|  |
| --- |
| ©Bluesmith Information Systems Ltd Commercial in Confidence |
| Introduction to Git  **Reference**: BP-11038-ALL-001  Version: Issue 1.2  Date: 10/05/2017  For:  Freemans Grattan Holdings PLC |
| [www.bluesmithIS.com](http://www.bluesmithIS.com) |

Document Control

|  |  |  |  |
| --- | --- | --- | --- |
| Issued Version Approval | | | |
| Author: | Afzul Mohmud | | |
| Bluesmith Approval: | Tom Robinson | Signature/Approval: |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Change History | | | | | |
| Version: | Issue 1.0 | Date: | 26/10/2015 | Change Author: | Afzul Mohmud |
| Issued following minor changes. | | | | | |
| Version: | Draft A | Date: | 11/01/2017 | Change Author: | Phil Tillotson |
| Modified for Freemans Grattan Holdings. | | | | | |
| Version: | Issue 1.1 | Date: | 13/01/2017 | Change Author: | Phil Tillotson |
| Modified for Freemans Grattan Holdings change Git exercises to make more relevant. | | | | | |
| Version: | Issue 1.2 | Date: | 10/05/2017 | Change Author: | Ian Long |
| Issued following internal review. | | | | | |

Table of Contents

[Document Control 2](#_Toc487090007)

[1. Introduction 5](#_Toc487090008)

[1.1. Git Source Control 5](#_Toc487090009)

[2. Git and Version Control Overview 6](#_Toc487090010)

[2.1. What is version control? 6](#_Toc487090011)

[2.2. Basic version control concepts 6](#_Toc487090012)

[2.3. Centralised version control system (CVCS) 7](#_Toc487090013)

[2.4. Distributed version control system (DVCS) 8](#_Toc487090014)

[2.5. Centralised vs. Distributed VCS 9](#_Toc487090015)

[2.6. Snapshots Rather Than Differences 11](#_Toc487090016)

[2.7. Local Operations 12](#_Toc487090017)

[2.8. Push and Pull 12](#_Toc487090018)

[2.9. Git Integrity 13](#_Toc487090019)

[2.10. Working, Staging, Local and Remote 13](#_Toc487090020)

[2.11. Stage, Commit and Push 14](#_Toc487090021)

[2.12. Git Extensions 15](#_Toc487090022)

[3. Common Git Operations 16](#_Toc487090023)

[3.1.1. Cloning a Git repository 16](#_Toc487090024)

[3.1.2. Browsing the Repository 16](#_Toc487090025)

[3.1.3. Modifying Files and Staging 16](#_Toc487090026)

[3.1.4. Committing Changes 17](#_Toc487090027)

[3.1.5. Amending a Commit 17](#_Toc487090028)

[3.1.6. Undoing Changes 17](#_Toc487090029)

[3.1.7. Deleting and Moving Files 18](#_Toc487090030)

[3.1.8. Working with Remote Repositories 18](#_Toc487090031)

[3.1.9. Pulling Changes 18](#_Toc487090032)

[3.1.10. Pushing Changes 19](#_Toc487090033)

[3.1.11. Stashing Changes 20](#_Toc487090034)

[3.1.12. Creating Tags 20](#_Toc487090035)

[3.1.13. Working with Branches 20](#_Toc487090036)

[3.1.14. Checking out branches 22](#_Toc487090037)

[3.1.15. Ignoring Files 22](#_Toc487090038)

[3.1.16. Integration with Other Tools 22](#_Toc487090039)

[3.1.17. Git Command Line 23](#_Toc487090040)

[4. Git Resources 24](#_Toc487090041)

[5. Git Exercises 25](#_Toc487090042)

[5.1. Create Repository 25](#_Toc487090043)

[5.2. Install Git Extensions 25](#_Toc487090044)

[5.3. Clone Repository 27](#_Toc487090045)

[5.4. Create, Commit and Push 27](#_Toc487090046)

[5.5. Tag It 28](#_Toc487090047)

[5.6. Branch Out 28](#_Toc487090048)

[5.7. Merge Changes 29](#_Toc487090049)

[5.8. Release 1.0 30](#_Toc487090050)

[5.9. Add “Person\_tables\_and\_views” Feature 31](#_Toc487090051)

[5.10. Stop – Hotfix Time 31](#_Toc487090052)

[5.11. Back To The Feature 32](#_Toc487090053)

[5.12. More New Features 32](#_Toc487090054)

[5.13. Browsing File History 33](#_Toc487090055)

[5.14. Additional Tasks 33](#_Toc487090056)

[5.15. Couple of useful commands 33](#_Toc487090057)

1. Introduction
   1. Git Source Control

This training module is designed as a self-teach exercise, enabling the student to gain familiarisation with the Git version control system and Git Extensions, which is a Git client for Windows.

This module aims to introduce the basic Git features. There are very few practical demonstrations or walkthroughs in this document – the trainee is expected to learn much of the practical side by doing the exercises at the end.

It is outside the scope of this module to describe all the different Git features. It is up to the trainee to supplement the learning from this module with additional reading and experimentation to build on their understanding. The Git Resources chapter of this document (section 4) provides several resources that you will find helpful and should supplement some of the information in this document.

While you are working through this training module you are strongly advised to refer to the following book about Git for further information and context: <http://git-scm.com/book/>. Although the book is primarily based on command line Git (whereas this training module is focussed on the Git Extensions tool), much of the material in this module is based on information from this book.

It is strongly recommended you read the Source Control Guidelines document before you work through this training document. The Source Control Guidelines document provides general guidance on how source control should be used and provides useful background knowledge that will help you through this training module.

You are also advised to read the Review Procedure document as it contains guidance about how to use Git “Merge Requests” for reviews. The exercises in this document will ask you to create merge requests, so it is important you are aware of how merge requests are used for reviews.

1. Git and Version Control Overview
   1. What is version control?

Version control is a system that allows you to record changes to one or more files over a period of time. This allows you to keep a historical record of all versions of a file and allows you to recall specific versions later if necessary. As part of your training you will be using version control for source code files, however in theory, you could use version control with any type of file on your computer.

Version control can be very useful, particularly with source code, because it allows you to revert files back to their previous state, you can compare changes to one or more files over time and you can also keep track of who changed files and when, which can be handy if someone has introduced a bug or an issue into your application.

Version control becomes even more important when you have multiple people all working on the same files, sometimes from different locations. How do you ensure people can work on the files at the same time? How do you ensure people don’t edit the same files and overwrite each other’s changes? This is where version control can help.

* 1. Basic version control concepts

This section describes the basic concepts and terminology you need to be aware of when working with a version control system (note that the content in this section is largely based on this webpage: <http://guides.beanstalkapp.com/version-control/intro-to-version-control.html>). These concepts and terms will be used and referred to throughout the rest of this training document.

A “Version Control System” (VCS) is primarily used for tracking changes to one or more directories and files over time. The set of directories and files placed under version control is commonly referred to as the **repository**. A user’s local copy of the repository is often referred to as a **checkout** or **clone** of the repository.

When you work with files in your repository (i.e. files that are under version control) any changes you make will be tracked as you make them. A version control system usually allows you to submit a set of changes to the repository (rather than having to submit each change individually). This set of changes is known as a **commit**.

When a commit is made to a repository, the version control system records the changes as a **changeset** and allocates a unique revision ID. Different version control systems use different forms of revision ID, but the aim is the same – the revision ID allows you to refer to a specific set of changes. A changeset is more than just a record of the files affected by the change, it also includes details of the person who made the changes, when they were made and usually an associated comment.

When multiple people are using a version control system it is important that each user has the latest version of the repository and has visibility of other people’s changes (this reduces the chance of people making conflicting changes to the same files). Getting the latest changes from the repository is often referred to as an **update** or **pull** operation.

A **conflict** occurs when two people have made similar changes to the same file and the version control system can’t determine which set of changes is the authoritative set. When a conflict occurs, a user must **merge** the two sets of changes to ensure changes are not undone. This is an operation that needs to be performed manually.

It is often necessary to view what has changed recently in a file e.g. to track when a bug was introduced into code. As version control systems record each commit as a changeset, they typically provide a facility to view the differences between two revisions of a file. Viewing the differences between two revisions is known as a **diff**.

If you want to experiment with the files in the repository, you might want to create a separate **branch** for this experimentation, rather than changing the main set of files (the main set of files is often referred to as **trunk** or **master**). A branch is like a snapshot or copy of the repository that you can modify in parallel without affecting the main set of files. You can commit changes to your branch while others can work with the main set of files (i.e. trunk/master) without any of the changes affecting either side.

When you’re happy with the changes in your branch, you may want to make them part of the master/trunk branch (i.e. the main set of files). This can be achieved by **merging** the changes. When a merge is performed, the version control system examines the changesets in each branch and tries to merge them together. If the version control system cannot automatically apply changes when performing a merge, then you have a conflict and this needs to be resolved manually by examining the changes in detail.

This section has covered the key concepts of version control at a high-level and introduced some of the main terms that you need to be aware of. Most of these concepts are covered in more detail in later sections of this document.

The next few sections of this document describe the difference between centralised and distributed version control.

* 1. Centralised version control system (CVCS)

In a Centralised VCS you have a central server that contains all the versioned files. Each user that wants to work on a file must take a copy of the files (known as a “checkout”) from the central server. When a user has finished editing a file they can “commit” their changes to the central server. This creates a new revision of the file on the central server and makes the latest changes available to other users who access the central server.

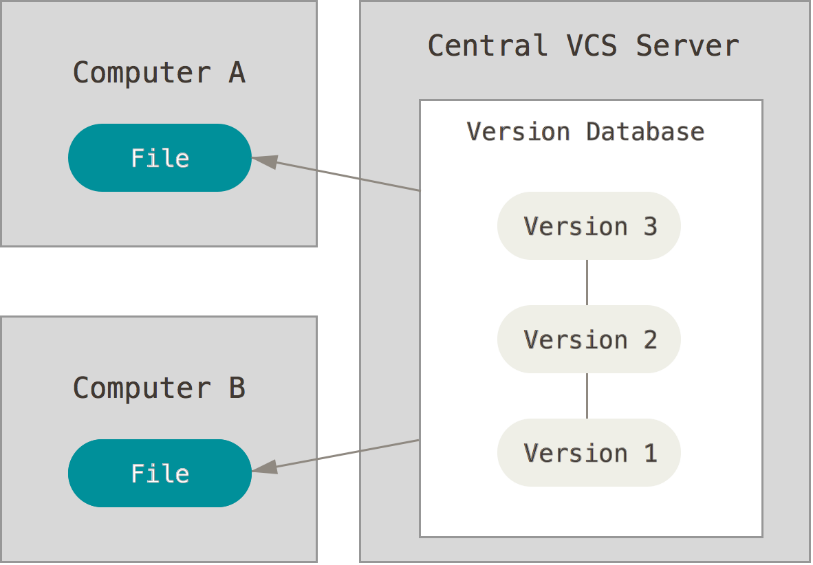


Figure 2: Visualisation of a centralised VCS

A Centralised VCS allows multiple people to collaborate on the files and also ensures there is a central copy of the files on the server (rather than just a local copy on an individual computer).

A Centralised VCS isn’t perfect, primarily because there is a single point of failure – the central server. If the central server is unavailable then it means people can only work on local copies of their files.

Centralised VCS is however a very easy concept to grasp, because most people are familiar with the client-server model. Examples of the Centralised VCS model are Visual Source Safe, TFS, CVS and Subversion.

* 1. Distributed version control system (DVCS)

More recently, Distributed VCS has become more popular. In a Distributed VCS users don’t just checkout the latest snapshots of files, but instead they checkout an entire copy of the repository – including all the metadata and the full history of the project. This means that most operations are now performed on the local copy of the repository rather than the central remote repository (as would be the case with Centralised VCS).

This usually means interaction with the repository is extremely fast because everything is done on the local hard drive and as no network access is required users can also work offline. Any operations that involve synchronisation with a remote repository would still have to be performed over the network to a remote server.

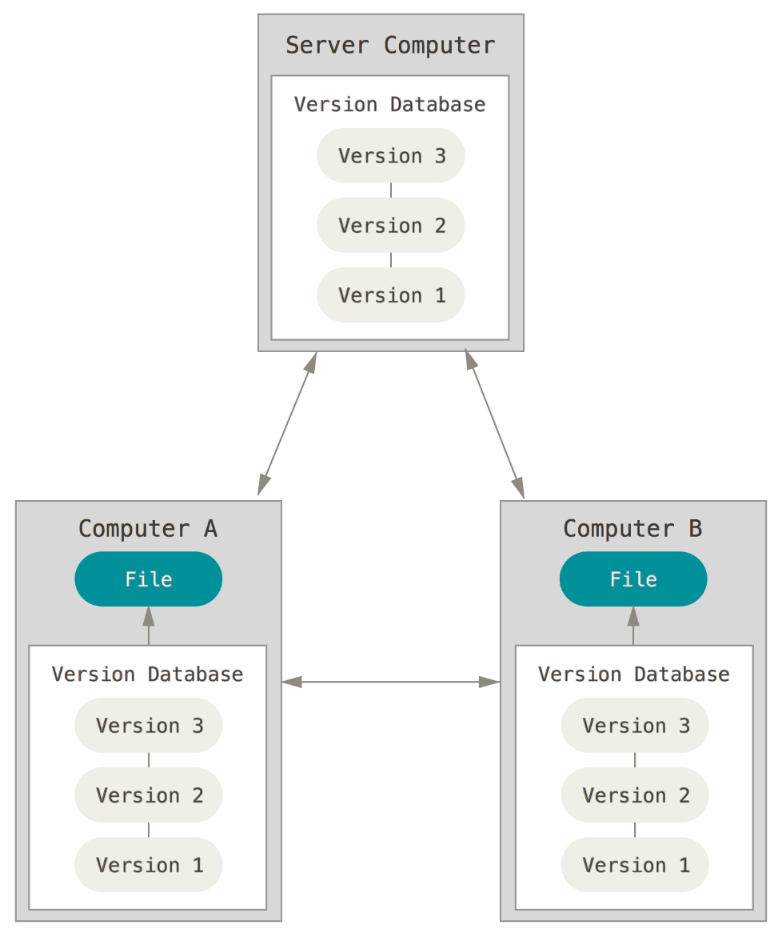


Figure 3: Visualisation of a distributed VCS

A Distributed VCS does not need to have a central remote repository. Each user could have their own local clone of the repository and they may share changes with each other without going via a central repository. In this scenario, each user has their own local repository and the other user repositories are treated as remote repositories that they can synchronise with. In reality however, most implementations of a Distributed VCS will still rely on a central server to act as the authoritative repository. The use of a central repository becomes optional with a Distributed VCS (rather than being required as with a Centralised VCS). Even with a central repository, users can still share changes with each other without going via the central repository if necessary.

The Centralised VCS model is good for backup, undo and synchronisation of changes. However it is not always intuitive for branching and merging. It is possible to branch and merge in a Centralised VCS but it is made much easier in a Distributed VCS (mainly because the distributed model is built around sharing changes and the model itself is reliant on the creation and merging of branches to work).

* 1. Centralised vs. Distributed VCS

If you’ve previously used a Centralised VCS (CVCS), such as Visual SourceSafe, and know how it works, then you’ll know that a CVCS has a single central server that contains all the versioned files. Clients can checkout files from the central server and then save changes back to the central server. Git, however, is a Distributed VCS (DVCS), where clients don’t just check out a snapshot of the repository, they mirror the entire repository.

The diagrams below (available at [betterexplained.com](http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/)) show the key difference between CVCS and DVCS.

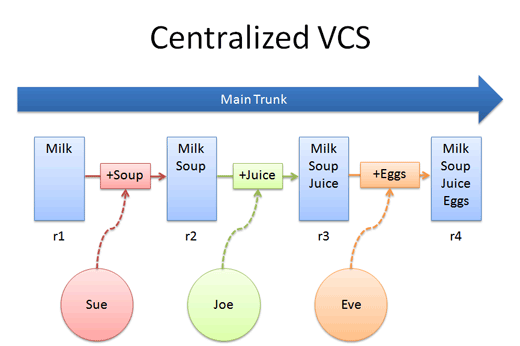


Figure 4: Centralised Version Control

The diagram above shows the typical actions performed on a CVCS. With a CVCS everyone syncs their local files with and checks their updated files into the central repository (commonly referred to as the “trunk” or the “main trunk”). Each change to a file must go into the central repository before it can be seen by other users of the CVCS. The diagram above shows a simple file, with Sue adding Soup, then Joe adding Juice and finally Eve adding Eggs. Each addition to the file results in a new revision of the file.

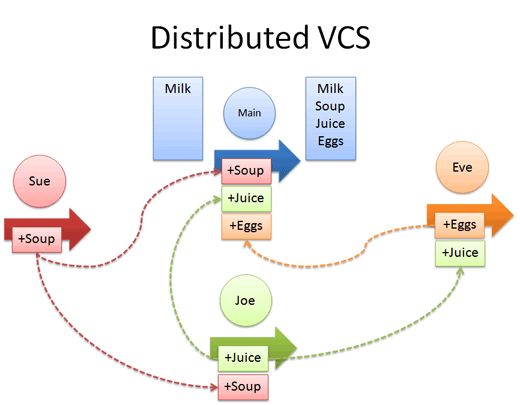


Figure 5: Distributed Version Control

With a DVCS each user has their own local repository and users can share changes between themselves without having to go via the central repository (indicated in the diagram above as “Main”). The arrows in the diagram above show changes being pushed between user repositories e.g. +Soup is pushed directly from Sue to Joe. Users can also push changes to the “Main” central repository if necessary e.g. +Soup is also pushed to the central repository.

The diagram below shows a more realistic and slightly more complex example of a DVCS. The graph/tree shown within each repository (e.g. alice, david, origin etc.) represents an instance of an entire repository and the history of all commits. In this example, origin is regarded as the central repository. As the diagram below shows, developers can push (commit) changes to and pull (retrieve) changes from origin, but they can also push changes to and pull changes from other user’s repositories.

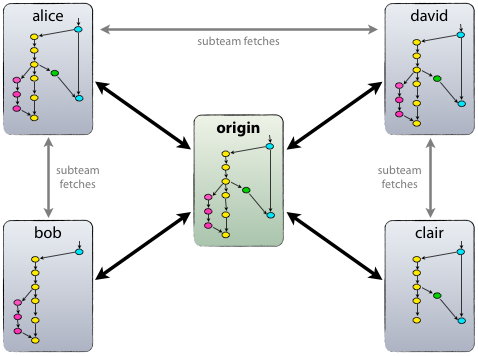


Figure 6: Distributed Version Control System

As covered briefly in a previous section, the main advantage of a DVCS is that each developer has the entire repository locally. This means they can work offline if necessary and they can commit changes locally, which encourages more frequent commits. As the bulk of your work is done locally with a DVCS, it is also very fast as there is no network latency involved. A DVCS also allows the concept of multiple remote repositories, which makes it easier to share changes directly between developers if required, without having to go via the central remote repository.

The following sections cover conceptual topics related to a DVCS, specifically Git, in more detail. These sections describe operations in a scenario where Git is used with a central remote repository with one or more users who have local repositories. Most of the concepts described still apply even if a central remote repository is not being used.

* 1. Snapshots Rather Than Differences

There is a major difference between Git and most other version control systems in the way they think about the data they store. Most version control systems store a set of base files and then track changes to those files over time.

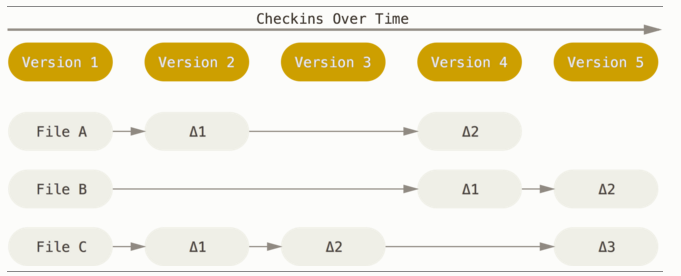


Figure 7: Storing data as changes to base files over time (https://git-scm.com)

Git doesn’t track changes in this way, it thinks of the data as a set of snapshots. Every time a change is made, Git takes a snapshot of all the files in the repository at that point in time and stores a reference to that snapshot. If a file hasn’t changed then Git won’t store it again, it just links to the previous identical revision of that file.

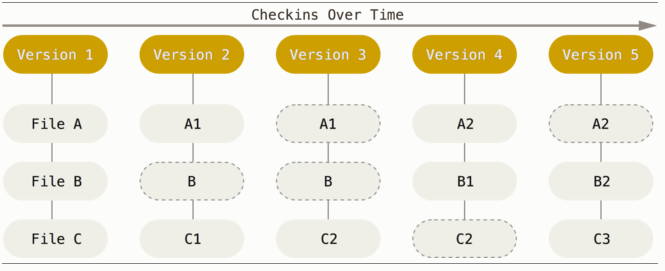


Figure 8: Storing data as a set of snapshots (https://git-scm.com)

This difference in the tracking of the data is vital because it has an effect on how you work with Git and it makes it easier to understand what is happening under the covers.

* 1. Local Operations

When you clone a repository in Git you get the history of the entire repository on your local machine. Most of the interaction you have with your Git repository will be on local files. Therefore most actions are very fast and can be performed offline. You can view file history, make changes and commit to local branches without being connected to a network. Generally, the only time you need to be connected and online is when you want to get changes from or send changes to a remote repository.

* 1. Push and Pull

In a Centralised VCS, such as Visual SourceSafe, you would use the “Get” command to retrieve changes from the central repository and the “Checkin” command to send changes to the central repository. In Git, you would use a “Pull” operation to get changes from a remote repository and a “Push” operation to send changes to a remote repository (notice this is a remote repository and not just a central repository). The Git “Pull” command consists of two separate commands, but we’ll come back to that later.

* 1. Git Integrity

Every operation on files in a Git repository is check-summed before being stored (using a SHA-1 hash). This allows revisions within a Git repository to be referred to by the checksum. These checksums are a fundamental part of how Git version control works – they ensure files do not get corrupted because if something is corrupted or lost in transit then Git can detect it from the checksum. You might notice these hash values (referred to as a “commit hash”) whilst working with Git so it’s worth knowing what they represent.

* 1. Working, Staging, Local and Remote

There are five main areas you need to be aware of when you are working with a Git repository:

* **The Git directory**: This is what you create when you clone a remote repository. This contains the metadata and object database for the repository. You will see a *.git* folder in the root of your local repository directory. This contains information about the Git repository you’ve cloned and all of its history. If you delete the local .git folder, the files on your machine are no longer linked to a Git repository and they simply become regular local files.
* **The working directory**: This is a single checkout of one version/revision (or branch) of the repository. When you checkout a branch or a specific revision of the files from the Git repository, the snapshots of those files are pulled out of the Git directory and stored in the working directory. If you switch to a different branch/revision at a later time, the files in the working directory change to represent the snapshot of the revision you’ve selected. This is a fundamental difference to how most Centralised VCS tools work. With a Centralised VCS, you would usually need a separate directory on your local machine for each branch/revision you work with (this is how Visual SourceSafe works for example).
* **The staging area**: This represents files that you have staged (added) ready for your next commit. When you make changes to a file or create a new file you must stage the change first before you can commit it. Notice this is referred to as staging changes, not staging files (there’s a bit more on this later).
* **The local repository**: This is the local repository on your machine – the files you created when you cloned the remote repository. When you stage and then commit a file, you are committing the change to the local repository i.e. the repository on your local machine. This means any other users of the remote repository cannot see your change (but they can see your change if they choose to configure your repository as a remote repository and connect directly to it).
* **The remote repository**: This is the remote repository or the central repository, which is typically hosted on a shared server such as GitLab. In a previous section of this document it was described how you could use a Distributed VCS without a remote repository, however from this point on the contents of this document assume a remote repository is being used.

When you run a “pull” operation on a remote repository you are fetching the latest changes from that repository and then merging them into the files on your local repository. When you run a “push” operation on a remote repository you are pushing committed changes from your local repository to the remote repository. Note that “push” and “pull” operations typically occur within the context of a single branch in Git i.e. you are not typically pushing or pulling the contents of the entire repository, just one branch.

* 1. Stage, Commit and Push

When you are working on files within a Git repository, you typically perform the following main operations when you want to make a change:

* Check out the local branch you want to make changes to.
* Modify the files in the working directory on your machine.
* Add the files that you want to commit to the staging area.
* Once staged, you can commit the changes to your local repository.
* Push the changes from the local repository to the remote repository.

The diagram below shows each of these stages. It also shows how you can retrieve changes from a local repository and merge these changes into your local branches.



Figure 9: Diagram showing the Git workflow

* 1. Git Extensions

There are several Windows tools available for working with Git repositories e.g. Git Extensions, SourceTree, GitHub, SmartGit, TortoiseGit etc.

For the rest of this training module you’ll be using the Git Extensions application, which is available for download here: <https://gitextensions.github.io/>.

You’ll be installing and working with the Git Extensions application when you go through the exercises in section 5.

1. Common Git Operations

This section briefly covers some of the main commands you might want to perform when working with a Git repository. This section explains the commands themselves but does not cover the detail of how to access them within the Git Extensions user interface. When you start working with Git Extensions you’ll see it is usually quite straight-forward to find and run each of these commands from the user interface.

* + 1. Cloning a Git repository

The vast majority of the time you will start by creating a Git repository via your hosting provider’s web interface. You will then clone that repository onto your machine. This will copy the remote repository to a local repository residing on your machine. The term “clone” is intentional in Git (as opposed to the similar “checkout” concept in Subversion). If you clone a repository that is already in use (rather than an empty one) you would get a local copy of every version of every file that has ever existed in that repository.

As mentioned previously, you don’t have to clone a remote repository as the starting point, because with Git you don’t even need a remote repository. You can also create a local repository on your machine without having any associated remote repository.

* + 1. Browsing the Repository

With a Graphical User Interface (GUI) like Git Extensions you can see the entire commit history for the repository in the GUI window. For each commit you can see the commit message, the author and when the commit occurred. The sequence of commits is also drawn up as a commit graph (showing branches, when they were merged, how many commits ahead a specific branch is etc.). From within Git Extensions you can also traverse the entire file tree for a single commit and see diffs of each file that was changed. You can also view the history for a single file, allowing you to run diffs and blames for each revision.

* + 1. Modifying Files and Staging

Once you have the files in your working copy you can change them as you normally would e.g. through Windows Explorer, Visual Studio etc.

Note that files in your working directory are either tracked or untracked. A tracked file is one that already exists in the repository i.e. it existed in the last Git snapshot or it didn’t exist but has been added to the staging area. Untracked files are all the other files i.e. files that didn’t exist in the last snapshot and haven’t yet been staged.

As you add, modify, delete and rename files in your repository they will have a different status e.g. untracked, unmodified, modified or staged. The diagram below shows the general stages for the lifecycle of a file within a Git repository.

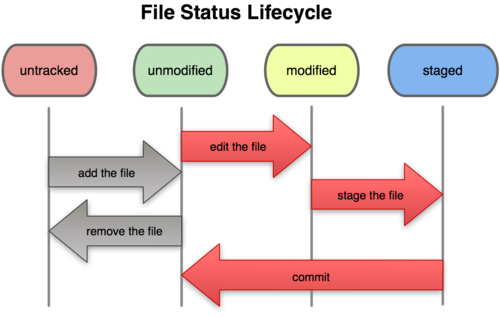


Figure 10: Diagram showing the lifecycle of files

What the diagram above doesn’t show clearly is that when you add a new file to the local repository, you add it to the staged area before it is committed. If you edit an existing file that is already tracked, then the diagram is accurate because as you modify and commit the file it will go through the unmodified -> modified -> staged path.

Note that when you stage a change to a file you are creating a snapshot of that change ready for commit. If you then decide to make further changes to the same file (without committing the staged version) this will result in two changes. One that is in the modified state waiting to be staged and another that is in the staged state in the staging area.

* + 1. Committing Changes

Once you’ve staged your changes and have prepared them to be committed, the next thing you want to do is actually commit them. When you perform a commit operation, you are saving your staged changes into the local repository. This is the same as a check-in in Visual SourceSafe, except here you’re committing the changes to the branch in the local repository, not to the central repository. When you commit you are provided with information about the commit e.g. the branch committed to, the SHA-1 checksum for the commit, how many files have changed etc.

* + 1. Amending a Commit

With Git it is possible to amend the previous commit – this is very useful if you’ve made a commit but you’ve forgotten to add a file. When you commit the next change you can choose to amend the previous commit. This will add the new changes to the previous commit, so your snapshot looks like it has only had one commit when in fact it has had two (the previous commit has been overwritten). This can be just as dangerous as it is useful, as it allows you to change the snapshot for a commit.

* + 1. Undoing Changes

If you’ve changed a file and moved it from the unmodified to the modified status, you can simply reset the file changes to undo what you’ve done. If you’ve changed a file and also staged it, you can un-stage the file first and then reset the file changes. Note that if you reset changes you are losing all history of those changes and you cannot recover them (since the change was never committed to a repository).

You can also run the “revert commit” operation through Git Extensions. Note that this will not actually delete a commit, but it will automatically create a new commit that allows you to undo the changes made in the original commit.

* + 1. Deleting and Moving Files

You can delete and move files in the way that you normally would with Windows Explorer or via your IDE. When you try to commit your change you will notice the files are waiting to be staged. If you have deleted a file it will be a single file marked as deleted. If you have renamed a file it will be the original file marked as deleted and the renamed file marked as a new item. Git has commands for deleting and renaming files that you can use from the command line if the default behaviour doesn’t do the job.

* + 1. Working with Remote Repositories

When collaborating with other developers you will be using remote Git repositories (whether this is a central repository or repositories on other developer’s machines). Remember that Git is a Distributed Version Control System (DVCS), which means changes are committed to your local repository first. To publish these changes to remote repositories you need to “push” them to the remote repository. If you want to see changes others have made, you will need to “pull” them from the remote repository.

When you clone a repository using Git Extensions you’ll find a remote repository named ‘origin’ is created for you automatically (based on the path you used for the clone). This is treated as your central remote repository e.g. the repository hosted on GitLab. You can create more remote repositories if required e.g. for other developers on your team.

With Git most of the work you do will be with the local repositories on your machine, but if you are working in a team you will need to interact with a remote repository to enable others to see your changes (by pushing to the remote repository) and to see changes others have committed (by pulling from the remote repository).

* + 1. Pulling Changes

When we refer to using the “pull” operation to get the latest changes, this is actually two operations – a “fetch” to retrieve the changes from the remote repository and then a “merge” to merge the changes with your local repository (i.e. merging the changes into any commits you have made in your local repository but you have not yet pushed up to the remote repository).

It is also possible to run just a “fetch” command if you want to retrieve changes from the remote repository but you don’t want to merge them into your local repository yet.

There is also a third option – fetch and rebase. Instead of merging, this will undo your local commits first, then fetch and apply the commits from the remote repository and then apply your local commits again on top of the remote changes. This operation often results in a cleaner commit history because it replays your local changes on top of the changes that already exist in the remote repository.

If you have uncommitted changes in your working directory and you need to pull the latest changes from the remote repository you will need to stash the changes first (there is a section on stashing changes later – see section 3.1.11). Stashing is essentially just storing your unfinished changes in a temporary area (without committing them) so you can work on your repository as if those unfinished changes don’t exist.

When you merge (or rebase) changes you could end up with “conflicts”. A conflict is where the commits in the remote repository collide with commits in the local repository, for example when the same file has been changed in the remote and local repositories. Git will try to resolve conflicts for you automatically by trying to create a combined version of the changes, but this is not always possible when the changes are complex, so you may need to resolve some of these conflicts yourself manually.

Resolving conflicts manually is something that you need to be very careful about, as it is possible you may inadvertently lose changes or break the compilation of the code. When resolving conflicts manually you need to use a diff tool (e.g. kdiff3) to analyse the conflicts for each file, as this allows you see how the code looked before and after the merge. This will allow you to judge which of the conflicting changes need to be present in the final merged version of the code file.

You may also need to speak to the other developers who have made conflicting changes. They will be able to advise how conflicting changes can be merged and can also verify whether you have inadvertently broken their code as a result of the merge.

Finally, when you’ve merged the code correctly you may have one or more new files with the .orig extension in your list of unstaged changes (the existence of this file depends on your Git settings and the diff tool you’ve used). The .orig file is a copy of the original file that contained conflicts, with markers to indicate the conflicts. You should delete these files once you have successfully completed your merge otherwise your repository will become littered with unnecessary .orig files (it is also possible to add the .orig file extension to your ignore list if you prefer).

* + 1. Pushing Changes

When you are ready to publish changes you have committed locally, you need to use the “push” operation to push the changes to a remote repository. You can push the changes from a single branch, from multiple branches or from all branches. As you would when working with any other source control system, you should pull changes from the remote repository first before you push changes. When using Git we generally pull changes and do any merging locally and then push the changes up to the server.

* + 1. Stashing Changes

Sometimes you will be in the middle of making some changes on a branch when you are asked to fix an unrelated issue in a different branch. Your current changes are half-written, so you don’t want to commit them but you don’t want to discard them either.

This is where you can use the Git feature that allows you to “stash changes”. This allows you to store a copy of your changes away into a stash area without committing them to the repository. You can then happily work on the issue in the other branch knowing you haven’t lost your changes. When you’re ready to come back to the original code change you can take it from the stash and apply it to your working directory.

The stash feature should only be used when stashing changes for a short period of time. If you need to place changes on hold for longer periods consider using a branch instead.

* + 1. Creating Tags

We use tags to mark a specific point in history (i.e. a commit) as being important e.g. a release version may be tagged “v1.0”. There are two types of tag – lightweight and annotated. A lightweight tag just points to a specific commit. Annotated tags are full objects in Git (have a checksum, tagger name, date, message etc.)

* + 1. Working with Branches

A branch is just a mechanism that allows you to work on a set of changes away from the main line of development. You can work on these changes, commit them and push them to the remote repository without affecting the code in any other branch (and therefore avoid affecting other developers). You might create a new branch when you start work on a new feature, when fixing a bug in production or when preparing for a new release.

A branch in Git is simply a lightweight pointer to a specific commit. When you create a new repository, a default branch named “master” is created for you automatically. The master branch points to the last commit you made. As you commit changes, the pointer is moved to the latest commit you’ve made. When you create a new branch, you are just creating a new pointer to a commit.

The series of diagrams below (borrowed from [here](http://git-scm.com/book/en/Git-Branching-What-a-Branch-Is)) demonstrates how branches work in Git and what happens behind the scenes when you work with branches. The green boxes in these diagrams represent commits, while the light grey boxes represent branches within the repository.

This first diagram below shows the state of your repository when you have a default “master” branch with several commits. You can see the branch pointer is pointing to the latest commit. If you then create a new branch named “testing” (from “master”) you end up with the diagram shown to the right below. As you can see from the diagram, there is now a new pointer to the latest commit for the “testing” branch.

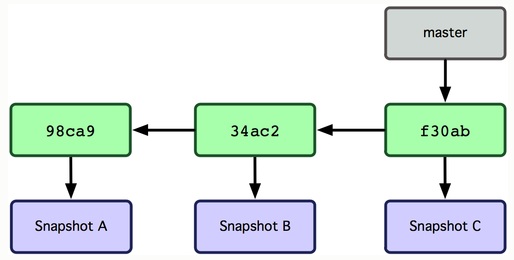
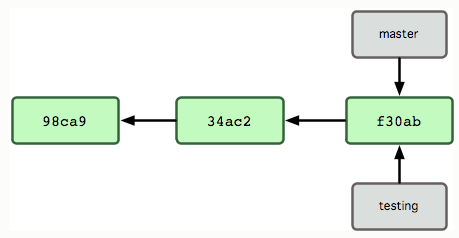
 

Figure 11: Showing master branch with commits and a new branch

Git keeps a special pointer called HEAD to track the branch you are currently working on (this is NOT the same as HEAD in SVN). In Git you run the “checkout” command to switch branches. This moves the HEAD pointer to point to a different branch. The diagrams below show what happens when you are working in the “master” branch (left diagram) and you switch to working in the “testing” branch (right diagram).

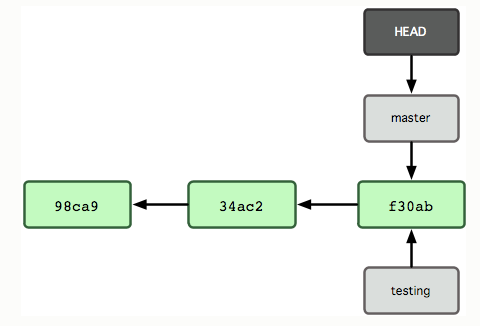
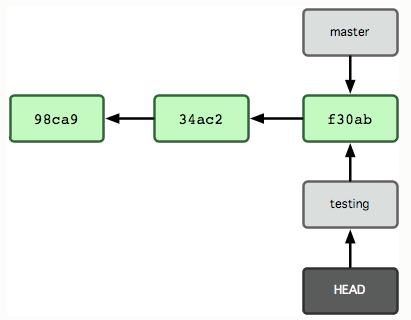
 

Figure 12: Indicating the current branch with the HEAD pointer

If you were to switch and make changes to each of the branches in turn the history diverges as shown in the diagram below, which shows the “master” and “testing” branches pointing to different commits within the repository.

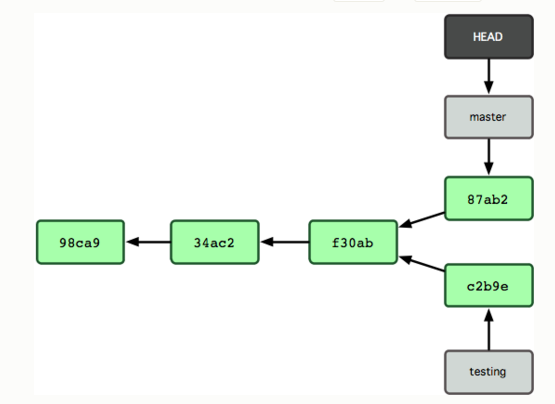


Figure 13: Changes made to both branches

As shown in the previous diagram, the branches have diverged and the changes are isolated in separate branches. You can work on each of these branches in isolation now without affecting the other. You can also switch back and forth between these branches and if necessary you can also merge changes between the branches when the changes are ready to be published.

Git makes it very easy to create and use branches and it is expected you will use branching all the time as part of your normal workflow. Branches are also very lightweight in Git (a simple pointer to a commit), which means they are very cheap to create and destroy. This is different to many other source control systems, which may involve creating a copy of all the project files and treat branching as an advanced task.

The one major difference between Git and any other source control system you’ve used in the past is likely to be the ease with which you can use branches. They are a central and very important part of using Git, so it is important you understand how they work.

* + 1. Checking out branches

When you clone a repository it will take a copy of the entire repository and it will place the objects/metadata inside a *.git* directory. The clone operation will also check out a working copy of the “master” branch in the repository. This means the working directory on your machine will contain the files belonging to the “master” branch by default.

If you want to work on a different branch you need to perform a checkout on that branch. You already have all the information about remote branches in your local repository, but you’re not tracking these branches locally yet. Upon checkout of a branch, if the branch isn’t already being tracked locally then Git will start tracking the remote branch. The checkout will also change the contents of your working directory to reflect the branch you’ve just checked out.

* + 1. Ignoring Files

With any set of files generated from an IDE like Visual Studio you’re going to have certain files that you don’t want to be part of your Git repository (e.g. compiled binaries, user specific configuration files etc.). You can choose not to stage these files, leaving them untracked, but this will clutter your commit window with files you never want to commit.

Git allows you to ignore these files by adding the file name (or file pattern) to the *.gitignore* file. With Git Extensions you can very quickly generate a *.gitignore* file with a set of default patterns. The default list isn’t exhaustive, so you’ll still need to occasionally add files to your ignore list manually.

* + 1. Integration with Other Tools

If you install Git Extensions on Windows, it will automatically provide extensions to Windows Explorer and Visual Studio. These extensions allow you to access several Git features from within the IDE or File Explorer e.g. viewing file history, blame, diff etc. These extensions aren’t covered in this training module.

* + 1. Git Command Line

If you’ve been reading any Git articles, tutorials or books as you’ve been going through this training you’ve probably guessed that Git was designed as a command line tool.

This training module focusses on the Git Extensions GUI, but everything we’ve covered boils down to one or more command-line statements behind the scenes.

It is important to be aware of this, as there may be times where you need to resort to the command line if the GUI doesn’t do exactly what you need. You don’t need to have any understanding of how to use Git on the command line to use the Git GUI, but some command line knowledge can go a long way if you want to become a Git expert.

1. Git Resources

Much of the online material you will find on working with Git will be based on the command line Git tools. It can sometimes be difficult to find good resources for learning how to use Git on Windows (as the GUI tools haven’t been around as long as the command line tools). The documentation for the command line tools is still applicable to the Git Extensions GUI and the general concepts are still the same.

The links below provide some further resources you can use to find out more about Git and how it works. You are advised to do some further reading to build upon the concepts that have been introduced in this training module. This document has covered the basic concepts and you will need more than the basics to tackle the exercises in the next section.

* A free Pro Git book (primarily command line Git): <http://git-scm.com/book>. Some of the material in this training module is based on information from this book.
* An article discussing a useful Git branching model (this is what the exercises are based on so this is definitely worth reading): <http://nvie.com/posts/a-successful-git-branching-model/>
* The Git Extensions user manual (the commands available in Git Extensions): <http://gitextensions.googlecode.com/files/Git%20Extensions%20-%20User%20Manual.pdf>
* A list of Git tutorials that may be helpful for developers who are new to Git: <http://sixrevisions.com/resources/git-tutorials-beginners/>
* Git for Windows tutorial series: <http://lostechies.com/jasonmeridth/2009/06/01/git-for-windows-developers-git-series-part-1/>
* Understanding Git conceptually: <http://www.sbf5.com/~cduan/technical/git/>
* Git concepts simplified: <http://gitolite.com/gcs.html#(1)> (note that this is a web based slideshow – you will need to move between the slides with your mouse or keyboard – press right arrow or spacebar).
* Git explained from a SVN perspective: <http://git.or.cz/course/svn.html>

1. Git Exercises

You will need to carry out the following exercises using a private Git repository in GitLab and the Git Extensions tool for Windows.

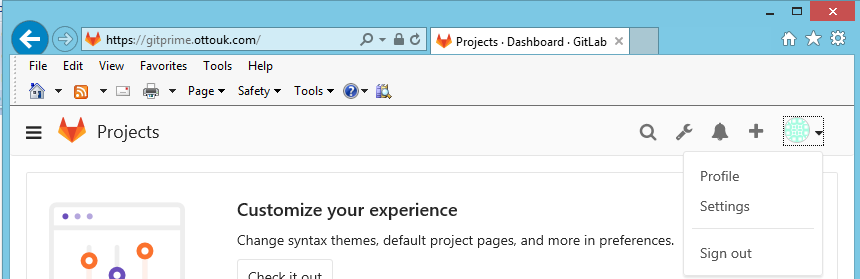
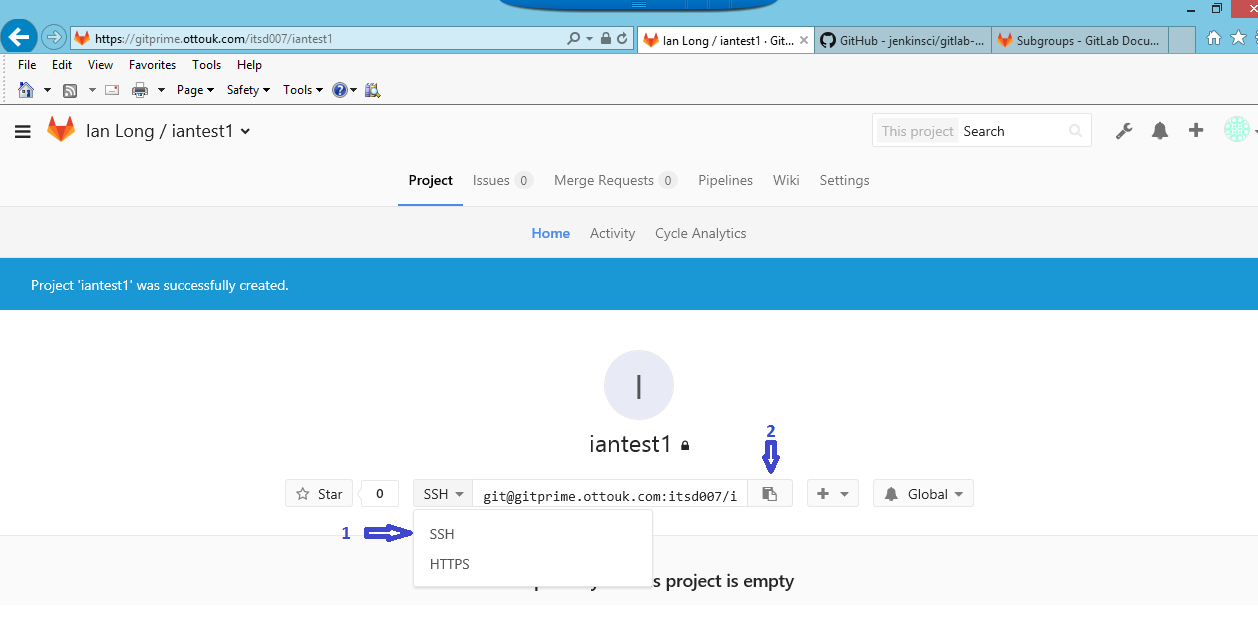
You will need to do some independent reading and research in order to complete these tasks. You should check in with your training mentor at regular intervals throughout these exercises (possibly as frequently as the end of each sub-section). If you get something wrong in an early exercise it will have an impact on your progress throughout the rest of the exercises so you need to ensure you are heading in the right direction.

**Important:** The branching model you should use for these tasks is described on this page: <http://nvie.com/posts/a-successful-git-branching-model/>. Please ensure you have read through this page and you are familiar with the branching model before you start working through these exercises.

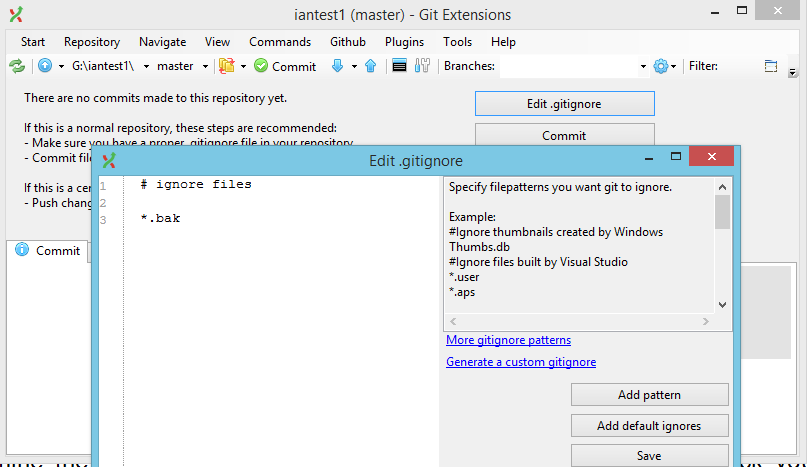
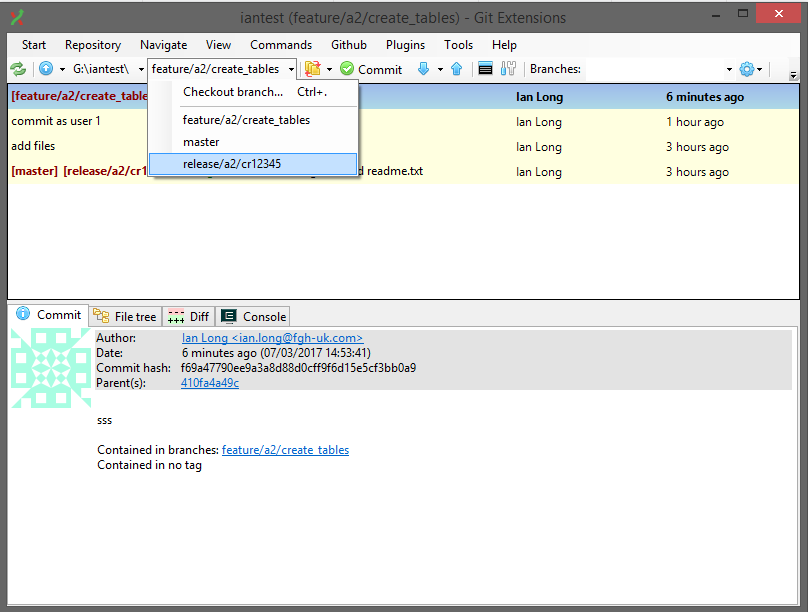
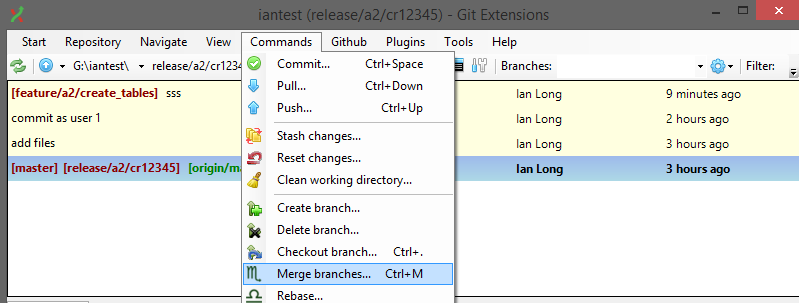
* 1. Create Repository
* Sign into GitLab <https://gitprime.ottouk.com> using your windows user\_id and password.
* Create a private Git repository on GitLab. Click on the Green New Project box, enter a project name and description.
  1. Install Git Extensions
* Install Git Extensions for Windows. There are guides available on the web that show you how to do this, but these steps summarise what you should choose:
  + Install KDiff (unless you already have a merge tool).
  + Choose OpenSSH as the SSH client.
  + You’ll see a security popup and two other installers during the installation:
    - KDiff3 Installer: Choose all default install options
    - Git Setup Wizard: Choose all default install options
* Once Git Extensions installed set up your username and email (these are used to track your commits). Make sure everything is green on the Checklist tab before you move onto the next step.
* For this module it is recommended you use SSH when cloning your repository. You can use HTTP if you wish, but you will need to keep entering your user\_id and password in Git Extensions. To set up SSH connection follow these steps:
  + In windows explore right click not on any files or folders and select Git Bash Here
  + In the window that opens type ssh-keygen -t rsa -C "youremailaddress"
  + You should get the following lines

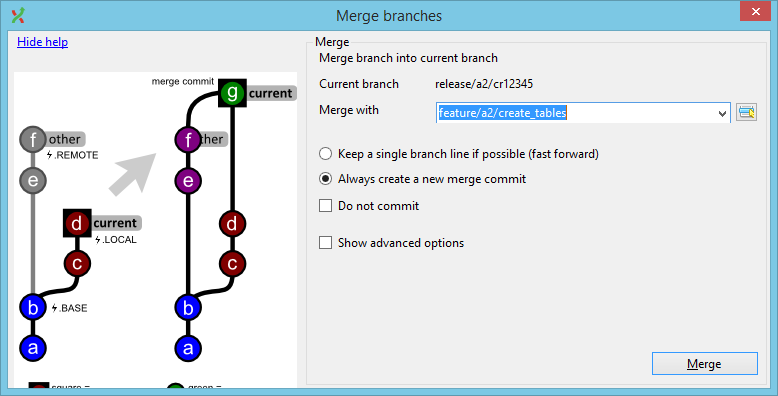
Generating public/private rsa key pair.

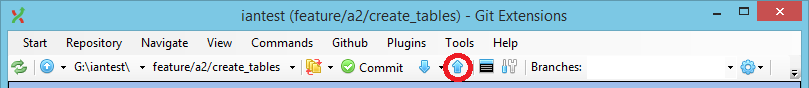
Enter file in which to save the key (/h//.ssh/id\_rsa):

* + If the file path in brackets is not the same as above enter /h//.ssh/id\_rsa and press return.
  + When the enter passphrase line appears either leave blank (this would normally be a bad thing but as it is putting it in a directory that only you and administrators can access it is less of an issue) or give a passphrase (you will need to remember this for when you clone a repository etc).
  + After the passphrase it will either generate a public/private key pair or if you already have one will ask you to overwrite it. I you already have one and know the passphrase the answer n.
  + Open a command window in administrator mode and type the following **mklink /D %userprofile%\.ssh h:\.ssh** close this window.
  + In an internet browser sign into gitlab select your user settingsand on the next page select SSH KEYS now open a command window(not in administrator mode) and type **type %userprofile%\.ssh\id\_rsa.pub | clip** back to gitlab and paste it in the box provided and click **Add key**.
  1. Clone Repository
* Clone your private Git repository via Git Extensions. You will need the HTTPS or SSH path for your repository depending on which you have chosen to use 

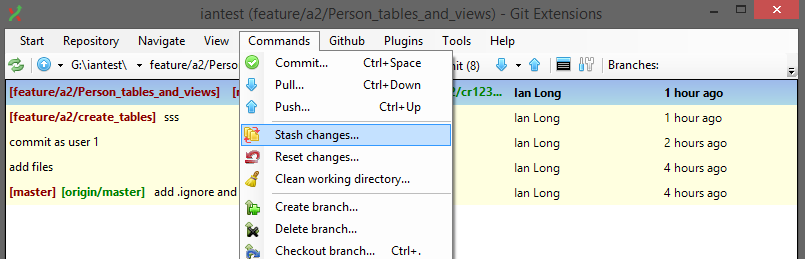
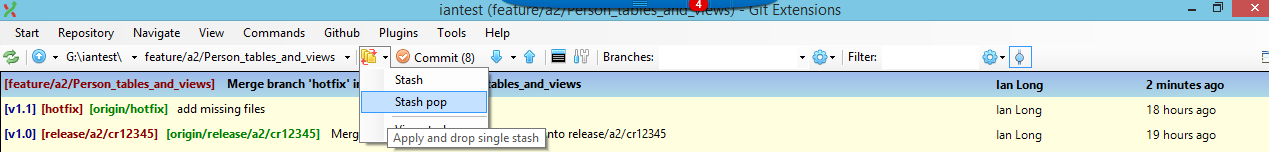
1. select SSH or HTTPS depending on your preference.
2. Copy the url to your repository.
   1. Create, Commit and Push

* At this point you should create a *.gitignore* file for your repository that contains the default ignore patterns. You can do this through Git Extensions. In Windows Explorer in the repository directory right click and select GitExt Browse. Add a comment (starts with #) and any file extension you want git to ignore. in this example I have told it to ignore file of type .bak.
* Open the directory for your Git repository on your machine (using Windows Explorer). Create a new README.txt file in this directory. Add a line to this file containing your name and the current date.
* Go back into Git Extensions and click the Commit button. You will need to stage your README and *.gitignore* files before you can commit them. Ensure you choose the Commit option and specify message(s) for the commit(s).
* Examine the commit messages shown in Git Extensions and also check your repository on the hosting provider site.
* From Git Extensions push your changes to the remote repository.
  1. Tag It
* This is the starting point for your repository, so create a v0.1 tag on master. In Git Extensions menu commands Create Tag and tick the push tag to origin box.
  1. Branch Out
* In Git Extensions create a new branch named “release/a2/cr12345” and checkout the branch.
* Add a code file such as a SQL script or ASP page to your “release” branch. Copy folder I:\is\_doc\FGH Git Training\files\_for5.6
* Add the changes you’ve just made to the staging area. See [I:\is\_doc\FGH Git Training\screen\_shots\ Stage files.docx](screen_shots/Stage%20files.docx)
* Commit the changes you have made to the release branch. If there are any files that shouldn’t be under source control, ensure these are not committed.
* Then create another new branch named “feature/a2/create\_tables” (from “release/a2/cr12345”) and checkout the branch.
  1. Merge Changes
* Clone your local repository – Gitext Clone Repository to clone, browse to your local repository, destination browse to where you want your new repository and click clone. Note the last change above do not appear in the newly cloned repository because they were not committed.
* While still working in the feature branch of the original repository, modify the files\_for5.6\mysqlscript.txt file add the following line **select @i/@a** and change **set @i=1234** to **set @i=1235** then stage and commit the changes
* Now switch to the new repository, ensure you checkout the “feature/a2/create\_tables” branch in the second repository so that you are working in that branch (ensure you are not working in the master branch, as the master branch may be checked out by default).
* In the second copy of the repository, while still in the feature branch, change the files\_for5.6\mysqlscript.txt change **declare @a int** to **declare @a varcar(20)** and **set @i=1234** to **set @i=1233** . Stage and commit this change.
* Still working in the second clone of the repository again, pull the latest changes from the remote feature branch in widows explorer right click in the repository and move the cursor to Git Extensions and then select pull, when the window opens make sure the “*Merge remote branch into current branch*” is ticked. As you made changes to the README file in this first clone and you committed them you should get a merge conflict at this point (because the README file was also changed and committed in the second clone of the repository). You will need to resolve any merge conflicts that occur click ok to merge the conflicts and then click start mergetool.
* See [I:\is\_doc\FGH Git Training\screen\_shots\Resolving merge conflicts.docx](screen_shots/Resolving%20merge%20conflicts.docx)
* You don’t need to worry about the second clone of your repository now – we will be working exclusively in the first clone of the repository from this point onwards.
  1. Release 1.0
* We’ve decided the code is complete and ready to go now. So checkout you release branch either from commands: Checkout branch or from the drop down
* Merge the feature branch into the release branch 



* Your code is now ready for release. Tag the branch to indicate the release version(v1.0).
* Push the changes from your local feature branch to the remote feature branch (note that the “Branch to push” bit in the popup window should say “release/a2/cr12345” to “release/a2/cr12345”).

When you do this, Git will automatically create a new remote branch named “release/a2/cr12345”. You may also be asked about adding a tracking reference when you push your changes. A tracking reference will link your local feature branch to the new remote feature branch.

* 1. Add “Person\_tables\_and\_views” Feature
* We need to add more functionality to our code.
* Create new feature branches for this change and make the necessary changes to the script. Copy folder I:\is\_doc\FGH Git Training\files\_for5.9 into your new feature branch
* Do not commit or stage the changes yet, just move onto the next section.
  1. Stop – Hotfix Time
* A bug has been raised with v1.0 of the code and it needs to be fixed immediately.
* Stash any changes you’re working on. Select all the files tick the include untracked and click save changes.
* Create a new branch for the hotfix.
* Fix the issue in the hotfix branch by adding the files in I:\is\_doc\FGH Git Training\hotfix to your files\_for5.6 folder and commit your changes.
* Tag the hotfix with a minor version(v1.1)
* Push the hotfix to GitLab and merge your hot fix branch into your local feature branch.
  1. Back To The Feature
* You can finish working on your “Person\_tables\_and\_views” feature now.
* You stashed the changes for this feature earlier. Restore these now , finish your changes and commit them.
* Push these changes up to the remote repository and this feature is ready for release now.
* Create a new release branch “release/a2/cr12346” from master branch and merge your feature branch into this branch, tag the release version and push it to GitLab.
  1. More New Features
* Create new release and feature branches for additional changes. Write your scripts and follow the process as before to prepare them for release until comfortable with the process.
  1. Browsing File History
* Look back over the commit log in Git Extensions and also on the website for the hosting provider. Consider how your changes are reflected in the commit graph.
* Use the blame function on the main SQL script and on the README file. Look at each of the commits and consider how this function helps you see how parts of the file have changed over time.
  1. Additional Tasks

These additional tasks are not part of this module, but you may choose to look into these if you want to learn more about Git and Git Extensions:

* We haven’t looked at how to use a rebase. What is the difference between a rebase and a merge? When might you want to use a rebase?
* There are several features that we haven’t experimented with as part of this training module e.g. creating patches, deleting tags, cherry pick commits, resolving complex merge conflicts etc. Read up on some (or all) of these and experiment with them on your training repository.
* We haven’t considered how Git can be used from Visual Studio or from Windows Explorer. Take some time to experiment with this functionality, particularly the extensions to Visual Studio. These can be useful during the development process as you don’t have to switch out of the IDE (although most people tend to use Git Extensions rather than Visual Studio).
* Everything we’ve done has been focussed around using a repository in GitLab as the remote repository. You can also connect to other developer’s repositories too. If you have any other people around you using Git then you can experiment with this if you want to try it out.
* Spend some time looking at the command line version of Git. Read up on the commands it provides and consider how you would achieve some of the tasks you’ve been doing from the command line. Everything you’ve done has issued one or more Git commands behind the scenes, so it’s important to be aware of these commands and how they relate to the GUI. There may be a time when you need to use the command line to achieve something the user interface doesn’t provide by default.
  1. Couple of useful commands
* To list files that have changed between branches open a command window and cd to you local repository enter *git diff --name-status branch1 branch2*
* To list all branches open a command window and cd to you local repository enter *git branch –a*
* If you need to restore a file to the last commit point delete the file and then check the file out from the command line with *git checkout <filename>*
* To find a string in all the files in the repository on a command line
* *Git grep –i ‘some text’* the –i switch makes it case insensitive.