TFS Branching Guidance - Scenarios

Bijan Javidi, James Pickell, Tina Erwee, Willy-Peter Schaub
Microsoft Corporation

VSTS Rangers

This content was created in a VSTS Ranger project. VSTS Rangers is a special group with members from the VSTS Product Team and Microsoft Services. Their mission is to provide out of band solutions for missing features or guidance.

Contents

Scenarios Overview:

This scenarios section provides examples of common situations that development teams find themselves in and specific branching patterns that address the situations. The most frequent situations that development organizations encounter are where it becomes necessary to have different teams working in parallel, or being able to have a subset of a team work in isolation.

The scenarios outlined here should allow teams to quickly understand the branching options that are available to them, and the scenarios should provide the necessary framework for a team to adopt a branching structure that better addresses their needs. It's important to note that while these scenarios present a solution to a specific situation, it is indeed possible and quite likely to setup an equally valid and different the branching structure.

One of the most valuable aspects of the scenarios section is that each scenario contains both the layout of the source structure and hierarchy of the branching structure. While the file structure and branching structure of most development teams is complex, re-occurring branching patterns routinely present themselves and we hope that the scenarios presented here help answer the questions of when to branch, what to branch, and the value that branching can provide to help development be faster and smoother.

In each scenario, a situation will be presented that is facing development team at Wood Grove Banking. The resulting branching structure as well as the branching and merging operations will be explored and examined in detail.

Before exploring each scenario, it's important to fundamentally understand Branching purpose in the wider discussion of source control management. Branching enables parallel development and build automation by providing each development phases (bug fix, new feature development, stabilization of code, etc.) its own self contained isolation snapshot of needed source codes, external dependent components and a stable build environment.

Source Control Management typically involves:

- Taking a snapshot of source code to create an isolation snapshot of a stable or milestone state of source code (such as the last successful build). The source where the code is obtained called the parent branch and the resulting code copy is the child branch.
- Developers can make code changes where they are contained and stabilized within the isolated snapshot in the child branch.
- Developers can perform bi-directional synchronization of changes, i.e. merging, with the parent branch. In general, a small delta changes are always the best way to utilize automated merge provided by VSTF and it provides the ability to perform self code review on the smaller delta and reduce human errors. This best practice on merging smaller chunk of codes as soon as possible is strongly recommended compared to waiting until the end of development cycle milestone with large amount of changes resulted in requiring manual merge process between the parent and the child branches.

Scenario #1: Single Team Branching Model

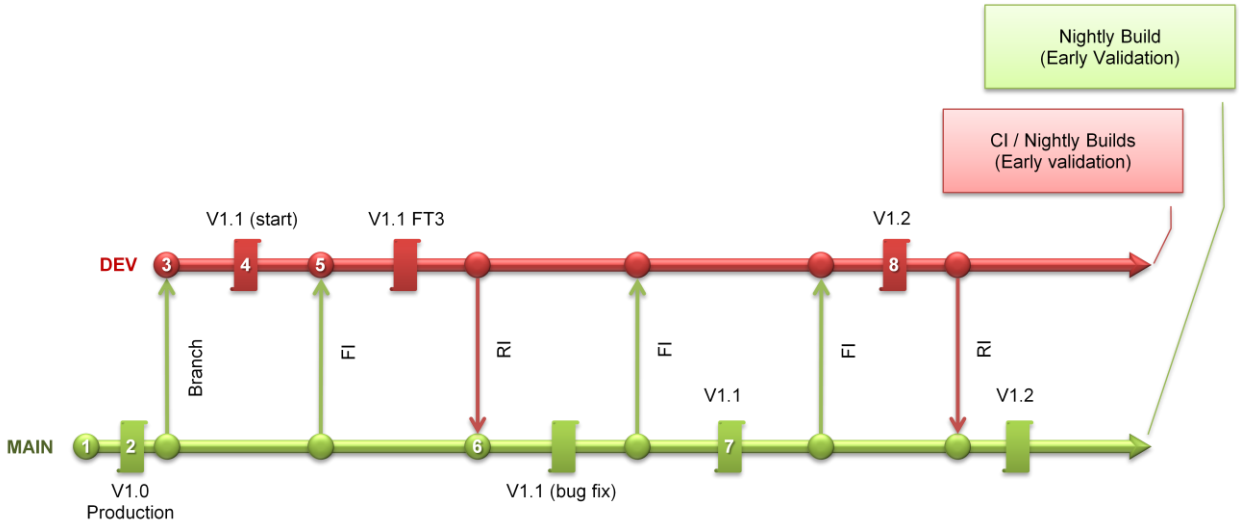


Figure 2.1 Single Team Branching Model

Scenario Overview: Single Team Branching Model

An organization is trying to improve their engineering practices and has decided to focus on the flow of source code throughout the development process. The development team has recently encountered numerous problems such as developers checking in code that hasn't been adequately tested, having a very volatile code base right before release, and having to have lengthy code freezes where the development team is unable to modify any source. To address these issues, the development team has decided to adopt a very simple branching model.

In this scenario, the branching model entails creating a **DEV** branch from the main source line. The purpose of creating this **DEV** branch from **MAIN** is that by branching **MAIN** to **DEV** a consistent forward and reverse integration model is created and this model allows for better integration consistency with the addition of future branches.

In this scenario, it's imperative that the **MAIN** branch be stable with the latest customer-ready version. Keeping the **MAIN** branch stable allows for more frequent milestone deliveries to be distributed to customers without the risk of shipping unstable or incomplete feature code. In keeping the **MAIN** branch stable the development team also reduces the time required to prepare the final release.

Code is labeled in **MAIN** with the actual version number and the **DEV** branch is labeled at the start of development as the next version, in this scenario this happens to be version 1.1. As development gets underway the version of code in **MAIN** remains stable at V1.0 while development is moving forward. At specific intervals code is forward integrated from the **MAIN** branch to the development branch to ensure that the development team is able to understand the implications of the changes they are making in V1.1.

At some specified point in time, the development branch has met certain quality standards and is ready to be put into production, so the **DEV** Branch is reverse integrated into the **MAIN** branch.

It's important to realize that the **DEV** branch has running CI builds while the **MAIN** branch only has nightly builds. The CI builds in the **DEV** Branch allow for the team to quickly understand if the code changes they've made have broken the development branch and potentially fix the **DEV** build before breaking the nightly build in the **MAIN** branch.

Branch Definitions for this scenario:

- **Dev** - The **DEV** branch is for staging all new development activity whether it is for feature development, bug fixing for coming developing release, or integration of breaking changes. This area is designed to isolate, contain, and stabilize new development activities.
- **Main** – The **MAIN** branch is used as a holding tank to integrate changes to and from development branch, and it should be kept as stable as possible.

Scenario Walkthrough:

1. The initial source code that is release 1.0 is checked into TFS in the **MAIN** branch.
2. The **MAIN** branch is labeled with V1.0.
3. A **DEV** branch is created from the **MAIN** branch.

   $/ WoodGroveBanking/Main/ to $/WoodGroveBanking/DEV
4. The **DEV** branch is labeled with V1.1 at the start of development of the next version.
5. The **MAIN** branch is forward integrated into the **DEV** branch at regular intervals.
6. Certain quality standards have been met the **DEV** branch is reverse integrated into **MAIN**

   $/WoodGroveBanking/DEV to $/ WoodGroveBanking/Main/
7. The **MAIN** branch is now labeled V1.1
8. The **DEV** branch is labeled V1.2 and development continues forward.

Structure in Team Foundation Server

To demonstrate this branching pattern in practice using Team Foundation, the source control structure and branching structure would be displayed as follows.

Figure 1.0 Scenario Source Structure

Branches:

File Name	Branched from version
$/WoodGroveBanking/Main/source	
$/WoodGroveBanking/Dev/source	1048

Figure 1.1 Scenario Branching Structure

Scenario Source Control Artifacts:

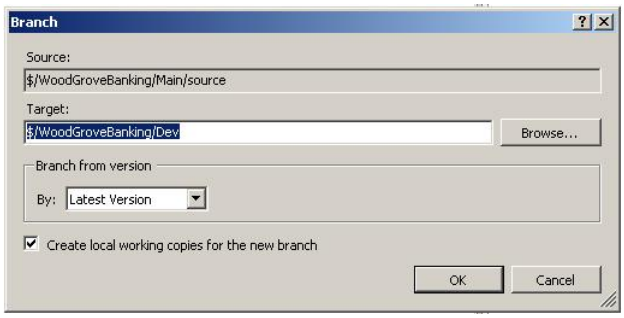


Figure1.3 Branch MAIN to Dev

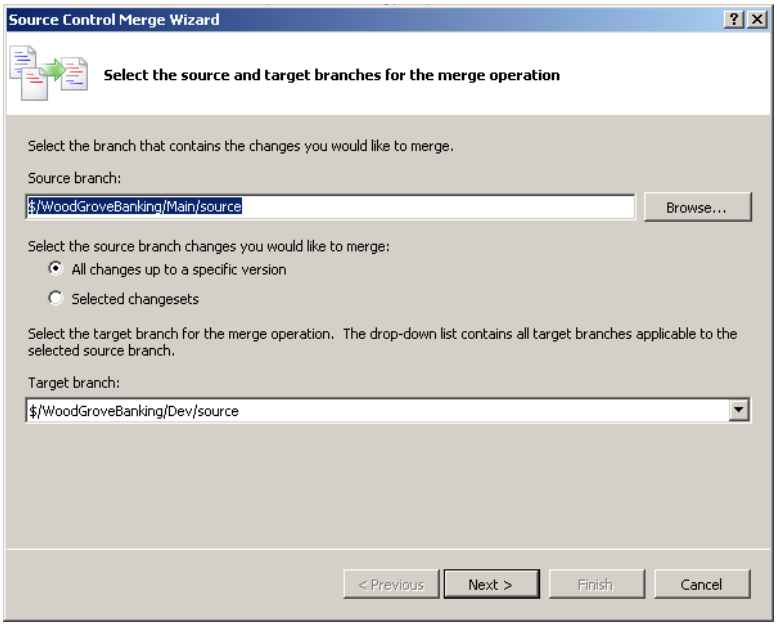


Figure1.4 – Merge from Main to Dev

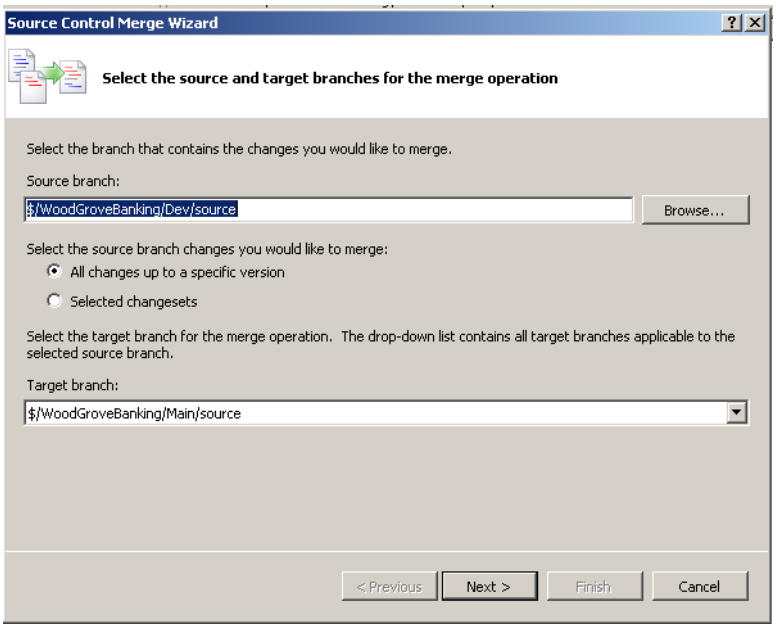


Figure 1.5 – Merge from Dev into Main

Scenario #2: Concurrent Hot Fix, Service Pack, and v.Next

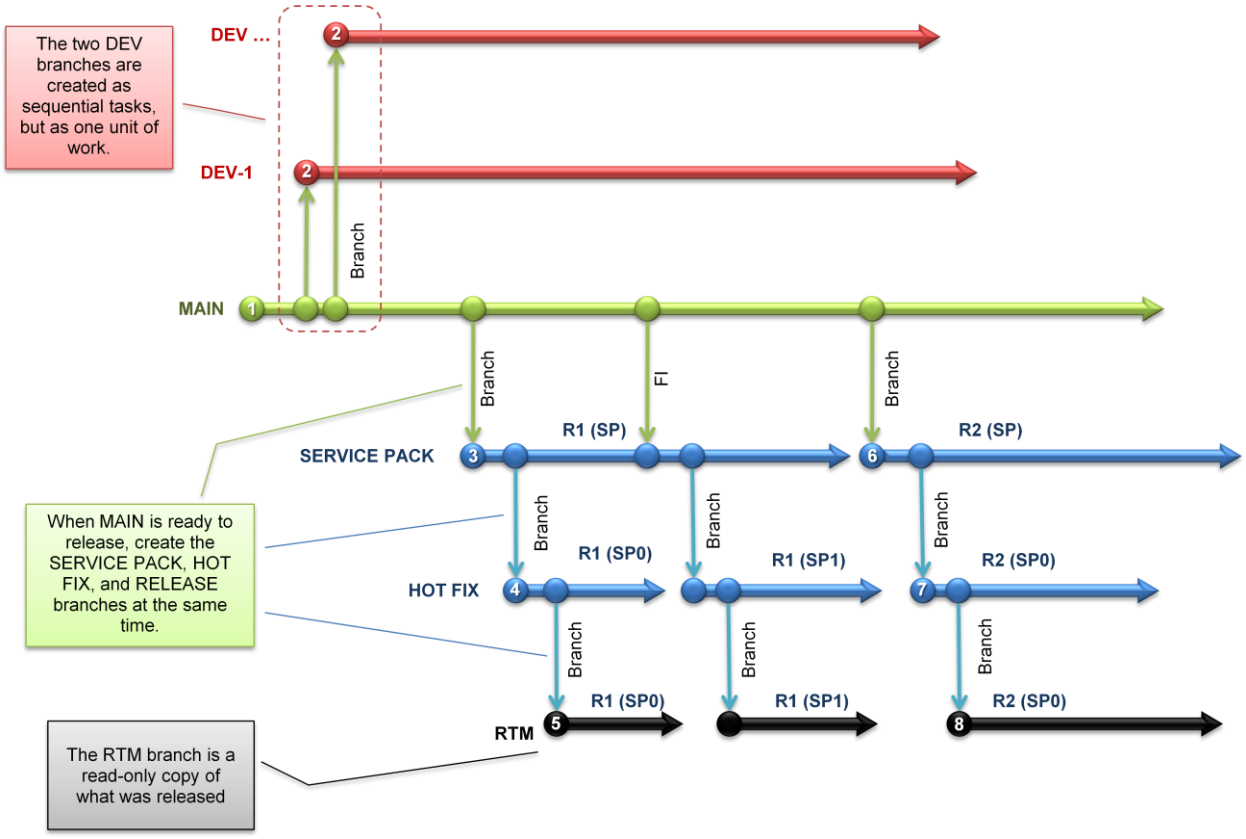


Figure 2.1 Concurrent Hot Fix, Service Pack, and v.Next

Scenario Overview: Concurrent Hot Fix, Service Pack, and v.Next

Many organizations find themselves in the position where they need to be able to service a released product with hot fixes, a cumulative service pack that includes all approved hot fixes, and the organization wants to be able to work simultaneously on the next version of the product. The organization also needs to ensure that there are few regressions from "lost" changes and that there are reduced merge conflicts, these 2 considerations are important because time shouldn't be spent reengineering solutions or spending lengthy cycles resolving complex merges.

The Windows development team at Microsoft has a comprehensive branching structure in place that provides the ability to work concurrently on hotfixes, service packs, and future releases. The Windows development team has a branching plan that precisely defines the ways that code leaves the organization and ends up in the hands of its customers. A simplified overview of the Windows branching structure involves a minimum of 3 different release vehicles:

- Major Release (RTM)
- Hot Fixes for a Major Release
- Service Packs for the Major Release

Within each of these 3 release vehicles there are separate branch plans that enable code to be checked in once and then merged back to main. From development to main there should be

regular forward (FI - parent to child) and reverse (RI - child to parent) integrations. The regular FI and RI integration ensure that the entire organization has a clear understanding of the structuring of the application and is aware of dependencies between various development efforts.

Once the code in main has reached some objective measure of release quality (e.g. bug count, test pass %, preview feedback, etc.) then it is time to create the production/release branch for this release. The key to this branch plan is that the SP, Hot fix and RTM branches must be created in order to assure the parent child branch relationships are maintained. Without enforcing the integrity of these branch relationships, the organization cannot begin to address regressions or merge conflicts.

For an organization looking to be able to work concurrently on releases there should be a strong association between the branching structure and the release vehicles so that the entire organization is aligned on very specific releases and in ensuring the associations it is possible to identify work that is not focused around a particular release.

Branch Definitions for this scenario:

- **Dev** - child branches of main where developers work on the current on future versions on the product.
- **Hot Fix** - specific fixes required to unblock a specific customer blocking issue.
- **Main** - this is the junction between the development and production branches. For most organizations this branch should be very stable.
- **Production** - child branches of main where servicing (i.e. hot fix and SP) check-in's are made.
- **RTM** - release to market, this is the branch that has the sources for your final release.
- **Service pack** - a cumulative package of hot fixes and updates targeting a previously released version on you product.
- **V-next** - the next version of the product.

Scenario Walkthrough:

1. The initial source code is checked into TFS in the [Main] Branch

2. A Dev Branch is created with the relevant team branches from [Main].
   - The first branch is: $/WoodGroveBanking/Dev/Dev_Team1
   - The second branch is: $/WoodGroveBanking/Dev/Dev_Team2

3. The organization intends to create a service pack for the application, a collection of cumulative bug fixes, so a Service Pack Branch is created under the Production Branch.

   $/ WoodGroveBanking/Main/ to $/WoodGroveBanking/Production/V1/ServicePack

4. The organization has to have the ability to ship immediate fixes and still be able to integrate all hotfixes into the next culmulative release without having any regressions so a child Branch for Hot Fixes is created under Service Pack.

$/WoodGroveBanking/Production/V1/ServicePack to $/WoodGroveBanking/Production/V1/Hot Fix

5. The organization wants to ensure that once code reaches some objective measure that a release branch is created so a child Branch for the RTM version is created from the Hot Fix branch.

   $/WoodGroveBanking/Production/V1/Hot Fix to $/WoodGroveBanking/Production/V1/RTM

6. The organization also has begun work on the next version of the product, V2, so a child Branch is created from Main for V2.

   $/ WoodGroveBanking/Main/ to $/WoodGroveBanking/Production/V2/ServicePack

7. At the onset of the V2 release the organization knows that at some point after V2 has shipped they will ship a service pack that will contain all hot fixes so the branching structure is identical to V1 and a child Branch for Hot Fixes is created under Service Pack.

$/WoodGroveBanking/Production/V1/ServicePack to $/WoodGroveBanking/Production/V2/Hot Fix

8. The V2 release will take considerable time to be developed and tested but at some point in the future a child Branch for the RTM version is created from the Hot Fix branch.

$/WoodGroveBanking/Production/V1/Hot Fix to $/WoodGroveBanking/Production/V2/RTM

Structure in Team Foundation Server

To demonstrate this branching pattern in practice using Team Foundation, the source control structure and branching structure would be displayed as follows.

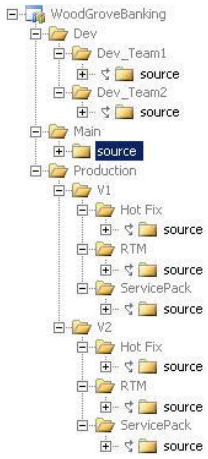


Figure 2.1 Scenario Source Structure

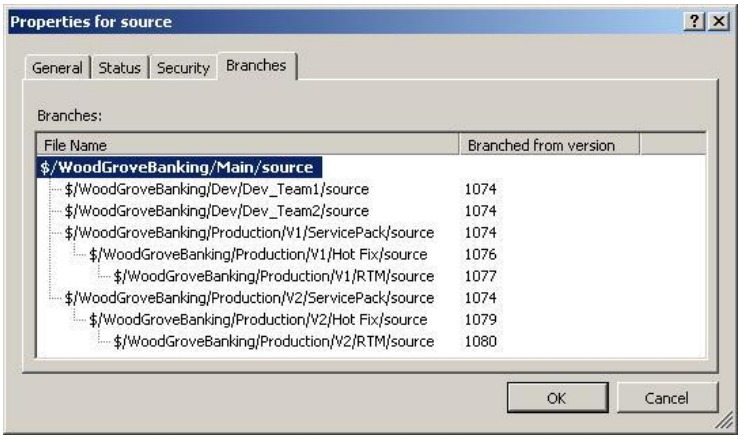


Figure 2.2 Scenario Branching Structure

Scenario Source Control Artifacts:

This section describes the artifacts relating to the implementation of the scenario in Team Foundation Server.

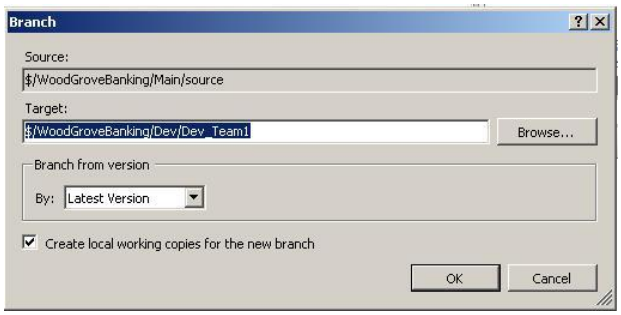


Figure 2.3 Branch to Dev_Team1

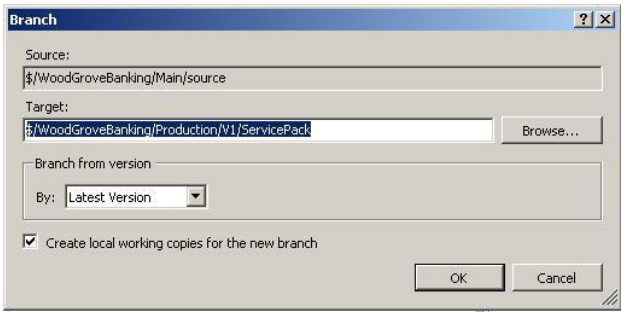


Figure 2.4 Branch to V1 Service Pack

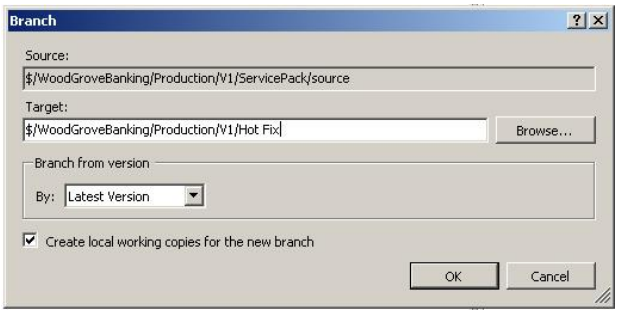


Figure 2.5 Branch to V1 Hot Fix

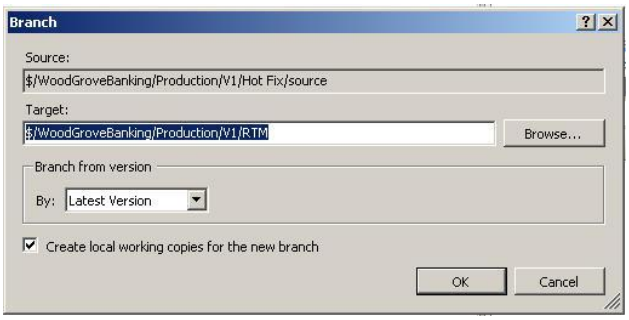


Figure 2.6 Branch to V1 RTM

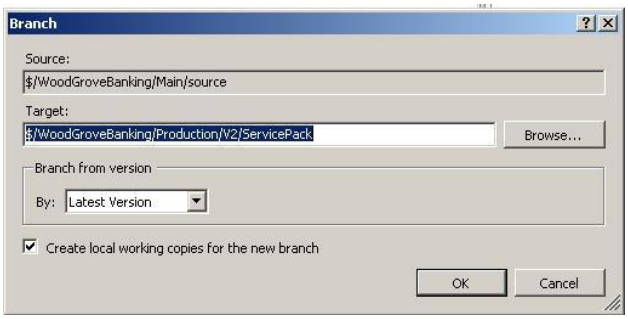


Figure 2.7 Branch to V2 Service Pack

Important Scenario Considerations:

This is a specific branch plan that enables concurrent hot fix, service pack and v-next releases. Each branch is associated with a release vehicle to give the developers checking in a clear map of where their changes should be checked in depending on what release they are working on.

This plan also allows for a single check-in to have a clear merge path to main to reduce regressions and merge conflicts.

Any changes for the V1 release happen in the appropriate branches based on the specific criteria below.

- Ship stopping bug fix for v1 RTM - check in to production/V1/RTM
- The RTM branch is marked read only after release.
- Hot fix for V1 release - check in to production/V1/Hot Fix
- SP 1 work - check into production/V1/Service Pack
- V2 work - check in to the development branch

Changes should be RI'd in from RTM->HF->SP->Main only. This will ensure future releases are regression free as fixes will integrate into the next cumulative release (i.e. all hot fixes will be merged into the next service pack). **Do not** FI changes from main to the Service Pack after the initial release branches are created. An exception may be for merge specific change sets as needed for a Hot Fix or Service Pack release.

Scenario #3: Branching and Labeling Scenario

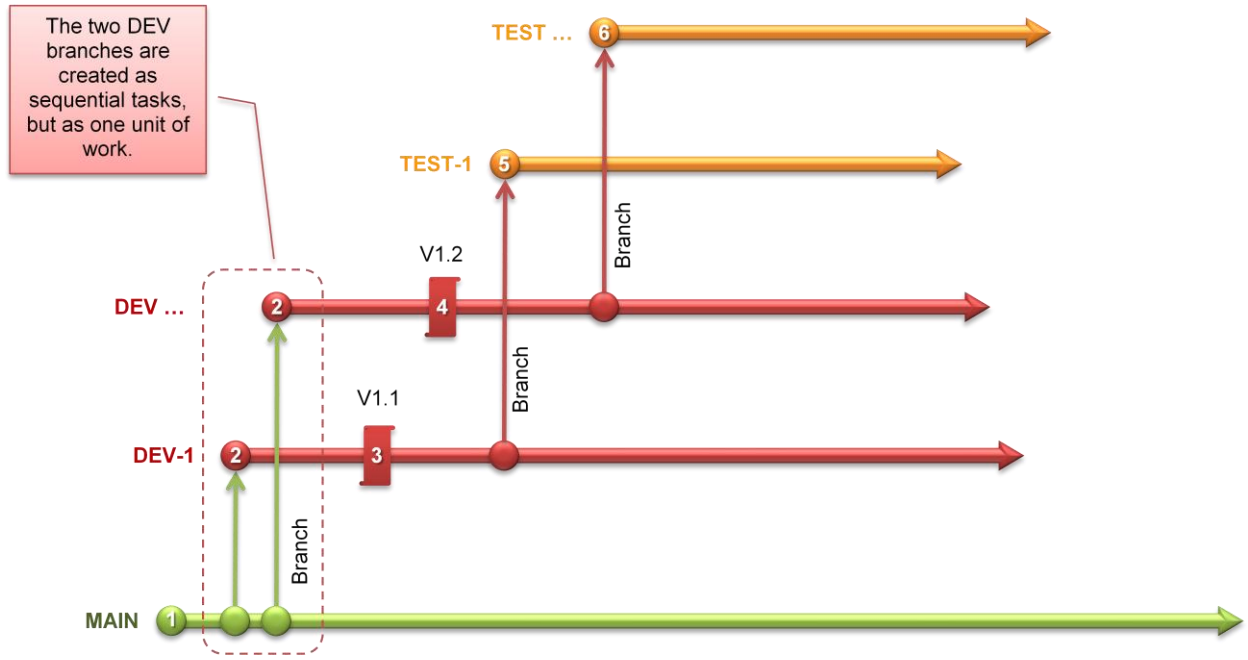
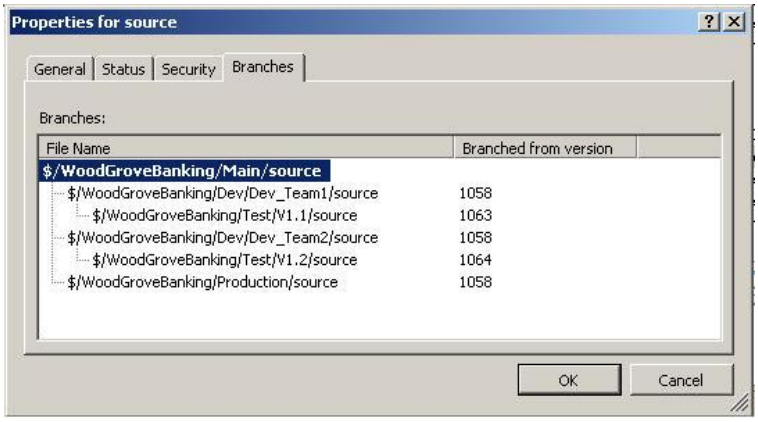


Figure 2.1 Branching and Labeling Scenario



Scenario Overview: Branching with Labels

Organizations that need a good deal of stability should consider create branches to capture specific versions of code that went into a release. Utilizing this type of branching strategy makes it relatively easy for a developer to be able to pull down a specific version of code and perform maintenance on that version. Creating a branch from specific versions of source allows the creation of branches for maintenance as needed instead of as a side-effect of every release. When branching by version using Team Foundation Server it's important to recognize the value that labels provide.

Labels in Team Foundation Server are a powerful tool that allow for a development team to be able to quickly identify files at specific version levels. Utilizing labeling as part of a branching strategy

allows developers to be able to isolate code from subsequent changes that may have happened in the source control system. For example if maintenance is needed on a specific version, a branch can be created at any time in the future based on a label. This can lead to a cleaner version control folder structure as new folders are only created as needed.

In this scenario let's look at how an organization manages version control across different environments where different branches are used to represent the different environments of development, test, and production. When labeling is applied on top of this branching structure, labels allow developers to more easily move files into the branches and then allow for easier merging of files and file versions from on one branch to another.

Branch Definitions for this scenario:

- **Dev** - child branches of main where developers work on the current on future versions on the product.
- **Main** - this is the junction between the development and production branches. For most organizations this branch should be very stable.
- **Production** - child branches of main where servicing (i.e. hot fix and SP) check-in's are made.
- **Test** – this is the where the quality assurance team will test the application at various internal releases.

Scenario Walkthrough:

1. The initial source code is checked into TFS in the [Main] Branch
2. A Dev Branch is created with the relevant team branches from [Main].
   - The first branch is: $/WoodGroveBanking/Dev/Dev_Team1
   - The second branch is: $/WoodGroveBanking/Dev/Dev_Team2
3. Team1 reaches the first milestone and labels the contents of their branch **Dev_Team1** with label **V1.1**
4. Team2 reaches the second milestone and labels the contents of their branch **Dev_Team2** with label **V1.2**
5. **Dev_Team1** branches by label into the Test Branch in order to allow QA to test their teams work.

   Branch by label using the V1.1 Label

   $/WoodGroveBanking/Dev/Dev_Team1 to $/WoodGroveBanking/Test/V1.1
6. **Dev_Team2** branches by label into the Test Branch in order to allow QA to test their teams work.

   Branch by label using the V.1.2 Label

   $/WoodGroveBanking/Dev/Dev_Team2 to $/WoodGroveBanking/Test/V1.2

Structure in Team Foundation Server

To demonstrate this branching pattern in practice using Team Foundation, the source control structure and branching structure would be displayed as follows.

Figure 3.1 Scenario Source Structure

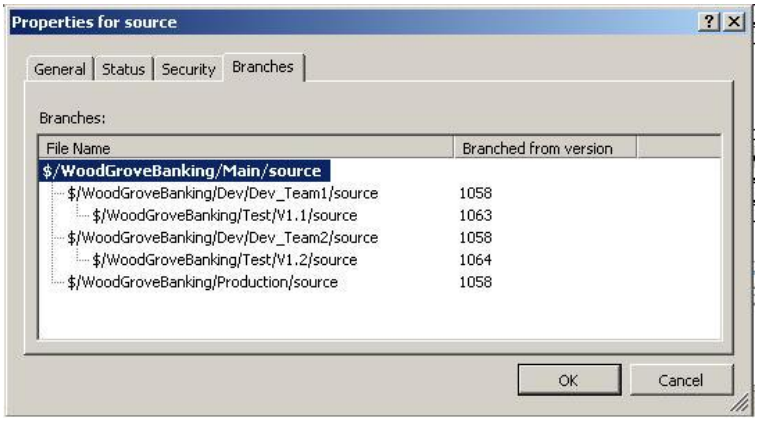


Figure 3.2 Scenario Branching Structure

Scenario Source Control Artifacts:

This section describes the artifacts relating to the implementation of the scenario in Team Foundation Server.

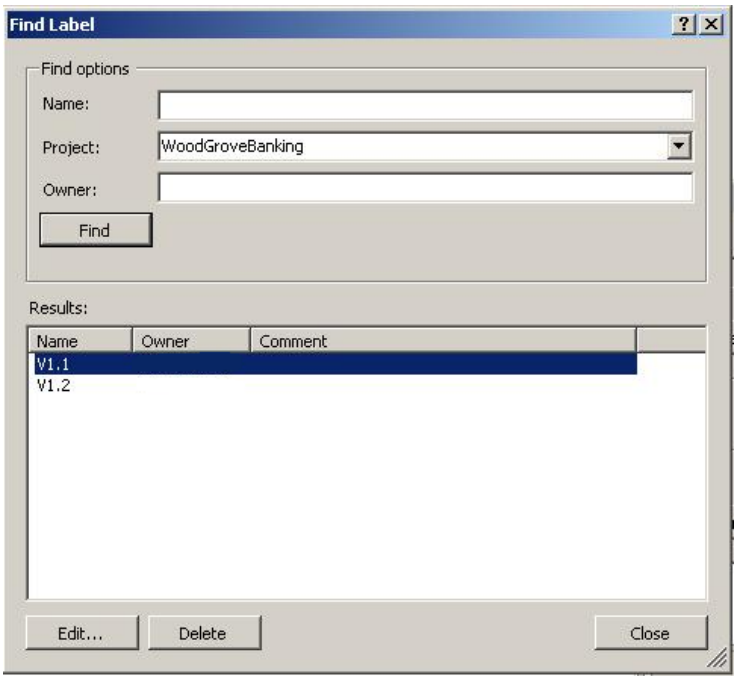


Figure 3.3 Dev_Team1 Find Label for V1.1

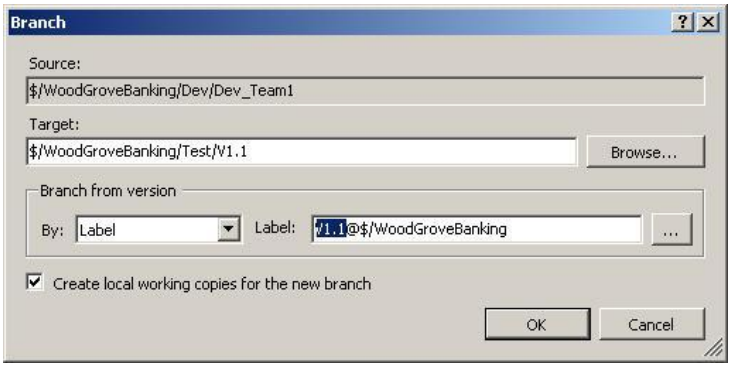


Figure 3.4 Dev_Team1 Branch by Label for V1.1

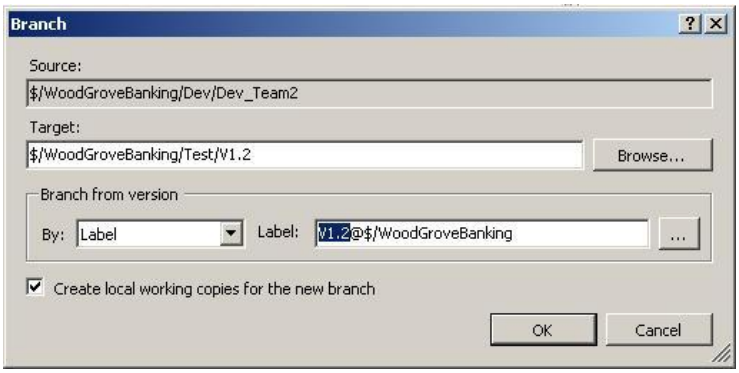


Figure 3.5 Dev_Team2 Branch by Label for V1.2

Important Scenario Considerations:

Keep in mind that files deleted during the development cycle will need to be manually flagged and deleted during the merge process as labels don't track deletes.

It’s important to caution against the over-use of labels given that:

1. Team Foundation Server does not retain a history of changes made to the label.
2. Given certain permissions, labels might be deleted or otherwise invalidated by changes, and there is no way of auditing those changes.
3. There can be contention for a given label if more than more person wants to use and modify the label or the files contained in the label.
4. As such, labels should only be used in cases where a snapshot of sources is needed and if it’s possible to guarantee via permissions that the label won’t be changed.

Environment and Build Location Considerations for Scenarios

IMPORTANT NOTE: Team build scripts (i.e. TeamBuildTypes) should be considered first-class artifact that should be isolated from all other branches in the source explorer tree. It should be isolated from the actual source code, preferably in a directory that is a sibling of the "source" directory of a given branch. For example, in the Main branch, you should have both **$/WoodGrooveBanking/Main/source** and **$/WoodGrooveBanking/Main/build.** As branches are forked off of main, the build scripts will travel alongside with the source code at the same relative path ensuring stability of builds regardless of the branch.

In addition to version control isolation, i.e. branch, we need to be able to deploy the software to corresponding hardware environments for previewing the state of the software at various stages and for supporting applications in production. The motivation for creating separate environments is the same as the motivation or creating branches: to provide isolation from changes to the software in a controllable fashion. Most enterprises typically have one or more of the following environments:

- **Development** – Environment where all of the development takes place. This is the most volatile environment since all changes take place in this environment. The development environment should reflect the latest version of the software that is stabilized and available in MAIN.
- **Test** – All testing by testers occurs in this environment. Usually only stable —drops (often bi-weekly) are delivered to the Test environment which allows the development team to focus on reaching a testable state before dropping to the Test environment.
- **User Acceptance Testing (UAT)** – The UAT environment is more stable than Dev or Test. It is often used for conducting demos and is the environment that project stakeholders use to test the software to reach sign off.
- **Staging/Pre-Production** – Not all enterprises use this environment because the cost of hardware associated with this environment can be high. The Staging/Pre-Production environment should have hardware which is identical or very similar to the Production environment so that issues that are only manifested in Production can be identified early (such as database clustering issues, Web farm issues, firewall issues, etc.)
- **Production** – The environment that supports the production use of the application.
- **Maintenance** – Once the application is released and the development team continues building the next major version of the application or product, the Maintenance environment is used to test hotfixes and changes made against the maintenance release code base.

Environment to Branching Structure Mapping

The table below depicts which branches are typically used to produce releases for each of the environments mentioned above.

Environment	Branch	Note
Development	DEV	Usually the DEVELOPMENT environment will reflect the latest good build from one of the DEV branch.
Test	Usually MAIN, sometimes a DEV branch	Usually the Test environment will contain a build of the software from the MAIN line. The Test environment at times can also contain a build from a DEV branch.
UAT	Usually MAIN, sometimes PRODUCTION when previewing a software release that needs to pass UAT.	In most cases the UAT environment will be running a high quality build from the MAIN line.
Staging/Pre-Production	PRODUCTION	Pre-Production will almost always have a build from the PRODUCTION line.
Production	PRODUCTION	PRODUCTION always contains a build from the PRODUCTION line.