# Conservez l'historique de vos commandes pour chaque projet, le retour

Alban Mancheron

# Conservez l’historique de vos commandes pour chaque projet, le retour

**Alban Mancheron**
**[Enseignant-Chercheur en bioinformatique à l’Université de Montpellier, linuxien depuis 1997 (convaincu depuis 1998)]**

***Pouvoir conserver un historique dédié pour chaque projet, voici l’idée géniale énoncée par Tristan Colombo dans un précédent article de GLMF [1]. Cet article reprend le concept génial (je l’ai déjà dit?) et l’étoffe en simplifiant son installation et en ajoutant quelques fonctionnalités (comme l’auto-détection de projets versionnés pour proposer à l’utilisateur d’activer un historique dédié si ce n’est pas le cas).***

*/// TAG = BASH ///*

*/// Titre Couv  Bash / Open source #7541 : Conservez l'historique de vos commandes pour chaque projet : histoire d'une amélioration  ///*

*/// Mots-clés ///*

Bash, historique des commandes, projet versionné, partage

*/// Fin Mots-clés ///*

Une histoire, ça se raconte. Une belle histoire, elle, ne se raconte pas ; elle se conte , on l’adapte, on la transforme, on se l’approprie et au final, cela en devient une nouvelle version.

Laissez vous conter ma version d’*history-guardian*.

## 1 Histoire d’un jour

Ce prologue pourrait ressembler à une page de publicité pour GLMF, mais il m’apparaît nécessaire pour comprendre ce qui m’a amené à faire évoluer la version présentée dans le numéro 232. Le lecteur pressé pourra passer directement à la partie intitulée « Un jour d’histoire », mais ce serait dommage…

### 1.1 Un début de journée ordinaire

Il est 6h20. Nous sommes mardi. Je me lève, non pas que j’en ai envie – loin s’en faut, mais en père attentionné et habitant à la campagne, j’emmène mes grandes en ville jusqu’au tramway. Comme bien souvent, ma première pensée en émergeant est que je ne suis vraiment pas du matin. Pourvu que la journée démarre bien.

Le temps du petit déjeuner, de la toilette, nous sommes en retard. La circulation est difficile à l’arrivée en ville, comme souvent. Je dépose mes filles et arrive enfin au boulot, il est presque 8h. Un petit café et la journée démarre vraiment. Quelle tristesse de vivre une journée si ordinaire.

Un peu plus tard, le va-et-vient des collègues donne un petit coup de fouet jusqu’à l’arrivée de l’un d’entre eux qui me ramène le Linux Mag’ **[1]** qu’il avait emprunté. Je dis « me ramène » car habituellement, je suis le premier à le récupérer encore sous blister. Je le range pour plus tard, mais cette journée s’annonce finalement plutôt bien.

## 1.2 Un journée fabuleuse

Ça y est, tout le monde est couché dans la maisonnée. Il est 23h et j'entame la lecture du magazine tant attendu. Il est tard, je serai raisonnable et ne lirait que l'édito et éventuellement le premier article sur les nouveautés de **Python** 3.8. Je ressens la fatigue qui me gagne, mais soyons fou et commençons la lecture du second article qui fait une comparaison entre Python et **Go**. Fin de l'article. Regardons le titre du troisième article avant d'éteindre la lumière. Du billard quantique ? Quel curieux titre! Le résumé m'éclairera, … Vous l'aurez compris, je n'ai clairement aucune volonté. Les articles s'enchaînent, tous aussi intéressants que diversifiés les uns que les autres. Arrivé à la page 70, je lis « Conservez l'historique de vos commandes pour chaque projet ». C'était sur la page de garde, mais cela m'était complètement sorti de la tête.

Je lis, je dévore, je m'enthousiasme. C'est une évidence, demain matin, je commencerai par récupérer le tout sur le **GitHub** de GLMF dès que ma machine sera allumée.

Je m'endors.

*/// Début note ///*

Pour ceux qui n'auraient pas eu la chance de lire l'article sur *history-guardian* de Tristan Colombo, un petit résumé s'impose.

L'idée est simple et, comme parfois avec les idées simples, elle est géniale : disposer non pas d'un unique historique des commandes exécutées au fur et à mesure de nos sessions, mais avoir la possibilité de disposer d'un historique dédié à certains répertoires choisis, de sorte à récupérer les commandes exécutées il y a 3 mois, 6 mois, 2 ans, pour les projets associés à ces répertoires.

Voici ce que propose donc la version originale d'*history-guardian* :

- lors du déplacement dans un répertoire *via* la commande **`cd`**, si le répertoire destination contient un fichier nommé **`.history`**, alors l'historique global est sauvegardé et l'historique local est chargé à sa place ;

- possibilité d'activer/supprimer la sauvegarde d'un historique local dans le répertoire courant *via* la nouvelle commande **`history_guardian`**.

Pour cela, Tristan Colombo a écrit un ensemble de fonctions **Bash**, distribuées dans 3 fichiers :

- le fichier **`history_guardian_toggle`**, qui regroupe l'essentiel des fonctionnalités nécessaires au bon fonctionnement d'*history-guardian* ;

- le fichier **`history_guardian`**, qui contient le programme permettant d'activer/supprimer l'historique local pour le répertoire courant ;

- le fichier **`history_guardian_cd`**, qui définit une surcharge de la commande **`cd`** permettant de passer automatiquement du mode « historique global » au mode « historique local » – et vice-versa – selon que le fichier **`.history`** est présent dans le répertoire destination ou pas et selon le mode d'historique courant.

Les codes présentés sont intégralement écrits et clairement expliqués, avec simplicité. La structure du code est limpide et sa lecture aisée.

Bref, si vous ne l'avez pas lu, vous avez manqué quelque chose.

*/// Fin note ///*

# 2 Un jour d'histoire

Mercredi, 8h10, ma machine est allumée, je vais directement sur le **GitHub** de GLMF et constate que les sources des programmes présentés dans le numéro 232 ne sont pas disponibles.

Je n'ai clairement pas la patience d'attendre. Ma décision est évidente, il y a moins de 200 lignes à recoder. C'est l'histoire d'un instant (voir encadré).

Quitte à récrire le code pour ma pomme, je me permets de changer les noms de deux des trois fichiers :

- le fichier **`history_guardian_toggle`** devient **`history_guardian_init`**. En effet, *to toggle* signifie *basculer*, alors que ce fichier correspond plutôt à l'ensemble des fonctions à charger pour pouvoir utiliser *history-guardian* et que c'est le fichier qui devra être chargé au démarrage de chaque session ;

- le fichier **`history_guardian_cd`** devient **`history_guardian_wrapper`** car je sais que j'envisage déjà d'étendre le programme à d'autres commandes.

Deuxième grand changement, dans la version originale, les fonctions sont nommées en utilisant la notation *Lower Camel Case*, j'opte pour une notation *Snake Case*. Ne cherchez pas de bonnes raisons à ces choix, il n'y en a pas. C'est tout simplement mon histoire !

*/// Début note ///*

La notation *Camel Case* (j'utilise la dénomination anglaise car elle est carrément mieux connue que ses variantes françaises : *casse chameau* ou *casse mixte*) consiste à accoler chaque mot d'une phrase en les faisant débuter par une majuscule. La notation *Lower Camel Case* est similaire, à la différence que le premier mot débute par une minuscule.

La notation *Snake Case* consiste à écrire chaque mot en minuscule et à les séparer par un tiret bas. La variante *Screaming Snake Case* consiste à écrire les mots en majuscule au lieu de les écrire en minuscule. En général, cette variante est utilisée en informatique pour dénoter des noms de constantes globales ou des variables d'environnement.

Ainsi le « truc qui ne sert pas » serait écrit **`TrucQuiNeSertPas`** en *Camel Case*, **`trucQuiNeSertPas`** en *Lower Camel Case*, **`truc_qui_ne_sert_pas`** en *Snake Case* et **`TRUC_QUI_NE_SERT_PAS`** en *Screaming Snake Case*.

*/// Fin note ///*

## 2.1 Retour aux sources

Comme l'a expliqué Tristan Colombo, les fichiers doivent être « sourcés » et non interprétés dans un *shell* distinct, sans quoi les changements opérés dans ce *shell* demeureraient sans effet sur le *shell* courant (e.g. changement de valeurs des variables – exceptées les variables d'environnement, changement de répertoire).

J'ai donc souhaité ajouter une protection sur chaque fichier pour empêcher leur exécution dans un shell distinct.

La première solution qui m'est venue à l'esprit fût de remplacer le *shebang* **`#!/bin/bash`** par **`#!/bin/false`**.

Ainsi, il n'est plus possible d'exécuter directement les scripts, puisque la commande **`false`** échoue nécessairement.

```
$ echo -e '#!/bin/false\necho "Bonjour"' | tee test-shebang.sh
#!/bin/false
echo "Bonjour"
$ chmod 755 test-shebang.sh
$ ./test-shebang.sh
$ echo $?
1
```

Cependant, cette solution ne m'a pas totalement satisfait car elle présente deux inconvénients. Le premier est qu'aucun message d'avertissement n'est émis permettant de savoir que la commande a échouée (et personnellement, je ne vérifie pas systématiquement le code de retour des commandes que j'exécute). Le second inconvénient est qu'il demeure possible de contourner cette protection en lançant explicitement la commande dans un shell.

```
$ sh ./test-shebang.sh
Bonjour
$ bash ./test-shebang.sh
Bonjour
$ dash ./test-shebang.sh
Bonjour
$ csh ./test-shebang.sh
Bonjour
```

La deuxième solution consiste à comparer le contenu de la variable **`${0}`** (qui contient le nom du fichier interprété le cas échéant ou l'interpréteur dans le cas contraire) avec la valeur de **`${BASH_SOURCE[0]}`** (qui contient le nom du fichier contenant les commandes en cours d'exécution). Ainsi, si ces deux valeurs sont identiques, cela signifie que nous sommes dans un mauvais cas d'utilisation.

```
$ echo -e '#!/bin/bash\necho "\${0}=${0}"\necho "BASH_SOURCE=${BASH_SOURCE[0]}"' | tee test-
bash_source.sh
#!/bin/bash
echo "\${0}=${0}"
echo "BASH_SOURCE=${BASH_SOURCE[0]}"
$ chmod 755 test-bash_source.sh
$ ./test-bash_source.sh
${0}=./test-bash_source.sh
BASH_SOURCE=./test-bash_source.sh
```

```
$ sh ./test-bash_source.sh
${0}=./test-bash_source.sh
BASH_SOURCE=./test-bash_source.sh
$ source ./test-bash_source.sh
${0}=/bin/bash
BASH_SOURCE=./test-bash_source.sh
$ bash ./test-bash_source.sh
${0}=./test-bash_source.sh
BASH_SOURCE=./test-bash_source.sh
$ dash ./test-bash_source.sh
${0}=./test-bash_source.sh
./test-bash_source.sh: 3: ./test-bash_source.sh: Bad substitution
$ csh ./test-bash_source.sh
\./test-bash_source.sh=./test-bash_source.sh
BASH_SOURCE: Undefined variable.
```

Cerise sur le gâteau (ou pas), cela évite également l'utilisation avec un autre interpréteur, ce qui a du sens dans la mesure où certaines instructions sont spécifiques à Bash.

Les trois fichiers débutent donc dorénavant (commentaires supprimés) par :

```
01: #!/bin/bash
...
34: if [ "${0}" = "${BASH_SOURCE[0]}" ]; then
35:   echo "This script has to be sourced and not executed…"
36:   exit 1
37: fi
```

Un autre aspect à prendre en compte dans le fait que les fichiers sont « sourcés » est le risque de se retrouver avec un nom de variable ou de fonction potentiellement conflictuel (par exemple, la fonction **`usage`** qui a toute sa place dans n'importe quel script devient ici source de problème). J'ai donc choisi de préfixer toutes les fonctions par **`history_guardian_`** et toutes les variables globales par **`HISTORY_GUARDIAN_`**. Les variables locales sont très majoritairement préfixées par **`hg_`** (et de toutes façons, elles sont déclarées comme étant locales).

## 2.2 Quelques nouveautés

Au fur et à mesure de l'écriture du code, j'en profite pour remodeler un peu son organisation et surtout pour m'en imprégner. Cela m'amène à de menues modifications dans le cahier des charges initial.

En effet, je m'aperçois rapidement que si j'active l'h7541_History_Guardian_Back_v2.odtistorique local pour un répertoire et que je vais dans un sous-répertoire je repasse à l'historique global. Cela ne me convient pas. *A priori*, si j'utilise un historique local pour un répertoire, celui-ci doit également être utilisé par défaut pour ses sous-répertoires. Bon, il est possible de contourner cela en tapant **`/cd`** (ou **`builtin cd`**) au lieu de **`cd`**, mais ce n'est pas pratique et de toute façon cela ne permet pas d'activer le mode local en allant directement dans un sous-répertoire sans passer par le répertoire contenant le fichier d'historique.

Autre constat : si je passe d'un répertoire ayant un historique local à un autre répertoire ayant un autre historique local, et bien ça ne fonctionne pas et *history-guardian* repasse en historique global (et oui, il *toggle*). Enfin, j'utilise également régulièrement les commandes **`pushd`** et **`popd`** qui permettent respectivement d'empiler/dépiler le répertoire courant avant de changer de répertoire (voir note). Malheureusement, la structure actuelle d'*history-guardian* ne permet pas de surcharger ces commandes.

*/// Début note ///*

La commande **`pushd`** suivie d'un nom de répertoire empile en mémoire le répertoire désigné avant de s'y rendre. Si la pile était vide, la commande ajoute également le répertoire que l'on quitte.

La commande **`pushd`** sans argument permet d'intervertir les deux premiers éléments de la pile et de se placer dans le répertoire qui vient de passer en tête. Il est également possible d'effectuer une permutation circulaire de la pile avec les options **`-n`** et **`+n`**.

La commande **`popd`** permet de supprimer le sommet de la pile (et de se rendre dans le répertoire qui est maintenant au sommet de la pile). De même, les options **`-n`** et **`+n`** permettent de supprimer un autre répertoire de la pile.

La commande **`dirs`** permet d'afficher (ou effacer avec l'option **`-c`**) la pile des répertoires.

Ces commandes s'utilisent assez intuitivement et sont décrites dans la page de manuel de Bash.

*/// Fin note ///*

C'est parti pour une réécriture !

### 2.2.1 Un gardien qui me ressemble

Plusieurs variables d'environnement sont définies essentiellement pour permettre un paramétrage personnalisé.

C'est le cas des variables suivantes :

| Variable | Description |
|---|---|
| `HISTORY_GUARDIAN_HISTORY_FILE` | Nom du fichier à utiliser pour conserver l'historique local |
| `HISTORY_GUARDIAN_IGNORE_FILE` | Nom du fichier à utiliser pour ignorer un éventuel historique local |
| `HISTORY_GUARDIAN_TAG` | Titre des boîtes d'affichage des informations |
| `HISTORY_GUARDIAN_GLOBAL_HISTFILE` | Chemin absolu vers le fichier d'historique global |
| `HISTORY_GUARDIAN_DISPLAY` | Configuration de l'affichage des informations |
| `HISTORY_GUARDIAN_CHECK_LATEST_VERSION` | Activer/désactiver la vérification de nouvelles versions |

D'autres variables sont utilisées pour connaître à tout moment l'état courant de l'historique utilisé (et ne doivent pas être modifiées manuellement) :

| Variable | Description |
|---|---|
| `HISTORY_GUARDIAN_CURRENT_DIRNAME` | Chemin absolu du répertoire contenant le fichier d'historique courant |
| `HISTORY_GUARDIAN_CURRENT_HISTFILE` | Chemin absolu du fichier d'historique courant |

Enfin deux variables en lecture seules seront définies lors de l'initialisation :

| Variable | Description |
|---|---|
| `HISTORY_GUARDIAN_INITIALIZED` | Positionnée à vrai après l'initialisation |
| `HISTORY_GUARDIAN_INSTALL_DIR` | Répertoire d'installation d'*history-guardian* |

### 2.2.2 Un gardien loquace

L'affichage des informations d'*history-guardian* lorsque je passe d'un répertoire à l'autre est [trop] sobre et rapidement je m'aperçois que je n'y prête pas attention. Je retravaille donc l'affichage de sorte que les messages apparaissent dans des boîtes aux bordures épaisses.

Une boîte doit avoir un titre, *a priori*, le nom du programme. Je me connais, je risque de changer d'avis sur le nom. La solution : le titre de la boîte sera donc stocké dans une variable globale `HISTORY_GUARDIAN_TAG`. Le texte (possiblement structuré en paragraphes) va devoir être re-formaté pour être affiché dans une boîte au nombre de colonnes fixé. Le texte sera donc passé comme argument sous forme d'un tableau (une valeur = un paragraphe).

```
145: function history_guardian_message() {
```

Comment choisir le nombre de colonnes ? Au minimum, il faut assez de colonnes pour afficher le titre. Au maximum, on ne dépassera pas le nombre de colonnes du terminal (variable `COLUMNS` « automatiquement configurée si l'option `checkwinsize` est activée ou dans un interpréteur interactif à la réception d'un signal `SIGWINCH` » [extrait de la page de manuel de Bash]). À défaut de valeur connue, on partira sur un maximum de `70`, ce qui est raisonnable. Deux situations me viennent à l'esprit : le texte est relativement court par rapport au nombre total de colonnes et l'utilisateur souhaite ajuster les bords de la boîte au plus près du texte ou alors l'utilisateur souhaite que la boîte utilise un nombre fixe de colonnes (le maximum disponible ou une valeur arbitraire qu'il peut choisir).

C'est le rôle de la variable globale `HISTORY_GUARDIAN_DISPLAY` de paramétrer tout ça. Cette variable peut prendre les valeurs suivantes :

- `X columns` qui fixe à `X` le nombre maximal de colonnes à utiliser ;

- `fixed columns` qui utilisera toutes les colonnes du terminal ;

- **X fixed columns** qui utilisera **X** colonnes du terminal.

J'essaie de me mettre dans la peau d'un autre utilisateur et je me dis qu'il pourrait ne pas vouloir de message. Ajoutons cette possibilité en acceptant également la valeur **quiet** pour cette variable.

Pour traiter tous ces cas de figure, commençons par copier le contenu de la variable (en le convertissant en minuscule) dans une variable locale **hg_display** (ligne 147). Si la fonction est appelée sans argument, si le premier argument est vide ou si **hg_display** vaut **quiet**, il n'y a rien à faire.

```
147:    local hg_display="${HISTORY_GUARDIAN_DISPLAY,,}" # downcase variable
148:    if [ -n "${1}" -a "${hg_display}" != "quiet" ]; then
          ...
213:    fi
214:
215: }
```

La variable locale **ll** récupère le nombre de colonnes minimal de la boîte (le nombre de caractères du titre + 2 espaces à droite et à gauche). Par défaut, le nombre de colonnes sera supposé adaptatif (variable **fixed_columns** positionné à **false** en ligne 154). La variable **nb_columns** est égale au contenu de **hg_display** tel que l'éventuel suffixe « **fixed columns** » est supprimé (ligne 157). Si le contenu de **nb_columns** est différent de celui de **hg_display**, cela signifie que **${hg_display}** se termine par « **fixed columns** » et la variable **fixed_columns** sera positionnée à **true** (ligne 166). Dans le cas contraire (test de la ligne 158 est vrai), on effectue la même manipulation affectant à **nb_columns** le contenu de **hg_display** privé de l'éventuel suffixe « **columns** » (ligne 160). Si les deux variables ont le même contenu, alors aucune des configurations n'a été reconnue et on positionne **nb_columns** à la chaîne vide (ligne 163). Si la variable **nb_columns** est vide alors on lui affecte la valeur de la variable **COLUMNS** (ou **70** si celle-ci est vide ou non définie, ligne 170). Il ne faut pas oublier de retrancher **4** au nombre de colonnes disponibles afin de prendre en compte les deux bordures et l'espace entre les bordures et le texte (ligne 172). Enfin, si le nombre de colonnes ainsi calculé ne permet pas d'afficher le titre sur une seule ligne, et bien dans ce cas on agrandit le nombre de colonnes (ligne 175).

```
149:    local ll=$((${#HISTORY_GUARDIAN_TAG}+2))
150:    local l=${ll}
151:    local old_IFS=${IFS}
152:    local lines=(" " "" "")
153:    local s
154:    local fixed_columns=false
155:
156:    # Check if number of columns is provided by environment variable
157:    local nb_columns="${hg_display%fixed columns}"
158:    if [ "${nb_columns}" = "${hg_display}" ]; then
159:      # The variable doesn't fix an absolute number of columns.
160:      nb_columns="${hg_display%columns}"
161:      if [ "${nb_columns}" = "${hg_display}" ]; then
162:        # The variable doesn't give a maximum number of columns.
163:        nb_columns=
164:      fi
165:    else
166:      fixed_columns=true
167:    fi
168:    if [ -z "${nb_columns}" ]; then
169:      # Using environment variable if set otherwise, limit to 70.
170:      nb_columns=$((${COLUMNS:-70}))
171:    fi
172:    nb_columns=$((nb_columns - 4)) # since there is a border
173:    # The box title gives the minimal number of columns
174:    if [ "${nb_columns}" -lt ${ll} ]; then
175:      nb_columns="${ll}"
176:    fi
```

Maintenant que ce nombre de colonnes est calculé, il est nécessaire de formater les paragraphes en vue de leur affichage. Pour cela, on stockera chaque ligne dans un tableau **lines** (déclaré à la ligne 152 et pré-rempli pour accueillir une ligne vide, le titre et une ligne de soulignement). Pour cela, on traite chaque paragraphe (chaque argument passé lors de l'appel de la fonction) de la même manière (boucle de la ligne 180 à 183). La variable **s** correspond donc à un paragraphe. Devant chaque paragraphe, on insère une ligne vide (ligne 181), puis on utilise la commande **fold** qui fait le travail de mise en forme (l'option **-w** précise le nombre de colonnes à utiliser et l'option **-s** permet de découper les lignes au niveau des espaces - sans cette option les lignes sont coupées exactement à la largeur désirée et ce n'est pas très beau). Cette commande lit le paragraphe stocké dans la variable **s** et les lignes sont ajoutées au tableau **lines** (ligne 182). Par défaut, chaque mot serait considéré comme une valeur du tableau. Nous souhaitons modifier ce comportement de sorte à ce qu'une valeur corresponde à une ligne. Ceci se fait en positionnant la variable **IFS** au retour chariot (ligne 179). C'est pourquoi la valeur originale de cette variable a été

préalablement sauvegardée (ligne 151) et est restaurée à la fin du traitement (ligne 185).

```
178:     # Wrap the text according to the maxumum number of columns
179:     IFS=$'\n'
180:     for s in "${@}"; do
181:       lines+=(" ")
182:       lines+=($(echo "${s}" | fold -s -w ${nb_columns}))
183:     done
184:     lines+=(" ")
185:     IFS=${old_IFS}
```

Selon que la configuration impose une largeur de boîte fixe ou non, la variable **l** sera positionnée à **${nb_columns}** (ligne 189) ou à la longueur de la plus grande ligne à afficher (lignes 192 à 196).

```
187:     # Set the inner length of the frame box
188:     if [ "${fixed_columns}" = "true" ]; then
189:       l=${nb_columns}
190:     else
191:       # Compute the length of the longuest line
192:       for s in "${lines[@]}"; do
193:         if [ "${l}" -lt "${#s}" ]; then
194:           l="${#s}"
195:         fi
196:       done
197:     fi
```

Il reste à ajouter le titre (centré pour une question d'esthétique). Le titre correspond à la deuxième valeur du tableau **lines** (donc **${lines[1]}**) et il sera souligné (**${lines[2]}**). Ces valeurs ont été initialisées (ligne 152) à la chaîne vide. Pour centrer il suffit d'ajouter un nombre d'espaces égal à la moitié de la différence entre la largeur utile de la boite (variable **l**) et la largeur du titre (variable **ll**) à chacune des deux valeurs (lignes 200 à 203) puis à ajouter respectivement le contenu de la variable **HISTORY_GUARDIAN_TAG** (ligne 204) et le contenu de cette variable où chaque caractère est substitué par le caractère **'='** (ligne 205).

```
199:     # Center the title
200:     for ((i = 0; i < $(((l - ll) / 2)); ++i)); do
201:       lines[1]+=" "
202:       lines[2]+=" "
203:     done
204:     lines[1]+="[${HISTORY_GUARDIAN_TAG}]"
205:     lines[2]+="$(echo "[${HISTORY_GUARDIAN_TAG}]" | sed 's,.,=,g')"
```

Il reste à afficher une ligne pleine de longueur **${l}** au début (ligne 208), parcourir chacune des valeurs du tableau **lines** et pour chacune d'elle, l'afficher sur une ligne de longueur **${l}** (lignes 209 à 211), puis afficher une ligne pleine pour fermer la boîte (ligne 212).

```
207:     # Print the text in a framed box
208:     history_guardian_print_line ${l} >&2
209:     for s in "${lines[@]}"; do
210:       history_guardian_print_line ${l} "${s}" >&2
211:     done
212:     history_guardian_print_line ${l} >&2
```

Le code permettant d'afficher une ligne d'une boîte ne présente pas de grandes difficultés. Si un seul argument est fourni (ou que le second argument est la chaîne vide) alors une ligne de **'#'** de longueur **${1}+4** est affichée (lignes 127 à 130), sinon un symbole **'#'** suivi d'une espace, suivit de **${2}** est écrit (ligne 132), la ligne est ensuite complétée par des espaces jusqu'à atteindre la colonne **${1}** (hors bordure et espacement, lignes 134 à 136), puis se termine par un **'#'** (ligne 137).

```
118: # Display a framed line of length given by the first argument.
119: # If the second argument is empty or not given, then print a
120: # separation line.
121: # ${1}: length of the framed line (without delimiters)
122: # ${2}: text to print (optional)
123: function history_guardian_print_line() {
124:
125:   local s;
126:   if [ -z "${2}" ]; then
127:     for ((s=0; s < ${1}; ++s)); do
128:       echo -n "#"
129:     done
130:     echo "####"
131:   else
132:     echo -n "# ${2}"
133:
134:     for ((s=${#2}; s < ${1}; ++s)); do
135:       echo -n " "
136:     done
137:     echo " #"
```

```
138:   fi
139:
140: }
```

### 2.2.3 Un gardien actif

Définissons la fonction **history_guardian_set_history** qui active le fichier d'historique d'un répertoire donné (lignes 219 à 239).

La première chose à faire avant d'activer un nouveau fichier d'historique est de mettre à jour l'historique courant (option **-a** de la commande **history**, ligne 221) car la mise à jour du fichier d'historique n'est pas synchrone.

Il faut ensuite récupérer le chemin absolu du répertoire cible afin de mettre la variable **HISTORY_GUARDIAN_CURRENT_DIRNAME** à jour (ligne 223).

Cela permet de redéfinir la variable **HISTFILE** (variable définie et utilisée par l'interpréteur Bash) en lui associant le chemin absolu vers le nouveau fichier d'historique (ligne 225) et de s'assurer que ce fichier est créé s'il n'existe pas déjà (ligne 226). Le contenu de la variable **HISTFILE** est ensuite copié dans la variable **HISTORY_GUARDIAN_CURRENT_HISTFILE** (ligne 228).

Il faut ensuite supprimer toutes les commandes en mémoire dans l'historique (option **-c** de la commande **history**, ligne 230), puis lire le contenu du nouveau fichier d'historique (option **-r** de la commande **history**, ligne 231).

Enfin, selon que le fichier d'historique utilisé est le fichier global ou non, on affiche un message précisant le mode d'historique activé (lignes 233 à 237).

```
219: function history_guardian_set_history() {
220:
221:   history -a
222:
223:   HISTORY_GUARDIAN_CURRENT_DIRNAME=$(readlink -m "${1}")
224:
225:   HISTFILE="${HISTORY_GUARDIAN_CURRENT_DIRNAME}/${HISTORY_GUARDIAN_HISTORY_FILE}"
226:   touch "${HISTFILE}"
227:
228:   HISTORY_GUARDIAN_CURRENT_HISTFILE="${HISTFILE}"
229:
230:   history -c
231:   history -r
232:
233:   if [ "${HISTORY_GUARDIAN_CURRENT_HISTFILE}" = "${HISTORY_GUARDIAN_GLOBAL_HISTFILE}" ]; then
234:     history_guardian_message "Global history activated."
235:   else
236:     history_guardian_message "Per directory history activated for '${1}' and its
subdirectories."
237:   fi
238:
239: }
```

Bien évidemment, la fonction précédente ne doit être exécutée que si le fichier d'historique est modifié. C'est le rôle de la fonction **history_guardian_update** (lignes 241 à 250) qui calcule le chemin absolu du répertoire passé en paramètre (ligne 245) et le compare au répertoire associé à l'historique courant (ligne 246).

Si ces deux répertoires sont différents, alors seulement le changement de fichier d'historique sera mis en place (ligne 247).

```
243: function history_guardian_update {
244:
245:   local hg_current_dirname=$(readlink -m "${1}")
246:   if [ "${hg_current_dirname}" != "${HISTORY_GUARDIAN_CURRENT_DIRNAME}" ]; then
247:     history_guardian_set_history "${hg_current_dirname}"
248:   fi
249:
250: }
```

### 2.2.4 Un gardien prévenant

Dans sa version initiale, si un répertoire **A** a un fichier d'historique dédié, celui-ci n'est valable que pour le répertoire **A**, mais il n'est pas actif pour les sous-répertoires de **A**. Je ne sais pas pour vous, mais mes projets

sont généralement structurés en sous-répertoires et je préfère avoir un historique par projet plutôt qu'un historique par répertoire.

Cependant, je n'exclus pas la possibilité d'avoir une sous-arborescence d'un répertoire pour laquelle je n'ai pas envie de conserver l'historique dans celui du projet.

Il devient donc nécessaire, étant donné un répertoire, de chercher quel est le répertoire racine du fichier d'historique à utiliser. C'est le rôle de la fonction **history_guardian_check_closest_history_file_directory** (lignes 360 à 375).

La première chose à faire est donc de calculer le chemin absolu du répertoire passé en paramètre (ligne 362), le chemin absolu vers le fichier d'historique qui serait à utiliser s'il existe (ligne 363), ainsi que le chemin absolu du fichier d'abandon (qui forcerait à ignorer un éventuel historique local, ligne 364).

Si le répertoire courant est le répertoire principal de l'utilisateur ou la racine du système, alors on affiche la chaîne vide (tests de la ligne 366). Dans le cas contraire, si le fichier d'historique local ou le fichier d'abandon existe, le répertoire courant est affiché (ligne 369). Enfin, si aucune des conditions précédente n'est vérifiée, on relance la recherche sur le répertoire parent du répertoire courant (ligne 371).

```
360: function history_guardian_check_closest_history_file_directory() {
361:
362:   local hg_current_dirname=$(readlink -m "${1}")
363:   local hg_history_file="${hg_current_dirname}/${HISTORY_GUARDIAN_HISTORY_FILE}"
364:   local hg_ignore_file="${hg_current_dirname}/${HISTORY_GUARDIAN_IGNORE_FILE}"
365:   if [ "${hg_current_dirname}" = "${HOME}" -o "${hg_current_dirname}" = "/" ]; then
366:     echo
367:   else
368:     if [ -f "${hg_history_file}"  -o -f "${hg_ignore_file}" ]; then
369:       echo "${hg_current_dirname}"
370:     else
371:       history_guardian_check_closest_history_file_directory "$(dirname "${1}")"
372:     fi
373:   fi
374:
375: }
```

Ceci étant fait, j'imagine déjà le cas où je vais dans un répertoire/sous-répertoire d'un projet versionné avec **Git** ou **CVS** (les deux gestionnaires de version que j'utilise) et je sais que la plupart du temps j'aurais envie d'avoir un historique dédié au projet, mais je sais également que je ne vais pas penser systématiquement à activer l'historique dédié pour tous mes projets. Il faut donc que je puisse me suggérer d'activer l'historique dédié si celui-ci n'est pas activé et si je peux détecter que le répertoire destination est versionné. C'est le rôle des fonctions **history_guardian_propose_per_directory_history** (non présentée dans cet article) et **history_guardian_check_for_versioned_project** (lignes 326 à 344).

La vérification d'un répertoire appartenant à un projet versionné dépend des mécanismes et outils proposés par les gestionnaires de version.

Pour l'outil Git, la commande **git rev-parse --show-toplevel** affiche le répertoire racine du projet versionné sur la sortie standard le cas échéant ou bien un message d'erreur sur la sortie d'erreur. Il suffit donc de récupérer la sortie standard de cette commande et si celle-ci n'est pas vide (et qu'elle existe), le tour est joué.

Pour l'outil CVS, il n'y a pas de mécanisme intégré permettant de récupérer cette information. Pour détecter si un répertoire fait partie d'un projet versionné avec CVS, j'ai défini la fonction **history_guardian_check_for_CVS_versioned_project** (non présentée dans cet article).

Vérifier si un répertoire est versionné consiste donc à vérifier si celui-ci est versionné avec Git (ligne 331). Si ce n'est pas le cas, alors il faut vérifier si celui-ci est versionné avec CVS (ligne 334). À la fin de la fonction, il reste à afficher le répertoire calculé le cas échéant (ligne 341).

```
326: function history_guardian_check_for_versioned_project() {
327:
328:   local hg_current_dirname=${1}
329:   # If git is not installed or if given directory is not (part
330:   # of) git versioned project, then return the empty string
331:   local hg_toplevel=$(\cd "${hg_current_dirname}" && git rev-parse --show-toplevel 2>/dev/null
|| true)
332:   if [ -z "${hg_toplevel}" -o ! -d "${hg_toplevel}" ]; then
333:     # If given directory is not (part of) CVS versioned project, then return the empty string
334:     hg_toplevel=$(history_guardian_check_for_CVS_versioned_project "${hg_current_dirname}")
335:     if [ -z "${hg_toplevel}" -o ! -d "${hg_toplevel}" ]; then
336:       hg_toplevel=""
337:     fi
338:   fi
```

```
339:
340:   if [ -n "${hg_toplevel}" ]; then
341:     echo "${hg_toplevel}"
342:   fi
343:
344: }
```

La dernière chose à faire est donc de choisir l'action à effectuer pour le répertoire courant. C'est le rôle de la fonction **history_guardian_mode_auto** (lignes 377 à 410).

Dans un premier temps, on calcule le chemin absolu du répertoire courant (ligne 380) ainsi que le répertoire parent le plus proche contenant un fichier d'historique ou d'abandon (ligne 381).

Si un répertoire parent a été trouvé (donc il existe soit un fichier d'historique local, soit un fichier d'abandon), on vérifie en premier lieu l'existence d'un fichier d'abandon (lignes 385 et 386). Le cas échéant (et si on n'est pas dans le répertoire principal de l'utilisateur), on affiche un message expliquant qu'un tel fichier a été détecté et on rappelle à l'utilisateur comment activer l'historique local et on passe en mode global (lignes 387 à 394). Dans le cas contraire, c'est donc que l'on a un fichier d'historique local. Il suffit donc de l'activer (ligne 396).

Si aucun historique local n'a été détecté, alors on vérifie si le répertoire courant est versionné (ligne 399). Le cas échéant on propose à l'utilisateur d'activer l'historique local (ligne 404), sinon on repasse à l'historique global (ligne 406 ; pour rappel, il ne se passe rien si on est déjà en mode global).

```
378: function history_guardian_mode_auto() {
379:
380:   local hg_current_dirname=$(readlink -m "${PWD}")
381:   local hg_parent_dirname=$(history_guardian_check_closest_history_file_directory "$
{hg_current_dirname}")
382:
383:   # If some parent directory has a per directory history file or an ignore file
384:   if [ -n "${hg_parent_dirname}" ]; then
385:     local hg_ignore_file="${hg_parent_dirname}/${HISTORY_GUARDIAN_IGNORE_FILE}"
386:     if [ -f "${hg_ignore_file}" ]; then
387:       if [ ${HISTORY_GUARDIAN_CURRENT_DIRNAME} != ${HOME} ]; then
388:         local hg_flag=enable
389:         [ "${hg_current_dirname}" != "${hg_parent_dirname}" ] && hg_flag=start
390:         history_guardian_message \
391:           "Unable to start history guardian for directory '${hg_current_dirname}' since some
file '${HISTORY_GUARDIAN_IGNORE_FILE}' exists in '${hg_parent_dirname}'." \
392:           "Please use 'history_guardian ${hg_flag}' to allow per directory mode for '$
{hg_current_dirname}'."
393:         history_guardian_update "${HOME}"
394:       fi
395:     else
396:       history_guardian_update "${hg_parent_dirname}"
397:     fi
398:   else
399:     hg_parent_dirname=$(history_guardian_check_for_versioned_project "${hg_current_dirname}")
400:     # If current directory is detected as a project (typically a git
401:     # versioned project), we may propose to activate a per directory
402:     # history at the top level of the project.
403:     if [ -n "${hg_parent_dirname}" ]; then
404:       history_guardian_propose_per_directory_history "${hg_parent_dirname}" "$
{hg_current_dirname}"
405:     else
406:       history_guardian_update "${HOME}"
407:     fi
408:   fi
409:
410: }
```

Toutes les fonctions essentielles sont définies pour pouvoir passer d'un historique à un autre. Il reste à configurer le *shell* pour appeler les fonctions en fonction du contexte.

## 2.2.5 Un gardien discret et efficace

Dans la version initiale, l'utilisation d'*history-guardian* requiert plusieurs lignes de configuration plus ou moins corrélées dans le fichier **.bashrc**. Je préfère me contenter de « sourcer » le fichier **history_guardian_init**.

Cela va avoir pour effet de définir globalement quelques variables dont le contenu est explicite (sauf peut-être à la ligne 39 ou la variable **HISTORY_GUARDIAN_PROGNAME** se verra attribuer le nom du fichier auquel le suffixe

**_init** aura été retiré, ce qui donne dans le cas présent la valeur **history_guardian**.

```
39: HISTORY_GUARDIAN_PROGNAME=$(basename "${BASH_SOURCE[0]%_init}")
40: HISTORY_GUARDIAN_PROGVERSION=0.5
41:
42: HISTORY_GUARDIAN_URL="https://gite.lirmm.fr/doccy/history-guardian/"
43: HISTORY_GUARDIAN_AUTHOR="Alban Mancheron"
44: HISTORY_GUARDIAN_MAIL="alban.mancheron@lirmm.fr"
```

Il s'en suit la définition de la fonction **history_guardian_init** qui assurera l'initialisation.

Premièrement, on s'assure qu'*history-guardian* n'est pas déjà initialisé, sinon on sort (lignes 53 à 56).

Les variables d'environnement sont définies à leurs valeurs par défaut, sauf si elles ont déjà été définies par l'utilisateur (lignes 58 à 70).

On définit ensuite l'alias **history_guardian** (ligne 73) qui sera la commande permettant de contrôler *history-guardian* (cette commande est détaillée plus loin), ainsi que les alias des commandes permettant de changer de répertoire (lignes 74 à 76). Personnellement, j'utilise exclusivement les commandes **cd**, **pushd** et **popd**, mais *a priori* toute autre commande devrait pouvoir se *wrapper* de la même façon (le détail du *wrapper* est donné dans la section suivante).

Enfin, on active l'historique correspondant au répertoire courant (ligne 79).

```
51: function history_guardian_init() {
52:
53:   # Do not perform initialization if already done
54:   if [ -n "${HISTORY_GUARDIAN_INITIALIZED}" ]; then
55:     return;
56:   fi
57:
58:   readonly HISTORY_GUARDIAN_INITIALIZED=true
59:   readonly HISTORY_GUARDIAN_INSTALL_DIR=$(dirname "$(readlink -m "${BASH_SOURCE[0]}")")
60:
61:   HISTORY_GUARDIAN_TAG="${HISTORY_GUARDIAN_TAG:-History Guardian}"
62:   HISTORY_GUARDIAN_DISPLAY="${HISTORY_GUARDIAN_DISPLAY:-}"
63:   HISTORY_GUARDIAN_CHECK_LATEST_VERSION=${HISTORY_GUARDIAN_CHECK_LATEST_VERSION:-true}
64:   local hg_history_file=$(basename "${HISTFILE:-.history}")
65:   HISTORY_GUARDIAN_HISTORY_FILE="${HISTORY_GUARDIAN_HISTORY_FILE:-${hg_history_file}}"
66:   HISTORY_GUARDIAN_IGNORE_FILE="$
{DEFAULT_HISTORY_GUARDIAN_IGNORE_FILE:-.history_guardian.ignore}"
67:
68:   HISTORY_GUARDIAN_GLOBAL_HISTFILE=${HISTFILE}
69:   HISTORY_GUARDIAN_CURRENT_HISTFILE=$(readlink -m "${HISTFILE}")
70:   HISTORY_GUARDIAN_CURRENT_DIRNAME=$(dirname "${HISTORY_GUARDIAN_CURRENT_HISTFILE}")
71:
72:   # Setting command aliases
73:   alias history_guardian="source '${HISTORY_GUARDIAN_INSTALL_DIR}/history_guardian'"
74:   alias cd="source '${HISTORY_GUARDIAN_INSTALL_DIR}/history_guardian_wrapper' cd"
75:   alias pushd="source '${HISTORY_GUARDIAN_INSTALL_DIR}/history_guardian_wrapper' pushd"
76:   alias popd="source '${HISTORY_GUARDIAN_INSTALL_DIR}/history_guardian_wrapper' popd"
77:
78:   history_guardian_check_latest_version
79:   history_guardian_mode_auto
80:
81: }
```

Et, pas si vite!!! C'est quoi la commande de la ligne 78 ?

Oups, j'ai failli oublier ce détail.

Il s'agit simplement d'une petite fonction qui permet de vérifier que la version courante d'*history-guardian* est la plus récente.

Bon, déjà comme ça peut ralentir un peu le système, si l'utilisateur a défini la variable **HISTORY_GUARDIAN_CHECK_LATEST_VERSION** à autre chose que **true**, et bien on ne vérifie rien (lignes 87 à 89).

On teste si la commande **curl** est installée ou à défaut la commande **wget**. Si aucune des deux commandes n'est disponible alors on affiche un message indiquant qu'il faut l'une de ces deux commandes pour récupérer l'information relative à la dernière version d'*history-guardian* (lignes 91  103).

Ensuite on construit l'*URL* permettant d'interroger le serveur **GitLab** afin de récupérer les *tags* associés au projet (lignes 104 à 108).

Il se trouve que dans tous mes projets, les *tags* correspondent aux numéros de version et sont souvent de la forme **v. 1.2.3**, **release 3.2a**, *etc*. Donc lorsque je récupère tous les *tags* du projet, je ne récupère que la

partie commençant par un chiffre. Ces numéros de version sont ensuite triés *via* la commande **sort** avec l'option **-V** qui fait très bien ce travail. La dernière ligne est donc la version la plus récente (tout cela se fait à la ligne 109).

*A priori*, si la version courante diffère de la plus récente, c'est très certainement qu'il y a une mise à jour disponible et cela mérite clairement un message à l'utilisateur lui indiquant quelle est la dernière version et où la trouver (lignes 110 à 114).

```
085: function history_guardian_check_latest_version() {
086:
087:   if [ "${HISTORY_GUARDIAN_CHECK_LATEST_VERSION}" != "true" ]; then
088:     return
089:   fi
090:
091:   local hg_cmd
092:   if which curl > /dev/null; then
093:     hg_cmd="curl --silent --request GET"
094:   else
095:     if which wget > /dev/null; then
096:       hg_cmd="wget -O- -q"
097:     else
098:       history_guardian_message \
099:         "Couldn't find 'curl' nor 'wget' program." \
100:         "Unable to get latest version information from ${HISTORY_GUARDIAN_URL}"
101:       return
102:     fi
103:   fi
104:   local hg_url_base="$(echo "${HISTORY_GUARDIAN_URL}" | cut -d'/' -f 1-3)/"
105:   local hg_url_project="${HISTORY_GUARDIAN_URL#${hg_url_base}}"
106:   hg_url_base+="api/v4/projects"
107:   hg_url_project="${hg_url_project%/}"
108:   hg_url_project="${hg_url_project//\//%2F}"
109:   local hg_version=$(${hg_cmd} ${hg_url_base}/${hg_url_project}/repository/tags/ | grep -Po
'"name":"[^0-9]\K[^"]*' | sort -V | tail -n 1)
110:   if [ ${hg_version} != "${HISTORY_GUARDIAN_PROGVERSION}" ]; then
111:     history_guardian_message \
112:       "You are currently using version ${HISTORY_GUARDIAN_PROGVERSION} of $
{HISTORY_GUARDIAN_PROGNAME}." \
113:       "A new version (v${hg_version}) is available at '${HISTORY_GUARDIAN_URL}'."
114:   fi
115:
116: }
```

Voilà, tout est prêt. Il reste à appeler la fonction d'initialisation lorsque le fichier est sourcé pour tout activer (ligne 382).

```
378: ##################
379: # Initialization #
380: ##################
381:
382: history_guardian_init
```

## 2.2.6 La surcharge des commandes d'accès aux répertoires

Comme indiqué précédemment, j'utilise essentiellement trois commandes pour changer de répertoire : **cd**, **pushd** et **popd**.

Quelle que soit la fonction utilisée, le principe de la surcharge de ces commandes est le même. Il faut suffit de se déplacer dans le répertoire (donc appeler la commande d'origine), puis d'appeler la fonction **history_guardian_mode_auto**, qui basculera automatiquement le mode d'historique approprié au répertoire courant.

Il suffit donc de définir une commande **history_guardian_wrapper**, qui prend en paramètre la commande à exécuter. De même que précédemment (et comme brillamment expliqué dans l'article de Tristan Colombo), il est nécessaire de sourcer le fichier qui contiendra l'appel à cette fonction pour que le changement de répertoire soit effectif après l'exécution de la commande. Donc on retrouve le même mécanisme que pour les lignes 1 à 37 du fichier **history_guardian_init**.

La fonction de surcharge **history_guardian_change_directory** exécute la commande originale ainsi que les éventuels arguments passés en paramètre puis, en cas de succès uniquement, met à jour le mode d'historique attendu (ligne 42).

```
41: function history_guardian_change_directory() {
42:   builtin "${@}" && history_guardian_mode_auto
```

```
43: }
```

Pour rappel, des alias pour chacune des commandes ont été définis qui « sourcent » ce fichier avec comme argument le nom de la commande à exécuter (lignes 74 à 76 de la fonction **history_guardian_init**). Il suffit donc que le fichier sourcé **history_guardian_wrapper** appelle la fonction **history_guardian_change_directory** avec les arguments passés en paramètre (ligne 48). Cependant, cette fonction n'a de sens que si le fichier **history_guardian_init** a été sourcé (et donc que la fonction éponyme a été appelé au préalable). Dans le cas contraire, on palliera à cette défaillance (ligne 46).

```
46: [ -n "${HISTORY_GUARDIAN_INITIALIZED}" ] || source "${BASH_SOURCE[0]/_wrapper/_init}"
47:
48: history_guardian_change_directory "${@}"
```

Ainsi donc, si vous utilisez d'autres commandes pour vous déplacer, il suffit d'ajouter un alias soit dans votre **.bashrc**, soit de modifier la fonction d'initialisation (et de soumettre une proposition de fusion, communément appelée *merge request* dans l'univers de Git).

### 2.2.7 La commande de gestion

La dernière commande qui a été surchargée lors de l'initialisation (ligne 73) est celle qui permet de contrôler la gestion de l'historique : la commande **history_guardian**.

Dans sa version initiale, celle-ci proposait deux actions, **start** et **stop** dont le rôle était respectivement d'activer l'utilisation d'un historique dédié pour le répertoire courant et celui de désactiver et supprimer l'historique dédié pour le répertoire courant.

Dans la nouvelle version, ces commandes sont conservées (mais réécrites) et de nouvelles actions sont ajoutées :

- l'action **reload** qui permet de recalculer le mode d'historique à appliquer pour le répertoire courant ;

- l'action **enable** qui active l'utilisation d'un historique dédié pour le répertoire courant et ses sous-répertoires mais ne le démarre pas ;

- l'action **disable** qui empêche l'utilisation d'un historique dédié pour le répertoire courant et ses sous-répertoires (mais qui ne le supprime pas) ;

- l'action **status** qui permet d'afficher le mode d'historique en cours d'utilisation ;

- l'action **help** qui affiche la description complète de la commande **history_guardian**.

Bien évidemment, la première fonction mise à jour est **history_guardian_usage**, qui permet d'afficher un message d'aide (lignes 43 à 213). Celle-ci commence par définir des codes d'échappement ANSI afin de souligner, mettre en gras ou en couleur des portions de message (lignes 47 à 54).

```
43: # Display usage message
44: # ${1}: if set and is equal to "full" display a complete description
45: #       if set display an additional message (typically an error)
46: function history_guardian_usage() {
47:   local hg_underline=$(echo -e "\033[4m")
48:   local hg_bold=$(echo -e "\033[1m")
49:   local hg_bold_cyan=$(echo -e "\033[36;1m")
50:   local hg_bold_yellow=$(echo -e "\033[33;1m")
51:   local hg_bold_purple=$(echo -e "\033[35;1m")
52:   local hg_bold_red=$(echo -e "\033[31;1m")
53:   local hg_yellow=$(echo -e "\033[33m")
54:   local hg_reset=$(echo -e "\033[0m")
```

Ensuite, le copyright, le synopsis et les actions possibles sont affichées sur la sortie d'erreur (lignes 56 à 88).

```
56:   cat <<EOF >&2
57:
58: ${hg_underline}${hg_bold}${HISTORY_GUARDIAN_TAG} v. ${HISTORY_GUARDIAN_PROGVERSION}${hg_reset}
59:
60:   Copyright © 2019 -- Tristan Colombo
61:   Copyright © 2019 -- Alban Mancheron <alban.mancheron@lirmm.fr>
62:
63: ${hg_bold}${hg_underline}Synopsis${hg_reset}
64:
65:   The command history is a wonderful tool that allows to easily
66:   retrieve past command lines. However, when working on several
67:   project, commands are tangled…
68:
69:   History guardian allows to set per directory history.
70:
```

```
71: ${hg_bold}${hg_underline}Usage${hg_reset}
72:
73:   ${hg_reset}${hg_bold}${HISTORY_GUARDIAN_PROGNAME} <action>${hg_reset}
74:
75:   Where ${hg_bold}<action>${hg_reset} can be one of:
76:     ${hg_bold}start${hg_reset}    Activate per directory history mode for the current directory.
77:     ${hg_bold}stop${hg_reset}     Delete local history file (and reload correct history mode for
78:               the current directory).
79:     ${hg_bold}reload${hg_reset}   Reload history mode for the current directory.
80:     ${hg_bold}global${hg_reset}   Force global history mode for the current directory (not
81:               permanent).
82:     ${hg_bold}enable${hg_reset}   Allow per directory history mode (but doesn't start).
83:     ${hg_bold}disable${hg_reset}  Prevent per directory history mode (but doesn't stop).
84:     ${hg_bold}status${hg_reset}   Show the current status of history guardian.
85:     ${hg_bold}help${hg_reset}     Display a complete description of history guardian features.
86:
87: EOF
```

Si la fonction **history_guardian_usage** a été appelée avec la chaîne **full** en argument, alors une description complète est affichée (non reproduite ici, lignes 89 à 206). Sinon, si la commande a été appelée avec un argument et que cet argument est différent de **full**, alors celui-ci est affiché (lignes 207 à 212).

```
089:   if [ "${1}" = "full" ]; then
090:     cat <<EOF >&2
091: ${hg_bold}${hg_underline}Description${hg_reset}
092:
...
206: EOF
207:   else
208:     if [ -n "${1}" ]; then
209:       history_guardian_message "${1}"
210:       echo >&2
211:     fi
212:   fi
213: }
```

Pour chacune des actions acceptées par le script, une fonction dédiée est crée. Comme pourrait le dire Marie-Joëlle, « Y a pas de quoi vermifuger un abris-bus » (La tour Montparnasse Infernale, 2001), aussi le code de ces fonctions n'est pas présenté dans cet article.

La commande **history_guardian** consiste donc, après s'être assurée que tout est initialisé (ligne 290), à appeler la fonction associée à une action donnée ou à afficher le message d'aide si aucune commande (connue) n'a été fournie en argument (lignes 292 à 306).

```
290: [ -n "${HISTORY_GUARDIAN_INITIALIZED}" ] || source "${BASH_SOURCE[0]}_init"
291:
292: if [ "${#}" -ne 1 ]; then
293:   history_guardian_usage
294: else
295:   case "${1}" in
296:     start) history_guardian_start;;
297:     stop) history_guardian_stop;;
298:     reload) history_guardian_reload;;
299:     global) history_guardian_global;;
300:     enable) history_guardian_enable;;
301:     disable) history_guardian_disable;;
302:     status) history_guardian_status;;
303:     help) history_guardian_usage full;;
304:     *) history_guardian_usage "Action '${1}' unknown";;
305:   esac
306: fi
```

Comme indiqué dans l'article de Tristan Colombo, cette commande (et les actions associées) peuvent donc facilement être associées à des raccourcis clavier pour encore plus de fantaisie…

# Fin de l'histoire

Mon histoire est presque terminée et elle me plaît (c'est déjà ça). J'avais très envie de la partager et j'ai tout à fait conscience qu'elle n'aurait jamais vu le jour si Tristan Colombo ne l'avait pas initiée.

J'aurais certes pu écrire

*/// Début note PAO ///*

*L'idée de l'encadré ci-après est de mettre en exergue le texte (genre citation, post-it ou encore fichier*

*README)*

*/// Fin note PAO ///*

*/// Début encadré ///*

J'ai fait un *fork* du projet *history-guardian* présenté par Tristan Colombo dans **[1]**. Ce *fork* ajoute les fonctionnalités suivantes :

- surcharge des commandes `pushd` et `popd` (en plus de la surcharge de la commande `cd`) afin d'activer automatiquement le mode d'historique associé au répertoire courant ;

- utilisation de l'historique d'un répertoire pour tous ses sous-répertoires ;

- possibilité de désactiver (sans supprimer) l'utilisation d'un historique dédié dans un répertoire ;

- détection automatique des projets sous gestionnaires de version (*git* ou *CVS*) afin de proposer le passage à un historique dédié si ce n'est pas déjà le cas ;

- amélioration de l'affichage des informations ;

- possibilité de paramétrer finement le comportement d'*history-guardian* ;

- simplification de l'installation ;

- ajout d'une documentation.

Vous pouvez récupérer la version originale à l'adresse **https://github.com/tcolombo/history-guardian** et ma version sur **https://gite.lirmm.fr/doccy/history-guardian/** ; ces deux versions sont distribuées sous licence GNU GPL 3.

Pour installer et utiliser cette version, allez sur le **GitLab** du projet et *RTFM*.

*/// Fin encadré ///*

Mais ainsi racontée, je trouve qu'elle manque profondément d'attrait ; d'autant que je souhaitais avant tout rendre hommage à l'ingénieuse et prodigieuse idée de Tristan Colombo, que je remercie encore de l'avoir partagée :

```
$ for ((i = 0; i < 1000; ++i)); do echo "Merci"; done
```

J'espère donc avant toute chose que ma version de l'histoire vous aura plu également. Je ne peux que vous inviter à raconter et pourquoi pas compléter ce récit.

# Références

**[1]** T. COLOMBO, *« Conservez l'historique de vos commandes pour chaque projet »*, GNU/Linux Magazine n°232, Novembre 2019, p. 70 à 75 : **https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-232/Conservez-l-historique-de-vos-commandes-pour-chaque-projet**