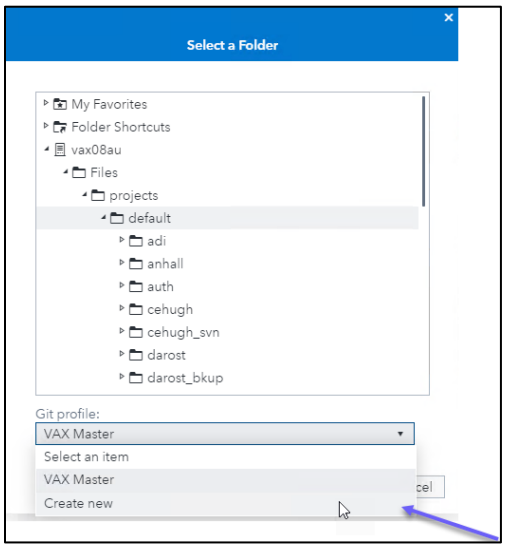
4. When the panel opens, navigate to the location of your playpen created in the section above.
5. Select your playpen, but before clicking **OK**, expand the Git profile drop down and select **Create New** (Figure 8).

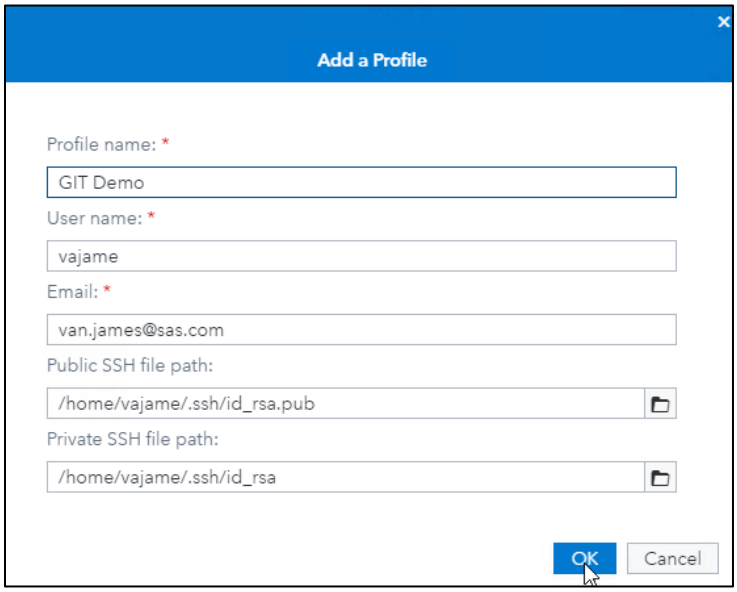
Figure 8: SAS Studio V Configure Existing Repository



Note: When adding a profile, it is important that the **Public SSH file path** and **Private SSH file path** fields match the location of the RSA key created when you set up your token authentication. This enables SAS Studio V to interact with GitLab without having to store your credentials on the machine.

6. When the fields are correct, click **OK** to add the profile (Figure 9), then select **OK** again to add the repository.

Figure 9: SAS Studio V Create Git Profile



For more information on using Git with SAS Studio, refer to the SAS Studio User's Guide.

3.6 Playpen Directory Structure

Once your playpen has been established you will see a collection of directories and sub-directories. Table 3 below describes how the directories are to be leveraged in the USPS environment. Depending on the nature of the project and the software that will be leveraged some of the directories listed below may not be available.

Table 3: USPS Playpen Directory Structure

Directory	Sub-Directory	Usage
auth		Contains encrypted password files
bin		Specific scripts to start sas for subsystem purposes (using separate autoexec/sasautos/etc)
conf		Specific configuration files to use for subsystem purpose
jmp		JMP-specific programs and projects files.
	jsl	JMP script files
	journals	JMP Journals (Projects)
logs		Used to store logs during batch runs. (Do not include in GIT)
sas		Root Folder for all .sas files. Do not store sas programs at this level.
	discoed	Data Integration Studio generated SAS code
	disjobs	Data Integration Studio exported Job XML
	distransforms	Data Integration Studio transform code
	equide	Enterprise Guide projects
	includes	SAS code snippets that are added to SAS programs with the %include command
	one-offs	Single jobs written for ad-hoc or single run purposes, typically named with corresponding JIRA ticket.
	programs	Stand-alone SAS programs
	sasautos	SAS macro functions that can be leveraged across SAS programs
	stp	Stored Process jobs if applicable

4 Trunk-Based Methodology

4.1 Overview

USPS follows the trunk-based development paradigm when working in a Git environment. Trunk-based development aims to keep development work close to the master branch to avoid complicated merges. For new features, a short-lived branch is created to add feature flagging. This branch is then merged back into the master branch and the new feature is developed directly in the master branch.

Bug fixes follow a similar flow. A branch is created off the master to fix the bug. The branch is then merged back into the master branch where the master branch undergoes quality assurance testing.

When ready for dissemination, the bug fix is cherry-picked from the updated code and pushed out to the release candidates already deployed to the development, testing, and/or production servers. Merge requests are leveraged so that the maintainer(s) of the Git repository can review any code before it is merged back into the main repository.

4.2 Naming Conventions

Bug fixes, feature branches, and release branches should all follow a standard naming convention. Using standard naming across projects reduces the learning curve as new resources are brought on to the project. It also facilitates flexibility among the delivery team by reducing time spent on knowledge transfer. In that vein, USPS should adhere to the following naming conventions when creating branches off the master.

4.2.1 Feature Branches

features/{feature-name}

Where *{feature-name}* is a descriptive name of the feature being added.

4.2.2 Bug Fix

bugfix/{description}

Where *{description}* is a short description of the bug that is being fixed within the branch. For complex bugs please reference the Jira ticket in lieu of the description.

4.2.3 Release Branch

release/{release number}

Where *{release number}* is a counter that increments after each release. It is not necessary to track major and minor release versions in the branch name. As long as each release has a unique release number, you are able to reconcile the release number to the corresponding major / minor release version.

4.3 Concepts

4.3.1 Merge Request

When changes are made in a developer's repository, a merge request is used to submit those changes to the main project repository. The merge request initiates a request to the project maintainer to review

the changes before the code is approved to be merged into the main code base. For large projects, it is recommended that the repository have several designated repository maintainers. This built-in peer review process ensures that all code merged into the master repository has been vetted.

4.3.2 Feature Flagging (Toggling)

Feature flagging, or feature toggling, is used to incorporate new features or update existing features in the code base, without impacting the user experience. To achieve this goal, a branch is created off the master. Within that branch, a conditional statement is added that allows the new or updated feature to be toggled on or off.

For updating or replacing older features, an additional conditional statement is used to toggle off the old code. This process is best achieved by using a database to maintain a list of feature statuses. Then an *if* or *if/else* statement is used to toggle on and off features.

See Section 4.4 for more details.

4.3.3 Cherry-Picking

Cherry-picking in Git means to take a commit from one branch and apply it to another branch. When code is committed in Git, an SHA key is created and serves as a unique key for that commit. Use the SHA key for a given commit to selectively choose which code updates can be pushed out to other branches. USPS leverages cherry-picking to fix bugs that have been discovered in a release branch.

4.4 Feature Flagging Workflow

Feature flagging allows the development team to toggle features on or off, thereby allowing new features to reside in the master branch along with production-ready code. A database table maintains the features' status, allowing lower environments (that is, Development and/or Test) to have new features turned on, while the same code base in the Production environment represses the feature. If the application does not have database support, then a configuration file can also be used to track these variables.

Note: An additional update to the code base is needed each time you want to toggle a feature using the configuration file.

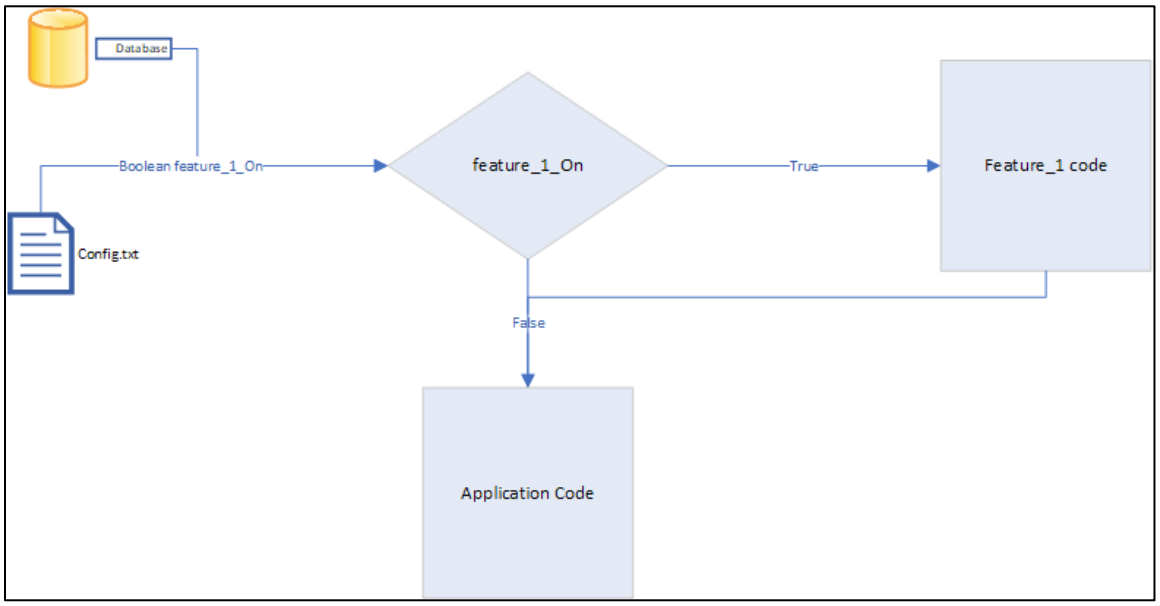
After the feature flagging conditional logic has been added to the new branch, a merge request can be initiated to merge the logic back into the master code base. At that point, development of the new feature can proceed in the master branch without impacting the end user.

4.4.1 Scenario 1: Adding a New Feature to the Existing Application

When adding a new feature to the existing code base, the code base needs to be updated with conditional logic to toggle the new feature on and off. In Figure 10, the Boolean variable *feature_1_on* flag is read from either a database table or a configuration file.

- If that flag is set to **True**, then the feature identified as *Feature_1* in Figure 10 is executed and then moved to the existing application code.
- If the variable *feature_1_on* is set to **False**, then the *Feature_1* code is not executed.

Figure 10: Feature Forking with a New Feature

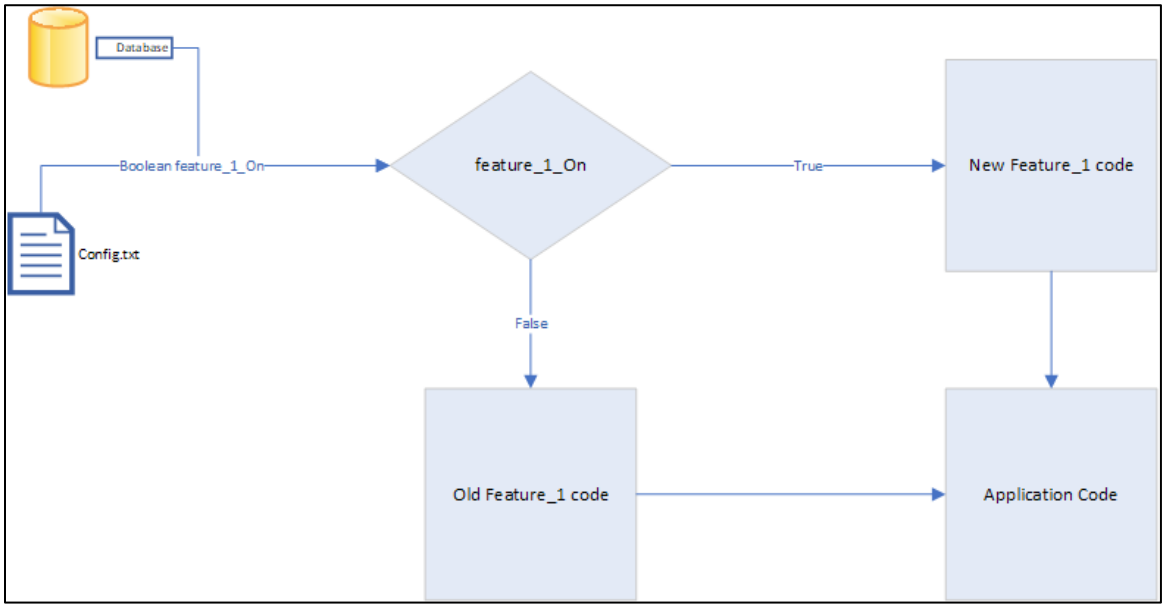


4.4.2 Scenario 2: Updating an Existing Feature

When replacing an existing feature with updated code, the code base needs to be updated with conditional logic to toggle the new feature code on or off, and the old code off or on, respectively. In Figure 11, the Boolean variable *feature_1_on* flag is read from either a database table or a configuration file.

- If that flag is set to **True**, then the feature identified as *New Feature_1* in Figure 11 is executed, while the code identified as *Old Feature_1* is skipped. After execution completes, the application code proceeds as usual.
- If the flag identified as *feature_1_On* is set to **False**, then the *Old Feature_1* code is executed.

Figure 11: Feature Forking to Update Existing Feature



4.5 Merge Request Workflow

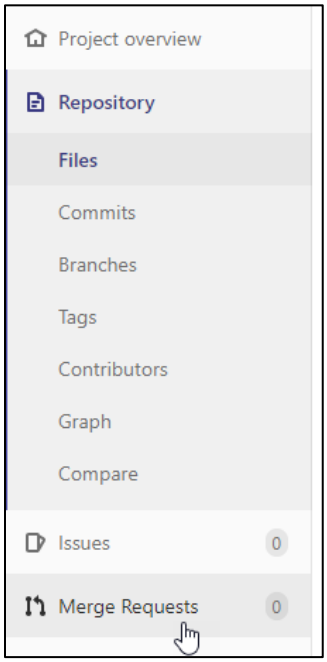
The following screen shots were taken from GitLab, but the process is similar for GitHub. A merge request opens a dialog between the developer and the maintainer of the project code base. After code is ready to be merged into the main repository, the developer opens a merge request, which alerts the maintainer that code is ready for review. The maintainer can now review the code before it is merged into the main code base. If questions arise, the maintainer can comment in the merge request, asking for clarification from the developer. This dialog is maintained with the merge to document the exchange.

4.6 Submitting a Merge Request

To initiate a merge request, use the following steps:

1. Log on to the Git web interface and navigate to your personal repository that is tied to your development playpen.
2. Select **Merge Requests** from the left navigation panel (Figure 12).

Figure 12: Merge Request Navigation Button



3. Confirm that you want to create a **New merge request**, as shown in Figure 13.

Figure 13: New Merge Request