Version Control

Let us start with the definition of what exactly is Version Control. It is a system that records changes to a file or a set of files over time so that you can recall specific versions later on. We can provide some examples of these systems below:

1. CVS

- It is one of the oldest of the revision control systems. First released in 1986, and up to this day Google Code still hosts the original Usenet post announcing CVS. It is the **de facto standard** and is installed virtually everywhere. However the code base isn't as fully featured as SVN or other solutions.

2. SVN

- Subversion is probably the version control system with the widest adoption. Most open-source projects use Subversion as a repository because other larger projects, such as SourceForge, Apache, Python, Ruby and *many* others, use it as well. Google Code uses Subversion exclusively to distribute code.

3. Git

- The new fast-rising star of version control systems. Initially developed by Linux kernel creator Linus Torvalds, Git has recently taken the Web development community by storm. Git offers a much different type of version control in that it's a **distributed version control system**. With a distributed version control system, there isn't one centralized code base to pull the code from. Different branches hold different parts of the code. Other version control systems, such as SVN and CVS, use centralized version control, meaning that only one master copy of the software is used.

  Git prides itself on being a fast and efficient system, and many major open-source projects use Git to power their repositories; projects like:

  i. Linux Kernel.
  ii. WINE
  iii. Fedora
  iv. e.t.c

4. Mercurial

- This is another **open-source distributed version control system**, like Git. Mercurial was designed for larger projects, most likely outside the scope of designers and independent Web developers. That doesn't mean that small development teams can't or shouldn't use it. Mercurial is extremely fast, and the creators built the software with performance as the most important feature. The name "mercurial" is an adjective that means "Relating to or having characteristics (eloquence, swiftness, cleverness) attributed to the god Mercury."

Aside from being very fast and scalable, Mercurial is a **much simpler** system than Git, which is why it appeals to some developers. There aren't as many functions to learn, and the functions are similar to those in other CVS systems. It also comes equipped with a stand-alone Web interface and extensive documentation on understanding Mercurial if you have been using another system.

Types of Version Controls

Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.
To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

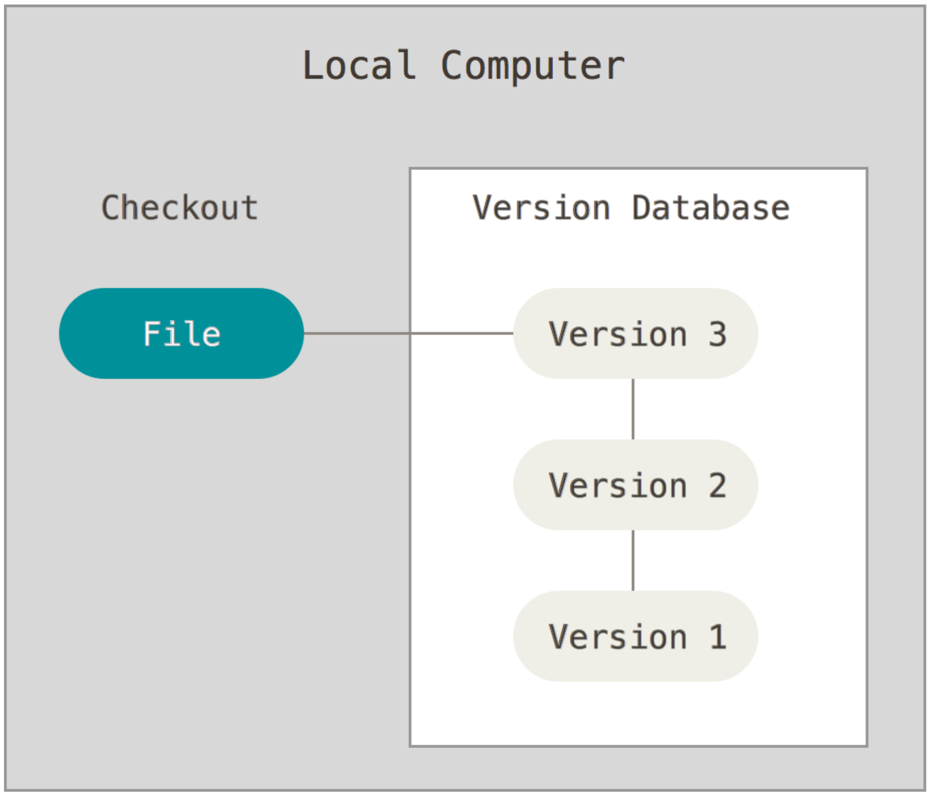


Figure 1-1. Local version control.

One of the more popular VCS tools was a system called RCS, which is still distributed with many computers today. Even the popular Mac OS X operating system includes the rcs command when you install the Developer Tools. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

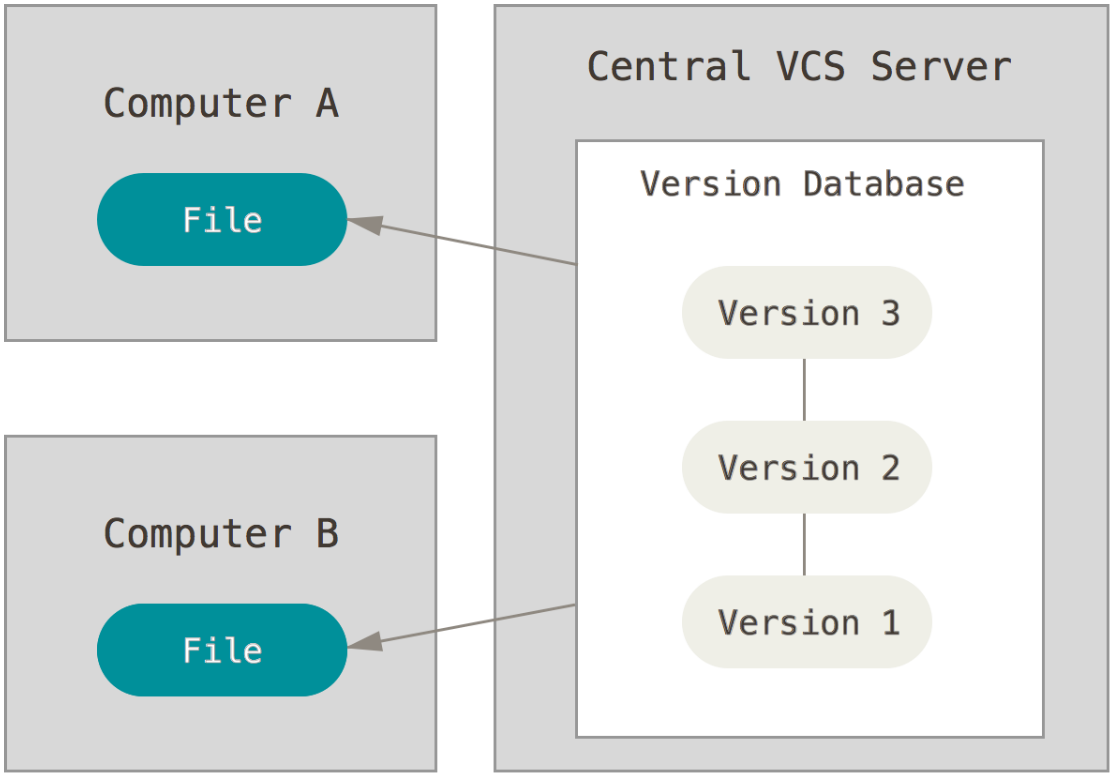


Figure 1-2. Centralized version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what; and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem – whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

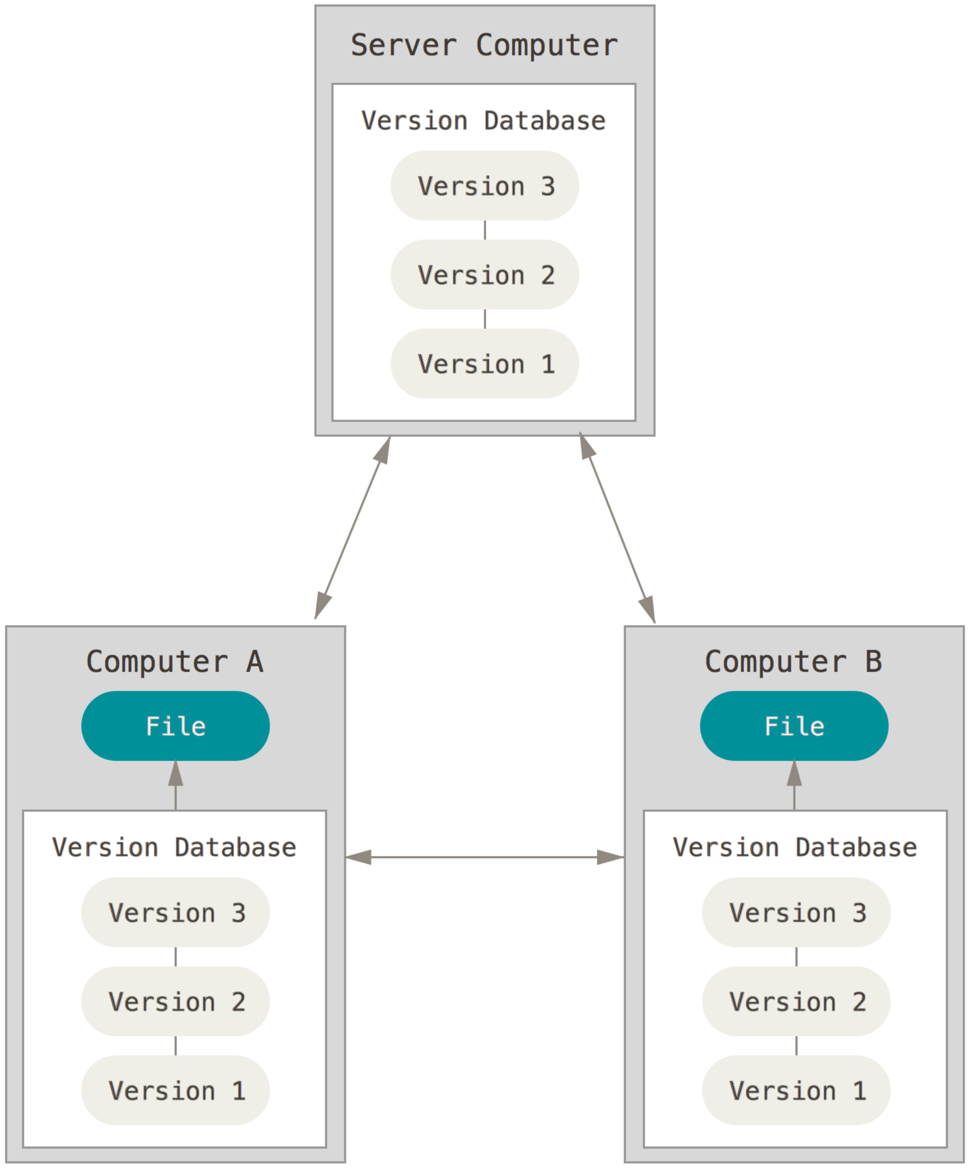


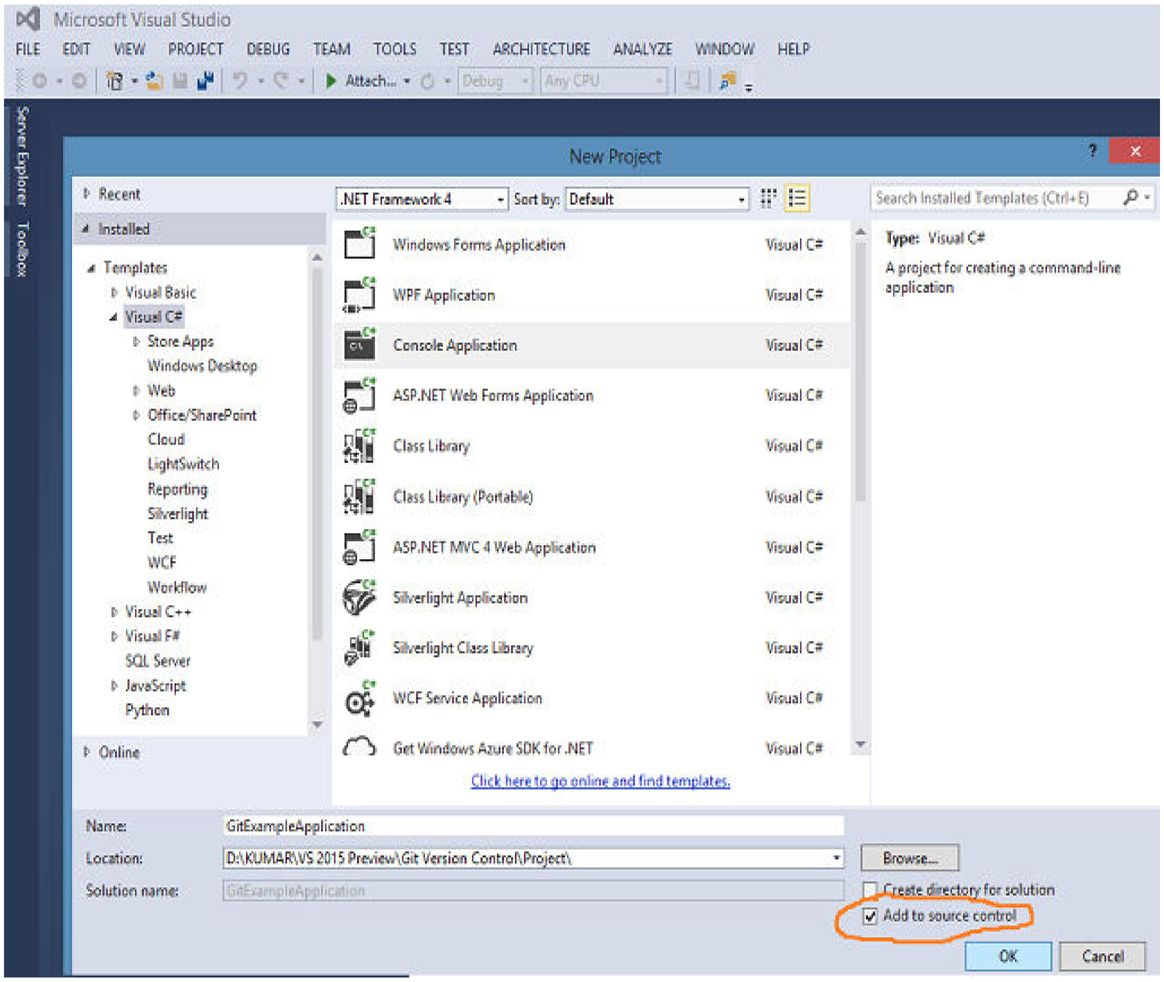
Figure 1-3. Distributed version control.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

Why should an Individual use Version Control?

Without a Version Control System in place, you're probably working together in a shared folder on the same set of files. Shouting through the office that **you** are currently working on file "xyz" and that, meanwhile, your teammates should keep their fingers off is not an acceptable workflow. It's extremely error-prone as you're essentially doing open-heart surgery all the time: sooner or later, someone will overwrite someone else's changes.
With a VCS, everybody on the team is able to work absolutely freely - on *any* file at *any* time. The VCS will later allow you to merge all the changes into a common version. There's no question where the latest version of a file or the whole project is. It's in a common, central place: your version control system.

Step by Step Tutorial


Microsoft Visual Studio
FILE EDIT VIEW PROJECT DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Attach... Debug Any CPU
Server Explorer Toolbox
Recent
Installed
Templates
Visual Basic
Visual C#
Store Apps
Windows Desktop
Web
Office/SharePoint
Cloud
LightSwitch
Reporting
Silverlight
Test
WCF
Workflow
Visual C++
Visual F#
SQL Server
JavaScript
Python
Online
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Forms Application Visual C#
Class Library Visual C#
Class Library (Portable) Visual C#
ASP.NET MVC 4 Web Application Visual C#
Silverlight Application Visual C#
Silverlight Class Library Visual C#
WCF Service Application Visual C#
Get Windows Azure SDK for .NET Visual C#
Click here to go online and find templates.
Search Installed Templates (Ctrl+E)
Browse...
OK
Cancel

New Project

.NET Framework 4 Sort by: Default

Type: Visual C#

A project for creating a command-line application

Name: GitExampleApplication

Location: D:\KUMAR\VS 2015 Preview\Git Version Control\Project\

Solution name: GitExampleApplication

[ ] Create directory for solution

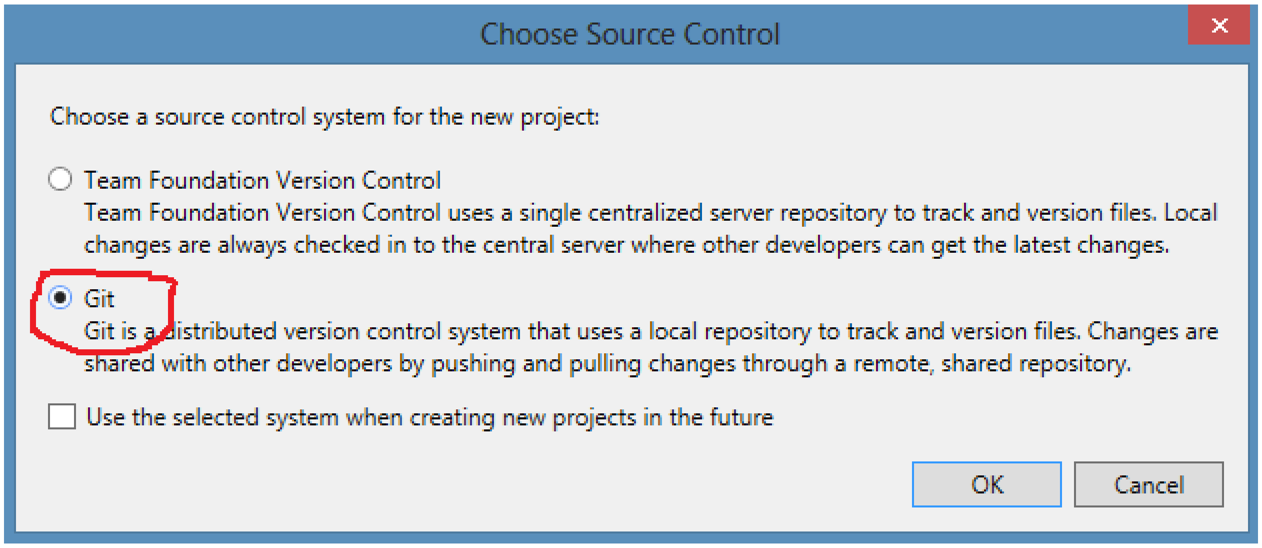
[✓] Add to source control

Now select Git from the popup window.

Example Program

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GitExample
{
  class Program
  {
    static void Main(string[] args)
    {
            Console.WriteLine("Git Example App Demo !");  //Firstly written
            Console.ReadLine();
        }
    }
}
```



Choose Source Control

Choose a source control system for the new project:

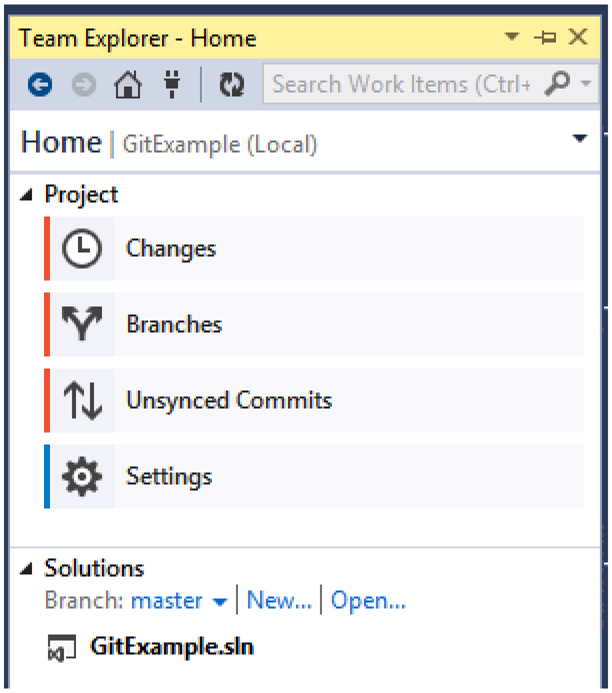
Team Foundation Version Control
Team Foundation Version Control uses a single centralized server repository to track and version files. Local changes are always checked in to the central server where other developers can get the latest changes.

[✓ Git]
Git is a distributed version control system that uses a local repository to track and version files. Changes are shared with other developers by pushing and pulling changes through a remote, shared repository.

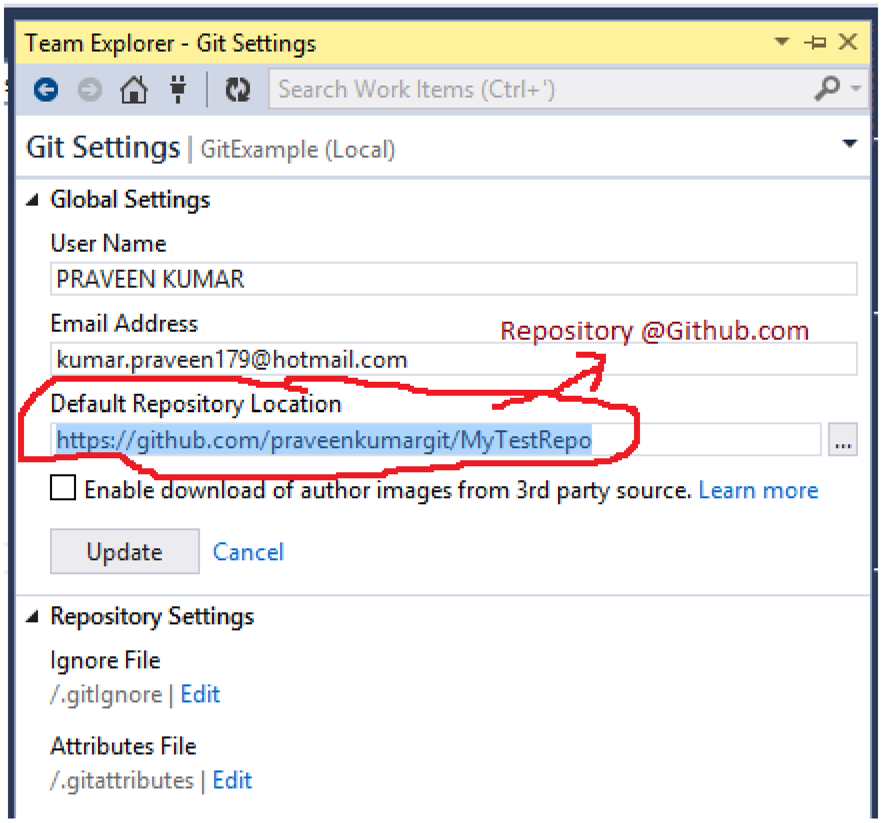
Use the selected system when creating new projects in the future

Now we have created our application that can work under the Git Version Control environment but some additional settings still need to be configured. To configure the

application go to **Team Explorer** in Visual Studio and configure the setting by providing **Username**, **Email Address** and **Repository Location.** Where Repository is the place on the server/local where the history of our work is stored.



We can publish the project either on Github.com or local IIS.


Team Explorer - Git Settings
Search Work Items (Ctrl+')
Git Settings | GitExample (Local)
Global Settings
Repository @Github.com
Update
Cancel
Repository Settings

User Name
PRAVEEN KUMAR
Email Address
kumar.praveen179@hotmail.com
Default Repository Location
https://github.com/praveenkumargit/MyTestRepo
☐ Enable download of author images from 3rd party source. Learn more
Ignore File
/.gitIgnore | Edit
Attributes File
/.gitattributes | Edit

Basically we can do the following operations:

- **Branching:** Branching in a Git is a lightweight movable pointer that points to the commits. Git has a default branch name that is **master**, changes. Suppose we want to do some changes in the given project then we would need to make a branches. If we do commit initially, it points to a master branch to the last commit.
- **Changes:** After implementing the new feature on a branch, we need to merge these changes with a master branch, so that everyone can use it.
- **Push:** Push command asks Git to fetch the new changes that have been applied to files/project. It means, before merging the changes the admin fetches the change from Git using **Push.**
- **Pull:** It merges the master branch from the remote repository.
- **Merge:** Basically the merge command identifies the changes to the other branch and makes changes to the master.
- **Commit:** Commit records the changes to the files in history. When we commit in Git, it stores a commit object that contains a pointer to the snapshot of the content we stayed, the author and message metadata.

Improvements with Visual Studio 2015 Preview

Let's see the new improvements that are in Visual Studio 2015. Now it's even easier to work with branches and see how the changes in our history diverged.

Branches: Now we can organize our branches hierarchically by specifying a prefix. Now with Visual Studio 2015, local branches as well as remote branches are
shown separately in a tree view. Let's see this new feature with an example screen.

History Details: Now we can see the commit diverge in the history that was not available with previous version of Visual Studio.