# Gestire la Semantic Versioning con GitFlow e GitVersion in una vNext Build

Nell’articolo precedente si è discussa la pubblicazione dei package nuget tramite la build vNext di VSO / TFS, ma non si è parlato di gestione di versione. Se si utilizza Git come controllo di codice sorgente e si adotta GitFlow come modello di gestione delle branch, si può ottenere un versioning automatico con poco sforzo. In questo articolo verrà quindi mostrato come integrare GitVersion nella propria build vNext.

Il punto di partenza è quello descritto nell’articolo precedente, con la build vNext che viene lanciata manualmente specificando il numero di versione del package nuget desiderato.

L’obiettivo è quello di rendere automatica la numerazione, così da ottenere il Continous Deployment, ovvero il deploy continuo su nuget di ogni cambiamento al codice.

## GitFlow and GitVersion

GitFlow Git non è altro che una **convenzione sul come gestire in maniera efficace le branch in un repository Git**. Di base si considera la branch *master* come quella di rilascio, la branch develop è invece usata per lo sviluppo e non è detto che sia stabile. A queste due si aggiungono branch per ogni feature e branch di supporto per hotfix e rilascio. Per chi non conoscesse GitFlow e fosse interessato all’argomento rimando a questi due link per una buona introduzione all’argomento:

1. [A successful Git Branching model](http://nvie.com/posts/a-successful-git-branching-model/)
2. [GitFlow CheatSheet](http://danielkummer.github.io/git-flow-cheatsheet/)

Per semplificare l’adozione di GitFlow esistono delle estensioni per riga di comando che introducono una serie di alias per lavorare semplicemente con GitFlow. Naturalmente è presente anche un plugin per Visual Studio: [[VS 2015 version](https://visualstudiogallery.msdn.microsoft.com/f5ae0a1d-005f-4a09-a19c-3f46ff30400a)] [[VS 2013 version](https://visualstudiogallery.msdn.microsoft.com/27f6d087-9b6f-46b0-b236-d72907b54683https:/visualstudiogallery.msdn.microsoft.com/27f6d087-9b6f-46b0-b236-d72907b54683)]; la prima volta che si utilizza il plugin di Visual Studio verranno tra l’altro installate automaticamente le estensioni GitFlow per la riga di comando.

Quando il team diventa padrone di GitFlow, il passo successivo è passare al [Semantic Versioning](http://semver.org/), un modello di gestione dei numeri di versione che è oramai adottato da moltissime realtà.

La buona notizia è che esiste uno strumento open source e gratuito chiamato [GitVersion](https://github.com/GitTools/GitVersion) che gestisce il semantic versioning in maniera automatica analizzando la storia di un repository Git. Lo strumento è semplice da utilizzare e se siete interessati esiste una buonissima documentazione online. In questo articolo verranno date le basi per raggiungere l’obiettivo richiesto, rimandando il lettore alla documentazione online per le funzionalità più avanzate.

## Come funziona GitFlow

GitVersion può essere scaricato direttamente da GitHub (<https://github.com/GitTools/GitVersion>) ed è un semplice strumento a riga di comando. Per utilizzarlo in una build è sufficiente copiarlo nella cartella radice del repository Git ed invocarlo la prima volta con l’opzione /ShowConfig per generare un file di configurazione di default.

GitVersion /ShowConfig > GitVersionConfig.yaml

Il file di configurazione può tranquillamente essere omesso, ma è utile generarlo in modo da avere immediata visibilità sulle opzioni di default attualmente attive. Il parametro probabilmente più importante e quello che solitamente viene immediatamente modificato è chiamato *mode* ed è specifico di ogni branch. I possibili valori sono ContinuousDelivery oppure ContinuousDeployment, ad indicare se si adotta per quella branch un deploy continuo o meno. Indicando ContinuousDeployment si sta specificando a GitVersion che ogni commit verrà deployato e quindi si deve sempre generare un nuovo numero di versione.

Spesso questo è il comportamento atteso e nel caso la versione non si incrementasse effettuando altri commit, la soluzione è modificare il file di configurazione per specificare ContinuousDeployment come mode per le branch per cui la versione non si incrementa. Non è scopo di questo articolo mostrare tutte le personalizzazioni disponibili, per questo rimando alla guida ufficiale <http://gitversion.readthedocs.org/en/latest/>.

Per lo scopo di questo articolo, basti sapere che è sufficiente invocare GitVersion senza alcun parametro per ottenere come output un file json, contiene tutte le informazioni riguardanti il Semantic Versioning

{

"Major":1,

"Minor":5,

"Patch":"0",

"PreReleaseTag":"unstable.9",

"PreReleaseTagWithDash":"-unstable.9",

"BuildMetaData":"",

"BuildMetaDataPadded":"",

"FullBuildMetaData":"Branch.develop.Sha.8ecde89ef5b97eabcf6e0035119643334ba40c4e",

"MajorMinorPatch":"1.5.0",

"SemVer":"1.5.0-unstable.9",

"LegacySemVer":"1.5.0-unstable9",

"LegacySemVerPadded":"1.5.0-unstable0009",

"AssemblySemVer":"1.5.0.0",

"FullSemVer":"1.5.0-unstable.9",

"InformationalVersion":"1.5.0-unstable.9+Branch.develop.Sha.8ecde89ef5b97eabcf6e0035119643334ba40c4e",

"BranchName":"develop",

"Sha":"8ecde89ef5b97eabcf6e0035119643334ba40c4e",

"NuGetVersionV2":"1.5.0-unstable0009",

"NuGetVersion":"1.5.0-unstable0009",

"CommitDate":"2015-10-17"

}

Questo è il risultato che si ha eseguendo GitVersion con la branch develop in checkout. In questo caso **la versione che lo strumento ha determinato è la 1.5.0-unstable.9**. Bisogna ora comprendere come lo strumento determini questi valori.

Prima di tutto la tripletta major.minor.patch viene determinata andando a ritroso nel log del repository e verificando il tag associato alla branch master. In questo caso la master branch ha un tag v1.4.1, ad indicare che la versione attualmente stabile e deployata è la 1.4.1. Dato che la branch develop rappresenta concettualmente il codice che andrà nella successiva versione, sulla branch develop il numero di versione è 1.5.0. Il fatto che venga incrementato il valore minor nella tripletta major.minor.patch è il default di GitVersion e può essere cambiato nel file di configurazione.

Il suffisso unstable.9 viene invece determinato in questo modo: il numero 9 rappresenta il numero di commit di distanza dalla master, mentre il valore “unstable” viene determinato dalla branch in cui ci si trova. La develop ha il suffisso unstable mentre ad esempio le branch di rilascio hanno il suffisso beta. In questo caso il numero 1.5.0-unstable.9 indica che questa versione è stata compilata nella branch develop, che è 9 commit avanti alla versione 1.4.x.

Per verificare il funzionamento, si può iniziare una release usando il comando *git flow release start 1.5.0*, questo provoca la creazione di una nuova branch *release/1.5.0* e facendo girare GitFlow con questa nuova branch in checkout, il valore di FullSemVer è 1.5.0-beta.0. Mano a mano che si aggiungono commit sulla branch di rilascio l’ultimo numero aumenta, ad indicare la distanza dal commit di creazione della branch.

Quando viene chiusa la branch di rilascio con il comando *git flow release finish*, la branch release/1.5.0 verrà sottoposta a merge con la master, verrà messo il tag v1.5.0 al commit corrispondente e verrà fatto il merge della release/1.5.0 sulla develop. A questo punto se si invoca GitVersion sulla develop il semVer sarà 1.6.0-unstable.0

## Come integrare GitVersion in una vNext Build.

In realtà esiste un articolo nella documentazione ufficiale di GitVersion (<http://gitversion.readthedocs.org/en/latest/build-server-support/build-server/tfs-build-vnext>) che mostra come effettuare l’integrazione. In questo articolo verrà adottato invece un approccio alternativo che garantisce maggiore flessibilità di utilizzo.

La ragione principale di questo approccio alternativo è che: GitVersion invocato con il parametro buildserver imposta un certo numero di variabili di ambiente con i valori di SemanticVersioning visti in precedenza. Purtroppo il valore delle variabili di ambiente non viene trasferito nei task successivi, per cui, a mio avviso, è più interessante un approccio di parsing del risultato json visto in precedenza con PowerShell.

La commandlet ConvertFrom-Json viene in aiuto perché **permette di effettuare il parsing di un testo Json** e di inserire il risultato direttamente in una variabile PowerShell.

$Output = & ..\GitVersion\Gitversion.exe /nofetch | Out-String

$version = $output | ConvertFrom-Json

A questo punto lo script può accedere a tutte le proprietà di cui ha interesse. Il primo passo è generare i tre numeri di versione per gli assembly:

$assemblyVersion = $version.AssemblySemver

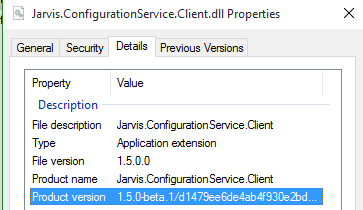
$assemblyFileVersion = $version.AssemblySemver

$assemblyInformationalVersion = ($version.SemVer + "/" + $version.Sha)

**Questi valori verranno poi usati per modificare i vari file assemblyinfo.cs o assemblyinfo.vb del progetto**. Uno script per ottenere questo risultato è già stato pubblicato nel mio blog inglese: <http://www.codewrecks.com/blog/index.php/2014/01/11/customize-tfs-2013-build-with-powershell-scripts/>. In quell’articolo si parla di build XAML classica, ma lo script powershell può essere tranquillamente utilizzato anche in una build vNext.

Uno degli aspetti interessanti è che si può utilizzare l’attributo AssemblyInformationalVersion per memorizzare una informazione di tipo stringa, in modo da poter inserire il numero completo di Semantic Versioning a cui solitamente viene aggiunto anche l’id del commit (il parametro Sha dell’oggetto Json ritornato da GitVersion).

Ecco quindi che direttamente dalle proprietà Windows degli assembly generati è possibile risalire alla numerazione ed al commit attualmente utilizzato per generare l’assembly. Massima tracciabilità con poche righe di PowerShell.

[](http://www.codewrecks.com/blog/wp-content/uploads/2015/10/image17.png)

## Inviare comandi al motore di Build vNext tramite PowersHell

Uno degli aspetti più interessanti della Build vNext è la possibilità di inviare comandi all’infrastruttura semplicemente scrivendo nell’output standard. È sufficiente infatti utilizzare una formattazione appropriata come descritto nell’apposito link di documentazione (<https://github.com/Microsoft/vso-agent-tasks/blob/master/docs/authoring/commands.md>) per dialogare con l’infrastruttura di build.

In questo modo si può semplicemente utilizzare il classico Write-Output per inviare comandi al motore di Build, ad esempio per richiedere l’aggiornamento di alcune variabili.

Write-Output ("##vso[task.setvariable variable=NugetVersion;]" + $version.NugetVersionV2)

Write-Output ("##vso[task.setvariable variable=AssemblyVersion;]" + $assemblyVersion)

Write-Output ("##vso[task.setvariable variable=FileInfoVersion;]" + $assemblyFileVersion)

Write-Output ("##vso[task.setvariable variable=AssemblyInformationalVersion;]" + $assemblyInformationalVersion)

In questo caso nella prima riga è stato cambiato il valore della variabile NugetVersion, che, come mostrato nel precedente articolo, viene utilizzata per gestire la numerzione del package di NuGet. In questo modo tutto il resto della build rimane identica all’articolo precedente, ma ora la numerazione dei package viene effettuata automaticamente in base alla branch/commit che viene compilata.

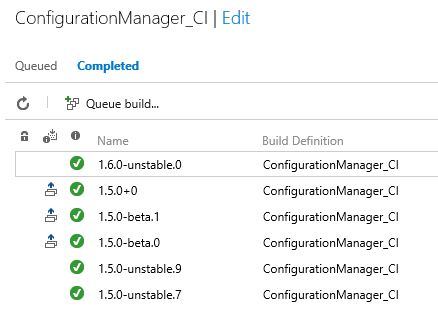
## Gli ultimi ritocchi

Prima di dichiarare completata la personalizzazione della build, rimane un’ultima personalizzazione da effettuare, ovvero cambiare il numero della build, operazione effettuabile con il seguente comando.

Write-Output ("##vso[task.setvariable variable=build.buildnumber;]" + $version.FullSemVer)

Write-Output ("##vso[build.updatebuildnumber]" + $version.FullSemVer)

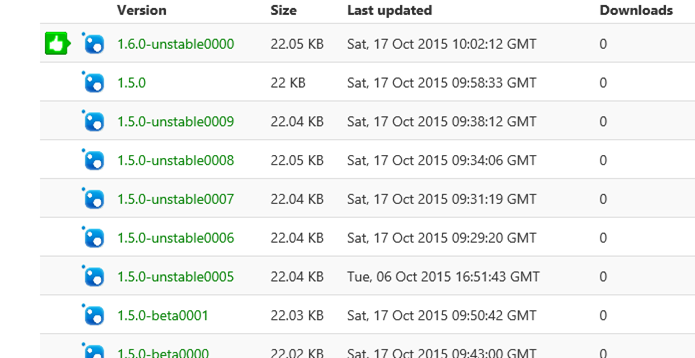
Questi comandi sono quasi equivalenti, entrambi aggiornano il numero della build, ma il primo aggiorna anche la variabile build.buildnumber con il nuovo numero. Ecco quindi come si presenta la lista delle ultime build nel sommario.

[](http://www.codewrecks.com/blog/wp-content/uploads/2015/10/image18.png)

Come si può vedere, il nome della build permette immediatamente di capire che branch è stata compilata e che versione di package è stato generato. Se si osserva attentamente la figura precedente si potrà notare che **le build beta e di rilascio (1.5.0-beta e 1.5.0.0) sono state accodate automaticamente, mentre le build unstable sono state accodate manualmente**.

Questa situazione è abbastanza normale, quando viene iniziata la branch di rilascio ha senso che ogni commit generi una nuova build e quindi un nuovo package di nuget, mentre ha meno senso che questo avvenga anche per ogni commit nella branch di develop. Per questa ragione la branch di develop non è sottoposta ad integrazione continua. Naturalmente se si vuole pubblicare un package unstable basato sulla develop, è sufficiente accodare una build in modo manuale, senza preoccuparsi però in questo caso del numero di versione, che verrà messo automaticamente.

L’ultimo passo è verificare che effettivamente i package pubblicati abbiano la numerazione corretta.

[](http://www.codewrecks.com/blog/wp-content/uploads/2015/10/image19.png)

## Conclusioni

Grazie alla build vNext, GitFlow, GitVersion e poche righe di PowerShell è stato possibile automatizzare la pubblicazione di un pacchetto NuGet con numerazione automatica, seguendo le specifiche di Semantic Versioning.

Questo è uno dei molti esempi che mostrano la grande flessibilità del nuovo sistema di build, in particolare quando viene accoppiato con repository basati su Git, non solamente se hostati su VSO/TFS. Si ricorda infatti che è anche possibile utilizzare la build vNext per effettuare build di repository presenti su GitHub o su altri servizi di hosting.