Migrate from other systems to Git

04/03/2017 • 7 minutes to read

By: Robert Outlaw

Before you try to migrate your source code from a legacy version control system to Git, be sure that you familiarize yourself with the differences between centralized version control systems and Git, and plan your team's migration.

Azure DevOps has simple tools to migrate from Team Foundation Version Control (TFVC), but migrating from a different centralized version control system is also straightforward.

In general, the migration process is:

For the mainline or first branch you wish to migrate

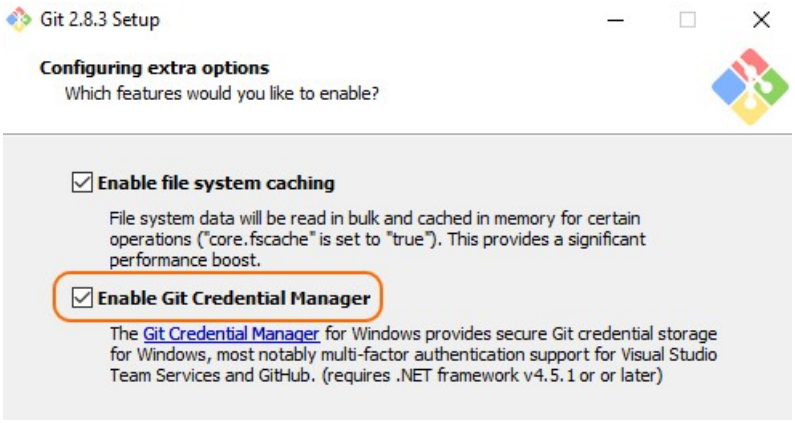
1. Create a repository in Azure DevOps and clone it locally.
2. Check out the latest revision or changeset in your legacy version control system.
3. Prepare large binaries; for dependencies like libraries and build tools, set up a package management system like NuGet. For binary assets like images, configure Git-LFS, the Git Large File Support extension.
4. Convert version control system-specific directives that you need to retain in Git. For example, convert `.svnignore` or `.tfignore` files to `.gitignore`, and convert `.tpattributes` files to `.gitattributes`.
5. Delete files or data that binds your code to the legacy version control system. For example, delete the `$tf` directory that contains TFVC metadata, or the `.svn` directory used by Subversion.
6. Optionally, create and commit a "breadcrumbs" file that provides information about how to access your legacy version control system.
7. Add and commit your files to the Git repository and push the repository to Azure DevOps.
8. Since you may need to run these steps multiple times, for test migrations before the final migration, you should create a script for this process that can be run repeatably.

For any remaining branches you wish to migrate

1. In the Git repository, checkout a new branch.
2. Replace the working directory's contents with the contents of your legacy branch.
3. Add and commit all files. (Due to the way Git stores history, only the files that differ in this branch will actually be uploaded.)
4. Push your new branch to Azure DevOps.
5. Checkout master again to prepare for the next branch.

Prerequisites

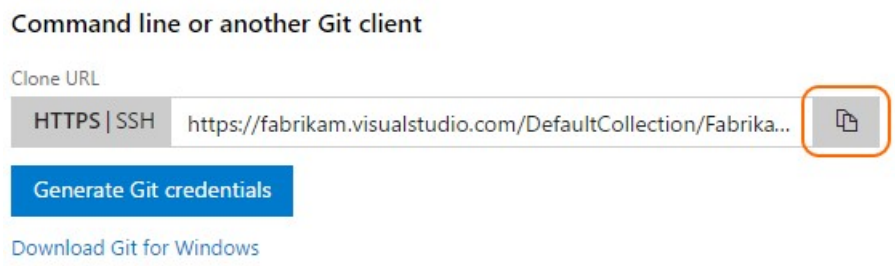
Install Git for Windows if you haven't done so already. Make sure to **Enable Git Credential Manager** during the installation so you can easily connect to Team Services or TFS.



Create and clone your Git repository

Create a Git repository in Visual Studio Team Services or an on-premises Team Foundation Server that will contain your code when the migration is complete.

Make note of the **Clone URL** for this repository.



Use this URL on the command-line to clone the new, empty repository that you just created. This will create an empty folder on-disk that you will populate with the contents of your legacy version control system:

```
git clone https://fabrikam.visualstudio.com/DefaultCollection/FabrikamApp
```

Populate your Git repository

Get or check out the latest version of the main branch of your application from your legacy version control system, and copy it into the folder that you created when you cloned your repository. Do not stage or commit your migrated code in Git yet.

Remove binary dependencies and assets

Due to the way Git stores the history of changed files, providing a copy of every file in history to every developer, checking in binary files directly to the repository will cause it to grow quickly and cause performance issues.

How you manage binary files with Git depends on what types of resources you have. If you have build dependencies like libraries that you depend on or if you check in tools used to perform the build, you should manage those with a package management solution like NuGet. If you have assets like artwork, game models or scientific data sets, you should set up Git-LFS, the Large File Support extension.

To discover the large files that you currently have in your centralized version control system, you can run the following PowerShell command from the folder containing your code:

```
gci . -r | sort Length -desc | select fullname -f 25
```

This returns the 25 largest files in your repository. You can tweak this command and filter the output on other criteria as needed.

Exclude tools and libraries

For tools and libraries, adopt a packaging solution with versioning support, such as NuGet. Many open source tools and libraries will already be available on the NuGet Gallery, but for proprietary dependencies, you will need to create your own NuGet packages. Once you have moved your dependencies into NuGet, make sure that they will not be included in your Git repository by adding them to your `.gitignore` file.

Manage asset files with Git-LFS

Look at the size and the history of the binary asset files in your repository, such as images and models. If they are small and very rarely updated, such as an icon file, then you can add them to Git directly without bloating the repository.

If you have large binaries, or assets that change frequently, then you should use Git-LFS to manage these assets. Simply follow the instructions to set up the Large File Support extension, then add the files to your local repository.

Convert version control-specific configuration

Many version control tools offer an "ignore" file to ensure that some files are not included in version control. For example, Team Foundation Version Control provides a `.tfignore` file and Subversion offers a similar `.svnignore`. If you rely on this behavior, convert these files to a `.gitignore` file.

Some version control tools also offer an "attributes" file that controls how files are placed on the local disk when checked out or translated when checked in to the repository. For example, a `Makefile` or a shell script may have specific line ending requirements that an attributes file can enforce. If you rely on this behavior, convert these files to a `.gitattributes`.

Delete legacy version control metadata

Many version control systems place metadata on-disk within the working folder. For example, Team Foundation Version Control creates a `$tf` directory on Windows (`.tf` on macOS and Unix platforms) to store information about the local filesystem and Subversion stores metadata in a directory named `.svn`.

These directories should not be included in your Git repository and should included in your `.gitignore` file or deleted from the local disk entirely.

Add a breadcrumbs file (optional)

If you're doing a "tip migration", only bringing in the latest version of your source, then you may want to add a "breadcrumbs" file that reminds people how to find older versions of the source tree.

It's helpful if this breadcrumbs file is the first commit in your new repository, even before the code migration. Create this file, then add it and commit it:

```
git add README-history.md
git commit README-history.md -m"Import from legacy version control: readme for earlier history"
```

Add and commit your files

Finally, you can add and commit your code to your local git repository.

```
git commit --all -m "Initial import from legacy version control"
```

This will commit your code to the `master` branch of your new Git repository, which is commonly the name of the default branch. Once the code is committed, push it to the `master` branch of your Git repository in Azure DevOps:

```
git push origin master
```

The main branch of your code is now in a Git repository in Azure DevOps. If you don't need to migrate any other branches, then your migration is complete.

Migrate branches

Once you've committed and pushed your code, migrate additional branches by creating a new branch from the `master` branch in your new Git repository and switching to it. Do this for each branch you want to migrate.

In this example, we are using a branch name `releases/20`:

```
git checkout -b releases/20
```

> Only migrate legacy branches that make sense in your team's Git branching strategy.

Copy over the most recent version of the code from your legacy version control system's branch into the working directory for your Git repository. Perform the same steps as before to filter out large files, binaries, and version control system metadata. Add and commit this branch to Git:

```
git commit --all -m "Initial import of the release 2.0 branch"
```

Then push your branch to Azure DevOps:

```
git push -u origin releases/20
```

Finally, switch back to the master branch by running `git checkout master`, and repeat this process for any other branches you need to migrate.

Update your workflow

Moving from a centralized version control system to Git is more than just migrating code. Your team needs training to understand how Git is different from your legacy version control system and how these differences affect day-to-day work. Learn more.

 Get started with unlimited free private Git repos in Azure Repos.

Robert is a content developer at Microsoft working on Azure DevOps and Team Foundation Server.