**Scaling Git for Enterprise DevOps**

In this video, you will learn how to structure your Git repository and how to use branching workflows. You also learn about pull requests, GitHooks, internal open source, Git versioning, and file handling.

**Mono vs Multi Repos**  
A repository is simply a place where the history of your work is stored. It often lives in a .git subdirectory of your working copy. So what's the best way to organize your code repository? Software development teams start off with the best intentions to keep clear separation of concerns in both the software being development and their code repositories. However, overtime it is not uncommon for the code repositories to be bloated with unrelated code and artifacts.

**Mono-Repo or Multi-Repo**There are two philosophies on how to organize your repos… Mono-Repo or Multi-Repo. Mono-repos are a source control pattern where all of the source code is kept in a single repository. This makes it super simple to get all of your employees access to everything in one shot. Just clone it down, and done. Multi-repos on the other hand, refers to organizing your projects each into their own separate repositories. The fundamental difference between the mono-repo and multi-repo philosophies boils down to a difference about what will allow teams working together on a system to go fastest. The multi-repo view, in extreme form, is that if you let every sub-team live in its own repo, they have the flexibility to work in their area however they want, using whatever libraries, tools, development workflow, etc. will maximize their productivity. The cost, obviously, is that anything not developed within a given repo has to be consumed as if it was a third-party library or service, even if it was written by the person sitting one desk over. If you find a bug in, say, a library you use, you have to fix it in the appropriate repo, get a new artifact published, and then go back to your own repo to make the change to your code. In the other repo you have to deal with not only a different code base but potentially with a different libraries and tools or even a different workflow. Or maybe you just have to ask someone who owns that system to make the change for you and wait for them to get around to it. The mono-repo view, on the other hand, is that that friction, especially when dealing with more complicated dependency graphs, is much more costly than multi-repo advocates recognize and that the productivity gains to be had by letting different teams go their own way aren’t really all that significant: While it may be the case that some teams will find a locally optimal way of working, it is also likely that their gains will be offset by other teams choosing a sub-optimal way of working. By putting all your eggs in the one basket of the mono-repo you can then afford to invest in watching that basket carefully. Clearly the friction of having to make changes in other repos or, worse, having to wait for other teams to make changes for you, is largely avoided in a mono-repo because anyone can (and is in fact encouraged) to change anything. If you find a bug in a library, you can fix it and get on with your life with no more friction than if you had found a bug in your own code.

**GitHooks**Why Care about GitHooks  
Continuous delivery demands a significant level of automation… You can’t be continuously delivering if you don’t have a quality codebase. This is where git fares so well, it gives you the ability to automate most of the checks in your code base even before committing the code into you local repository let alone the remote.

**GitHooks**  
GitHooks are a mechanism that allows arbitrary code to be run before, or after, certain Git lifecycle events occur. For example, one could have a hook into the commit-msg event to validate that the commit message structure follows the recommended format. The hooks can be any sort of executable code, including shell, PowerShell, Python, or any other scripts. Or they may be a binary executable. Anything goes! The only criteria is that hooks must be stored in the .git/hooks folder in the repo root, and that they must be named to match the corresponding events (as of Git 2.x): applypatch-msg pre-applypatch post-applypatch pre-commit prepare-commit-msg commit-msg post-commit pre-rebase post-checkout post-merge pre-receive update post-receive post-update pre-auto-gc post-rewrite pre-push

**GitVersion**

The goal of adding a new dependency to your code is to avoid reinventing the wheel. If there is code already available that does what you want, normally you would prefer to re-use it (there are exceptions, though) rather than investing your time in re-writing new code to solve the same task. On the other hand, once your software project has matured over time and is ready for production it may end up being a dependency for other software projects as well. Initially, re-using software packages as much as possible looks like the best way forward to save time and effort in the long run; and it is indeed true. However, creating dependencies with other software packages also brings its own disadvantages, specially when the number of dependencies becomes larger and larger. Grouping off-the-shelf functions into software packages and defining dependencies between them has been traditionally at the core of software development. If you're building libraries, products or any other software system, versioning is usually a pretty big deal. It's the only way to determine what version of that library, product or system you're looking at. Before an organization settles on a versioning strategy, many discussions have been held on what constitutes a major release versus a minor release, how to version the component from a marketing perspective, and how to deal with bug fixes. In addition to that, if that software system involves a library or framework, or just component, then you'd be pretty interested to know when an update to that component involves breaking changes.

From the version of an app you should, for instance, be able to guess whether it’s a bugfix or it has introduced new features. Without properly set version codes you could totally lose your compass on what’s happening. So, when you set the version of your software always be careful and remember that, given a major.minor.patch version number, you should increment: ● patch when you make bug fixes ● minor when you introduce new features ● major when you make changes that break backwards-compatibility or completely change the user experience

That being said, in the daily practice, if you want to start with a development phase before going to production the simplest thing could be to start with a 0.1.0 version number and increment minors at each new development release. Also keep in mind that if you’re working on a production project it’s always a good idea to start with a 1.0.0 version code.

**Public Projects**

Azure DevOps offers a suite of DevOps capabilities to developers including Source control, Agile planning, Build, Release, Test and more. But to use Azure DevOps features require the user to first login using a Microsoft Account. This however blocks a lot of interesting scenarios where you would want to share your code and artifacts publically or simply provide a wiki library or build status page for unauthenticated users…. With public projects, users will be able to mark a Azure DevOps Team Project as public. This will enable anonymous users to be able to view the contents of that project in a read-only state enabling collaboration with anonymous (un-authenticated) users that wasn’t possible before. Anonymous users will largely see the same views as authenticated users, with non-public functionality such as settings, or actions (such as queue build) hidden or disabled.

**Public versus private projects**

Projects in Azure DevOps provide a repository for source code and a place for a group of developers and teams to plan, track progress, and collaborate on building software solutions. One or more projects can be defined within an organization in Azure DevOps. Users that aren't signed into the service have read-only access to public projects on Azure DevOps. Private projects, on the other hand, require users to be granted access to the project and signed in to access the services. **Supported services**

Non-members of a public project will have read-only access to a limited set of services, specifically: ● Browse the code base, download code, view commits, branches, and pull requests ● View and filter work items ● View a project page or dashboard ● View the project Wiki ● Perform semantic search of the code or work items