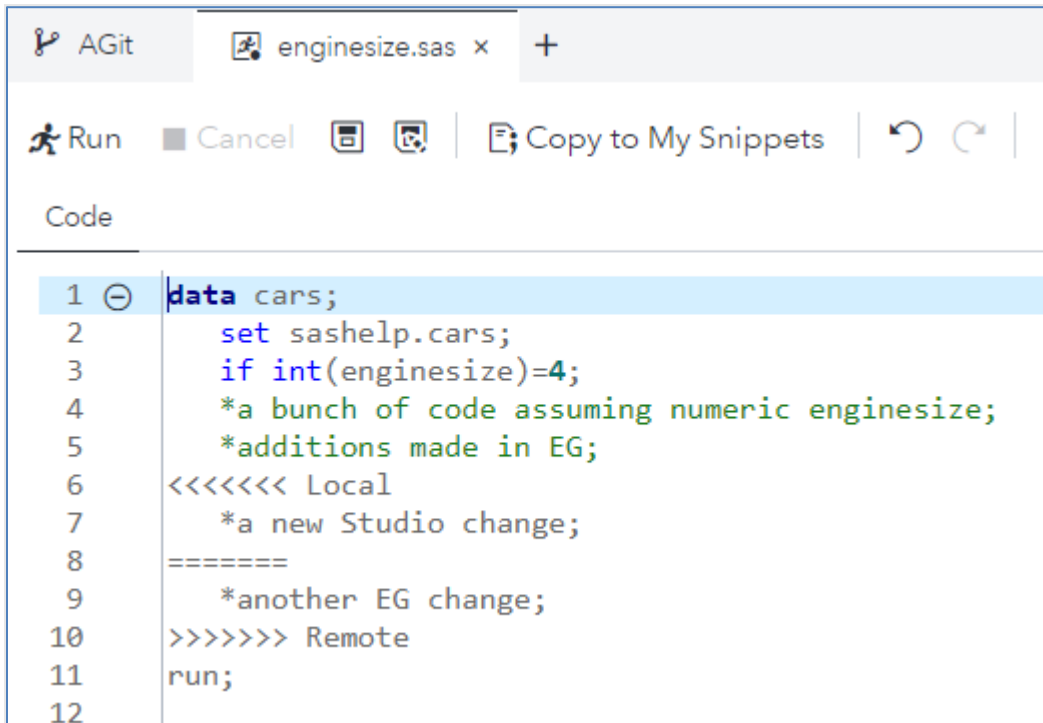
DEALING WITH CONFLICTS

This all works great when you are not working on the same file at the same time. If, however, your co-worker makes a change and pushes it, and you make and try to push a change without having first pulled the co-worker's changes, you will get an error.



Display 6. Conflicts in the Remote Repository

Information will be offered to help handle conflicts when possible. For this example, you do a pull to try to get the most recent copy of a file multiple people are working on and get an error that mentions the conflicts. Opening ENGINESIZE.SAS after the pull attempt, you see documentation of the conflict. The local section contains the change from your local commit. The remote section contains the conflicting line(s) from the commit(s) that were just pulled down. It's up to you to update the code, taking into account both sets of changes. When saving the program, you will commit this updated version of the program as part of the merge commit.



Display 7. Documentation on Conflicts

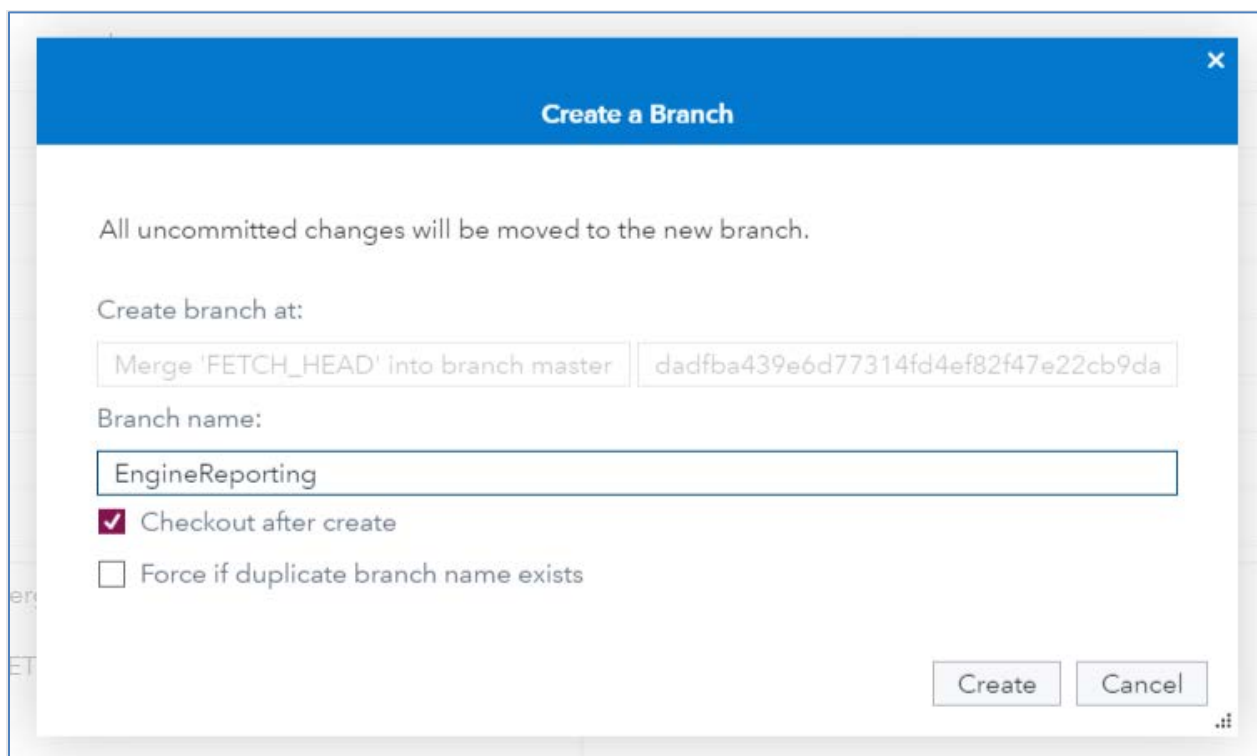
CREATING AND MERGING BRANCHES

One of the big advantages of using Git is its ability to manage changes to a repository across multiple people. This can be as simple as sharing a remote repository, as we have

already seen, or something more complex, such as creating branches to segregate work and then merging the new work back into the master branch. Branching can be used to work on a new feature or experiment while protecting the “production” version. You can also use branches to freeze in time a set of changes, as in the case of a patch or hotfix that will be merged later.

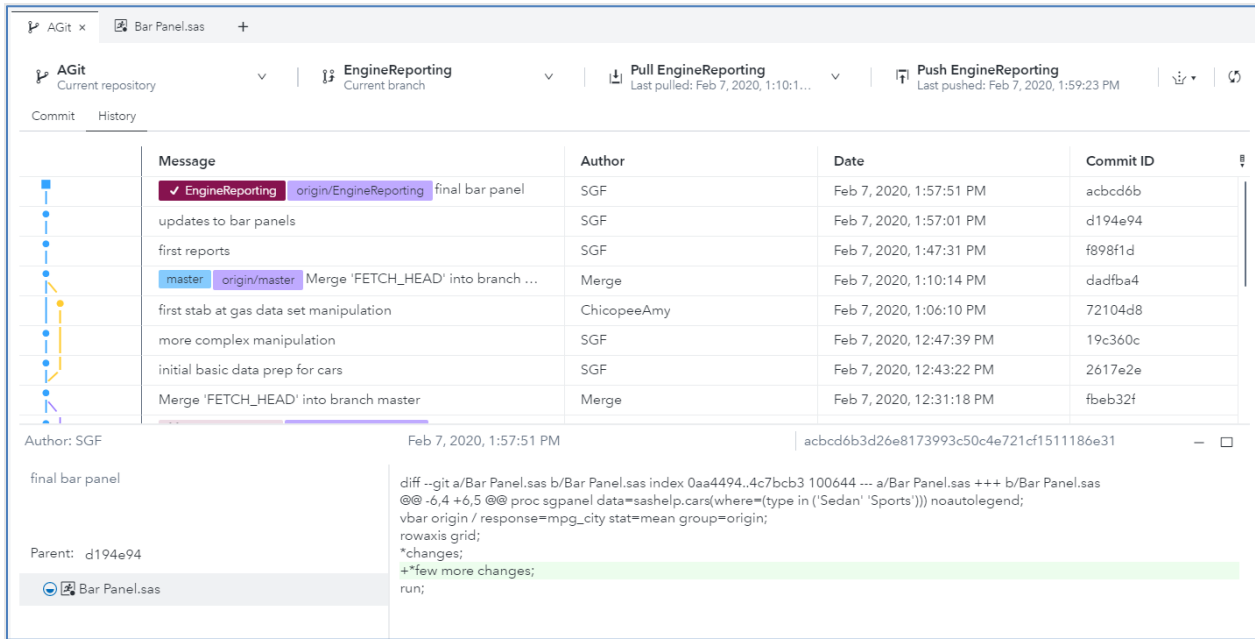
Master is the default branch in which you have been working so far. You can create a different branch to isolate your work by selecting any commit row (i.e., a point in time) from the History grid. When Git creates your new branch, the files that appear in your local repository when you **check out** that branch come from the point in time where you elected to create the branch.

In our example, while your co-worker continues to make changes in master to the data prep steps, you can create your own branch to work on some reporting by right-clicking on the master branch in the History grid and selecting **Create new branch**. You will see a new “branch badge” in that row of the History grid to indicate where your branch begins.



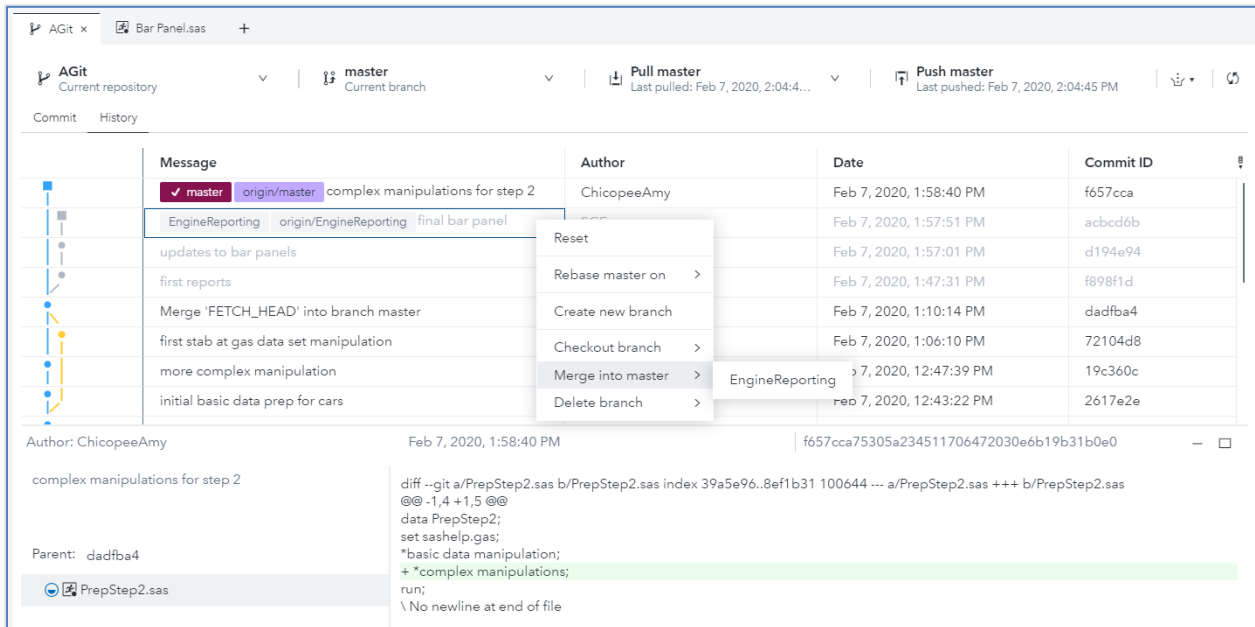
Display 8. Creating a Branch

Now that you are working in your own branch, add new code and go through a few cycles of saving, staging, committing, and pushing. The history shows the timeline, and the code differences reflect just what’s in the branch.



Display 9. Working in a Branch

To merge your branch into the master, in the History tab, select the *current-branch* pulldown menu and click **master** to check out the master branch. Click **Pull current-branch > Pull** to make sure you have the most recent updates. And, for this example, it's a good thing you did, since more work had been done on the data prep files. Now you can right-click on your branch and select **Merge into master** and the branch name. Note that the History grid now displays a graph line on the left to indicate where your branch was originated and merged.



Display 20. Merging a Branch into Master

There is a lot more you can explore here including fetching versus pulling, rebasing, and stashing. Consult the "Understanding Git Integration" section in the documentation of your preferred interface.

SUMMARY OF GIT TERMS AND PROCESSES

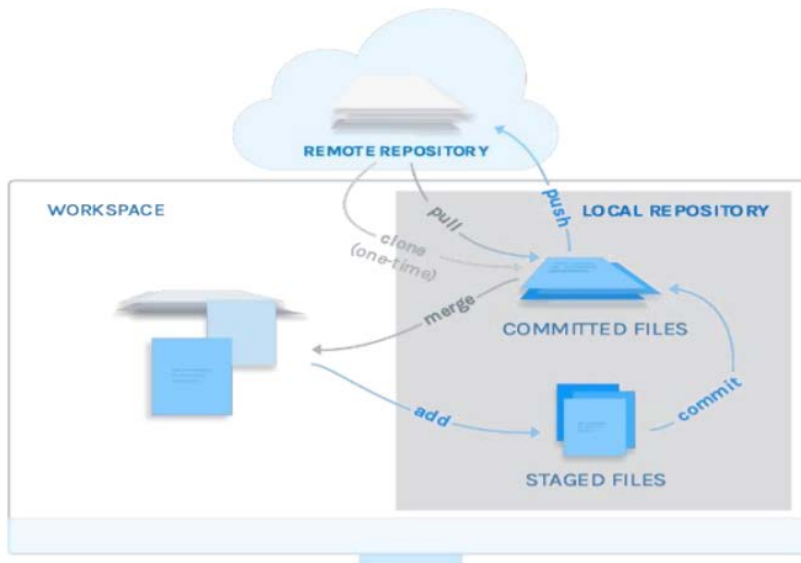
Here is a brief summary of the terms described in the above examples.

- **Remote repository** – project files and folders that are stored on a website (e.g. GitHub, Bitbucket) that provides support for Git version control functions.
- **Clone** – creates a local copy of a remote repository on your computer, including a set of files and folders that are used by Git for managing changes.
- **Local repository** – a personal work area for developing and testing files within the Git version control framework where you can create, modify, or delete files and folders.
- **Unstaged** – local repository files that have been edited and saved but are still considered as “in progress”.
- **Staged** – local repository files that you have tested and are ready for moving to the shared remote repository, often grouped with related files in the same state.
- **Commit** – designates a set of staged files as an approved “unit of work” intended for sharing with colleagues who have access to the remote repository.
- **Push** – promotes the files within a commit to the remote repository if the user has proper permissions.
- **Reset** – undoes the changes in a staged or unstaged local repository file and removes it from the Commit view.
- **History** – a log of all committed changes in a Git repository from which you can create or merge branches, reset your local repository to a previous commit.
- **Pull** – updates your local repository with any changes that have been pushed to the remote repository since the last time your local history was updated.
- **Branch** – a snapshot of the state of all files that comprise a selected commit in the History grid.
- **Checkout** – switches to the selected branch and updates your local repository with the files that belong to that branch.
- **Merge** – integrates the changes made in a secondary branch with the current branch; usually followed by deletion of the secondary branch.

SAMPLE SCENARIO – WHAT’S HAPPENING BEHIND THE SCENES

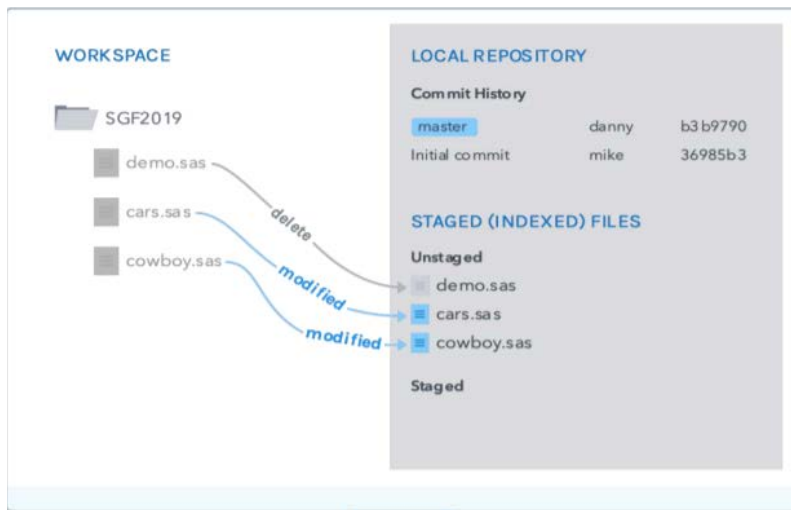
To get a copy of an existing Git repository, you’ll need to clone it (typically you do this only once) to your server or computer. The Git clone command creates a local repository in the specified folder on your computer and pulls down all the data and history for that repository.

When you first clone a repository, all your files are tracked by Git. At this point, the status of all the files is “unmodified” because you haven’t edited anything. You see them in the file navigation, but they do not show up in the Git interface. Git has three main file states: committed, modified, and staged. Committed means that the data is safely stored in your local repository. Modified means that you have changed a file in your development workspace but have not committed it to your local repository yet – these show as unstaged in the Git interface. Staged means that you have marked a modified file to be committed.



Display 31. Parts of a Local Repository

Suppose you need to fix two files (for example, `cars.sas` and `cowboy.sas`) and delete one file (`demo.sas`). In your development workspace, you make the changes and test them. As you edit files, Git sees them as “modified,” because you’ve changed them since your last commit. They are still in your development workspace as you have not marked them as ‘staged.’ You can’t commit files until they are staged.



Display 42. Staging Files

Now in your development workspace, you have a list of unstaged files that have been modified: `cars.sas`, `cowboy.sas`, and `demo.sas`. Git commits only files in the “staged files” list. These are also referred to as “indexed files.” The next step is to indicate which of these modified files you want to include in your next commit. Put all of these files into the Git staged area. Git commits all staged files together as a single commit to the local repository.



Display 53. Executing a Commit

Remember that the commit records the snapshot you set up in your staging area. Anything you didn't stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later. This is done by storing only the lines modified as part of each commit. All changes made to the file put together in sequence constitute its state at any given point in the history. This ensures that the repository remains as small as possible from a storage perspective. Prior to pushing your changes to the remote repository, you first want to pull to get any changes someone else might have pushed to the remote repository.

GETTING STARTED

There are two steps that need to be performed in SAS Studio to set up the Git interface.

1. Create a Git profile.
2. Set up a connection to a remote repository and clone to a local repository.

We are going to walk through setting things up to use the Git interface in SAS Studio. Before the two steps above can be performed, we need a remote repository and a URL to access it. If you are fortunate enough to have an administrator to set this up for you, enjoy skipping this section. If you are on your own, follow these instructions to create a repository on GitHub.

CREATE A GITHUB ACCOUNT AND A REPOSITORY

1. Create an account on <https://github.com>. You can open an account and create repositories for free.
2. Create a repository on GitHub. Once you have verified your account from the email address you chose, you will be taken to a screen to create a repository. Choose a name for your repository and decide whether it will be Public or Private. You can check **Initialize this repository with a README** as an easy way to add the first file to the repository. Then add or create a new SAS program. You can upload files from your PC or choose to create a new one. Once the .sas file is created or uploaded, commit it. You are required to add a comment with every commit.

3. While you're here in GitHub, make a note of the location of the repository URL address that we will need later. Click the green **Clone or Download** button. At a later point, you will click the clipboard to the right of the URL to copy the address to your clipboard. (You can do it now if you think you might make it through the next few steps with it still on the clipboard.)

Now you have a remote repository to use. The other step you need to do outside of SAS Studio is to generate your SSH keys.

GENERATE YOUR SSH KEYS

SSH keys are authentication credentials. By default, SAS Studio uses SSH keys to authenticate with your repository hosting service. Your keys will be used in both your SAS Studio profile and in your GitHub account.

You do not need to install Git on your Windows machine for the Git functionality; however, you can use it to generate the required SSH keys. If your SAS Studio server is a Unix machine, and you can access the operating system, you can skip the Git for Windows setup and run the ssh command below.

You can download Git for Windows at <https://git-scm.com/download/win>. Once the software is installed, you can use Git Bash to generate the SSH keys. Git on Windows, like GitHub, has a graphical user interface to Git. Git Bash allows you to use a command line to run Git commands. After opening Git Bash, either copy and paste or enter the following command:

```
ssh-keygen -t rsa -m PEM -b 4096 -C your_email@example.com
```

You can accept the default file locations and not enter a passphrase to get the keys created quickly. Now you need to add your ssh key to your GitHub account. See <https://help.github.com/en/github/authenticating-to-github/adding-a-new-ssh-key-to-your-github-account>.

Now you have a remote repository, and you have your SSH keys. You are ready for the two steps needed within SAS Studio.

CREATE A GIT PROFILE

You use the Add a Profile window to create a Git profile.

In SAS Studio 5.2, select **Options > Manage Git Connections**. Make sure **Profiles** is selected in the navigation pane and click +.

In SAS Studio 3.8, click  and select **Preferences**. Select **Git Profiles** in the navigation pane and click +.

The screenshot shows a dialog box titled "Add a Profile" with a blue header bar. The dialog contains the following fields and values:

- Profile name: * (required): My Profile
- User name: * (required): MyUserName
- Email: * (required): user@company.com
- Public SSH file path: C:/Users/Documents/MySASFiles/ssh_keys/id_rsa.pub
- Private SSH file path: C:/Users/Documents/MySASFiles/ssh_keys/id_rsa

At the bottom right of the dialog are two buttons: "OK" (highlighted in blue) and "Cancel".

Display 64. Adding a Profile

Choose and enter a profile name, and enter your user name and email address. The user name and email address will be used to identify you as the one who made a change to the remote repository. If you'll be working with a public repository, you may want an alias to keep your identity private. If you're working with a repository in your organization, you may use your user id or some other organization standard. If you're just working on your own personal repository, you can put any text there.


Navigate to the location, including the filename, of your SSH keys. If the path to your keys is not accessible from the server SAS Studio is running on, you will need to first upload the files to a location SAS Studio can access.

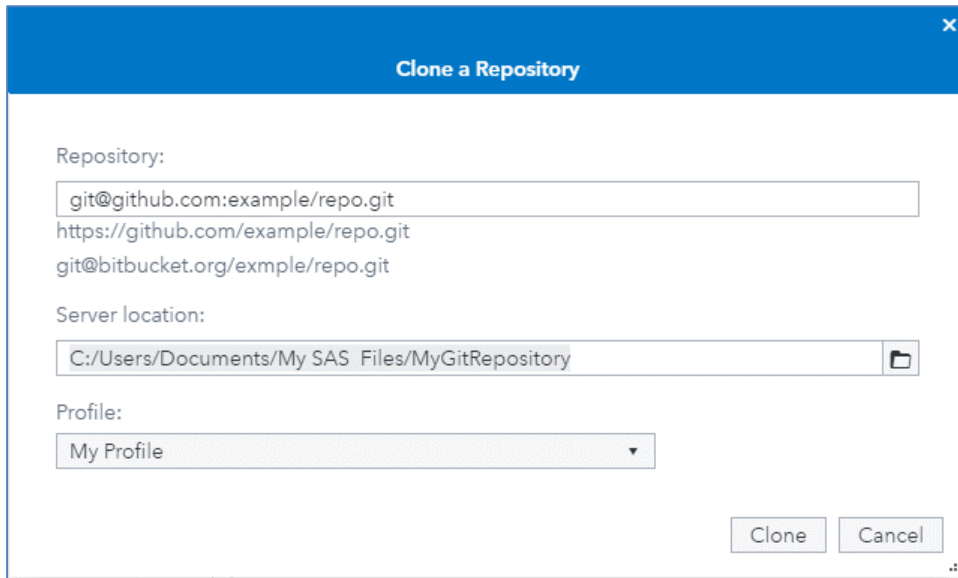
Press **OK** to save your profile.

SET UP A CONNECTION TO A REMOTE REPOSITORY AND CLONE TO A LOCAL REPOSITORY

You use the Clone a Repository window to clone a Git repository.

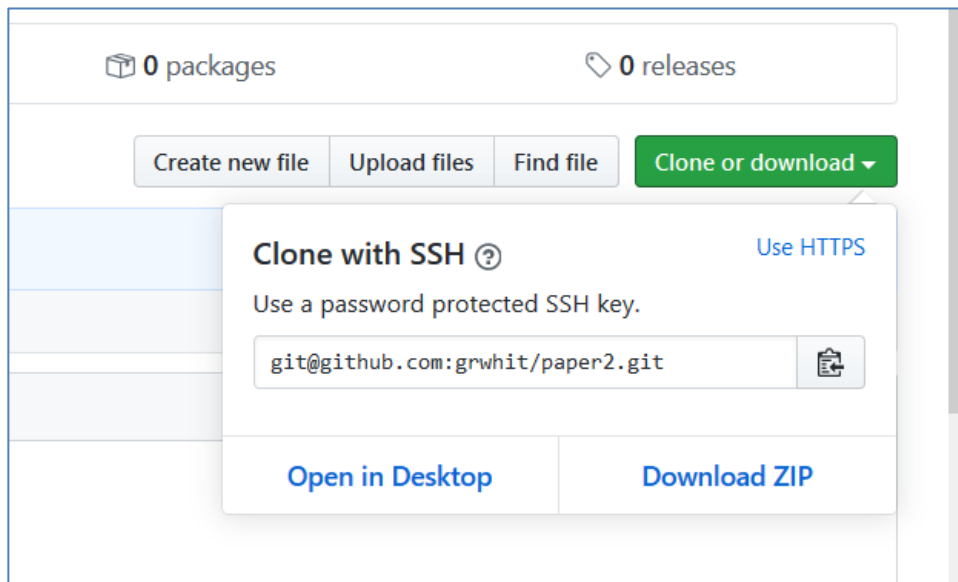
In SAS Studio 5.2, select **Options > Manage Git Connections**. Select **Repositories** in the navigation pane and click **+ > Clone a repository**.

In SAS Studio 3.8, click  and select **Preferences**. Select **Git Repositories** in the navigation pane and click **+**.



Display 75. Cloning a Repository

At this point, you will need the URL of the repository you created on GitHub. Remember that green **Clone or Download** button? Click that button to open a window with the path. Make sure that the window displays “Clone with SSH” like below.




Display 86. Getting the Repository Information from GitHub

Click the clipboard icon to copy the address to your clipboard. Then paste it into the remote repository name field in SAS Studio.

Determine the location where you want your local repository. The folder at the end of the path you choose does not have to currently exist. If it does exist, it must be empty. This field is labeled **Local repository** in SAS Studio 3.8 and **Server location** in SAS Studio 5.2.

Select the Git profile that will be used with this repository. At this point, you have just the one profile that you defined previously, but in the future, you might have multiple profiles that correspond with different repositories.

Click **OK** (SAS Studio 3.8) or **Clone** (SAS Studio 5.2) and your remote repository on GitHub will be cloned (copied) to the folder location you have specified on the SAS Studio server.

You should now be able to go to the specified folder location in the navigation pane and see that it is marked by the repository icon . Within the repository should be all the files you had created in the repository on GitHub.

Now you are all set up to use Git functionality through the SAS Studio UI! For more details about how to use all the features see *SAS Studio: User's Guide*.

SAS Studio 3.8 and SAS Studio 5.2 also support HTTPS authentication. To use HTTPS, your admin would need to configure SAS Studio to do so. The SAS Studio administration documentation includes information on how to accomplish this. SAS Enterprise Guide only uses HTTPS authentication when authenticating with a remote repository.

UNDERSTANDING THE DIFFERENCES USING GIT WITH SAS ENTERPRISE GUIDE VERSUS SAS STUDIO

SAS Enterprise Guide 8.2 includes several features that are modeled after the SAS Studio 5.2 Git interface. Differences between the two versions include:

- SAS Enterprise Guide 8.2 supports local repositories that are stored on the client machine rather than the SAS server.
- SAS Enterprise Guide 8.2 supports HTTPS authentication only; SAS Studio 5.2 supports SSH and HTTPS.
- SAS Enterprise Guide 8.2 does not support the Git Rebase or Stash functions; SAS Studio 5.2 supports both Rebase and Stash.
- SAS Enterprise Guide 8.2 allows use of a Windows Diff tool for side-by-side file comparisons.
- SAS Enterprise Guide 8.2 allows use of a Windows Merge tool for managing file conflicts. In SAS Studio 5.2, such differences are highlighted in a file tab and reconciled manually by the user.
- SAS Enterprise Guide 8.2 carries forward the program history feature from previous releases, allowing file changes to be tracked independently without requiring a local repository definition.
- Both applications allow limiting of commit history row retrieval for performance reasons. To access this option in SAS Studio 5.2, select **Options > Manage Git Connections > Options**.
To access this option in SAS Enterprise Guide 8.2, select **Tools > Options > Version Control**.

A table summarizing the differences is available in the SAS Studio FAQ - http://support.sas.com/software/products/sas-studio/faq/SASStudio_vsEG.htm .

USING CODE TO DO ALL OF THIS

Yes, you can use Git programmatically within the SAS language in the form of DATA step functions. In fact, the entire SAS Studio 3.8 and SAS Studio 5.2 interfaces are driven by these functions. In this section, you will go over the basic Git scenario of cloning a remote repository, staging a file, committing, and then pushing that commit back to the remote repository all within the SAS language. In this scenario, we will be using the Git functions that were released with SAS Viya 3.5. They have refactored names compared to the SAS 9.4M6 versions of the functions that are easier to read.

The first thing you want to do is clone a remote repository. For this example, we will be using the SAS Communities SAS Global Forum 2020 repository.

<https://github.com/sascommunities/sas-global-forum-2020>

NOTE: You will not be able to push changes to the SAS Global Forum 2020 remote repository.

When using SSH authentication, the clone function takes 6 parameters.

- The SSH remote repository URL - If you're logged in to GitHub, the green **Clone or download** button opens a fly-out window that gives you the option to clone with HTTPS or SSH. You need the SSH URL. In this scenario, you are going to use the sascommunities/sas-global-forum-2020 GitHub repository.
- The location on the local file system where you want to clone your repository to - For this scenario, choose a different location than the previous examples.
- The username - When using SSH authentication, the username needs to be whatever comes before the @ sign in the SSH remote URL. With GitHub, it is "git".
- The password - This is not required for SSH authentication, so leave as an empty string as seen below.
- The path to the public SSH key on the file system - It would be the same value you put in the SAS Studio Git profile in the previous example.
- The path to the private SSH key.

The clone function:

```
data _null_;
  rc = GIT_CLONE(git@github.com:sascommunities/sas-global-forum-2020.git,
                C:\SGF2020_SASCommunities,
                "git", "",
                "C:\MySSHKeys\id_rsa.pub",
                "C:\MySSHKeys\id_rsa");
run;
```

Now that you have cloned the remote repository successfully, navigate to the local repository that was created and add your SAS Global Forum paper to the repository. When you're happy with your changes, the next step in this scenario is to stage your paper for commit. That brings us to the GIT_INDEX_ADD DATA step function. Staging a file is adding that file to the Git Index. The index is where changes to files that are waiting to be committed are stored.

The staging function has 3 required parameters and N number of additional parameters, which allows you to stage multiple files at one time.

- The path of the local repository.
- The path of the file that you want to stage relative to the local repository.
- The status of the file, in this case "new".

The staging function:

```
data _null_;
  rc = GIT_INDEX_ADD("C:\SGF2020_SASCommunities",
                    "papers\4197-2020-Peters\SAS4197-2020.docx",
                    "new");
run;
```


Now that your paper has successfully been staged, you can commit the staged paper to the local repository. To do this use the GIT_COMMIT function. The GIT_COMMIT function takes 5 required parameters:

- The path of the local repository.
- The update reference - Use "HEAD" in this scenario. "HEAD" means you are committing to the top, or head, of the current branch.
- Committer's Name - This is informational (so that people can come find you when you break something; you can use a fake name if you don't want to be found).
- Committer's Email - This is also informational. The name and email will show up in the commit history.
- Commit Message - The message includes a detailed description of the commit.

The commit function:

```
data _null_;
  rc = GIT_COMMIT("C:\SGF2020_SASCommunities",
                 "HEAD",
                 "James Bond",
                 "007@MI6.gov",
                 "My SGF Paper");
run;
```

Now that your paper has been successfully committed to the local repository, it's time to push it up to the remote repository so that everyone who uses the SAS Communities GitHub repository can see your paper. The Push function is very similar to the clone function except that you only need to supply your local repository path. The local repository remembers the remote repository URL you used when you cloned.

The push function:

```
data _null_;
  rc = GIT_PUSH("C:\SGF2020_SASCommunities",
               "git", "",
               "C:\MySSHKeys\id_rsa.pub",
               "C:\MySSHKeys\id_rsa");
run;
```

You can now navigate to the remote repository on GitHub and see your commit. To follow best practices, you will want to pull before pushing, because you cannot push unless your local repository is up-to-date with the remote repository. The pull function is GIT_PULL and uses the same parameters as push. We left the pull example out to keep the scenario simple.

You can find documentation on all the Git functions in the programming references.

CONCLUSION

Whether you choose to use basic versioning functionality in the program history feature in SAS Enterprise Guide, go all out with branching and merging in the full Git interface in SAS Enterprise Guide or SAS Studio, or write your own SAS Git function utilities, you have plenty of options for keeping track of your own code and for collaborating with others.

CONTACT INFORMATION

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

Amy Peters

SAS
Amy.Peters@sas.com

Danny Zimmerman
SAS
Danny.Zimmerman@sas.com

Grace Whiteis
SAS
Grace.Whiteis@sas.com

Joe Flynn
SAS
Joe.Flynn@sas.com

Stan Polanski
SAS
Stan.Polanski@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.