CollabNet Subversion 1.9 for Configuration Managers

Lab Exercises

COLLABNET

4000 Shoreline Court, Suite 400
South San Francisco, California 94080 U.S.A.

888.778.9793 toll free
650.228.2500 voice
650.228.2501 fax
www.collab.net
E-mail info@collab.net

Table of Contents

1. Essential Concepts 3

1.1. Basic repository operations 3
1.2. Restructuring a repository 3
1.3. Mapping svn:external 4

2. Best Practices 5

2.1. Background 5
2.1.1. Your product 5
2.1.2. Your team 5
2.1.3. General lab rules 5
2.2. Scenarios to consider 6
2.2.1. How would you organize your infrastructure to work effectively? 6
2.2.2. What should be? your branching/merging scheme? 6
2.2.3. What if some additional requirements were added? 6
2.2.4. How would you handle recording your promotional process? 6
2.2.5. How would you define roles and permissions? 7

3. Solutions for Lab 1 – Essential Concepts 1 9

3.1. Suggested Solution for Basic Repository Operations 9
3.2. Suggested Solution for Restructuring a Repository 10
3.3. Suggested Solution for Mapping svn:external 12

4. Solutions for Lab 2 – Best Practices 14

4.1. Suggested Solution for Scenarios to Consider 14
4.1.1. How Would You Organize Your Infrastructure to Work Effectively? 14
4.1.2. What Should be Your Branching/Merging Scheme? 14
4.1.3. What if Some Additional Requirements Were Added? 15
4.1.4. How Would You Handle Recording Your Promotional Process? 15
4.1.5. How Would You Define Roles and Permissions? 16

1. Essential Concepts

1.1. Basic repository operations

Steps	Comments
1. Create a new repository named *CMlab*.	Optional (based on your ability to execute this step)
2. Create branches, tags and trunk directories in your repository.	Structural change.
3. Import an existing structure into your trunk.	Structural change.
4. Create a branch from trunk named *release1*.	Structural change.
5. Create a tag of trunk's HEAD and name it *Rel1.0*.	Snapshot.

1.2. Restructuring a repository

Steps	Comments
1. Create top level directories named *module1*, *module2* and *module3*.	Structural change.
2. Create *branches*, *tags* and *trunk* directories in each of the top level directories you created in step 1.	Structural change.
3. Create *Dev* and *SCM* directories in each *tags* directory.	Structural change.
4. Move the appropriate module directory under the repositories top level *trunk* directory to the appropriate module's *trunk* directory (e.g., */trunk/module1* to */module1/trunk*).	Structural change.
5. Move the appropriate module directory under the *release1* branch to the appropriate module's *branches* directory (e.g., */branches/release1/module1* to */module1/branches/release1*).	Structural change.

6. Move the appropriate module directory under the *Rel1.0* tag to the appropriate module's *tags* directory (e.g., */tags/Rel1.0/module1* to */module1/tags/Rel1.0*).	Structural change.
7. Delete the top level *branches*, *tags* and *trunk* directories.	Structural change.

1.3. Mapping svn:external

Steps	Comments
1. If you haven't created a working copy previously, then create one checking out module 1's trunk.	
2. Add the svn:external property to module1's trunk directory creating a *mod2* directory by mapping in the *Rel1.0* tagged revision of module2.	Property setting.
3. Commit the change.	
4. Update your working copy and validate the external works.	Structural change.

2. Best Practices

2.1. Background

2.1.1. Your product

Your company, ***Brilliant Ideas Inc.***, develops a product that consists of:

- 5 standard components that you use as is,
- 2 modified libraries, and
- 4 main components that contains your added value.

(Your development owns the 4 components and maintains patches on 2 libraries.)

Releases happen every 6 months with development overlapping. Bug fixes are delivered as patches. Your product is sold to external customers and deployed as a BTF (behind-the-firewall) application.

2.1.2. Your team

Your development team is distributed across three global sites:

- Amsterdam, The Netherlands: Team that started the product 8 years ago and "owns" it. Currently 50 software engineers and 10 QA engineers. Focussed on developing 3 components – 2 of which are fairly mature and stable and 1 is brand new with new features – and integrating and testing the product as a whole.
- Bangalore, India: Team has been involved for 5 years. Currently 35 developers and 20 QA engineers. Focussed on integrating and testing the 5 standard components and the 2 libraries as well as the patches on the modified libraries.
- Sunnyvale, California. Outsource partner, currently 10 developers and 2 QA engineers. Focussed on 1 component that delivers a set of brand new features.

2.1.3. General lab rules

- Team up in groups of four
- There are no right or wrong answers – just different opinions, though best practices might lead you to particular choices
- One person from each group will present their result from each part of the lab

2.2. Scenarios to consider

2.2.1. How would you organize your infrastructure to work effectively?

- Infrastructure: how many servers and where and what about business continuity?
- Repository scheme: Single or multiple repositories and what goes in those repositories?
- Responsibilities for the infrastructure assignments (e.g. development, QA, integration/release mgt)?
- Try to finish in 30 minutes!

2.2.2. What should be? your branching/merging scheme?

- Branching scheme?
- Responsibilities for creating and using the branches (e.g. development, QA, integration/release mgt)?
- Try to finish in 15 minutes!

2.2.3. What if some additional requirements were added?

Your organization changes the product to be a SaaS (Software-as-a-Service offering) and each team needs to use one of the libraries from logically within multiple components. You also find that you've got some features that take a considerable amount of time to develop and put other work at risk. In addition, your customers are now requesting customizations for their deployments.

- Infrastructure: any changes?
- Branching scheme: any changes?
- Responsibilities (e.g. development, QA, integration/release mgt): any changes?
- Try to finish in 30 minutes!

2.2.4. How would you handle recording your promotional process?

Individual modules are tested through a release candidate approach much like how Subversion is qualified as an overall product. The complete product/system goes to QA (Quality Assurance), then to UAT (User Acceptance Testing), and System Testing before being declared production worthy. Any failure in one stage causes the process to start again with QA. Developers also have the occasional need to tag revisions.

- Infrastructure: any changes?

- Promotional process: what tags are applied when?
- Responsibilities (e.g. development, QA, integration/release mgt): any changes?
- Same groups, 20 minutes to execute the exercises

2.2.5. How would you define roles and permissions?

- Fill in the table below outlining what role should have what read and write permissions on repositories (you may only have one), the trunk branch, specific branches (all the types defined to this point), specific directories, and tags (there may be multiple types). Keep in mind that every role may not have unique permissions and it is up to you to define reasons for the permissions you give or don't give to a specific role.

Role	Description	Repositories	Trunk	Branches	Directories	Tags
Dutch Developer	Company developer in The Netherlands					
Indian Developer	Company developer in India					
U.S. Developer	Outsource developer in California					
Dutch Team Lead	Development lead in The Netherlands					
Indian Team Lead	Development lead in India					
U.S. Team Lead	Development lead in California					
Dutch QA	Quality assurance engineer in The Netherlands					
Indian QA	Quality assurance engineer in India					
U.S. QA	Quality assurance engineer in U.S.					
Dutch Release	Formal build and process tagging					

Engineer	engineer in The Netherlands					
Indian Release Engineer	Formal build and process tagging engineer in India					
Configuration Manager	Configuration management team based in The Netherlands					
Management	Any type of management that might have an interest in product development					

3. Solutions for Lab 1 – Essential Concepts 1

3.1. Suggested Solution for Basic Repository Operations

Steps	Comments
1. If using command line, 'svn admin create ***/repos/CMlab***'. If using TortoiseSVN, create a directory and then right click on it selecting 'TortoiseSVN->Create repository here'.	Optional (based on your ability to execute this step)
2. If using command line on server (otherwise change prefix to either http, https, svn or svn+ssh), 'svn mkdir ***file:///repos/CMlab/branches*** ***file:///repos/CMlab/tags*** ***file:///repos/CMlab/trunk*** ' If using TortoiseSVN, start the repo browser (TortoiseSVN->Repo-browser supply the URL for your repository as done with the command line above), then right click on the URL and select 'Create folder' entering in '***branches***' and click on OK. Repeat for ***tags*** and ***trunk***.	
3. If using command line, 'svn import –m "Importing data" /target/dir' If using TortoiseSVN, then right click on the directory you wish to import and select 'TortoiseSVN->Import...". Enter your log message and hit OK.	
4. If using command line, 'svn copy file:///repos/CMlab/trunk file:///repos/CMlab/branches/release1 -m "Creating release1 branch' (use the appropriate URL for your repository). If using TortoiseSVN, then use the repo browser and right click on the trunk selecting "Copy to…". Enter the appropriate URL (e.g., file:///repos/CMlab/branches/release1), click OK, enter in your log message and click OK.	
5. If using command line, 'svn copy file:///repos/CMlab/trunk file:///repos/CMlab/tags/Rel1.0 -m "Creating release1 tag' (use the appropriate URL for your repository).	

If using TortoiseSVN, then use the repo browser and right click on the trunk selecting “Copy to…”. Enter the appropriate URL (e.g., file:///repos/CMlab/tags/Rel1.0), click OK, enter in your log message and click OK.	

3.2. Suggested Solution for Restructuring a Repository

Steps	Comments
1. If using command line on server (otherwise change prefix to either http, https, svn or svn+ssh), ‘svn mkdir ***file:///repos/CMlab/trunk/module1*** ***file:///repos/CMlab/trunk/module2*** ***file:///repos/CMlab/trunk/module3*** ’ If using TortoiseSVN, use the repo browser, then right click on the trunk and select ‘Create folder’ entering in ‘***module1***‘ and click on OK. Repeat for ***module2*** and ***module3***.	
2. If using command line on server (otherwise change prefix to either http, https, svn or svn+ssh), ‘svn mkdir ***file:///repos/CMlab/module1/branches*** ***file:///repos/CMlab/module1/tags*** ***file:///repos/CMlab/module1/trunk*** ***file:///repos/CMlab/module2/branches*** ***file:///repos/CMlab/module2/tags*** ***file:///repos/CMlab/module2/trunk*** ***file:///repos/CMlab/module3/branches*** ***file:///repos/CMlab/module3/tags*** ***file:///repos/CMlab/module3/trunk*** ’ If using TortoiseSVN, use the repo browser, then right click on the URL for module1 and select ‘Create folder’ entering in ‘***branches***’ and click on OK. Repeat for ***tags*** and ***trunk***. Right click on the URL for module2 and then module3 repeating the process done for module1.	
3. If using command line on server (otherwise change prefix to either http, https, svn or svn+ssh), ‘svn mkdir ***file:///repos/CMlab/module1/tags/Dev*** ***file:///repos/CMlab/module1/tags/SCM*** ***file:///repos/CMlab/module2/tags/Dev*** ***file:///repos/CMlab/module2/tags/SCM***	

file:///repos/CMlab/module3/tags/Dev
file:///repos/CMlab/module3/tags/SCM '

If using TortoiseSVN, use the repo browser, then right click on the URL for module1/tags and select 'Create folder' entering in '***Dev***' and click on OK. Repeat for ***SCM***. Right click on the URL for module2/tags and then module3/tags repeating the process done for module1.

4. If using command line, 'svn move file:///repos/CMlab/trunk/module1 file:///repos/CMlab/module1/trunk -m "Moving module1 to new structure' (use the appropriate URL for your repository). Repeat for module2 and module3.

   If using TortoiseSVN, then use the repo browser and left click (and hold) on trunk/module1 and drag & drop to module1/trunk. Enter the appropriate log message and click OK. Repeat for module2 and module3.

5. If using command line, 'svn move file:///repos/CMlab/branches/release1/module1 file:///repos/CMlab/module1/branches/release1 -m "Moving module1 release 1 branch to new structure' (use the appropriate URL for your repository). Repeat for module2 and module3.

   If using TortoiseSVN, then use the repo browser and left click (and hold) on branches/release1/module1 and drag & drop to module1/branches/release1. Enter the appropriate log message and click OK. Repeat for module2 and module3.

6. If using command line, 'svn move file:///repos/CMlab/tags/Rel1.0/module1 file:///repos/CMlab/module1/tags/Rel1.0 -m "Moving module1 Rel1.0 tag to new structure' (use the appropriate URL for your repository). Repeat for module2 and module3.

   If using TortoiseSVN, then use the repo browser and left click (and hold) on tags/Rel1.0/module1 and drag & drop to module1/tags. Enter the appropriate log message and click OK. Repeat for module2 and module3.

7. If using command line, 'svn delete file:///repos/CMlab/branches file:///repos/CMlab/tags file:///repos/CMlabs/trunk'. If using TortoiseSVN, then use the repo browser and right click on the top level branches and select Delete. Repeat for /tags and /trunk.	

3.3. Suggested Solution for Mapping svn:external

Steps	Comments
1. If you haven't created a working copy previously, then create one checking out module 1's trunk. If you are using command line, then 'svn checkout file:///repos/CMlab/module1/trunk .' Use the appropriate URL and make sure you're in an empty directory where you want the working copy. If you're using TortoiseSVN, then right click on the directory where you want to put the working copy and select 'SVN Checkout'. Enter the appropriate URL (e.g., file:///repos/CMlab/module1/trunk) and confirm the directory that you want to checkout to, then click on OK.	
2. If you're using the command line, 'svn propset svn:externals file:///repos/CMlab/module2/tags/Rel1.0 mod2 .' at the top of your working copy (using the appropriate URL). If you're using TortoiseSVN, then right click on the working copy's top level directory and select 'TortoiseSVN->Properties'. Click on New and select svn:externals from the pull-down list. Enter file:///repos/CMlab/module2/tags/Rel1.0 as the value and click on OK. Click on OK again.	
3. If you're using the command line, 'svn commit –m "Adding external to module2 Rel1.0 revision"'. If you're using TortoiseSVN, then right click on the working copy's top level directory and select "SVN Commit". Enter an appropriate comment and validate that just the directory is going to be commited and click on OK.	

4.	If you’re using the command line, ‘svn update’. If you’re using TortoiseSVN, right click on the working copy’s top level directory and Select “SVN Update”. In either case, validate that the new directory, mod2, exists and is populated.

4. Solutions for Lab 2 – Best Practices

4.1. Suggested Solution for Scenarios to Consider

4.1.1. How Would You Organize Your Infrastructure to Work Effectively?

Subversion was designed to work over the WAN so a single repository server would probably be the preferable approach. Alternatively, there could be a repository for the component developed by the outsource partner in the U.S. which could be replicated in Amsterdam.

Business continuity is best handled by replicating the server so that there is alternative location where code can be accessed if the primary server is down for more than a few minutes, but less than what it takes to bring up a new server. Best practice is just to use this replica in read-only mode while the primary server is down in this case. The replica might be placed in Bangalore so that if the failure was more than just a server in Amsterdam, then at least the rest of the team could access the replica.

If one was to look at disaster recovery, then replication is a good approach for that as well with procedures defined as to how to make a replica the master and how to change the DNS mapping to make the switchover as painless as possible for users.

There really isn’t enough information to definitively determine whether multiple repositories are best and ultimately such decisions are not absolute. In general, it is best to start with a single repository if that seems at all appropriate and then be able to break it up if needed as more experience is gathered. If a separate server is established in the U.S. for their work on a module, then logically that module would probably be in its own repository and potentially exposed in the larger one via svn:externals. Other repositories might be established for artifacts for non-developer project team members, but that might also be accomplished via separate projects within the same repository.

Configuration management should really “own” the infrastructure decisions along with any assigned system administrators. Business continuity and disaster recovery approaches should be “owned” by the system administrators with input from configuration management.

4.1.2. What Should be Your Branching/Merging Scheme?

Since there is overlapping releases, this would seem to be a great candidate for the stable trunk approach where branches are created for individual releases and merged back into the trunk when the release is completed. Bug fixes could be continuations of the release branches (assuming appropriate measures are taken to allow for multiple reintegration efforts) or new branches established off of the release merge point in the trunk. Bug fixes should be merged into other ongoing releases (both under support and under development), but can not be blindly merged as an overwrite back into the trunk.

Merging should be done periodically from the assumed predecessor release branch. This allows for the cascading of changes rather than having to worry about merging a change to multiple other branches and allows the release team to determine when they want to take the potential impact of merging such changes into their on-going work.

Things can be more complicated than what is documented here depending on other use cases that are not explicitly defined in the lab and maybe assumed in multiple ways.

4.1.3. What if Some Additional Requirements Were Added?

The change from BTF to SaaS might make it possible to support a single release of the product, but only if the product is handled in a multi-tenant manner. If individual customers get their own instantiation, then it is likely that multiple versions will have to be supported. Again, this may mean less supported versions, but it really causes no change in what we discussed to this point.

The need to use a library from within multiple components would best be satisfied by the use of svn:externals which would allow the shared library to be structurally placed appropriately to the individual components utilizing it when a working copy is checked out.

Features that bring risk to the overall development work call for isolation of that work on feature branches. This allows developers to work and to version control their work along the way without impacting other work. Such branches would come off of the associated feature branch and be merged back into it either when the work is completed or when there is no perceived risk from additional work being checked into the release branch. There is the flexibility that should a feature not be ready when a release is desired, the feature branch could be merged into the next release branch. Merging should be done as often as possible from the release branch to the feature branch in order to keep them in sync. And where possible, merging should be done as often as possible from the feature branch to the release branch.

Customer customizations require the use of customer branches. These would be established from the trunk based on the product release that the customer initially received. As the product continues to evolve and the customer is willing to take releases, the trunk should be merged into the customer branch. If certain features or defect fixes are determined to be candidates for the general product, then the specific revisions involved on the customer branch should be merged into the designated release branch.

Feature branches can either be established by the team lead or by the individual developer who recognizes the need for them. More commonly it is the latter approach that is used in order to facilitate getting to work quicker. Merging should be the responsibility of the developer using the feature branch unless it is a team and then that responsibility may be the team lead's. The creation of customer branches should be the responsibility of the team lead and merges from the trunk should be under their direction. Merging features from the customer branches should also be driven by the team lead.

4.1.4. How Would You Handle Recording Your Promotional Process?

Module tagging would be the responsibility of the release engineer for the team developing that module. In the case of the U.S. team, that may be a shared responsibility or one held by the team lead. The module should be built and the source tagged with a release candidate number and the release number (e.g., RC1_Rel1.0_1 where the final number is an indication of how many builds it took to get this release candidate). When a release candidate has been determined to be production worthy, then the release candidate should be tagged to indicate that (e.g., Prod_Rel1.0). These tags should be in the tags/SCM directory.

Product tagging would be the responsibility of the Amsterdam release engineer. The system should be built and the source tagged with a QA tag (e.g., QA1_Rel1.0_1). If that revision passes QA, then it would be tagged with a UAT tag (e.g., UAT1_Rel1.0). If it failed QA, then it would be sent back for rework and the next candidate from development would get a new QA tag (e.g., QA2_Rel1.0_1) and if it passed QA, then a new UAT tag (e.g., UAT2_Rel1.0). If the UAT stage is passed, then the revision would get a system test tag (e.g., ST_Rel1.0). If the system test stage is passed, then the revision would get a production worthy tag (e.g., Rel1.0). It would also be helpful (if possible) to identify what actually has made it into production versus being production worthy so an additional tag may be used or the production tag not applied until the revision is put into production (deployed to customers or deployed to a server). These tags should be in the tags/SCM directory.

Developers can apply tags when they feel it is necessary and nomenclature can probably be left up to the individual developer versus standardized. They would place such tags in the tags/Dev directory. Note that there may be little to no need for such tags given the global revision approach taken by Subversion.

4.1.5. How Would You Define Roles and Permissions?

The table below is a possible answer to the question given the information provided. There could be alternatives that would still satisfy the requirements and thus be correct answers. See your instructor to validate your answer.

Role	Repositories	Trunk	Branches	Directories	Tags
Dutch Developer	Assuming 1, read access to everything	Write access (could be limited to just the modules the Dutch team is expected to modify)	Write access (could be limited to just the modules the Dutch team is expected to modify)	No limits (could control write access to QA and rel eng)	Write access to tags/Dev (could be limited to just the modules the Dutch team is expected to modify)
Indian Developer	Assuming 1, read access to everything	Write access (could be limited to just the modules the Indian team is expected to modify)	Write access (could be limited to just the modules the Indian team is expected to modify)	No limits (could control write access to QA and rel eng)	Write access to tags/Dev (could be limited to just the modules the Indian team is expected to modify)
U.S. Developer	Assuming 1, read access to be limited	Write access to just the module the U.S. team is expected to modify	Write access limited to just the module the U.S. team is expected to modify	No limits (could control write access to QA and rel eng)	Write access to tags/Dev (could be limited to just the modules the U.S. team is expected to modify)
Dutch Team Lead	Assuming 1, read access to everything	Write access (could be limited to just the modules the Dutch team is expected to	Write access (could be limited to just the modules the Dutch team is expected to modify)	No limits (could control write access to QA and rel eng)	Write access to tags/Dev (could be limited to just the modules the Dutch team is expected to

		modify)			modify)
Indian Team Lead	Assuming 1, read access to everything	Write access (could be limited to just the modules the Dutch team is expected to modify)	Write access (could be limited to just the modules the Indian team is expected to modify)	No limits (could control write access to QA and rel eng)	Write access to tags/Dev (could be limited to just the modules the Indian team is expected to modify)
U.S. Team Lead	Assuming 1, read access to be limited	Write access to just the module the U.S. team is expected to modify	Write access limited to just the module the U.S. team is expected to modify)	No limits (could control write access to QA and rel eng)	Write access to tags/Dev (could be limited to just the modules the U.S. team is expected to modify)
Dutch QA	Assuming 1, read access to everything	Write access not broadly granted	Write access not broadly granted	Write access only to QA directory (could be an external to a separate repository where control would be easier)	No write access
Indian QA	Assuming 1, read access to everything	Write access not broadly granted	Write access not broadly granted	Write access only to QA directory (could be an external to a separate repository where control would be easier)	No write access
U.S. QA	Assuming 1, read access to be limited	Write access not broadly granted	Write access not broadly granted	Write access only to QA directory (could be an external to a separate repository where control would be	No write access

				easier)	
Dutch Release Engineer	Assuming 1, read access to everything	Write access not broadly granted	Write access not broadly granted	Write access only to rel eng directory (could be an external to a separate repository where control would be easier)	Write access to tags/SCM (could be limited to just the modules the Dutch team is expected to modify)
Indian Release Engineer	Assuming 1, read access to everything	Write access not broadly granted	Write access not broadly granted	Write access only to rel eng directory (could be an external to a separate repository where control would be easier)	Write access to tags/Dev (could be limited to just the modules the Indian team is expected to modify)
Configuration Manager	Assuming 1, read access to everything	Write access to all modules	Write access to all modules	No limits	Write access to all tags
Management	Assuming 1, read access to everything	No write access	No write access	No write access	No write access