git Version Control

Hany Greiss <Hany.Greiss@DXC.com>

Version 1.0, 01/26/2022: DRAFT

Table of Contents

Figures and Tables

Summary

The following is a high-level overview of a team development process based on **git**. The overview is based on accepted industry best practices. The workflow is not specific to Microsoft Dynamics 365 and it can be applied whenever version control and tracking are required. It can be applied to versioning a web site's contents, source code or any other file types. It can be used by a single developer tracking his or her changes or by a team collaborating on a project.

git Architecture

Architecturally, the concept essential to understanding **git** is that there is *no* central repository. We do however, designate and use a repository as being the *origin*. But, this is by convention and not enforced. The following diagram shows 5 repositories with each one equal to one another. The repository named *origin* is designated as the **central** repository, by convention. Theoretically, the repository named *clair* could be used as the origin. Team members can just as easily fetch from each others repositories.

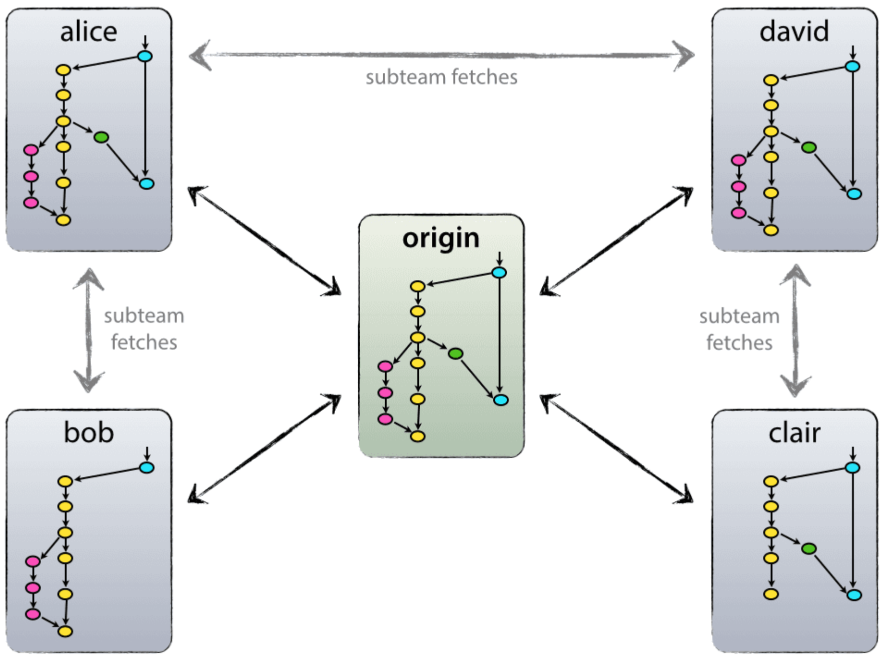

Figure 1. git Central Repository

Industry Best Practices

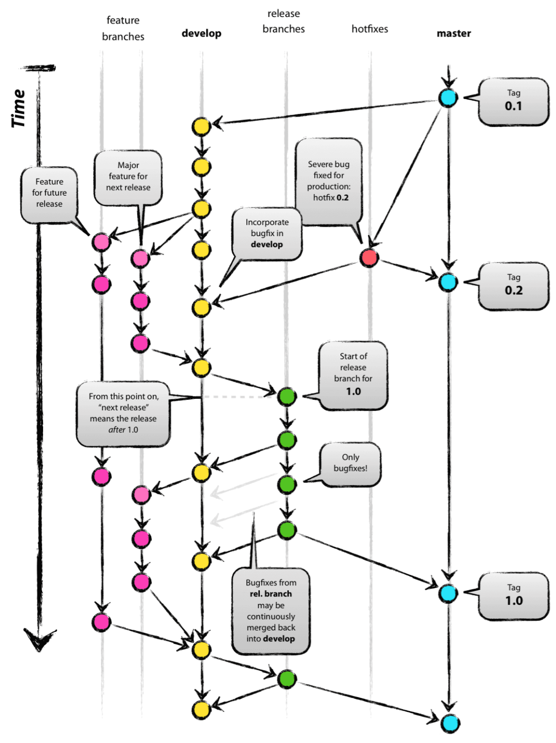
feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Figure 2. git Model

Using **git**, branching and merging operations are simple and safe. This is in contrast with other Version Control Systems (VCS) such as CVS and Subversion. The recommended approaches of managing source code using **git** are different because of these differences.

This section provides a high-level outline of some the best practices using **git** that are currently adopted and supported by various tools, such as **GitHub Flow**.

With **git**, probably the most significant difference, as compared with other VCS, is the distributed architecture/model. Technically, there is no central source repository, although one is typically designated as the *origin*. In actual fact, it is just another repository as are each of the developer's local repositories.

In the **git** team development model, developers will typically pull and push from/to the origin, whereas in fact, they could just as easily pull and push from any other developer's repository. There is nothing inherently different in a technical sense.

The origin repo is different in the sense that there is no working copy concept. Updates to the origin repo are merged in as updates are pushed. There are many ways that developers can cooperatively work together using Git. The collaboration process is known as a workflow and this section describes a well-accepted model. It starts with a key structural element, the branches.

Main Branches

The proposed workflow is centered on two branches, develop and master, that have infinite lifespans. The *origin/master* branches **HEAD** always points to the production-ready state. The *origin/develop* branch is where the next release is being worked on.

Developers working on the next release will merge in their changes on the origin/develop branch. Nightly builds are based on this branch. Production releases are merged in from develop onto master.

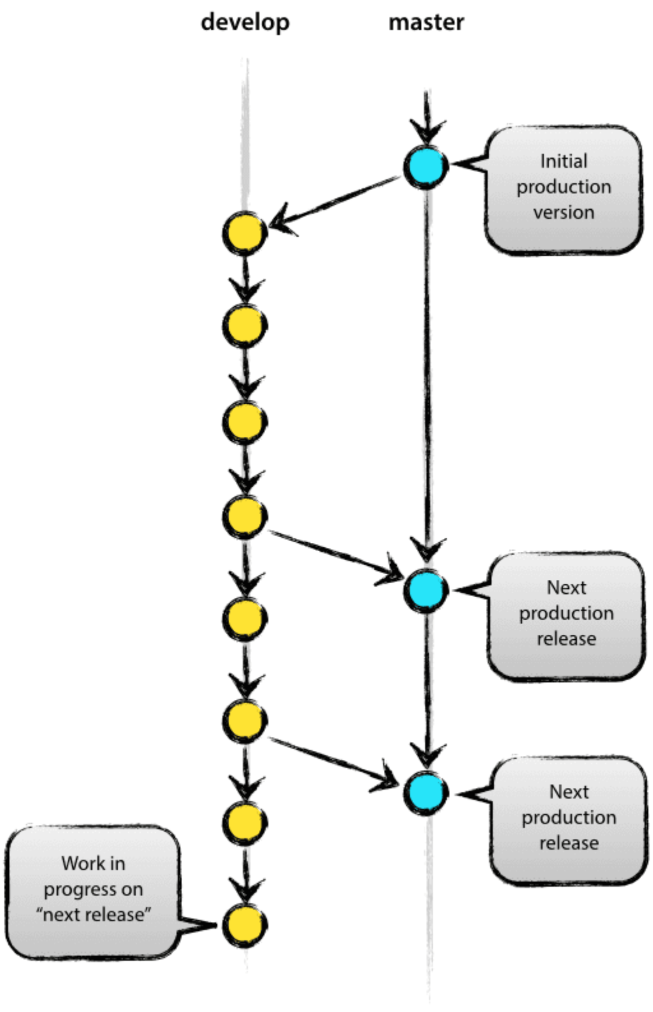
develop
master
Initial production version
Next production release
Next production release
Work in progress on "next release"

Figure 3. Main Branches

Working Branches

The day to day workflows are centered on the typical activities involving; adding features, creating releases and applying hotfixes.

- Features
  - A local feature branch is created, based on the local develop branch, and changes merged back into develop. The updates are pushed to the tracked origin/develop branch.
- Releases
  - A local release branch is created, based on the local develop branch, and changes merged back into develop. The updates are pushed to the tracked origin/develop and origin/master branches.
- Hotfixes
  - A local release branch is created, based on the local master branch, and changes merged back into master. The updates are pushed to the tracked origin/develop and origin/master branches.

The workflow solves the most common development use cases. Extensions to Git, such as **GitHub Flow**, provide direct support of this model. The next section deals with the recommendations specific to this project.

Proposed Workflow Summary

Structurally, the main repository will contain the two main branches, develop and master. The first step in the process is to clone the remote repo:

Clone the Repo

```
cd ~\sources\repos ①
git clone URL ②
```

① Or any folder where the repo will stored
② The *URL* will be provided by the provider and usually with credentials

Dynamics 365 Solutions

Configuration and customization of a Dynamics 365 application are primarily done using the user interface. Out of the box, there are no inherent tools to orchestrate the simultaneous activities of multiple developers working on the same organization. Inherently, solutions contain a collection of components. The components are described using XML and contain the configurations and customization that make up a portion of the application. More than one solution may exist in an organization. Collectively, they represent the application on that organization.

Two or more developers working on the same organization on their respective solutions may still conflict with one another. That is because a solution component referenced from different solutions refers to the same component.

To mitigate these collisions, each developer should work on his or her individual components. Configuration and customizations are isolated to their respective components. This model only gets us partially towards a working collaborative model. Ultimately, the individual contributions are merged in with the changes made by the team.

To address this version control, as described in this document, is used to manage this part of the process. Although Dynamics 365 may appear different to traditional programming where version control has been used for years, it is quite suitable nonetheless.

Using tools provided by Microsoft that are part of the **SDK**, an exported solution can be extracted into individual components, as **XML** fragments. The same tool can be used to pack these individual components into a solution and then imported into an organization. Thus, a working versioning strategy can be applied:

- Each developer works on their own components within a designated solution.
  - For example, the solution may be named for the current sprint.
- Daily, a designated developer gets the working copy of the solution from source control.
  - The solution in source control is stored using the individual component structure.
  - The developer packs the components into a solution which is then imported into the organization.
- Each developer creates a new branch, configures and customizes the application on Dynamics 365.

  - This could be for a feature or topic, a hotfix or an upcoming release.
- The developer then exports the same solution from Dynamics 365 and extracts the components into the working copy.
- These will appear as component changes, new or modified, from the local source control perspective.
- These changes can be committed locally along with a message describing the changes.
- Before pushing and merging the changes to the remote server, the developer fist merges in any changes that have occurred since the solution components were pulled down.
  - It is possible that merging these changes may overwrite one or more of the same components that have been updated by the developer.
  - In the event of this merge conflict, coordination and review are necessary to ensure that the changes are preserved. Git will try to merge the changes, but it is not always possible and manual oversight may be needed.
  - Once the merge conflicts, if any, have been resolved the developer can push the updates to the server.
- The developer then issues a pull request so that their changes can be merged to the develop branch.
- The process is repeated as work on the next feature or topic resumes.

Working with Solutions

In the working copy, locate the expanded solution folder where each the sections of the solution is expanded, e.g. Entities, Option Sets, and so forth exist. Using the *Solution Packager* tool from the **SDK** pack the files to create a solution. The solution is temporary and only used to import into your organization. It can be named anything you want.

Pack the Solution

```
SolutionPackager.exe /action:pack /zipfile:mySolution.zip /map:Mapping.xml ①
```

① Tool is part of the Dynamics 365 Tools

There are dependent solutions required to be built, e.g. plugins, workflow activities, etc. The map option is used because some of the solution components, e.g. Plugin Assemblies, Web Resources, etc., are built in locations outside of the current solution build area. The map file is used to specify the locations of each of these individual build artifacts and to specify where they were expected so that the PluginAssemblies folder can be refreshed. These individual solutions are built and the plugins are updated accordingly.

Import the *mySolution.zip* into your development organization. At this point it is just pure Dynamics 365 configuration and customization activities.

Feature Branch

A developer would typically work on one feature which will get merged into the working develop branch and then merged up to origin/develop. This is as described earlier. These steps will be repeated throughout each day during the sprint. Patch solutions are no longer required. The solution packed and imported from source control is the only solution that should be used.

git best practices recommend to commit changes often and to work on one feature at a time. If several unrelated changes are committed together, rolling back changes where part of the commit is still required and the buggy portion should be removed becomes problematic. The changes made on CRM are merged onto the local develop branch by first exporting the solution containing the changes and the extracting the solution over the working copy using the extract action of solution packager.

Extract the Solution

```
SolutionPackager.exe /action:extract /zipfile:mySolution.zip ①
git status ②
git commit –m "commit message" ③
git fetch   ④
# We then push changes to the remote – updating the remote develop branch
git push origin/develop ⑤
```

① Tool is part of the Dynamics 365 Tools

② Shows the changes

③ Changes are committed to the local repo

④ Do not just push changes because changes may have been checked in that we do not have yet. Check for merge conflict and fix any conflicts before proceeding

⑤ We would actually issue a *Pull Request* but we haven't covered that yet!

Tags

Tags lets us mark points in the repositories history typically used to mark release points. There are two types of tags available:

- *Lightweight*
    - Just a pointer to a specific commit
- *Annotated*
    - Full-fledged **git** object that contains detailed information on the tag:
        - Taggers name
        - Taggers email
        - Tag date
        - Tag message

Since annotated tags provide all of this information, it is almost always preferrable over lightweight tags. Here are a few examples:

git tag commands

```
$ git tag ①
1.0
1.1

$ git tag -a 1.2 -m "added git tag commands to documentation" ②

$ git tag ③
1.0
1.1
1.2

$ git push origin 1.2 ④
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 176 bytes | 88.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/hanyg/gitdemo.git
 * [new tag]         1.2 -> 1.2
```

① List tags on this branch
② create a tag
③ list the new tag just added
④ push the tags to the remote

Hotfixes

Applying a hotfix is similar to the normal feature/topic branch workflow previously described. The main difference is that because it is urgent to update the production release, the hotfix is branched off of origin/master. Work on the hotfix branch continues as usual and the changes are merged into the master and develop branches. The remote server is updated as usual.

Use a tag to mark the hotfix.

Releases

Once the develop branch as reached the point where a release is ready, we create a release branch. All the final bits and pieces of the release can be applied at this point. The release branch is created off the develop branch and once completed, merged into master and develop. The remote server is updated as usual.

Use a tag to mark the release.

GitHub Flow

GitHub flow is a lightweight, branch-based workflow. The **GitHub flow** is useful for everyone, not just developers. GitHub Flow

Resources

You can get **git** from the main site git. A really useful resource is the online Pro git Book.

Another helpful resource is the git Cheat Sheet

The workflow and concepts desribed in this document are based on the blog A successful Git branching model. Although more that ten years old, but recently updated, the core concepts remain tried and true. A worthwhile read!

The XrmToolbox is an indispendible tool. The Albanian Solution Packager simplifies the solution packing and unpacking operations described in the document.

Dynamics 365 Tools

The following *powershell* script will download the rquired tools and more from **NuGet**.

Clone the Repo

```
$sourceNugetExe = "https://dist.nuget.org/win-x86-commandline/latest/nuget.exe" ①
$targetNugetExe = ".\nuget.exe"
Remove-Item .\Tools -Force -Recurse -ErrorAction Ignore ②
Invoke-WebRequest $sourceNugetExe -OutFile $targetNugetExe
Set-Alias nuget $targetNugetExe -Scope Global -Verbose

##
##Download Plugin Registration Tool
##
./nuget install Microsoft.CrmSdk.XrmTooling.PluginRegistrationTool -O .\Tools ③
md .\Tools\PluginRegistration
$prtFolder = Get-ChildItem ./Tools | Where-Object {$_.Name -match 'Microsoft.CrmSdk.XrmTooling.PluginRegistrationTool.'}
move .\Tools\$prtFolder\tools\*.* .\Tools\PluginRegistration
Remove-Item .\Tools\$prtFolder -Force -Recurse

##
##Download CoreTools
##
./nuget install Microsoft.CrmSdk.CoreTools -O .\Tools ④
md .\Tools\CoreTools
$coreToolsFolder = Get-ChildItem ./Tools | Where-Object {$_.Name -match 'Microsoft.CrmSdk.CoreTools.'}
move .\Tools\$coreToolsFolder\content\bin\coretools\*.* .\Tools\CoreTools
Remove-Item .\Tools\$coreToolsFolder -Force -Recurse

##
##Download Configuration Migration
##
./nuget install Microsoft.CrmSdk.XrmTooling.ConfigurationMigration.Wpf -O .\Tools ⑤
```

```
md .\Tools\ConfigurationMigration
$configMigFolder = Get-ChildItem ./Tools | Where-Object {$_.Name -match 'Microsoft.CrmSdk.XrmTooling.ConfigurationMigration.Wpf.'}
move .\Tools\$configMigFolder\tools\*.* .\Tools\ConfigurationMigration
Remove-Item .\Tools\$configMigFolder -Force -Recurse

##
##Download Package Deployer
##
./nuget install Microsoft.CrmSdk.XrmTooling.PackageDeployment.WPF -O .\Tools ⑥
md .\Tools\PackageDeployment
$pdFolder = Get-ChildItem ./Tools | Where-Object {$_.Name -match 'Microsoft.CrmSdk.XrmTooling.PackageDeployment.Wpf.'}
move .\Tools\$pdFolder\tools\*.* .\Tools\PackageDeployment
Remove-Item .\Tools\$pdFolder -Force -Recurse

##
##Remove NuGet.exe
##
Remove-Item nuget.exe
```

① The *nuget* command that will be used to fetch the required tools

② Cleanup previous downloads

③ Plugin Registration Tool

④ Core Tools. Includes the required *SolutionPackager* tool decsribed in this document.

⑤ Configuration Data Migration Tool

⑥ Package Deployment Tool