# Chapter 5: Tracking Code

Version Control Systems (VCS) have fully matured over the last twenty years, and the Git system has become the de facto standard in the software world. In fact, the largest active source control repository on the planet, the Microsoft Windows source code, has been converted to Git. Centralized source control systems like Subversion and Team Foundation Version Control (TFVC) have given way to Mercurial and Git. Of those two, Git has become the version control tool of choice for developers on all the modern platforms. Needless to say, tracking your code in Git version control is part of a modern DevOps process. Throughout the industry, practitioners use version control, source control, VCS, and SCM interchangeably.

Azure Repos is the version control system in the Azure DevOps family. It supports the old TFVC format of source control as well as an unlimited number of private or public Git repositories. There are import tools for migrating existing code repositories into Azure Repos, so regardless of where your code is now, you can move it in. Azure Repos not only works with Visual Studio, but it also works with any other Git client, such as TortoiseGit, which is one of my favorites. For the purposes of this chapter, and much of the book, we will equate Azure Repos with Git version control. While TFVC will be supported for well over another decade, any new investment you make should use the Git technology.

## How many repositories?

In setting up your Git repository in the professional way, there are some principles to keep in mind. First, your team will likely have multiple repositories, unless you ship only one product. The architecture of your software will have something to do with the granularity of your repository design. For example, if you deploy your entire system together and the architecture doesn’t support deploying only a subset of the system, it’s likely that you will put the entire system into a single Git repository. As an organization, you may have multiple software teams. Here is an Entity Relationship Diagram to help you understand how to factor your system into Git repositories.



Figure: Relationship rules when designing Git repositories

Your team will own your Git repositories. A single Git repository cannot be owned or developed by multiple teams. With centralized version control systems of years past, this was possible only because these systems supported child-level branching. These systems hosted a repository of a different type. The reuse of the term “repository” has led to some confusion among users of TFVC and Subversion, which happily hosted multiple software systems while allowing branching at the child level. While merging was difficult much of the time, these tools did support it. Git’s repository design is different. Cloning and branching is done at the top level only. Therefore, to manage multiple pieces of software, you create multiple Git repositories. In Azure Repos, a single project can have an unlimited number of Git repositories, so you do have a way to maintain groupings of related Git repositories.

Now that we understand that a team must have a dedicated Git repository, our next relationship is the software application itself. Regardless of the size of the software application, there should be only one. Your application can be a small microservice with nothing more than an Azure Function, or it can be a very sizable application. If it maintains independent versioning and deployability, it must reside in its own Git repository, which owns the concept of versioning. If you are a Git and build expert reading this, you may be able to invest a custom paradigm that can violate this rule, but for the rest of us, this rule holds true. Let’s consider some examples of this.

1. You have a very large Visual Studio solution for a software system that is over 10 years old. It has a few web applications, some Windows services, some schedule jobs, and a SQL Server database. The question to ask is “do any parts of it build or version independently of the rest?” If the answer is no, then all of it belongs in the same Git repository. Don’t fret if sometimes you make changes to the website and then decide not to deploy the rest to production. That’s not the same as being versioned independently.
2. You have designed a system with independent sub-applications or microservices. Each of these applications owns its own small database, and the parts communicate asynchronously via queues. Each can change and deploy at a completely different cadence. In this scenario, you would segment each into it’s own Git repository in order to preserve the ability to maintain version independence.

There are some examples where you might have a system decomposed into mostly independent applications but want to keep them in the same Git repositories. Azure DevOps itself is a perfect example of this. The segmentation is to benefit the deployment architecture rather than version independence. There are dozens of services that make up the Azure DevOps product, but they all reside in a single Git repository, and a single (but large) team develops the system in Git. The whole system is built together and deployed together with a single version number. You can read more about how the Azure DevOps team does DevOps online[[1]](#footnote-1). To drive to a rule of thumb: put your current Visual Studio solution in its own Git repository and only put one.

## Structuring your Git repository

Sldkal ksldaklsdk alksd

* One team can own multiple repositories, but one repository cannot be effectively owned by multiple teams.
* Don’t use spaces in Azure DevOps project names. These spaces become %20 in the Git URL and that causes problems down the road.
* Use one branching pattern. Branch for features off the master. Don’t attempt to branch only a child path.
* When importing TFVC or Subversion, don’t maintain a structure where sub-repositories are designed with multiple independent applications. Break them apart.

There’s often a discussion about what to store in the application’s Git repository. The short answer is “store everything.” Absolutes are never right. (Except for the previous sentence.) However, you do store almost everything in your Git repository, including:

* Database schema migration scripts
* Azure Resource Manager (ARM) JSON files
* PowerShell scripts
* Tests
* Build scripts
* Images
* Content assets
* Visio architecture blueprints
* Documentation
* Dependencies, including libraries and tools

Given that there are some exceptions not to store, I’ll go through a few of the items required for developing software that you do not store in your Git repository. You can see that the items on this list are already impractical to store. Although it may be technically possible to store some of these items, the pain starts to become a losing trade-off in risk.

* Windows, the obvious one
* Visual Studio or VSCode, even if it’s possible to run it straight from disk
* Environment-specific data and configuration; this doesn’t belong to the software, it belongs to the environment
* Secrets; they are secret, so you shouldn’t know them anyway
* Large binary files that change very frequently, such as files from Autodesk products like AutoCAD and Revit

I want to address .NET Core specifically because the architecture of the .NET Framework has some fundamental differences here. With .NET Framework applications, the framework versions are installed on the computer as a component of the operating system itself. So it’s obvious that you don’t check it in. You check in only your libraries that your application depends on. If you need 7Zip or Log4Net, you obtain those libraries and check them into your Git repository because you depend on a particular version of them. With the advent of package managers, the debate has raged over when to not check in packages from npm or NuGet. That argument isn’t settled, but for .NET Framework applications, my advice has been to check in all your dependencies, including packages.

This fundamentally changes with the architecture of .NET Core. With .NET Core, the framework isn’t installed as a component of the operation system. The framework is delivered by NuGet to the computer running the build process. Furthermore, .NET Core libraries that are packaged as NuGet components have been elevated to framework status and are delivered in exactly the same way as .NET Core SDK components are. Therefore, my advice for .NET Core applications is to leave the defaults in place and do not commit the results of the dotnet.exe restore process into your Git repository. Under active development, this mix of SDK components and other NuGet packages will change quite a bit. Once the system reaches maturity and the rate of change slows, it may be appropriate to move and commit the **packages** folder in order to lock in that mix of dependencies given that package managers do not absolutely guarantee that the same mix of dependencies will be restored next month or next year. If you want to evaluate this for yourself and determine your risk tolerance, you can examine the packages easily by application by adding a Nuget.config file to your solution with the following configuration.

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<config>

<add key="globalPackagesFolder"

value=".\packages" />

</config>

</configuration>

Before you move on from Azure Repos, you must discuss the proper structure for a Git repository. Although you may make a different pattern work for you, the following generally works for most Visual Studio applications. **Figure 17** shows the top-level folder structure of an Azure Repos Git folder structure.

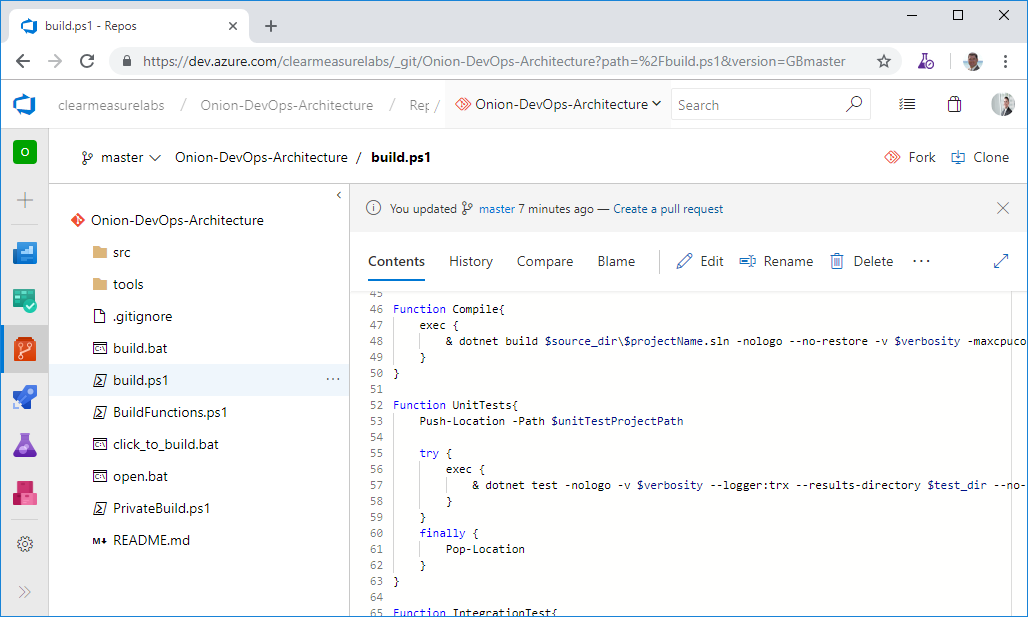


Figure 17: The top-level of a Git repository can be quite standard regardless of the type of Visual Studio software you’re developing.

You can see some directories and some script files at the top level. Notice that you don’t see a Visual Studio solution at the top level. That’s intentional. Let’s take the directories and files that you need in a properly organized Git repository.

* **/src/**: The application code is in this directory, beginning with the solution file. This is a common convention in multiple programming platforms
* **/tools/**: Any tools needed for the build process go in this directory. Common needs are 7Zip, Octo.exe, etc.
* **/build.ps1**: This is the private build script. Whether you name it this or not, you need your private build script in the top-level directory
* **/click\_to\_build.bat**: A mouse-friendly helper that adds an “& pause” to the build script so that the console window remains open for the examining of the build output
* **/open.bat**: A mouse-friendly helper that opens the Visual Studio solution via a double-click
* **/build/**: This directory is automatically created and destroyed by the build script. It shouldn’t be committed to source control

## Common Git mistakes

## Wrap up

1. (DevOps at Microsoft, n.d.) [↑](#footnote-ref-1)