# Chapter 5: Tracking Code

Version Control Systems (VCS) have fully matured over the last twenty years, and the Git version control 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. 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 as synonyms.

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 Windows Explorer “right click” integration. 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 can be deployed independently, 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 invent 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 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 its 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.

## What should be in your Git repository

There’s often a discussion about what to store in the application’s Git repository. The short answer is “store everything you can.” 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 that don’t come from a package manager

Given that there are some exceptions that cannot be committed to VCS, 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 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>

## The structure of the Git repository

We have discussed how to determine how many Git repositories we need for our system. Now we need to factor an individual application appropriately. Regardless of the architecture of the application, our relationship with the VCS comes from the Visual Studio solution. That solution can contain a large code base you call an application or a tiny code base that you might call a microservice. For the purposes of this guidance, these are the same.

If you are an experienced developer, you might be able to design a different structure that works for laying out a Git repository; however, if you would like prescriptive guidance in where everything should go within Git, consider the following advice. Shown within Azure Repos, the following figure depicts the top level of a well-factored Git repository.

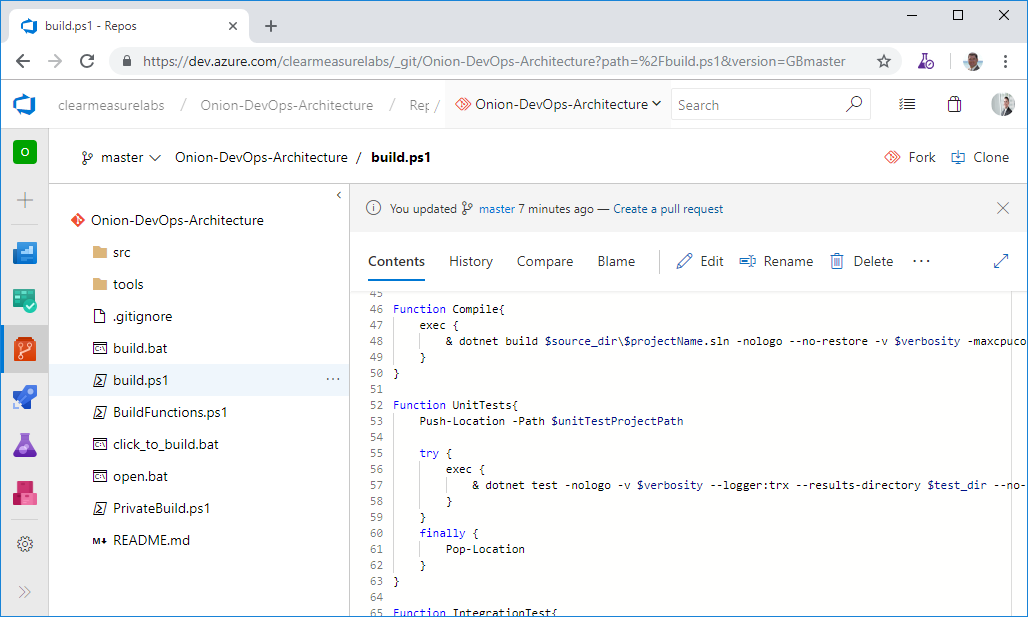


Figure: The top-level of a Git repository can be quite standard regardless of the type of Visual Studio software you’re developing.

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/keyboard-friendly helper that adds an “& pause” to the build script so that the console window remains open for examination of the build output
* **/open.bat**: A mouse/keyboard-friendly helper that opens the Visual Studio solution via a double-click or Enter
* **/build/**: This directory is automatically created and destroyed by the build script. It shouldn’t be committed to source control. This is the destination for test/publish output that is temporary in nature.

The above implementation is our example application for this book. More generally, the structure should be as shown to the left. This structure works for small microservices as well as very large applications with hundreds of screens and functions. Notice that this structure only goes down to the Visual Studio solution level and the folders within that solution. The separation within Visual Studio projects will vary.

Here are the rules:

1. The top of the Git repository will contain your private build assets. This includes the actual private build script, helper functions, and any assets/shortcuts in order to very run a private build on a local workstation.
2. The Git repository needs some basic documentation about what it contains and how to build what it contains. This is where the Readme.md comes in. You can use .txt or .docx, but Azure Repos, GitHub, and many other tools work well with the Markdown format and can show this file as a dashboard page.
3. The “build” directory is a temporary directory that will contain things generated by your build scripts. Call it what you like but this folder does not get committed to Git. Make sure to add it to your .gitignore file. Whether it is test output or artifacts created through running a “publish” command, use this folder for generated output both locally in the private build process and online in the continuous integration build.
4. The “tools” directory can contain any tools needed by the build process including a build framework if you choose to use one. With .Net Core, the process for building has been simplified. Many developers enjoy build tools like psake and others.
5. The “src” directory (source) contains the visual studio solution and all the code of the application. The Visual Studio solution file should be inside this folder
6. Each project/assembly within the Visual Studio solution will have its own folder. Take care to keep these folders at the same directory level as the Visual Studio solution.

Regardless of application type, nest your code in the /src folder. .Net Framework, .Net Core, Xamarin, TypeScript, etc. Reserve the top level of the repository for build assets.

## Choosing a branching pattern

If you are reading these chapters out of order, we discussed branching and merging while working on work items in chapter 4. Since branching is a means by which related code changes can be grouped, make sure to use branching for every change to the application. This rule of thumb can raise additional questions. Google, Facebook, and the Microsoft team responsible for the Azure DevOps family of products all use “Trunk-Based Development”[[2]](#footnote-2). You can also research other available branching strategies on Microsoft’s branching documentation[[3]](#footnote-3).

Use trunk-based development. In trunk-based development, the main branch (master) is build up like the trunk of a tree, thicker than branches and always getting longer. Branches are very short-lived, small, and exist to facilitate a reviewed pull request process. Aim to merge branches every day.

The sound of “trunk-based development” may cause you to think that adopting it means that you don’t branch. Not at all. Every branching strategy includes branches. Branches are a very effective way to group commits that are destined for the main code line (master in Git). As covered in chapter 4, a pull request is your method for reviewing with the team a group of changes to the product. When a pull request is approved, the branch in merged into master and deleted. Until the pull request for the branch is approved, these commits stay on the branch, guaranteed not to destabilize master. Through this process of review, the team together keeps the master branch stable. Here are some rules for working effectively with branches:

* Make them insanely short-lived: Don’t try to add large new capabilities on a single branch. If your user stories are small and discreet, you’ll be able to create/pull request/merge/delete a branch in a 24-hour period. Branches living into their second day are a red flag that should generate a team discussion.
* Tie them to a work item so that every change on the branch is related to the work item.
* Avoid large application refactorings while others have open branches – expect painful merge conflicts if you break this rule.
* Configure your DevOps pipeline to operate on all branches in order to get the benefit of the build and first deployment. If the first time your changes are deployed is after a pull request, you will be passing through more bugs onto master.

## Useful tips in Azure Repos configuration

Because Azure Repos is integrated within an Azure DevOps project, you need to be aware of how names and URLs are built. A project is meant to house multiple Git repositories. Because of this, the project name is included in the Git URL. Consider this same project name.

My Clunky Project

Azure Repos will produce a URL for initial cloning like this.

https://clearmeasurelabs@dev.azure.com/clearmeasurelabs/My%20Clunky%20Project/\_git/My%20Clunky%20Project

Notice that the spaces in the project name are transformed into %20 in the URL. While this can function just fine, there are some automation scenarios and tools some have run into that don’t properly handle the %20 in the URLs. If it is within your control when creating the project, simply avoid spaces in the project name. The public Azure DevOps project used for this book is a good example of this technique.

https://dev.azure.com/clearmeasurelabs/Onion-DevOps-Architecture

This yields a GIT URL as follows.

https://clearmeasurelabs@dev.azure.com/clearmeasurelabs/Onion-DevOps-Architecture/\_git/Onion-DevOps-Architecture

When you create your second Git repository within the same project, take care to avoid spaces so that you have a clean URL.

https://clearmeasurelabs@dev.azure.com/clearmeasurelabs/Onion-DevOps-Architecture/\_git/My-New-Repository

## How does GitHub fit in?

Microsoft acquired GitHub in 2018[[4]](#footnote-4). GitHub is intended to be the premiere offering for Git VCS hosting within the Microsoft ecosystem. At the time of this writing, Azure Repos is the fully integrated Git hosting offering. As such, it supports all the enterprise scenarios needed across customers including seamless identity management and logins with Office 365 and Active Directory accounts. GitHub has this integration on the roadmap and will become just as seamlessly integrated as Azure Repos, but as of this writing, that work has not been completed. If your code is already on GitHub, don’t move it. Keep it where it is and integrate the capabilities of the Azure DevOps family of products. If your code is already in Azure Repos, don’t move it.

## Wrap up

In this chapter we covered how to properly track your code when implementing a proper DevOps environment. We covered how to determine the size and scope of a Git repository and how many you should have. We discussed what should and should not be committed to your VCS. We analyzed the structure within the repository as well as how to think about and use branches. As you create or modernize source code, follow this guidance, and you won’t go wrong. As you encounter complex scenarios, plan adjustments while keeping the core principles in mind.

Now that we understand how to properly organize and track our code in our .Net DevOps environment, continue with chapter 6 where you will learn how to design and configure your build process.

# Bibliography

(n.d.). Retrieved from DevOps at Microsoft: https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/

*Adopt a Git branching strategy*. (n.d.). Retrieved 2 18, 2019, from https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops

Hammant, P. (n.d.). Retrieved from Trunk Based Development: https://trunkbaseddevelopment.com/

*Microsoft to acquire GitHub for $7.5 billion*. (n.d.). Retrieved 2 18, 2019, from https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/

1. (DevOps at Microsoft, n.d.) [↑](#footnote-ref-1)
2. (Hammant) [↑](#footnote-ref-2)
3. (Adopt a Git branching strategy, n.d.) [↑](#footnote-ref-3)
4. (Microsoft to acquire GitHub for $7.5 billion, n.d.) [↑](#footnote-ref-4)