mer. 22 septembre 2010

Gérer les versions de son code avec Bazaar

L'évolution du code (ajout de fonctionnalité, réécriture de modules, recherche d'erreurs) dans un projet de programmation crée le besoin d'enregistrer des versions successives. Une façon de procéder est de donner un nouveau nom de fichier à chaque amélioration de son contenu. Vous pourriez également noter dans un fichier texte à quoi correspondent chacune de ces versions. Ce tutoriel vous apprend à le faire dans un **gestionnaire de versions**. Le gestionnaire surveille les modifications de votre code et vous autorise à ne conserver qu'un seul fichier de manière apparente. Ainsi vous n'avez plus de copies du fichier originel avec des noms différents à chaque version.

Auparavant, lorsque vous vouliez finalement utiliser un ancien morceau de code à la place de votre nouveau code qui s'est avéré ne pas fonctionner, il fallait ouvrir l'ancien fichier, y reprendre le code intéressant puis le coller dans le nouveau fichier à la place du code qui ne fonctionnait pas. **Retourner à un état antérieur** est plus simple avec le gestionnaire de versions : il suffit de lui dire de revenir à une version précédente précise et le contenu de votre fichier sera changé automatiquement.

Enfin, il vous arrive sans doute de **développer en parallèle une version du code** avec des fonctionnalités hasardeuses et instables tandis que la branche principale ne voit que des corrections mineures. Lorsque vos nouvelles fonctions sont prêtes, il est temps d'en faire profiter l'utilisateur en les ajoutant dans la branche principale. Le processus de fusion de la branche parallèle est permis par le gestionnaire de versions.

Dans ce tutoriel, le terme *version* désigne les états successifs d'un fichier (par exemple, ajouter une ligne de code à un fichier et enregistrer celui-ci aboutit à une nouvelle version). Le terme de *révision* s'applique à l'état des fichiers d'une branche surveillés par le gestionnaire. Chaque révision est enregistrée avec une commande de soumission du gestionnaire. Chaque révision contient la dernière version enregistrée des différents fichiers surveillés. Le terme de *journalisation* est appliqué aux fichiers surveillés par le gestionnaire. Ces conventions de nommage sont restreintes au cadre de ce document et on lit souvent d'autres termes issus de la traduction approximative de l'anglais lorsque l'on cherche des informations sur Bazaar.

1. Initialisation de la branche

Créer un répertoire de projet. Aller dans ce répertoire. Dans ce répertoire on place la commande

```
bzr init
```

```
Created a standaline tree
```

Le répertoire est versionné en branches.

2. gérer son identité.

```
bzr whoami « Prénom Nom <adresse@privder.com> » (car le mail est unique)
```

On peut la vérifier :

bzr whoami

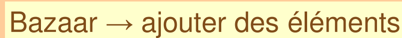
bzr help renvoit les commandes disponibles.

Imaginons que nous avons fait un script dans ce répertoire.

3. Ajouter le script à la journalisation

Bzr add script.py

ou en interface graphique :

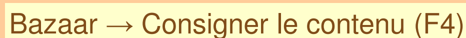


Il faut ensuite enregistrer le script dans le gestionnaire de version.

4. Enregistrer la version courante du script dans le gestionnaire

bzr commit -m "Premiere soumission du fichier"

ou en interface graphique :




```
Commited revision 1
```

l'option -m est obligatoire : expliquer explicitement pourquoi on crée une nouvelle révision.

Bzr status montre si on a modifié le script.

```
modified script.py
```

Le script apparaît uniquement s'il a été modifié et enregistré (dans gedit par exemple). Autrement Bazaar détecte qu'on a pas modifié le script depuis la dernière version et ne l'affiche pas (nettoyage de l'affichage pour ne montrer que les fichiers où l'action *bzr* présente un intérêt).

Lorsque l'on est arrivé à une version satisfaisante (la modification du script est terminée), je dois l'enregistrer dans le gestionnaire en produisant une nouvelle révision de la branche.

5. Enregistrer la nouvelle version dans le gestionnaire : créer une révision de la branche

bzr commit -m "Amélioration de la fonction bidule"

Il est important de comprendre que comme Bazaar surveille si vous avez modifié et enregistré une nouvelle version de votre fichier, cette commande retourne une erreur si vous tentez de publier une nouvelle révision alors qu'aucun des fichiers ajoutés à la journalisation par la commande bzr add n'a changé.

Il faut très souvent soumettre de nouvelles versions des fichiers modifiés pour produire de nouvelles révisions. C'est la garantie que vous pourrez revenir en arrière pas à pas si besoin est.

6. Contrôler l'évolution des révisions

Bzr log pour voir les différentes versions

Pour comparer les modifications entre deux versions successives:

bzr diff -r1..2

Bazaar → montrer les différences (F3)

Cette action compare la révision 1 et la révision 2.

Si, de la révision 1 à la révision 2, un fichier a été ajouté, le résultat de la commande diff sera l'affichage du fichier ajouté. Si un fichier présent à la révision 1 a été modifié avant la révision 2, diff affichera le fichier ajouté (comme précédemment) et les différences dans le fichier modifié. Si aucun fichier n'a été ajouté mais que la nouvelle révision n'a été décidée que pour enregistrer les changement d'un des fichiers journalisés alors seules les différences pour ce fichier seront affichées.

bzr diff -r..2

bzr diff -r2..

Ces commandes permettent de montrer toutes les différences entre versions à partir de la révision 2 ou bien jusqu'à la révision 2. Le retour de cette commande risque cependant d'être un peu trop complexe pour un usage pratique.

Pour une interface plus claire :

bzr diff -r1..2 –using meld

Tester le script pour éviter les régressions avant tout enregistrement de révision.

Si on s'en rend compte un peu trop tard on enregistre une version régressée. Imaginons que nous avons travaillé sur un script « script.py » qui a fait l'objet de 4 révisions successives. Malheureusement j'ai implémenté une erreur qui rend inopérant le script. Je ne m'en suis pas aperçu et lorsque j'ai soumis les fichiers pour produire la révision 4 avec la commande bzr commit j'ai gravé dans le marbre cette erreur. Je souhaite revenir à la version du fichier tel qu'il figure dans la révision 3. Alors on peut utiliser l'instruction suivante :

Bzr revert script.py -r3

Cette instruction enlève l'erreur commise à la dernière version ! Elle retourne le fichier à son état antérieur, par exemple à l'état de la révision 3 comme montré ici.

Note : on peut retourner dans son éditeur si le script était ouvert. On verra, c'est le cas de gedit, que l'éditeur signale que le fichier a été modifié et qu'il faut le recharger (pour voir son nouvel état) en cliquant sur un bouton qui apparaît alors. On observe que le fichier est restauré dans l'état antérieur choisi avec la fonction bzr revert !

Il faudra ensuite soumettre à nouveau au gestionnaire les fichiers journalisés (parmi lesquels notre fameux script dans sa version sauvée) :

bzr commit -m "script.py : retour rev.3, correction du bug implémenté par erreur à la rev.4"

La révision 5 ainsi produite contient les dernières versions des fichiers (celles contenues dans la révision 4) et le fichier sauvé. La révision 5 contient donc le script tel qui était dans la révision 3 et

tous les autres fichiers tels qu'ils étaient dans la révision 4. La révision 4 (associée à une version régressée de « script.py ») deviendra définitivement inaccessible.

Pour enlever un fichier de la journalisation effectuée par le gestionnaire de versions :

```
bzr remove –keep script.py
```

Pour le supprimer du gestionnaire *et* du disque dur :

```
bzr remove –force script.py
```

La copie de branche

Le développement d'une branche parallèle exige de partir de la dernière révision du projet. Il faut donc copier une branche.

```
cd ..
bzr branch projet/ projet2
```

toute l'historique de projet est copié dans projet2 : permet de développement en parallèle une branche avec des fonctionnalités nouvelles, un peu hasardeuses.

Fusion de branche :

On fusionne deux branches lorsqu'il est temps d'ajouter les nouvelles fonctionnalités développées à côté de la branche disponible aux utilisateurs (à côté de la branche principale pour faire simple). Fusionner les deux branches vise à verser dans la branche principale le code développé dans la branche parallèle.

```
cd ../projet
bzr merge ../projet2
succeeded
```

C'est une manière très puissante de gérer les branches. En effet, le gestionnaire de versions va garder en mémoire les branches parallèles qui ont été issues de la branche parentale pour finalement si rebrancher.

Il n'y a pas de conflit lorsque la branche principale n'a pas été modifiée pendant le développement de la branche parallèle. Bazaar reconnaît quelles sont les lignes de code nouvelles et les verse dans la branche principale. En revanche des conflits vont survenir lorsque la branche principale a aussi évoluée :

```
bzr merge ../projet2
conflicts encountered
```

C'est le cas lorsque des corrections mineures (par exemples de sécurité) sont fournies aux utilisateurs. Les utilisateurs utilisent un programme disponible dans une branche stable, par exemple la branche principale pour reprendre notre phraséologie. Le développement des branches parallèles aboutit au fait qu'à la fusion le gestionnaire de versions est incapable d'identifier avec certitude quel

est le code à verser présent dans la branche parallèle mais absent dans la branche principale. En effet, les conflits apparaissent généralement lors de changements divergents au même endroit dans un fichier.

Dans ce cas, Bazaar laisse un marqueur dans le fichier contenant le conflit, et crée trois nouveaux fichiers contenant les différentes versions du fichier, dont l'ancêtre commun aux deux versions en conflit. Pour voir les fichiers contenant des conflits, faites

```
bzr conflicts
```

Vous devez alors modifier le fichier pour résoudre le conflit et enlever le marqueur.

Résoudre à la main, enregistrement du THIS dans le script

```
cp script.py THIS script.py
```

Lorsque la résolution de conflit est terminée, choisir :

```
bzr resolve
```

puis faire un commit.

```
Bzr export
```

fait une archive où il prend que les fichiers journalisés sans les fichiers intermédiaires.

> BZR EXPLORER : une interface graphique à Bazaar
>
> Son principal avantage est de fournir une représentation graphique des branches (et de leurs histroriques), ainsi que de leurs fusions.

Mots : 1658 Caractères : 10295 Nom : bazaar_frndz Pages : 5