N°192 avril 2016

GNU LINUX MAGAZINE / FRANCE

DÉVELOPPEMENT ET ADMINISTRATION SUR SYSTÈMES OPEN SOURCE

OUTIL / PILLAR

Générez simplement des documents textuels et des slides p.56

CLUSTER / GALERA

QUELS RISQUES ÊTES-VOUS PRÊTS À PRENDRE AVEC VOS DONNÉES ?

CRÉEZ UN CLUSTER HAUTE DISPONIBILITÉ

AVEC MYSQL/MARIADB p.36

BUGTRACKER / APHRODITE

Prenez le contrôle de vos processus de développement p.71

BONNES PRATIQUES / BASH

Utilisez pleinement la puissance des scripts shell p.20

C++ / LISP

Concevez votre propre langage de programmation p.64

V8JS / PHP

Intégrez du JavaScript dans PHP p.76

SHELL / SERVICE

Testez quels sont vos ports ouverts p.32

ET AUSSI : Envoi de mails en PHP avec SwiftMailer - Drivers virtuels audio et vidéo

L 19275 - 192 - F: 7,90 € - RD

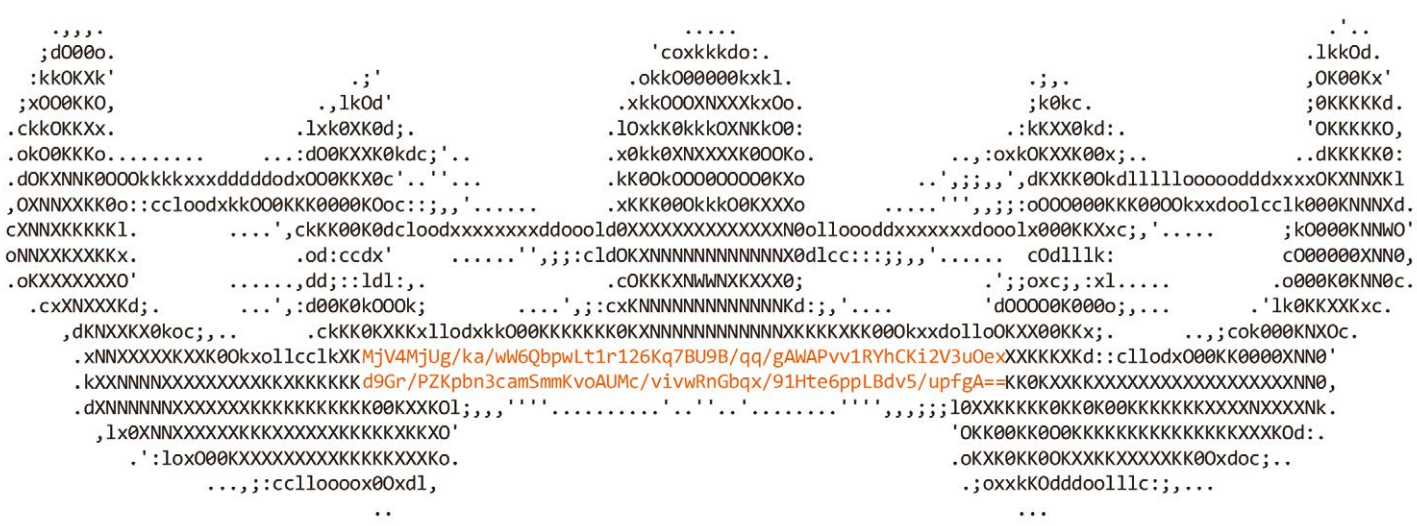
France MÉTRO. : 7,90 € – CH : 13 CHF – BEL/LUX/PORT.CONT : 8,90 € – DOM/TOM : 8,50 € – CAN : 14 $ CAD

Une énigme se cache dans le message suivant, saurez-vous la résoudre ?

MjV4MjUg/ka/wW6QbpwLt1r126Kq7BU9B/qq/gAWAPvv1RYhCKi2V3uOex
d9Gr/PZKpbn3camSmmKvoAUMc/vivwRnGbqx/91Hte6ppLBdv5/upfgA==

(message également accessible à cette adresse: http://goo.gl/sjWgrt)

Rendez-vous sur le lien caché dans l'énigme
pour envoyer votre solution et son explication.



Parrot®

Rejoignez les 900 passionné(e)s qui inventent les drones de demain

parrot.recrute.com

GNU/Linux Magazine France
est édité par Les Éditions Diamond

10, Place de la Cathédrale - 68000 Colmar - France
Tél. : 03 67 10 00 20 – **Fax :** 03 67 10 00 21
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com – www.ed-diamond.com

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Tristan Colombo
Responsable service Infographie : Kathrin Scali
Responsable publicité : Valérie Fréchard,
Tél. : 03 67 10 00 27 – v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, **N° ISSN :** 1291-78 34
Commission paritaire : K78 976

Périodicité : Mensuel
Prix de vente : 7,90 €

SUIVEZ-NOUS SUR :

https://www.facebook.com/editionsdiamond

@gnulinuxmag

LES ABONNEMENTS ET LES ANCIENS NUMÉROS SONT DISPONIBLES !

En version papier et PDF :
www.ed-diamond.com

Codes sources sur https://github.com/glmf

ÉDITORIAL

Il est amusant de constater comme nous fonctionnons tous selon des systèmes d'appartenance à des clans. Il y a le clan des partisans du logiciel libre contre le reste du monde, le clan des administrateurs système contre les développeurs et si on continue à affiner on peut arriver, au sein des développeurs, à trouver des oppositions entre développeurs Python, Java, C, etc. On doit même pouvoir trouver des écoles s'opposant au sein d'un même langage !

Quand j'étais jeune (non pas que je sois maintenant vieux...), je lisais déjà *GNU/Linux Magazine* et j'aimais dans cette revue l'ouverture qu'elle m'offrait, les découvertes technologiques que je pouvais y faire. Sans appartenir à une chapelle autre que « GNU/Linux », j'ai toujours été curieux de tout ce que je ne connaissais pas. En tant que rédacteur en chef, je suis toujours heureux de lire un article soumis en me disant « là j'ai appris quelque chose ». Ce sont ces découvertes que j'aime partager avec vous.

Dans les mails que je reçois, certains trouvent qu'il y a trop d'articles d'administration système, et d'autres trop d'articles de développement. Je pense chaque numéro de *GNU/Linux Magazine* de manière à essayer de trouver le meilleur équilibre possible entre les différentes rubriques et technologies de façon à ce que chaque lecteur puisse trouver son bonheur et, en même temps, élargir ses connaissances en ayant accès à des articles qui n'appartiennent pas forcément à son domaine de prédilection.

Le contenu de votre magazine dépend de deux paramètres fondamentaux :

- le point de vue de nos auteurs ;
- le point de vue de nos lecteurs.

Concernant le premier groupe, je vous invite à ne pas hésiter à venir nous rejoindre. Vous travaillez dans un domaine spécifique, vous souhaiteriez voir apparaître certains champs de compétences particuliers ? Sentez-vous libre de prendre la plume (ou plutôt le clavier) ! Certains diront « je ne suis pas écrivain », mais tout un chacun sait écrire et partager ses expériences. Nous n'attendons ni du Balzac ni du Tolkien, mais simplement vous qui avez une expérience à partager, une façon de programmer que vous présenterez à votre manière. C'est la première ligne qui compte, le reste suit naturellement ; alors j'attends que vous nourrissiez votre magazine et grossissiez les rangs du groupe des auteurs. Osez nous faire part de vos propositions de thèmes d'articles sur contrib@gnulinuxmag.com. Nous pourrons alors en discuter.

Concernant maintenant le second groupe, pour essayer de répondre toujours de manière plus poussée à vos attentes, j'aurais maintenant besoin de votre concours : deux petites minutes de votre temps pour répondre à un sondage en ligne qui permettra à la rédaction de mieux vous connaître (en tant que lecteur anonyme de *GNU/Linux Magazine* bien entendu, nous n'avons pas signé de partenariat avec la NSA...). Ce sondage est accessible à l'adresse suivante :

http://www.gnulinuxmag.com/sondage-gnulinux-magazine-2016/

N'oubliez pas que l'on n'entend que ceux qui s'expriment et qu'en ne participant pas vous donnez plus de poids aux avis des autres lecteurs.

En attendant les résultats de ce sondage, j'espère que vous prendrez plaisir à lire le présent numéro !

Tristan Colombo

ACTUELLEMENT DISPONIBLE

GNU/LINUX MAGAZINE HORS-SÉRIE N°83 !

LE GUIDE POUR APPRENDRE LES PRINCIPES CLÉS DU LANGAGE C++ !

NE LE MANQUEZ PAS

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

www.ed-diamond.com

SOMMAIRE

GNU/LINUX MAGAZINE FRANCE N°192

IONIC, LE FRAMEWORK HTML5
QUI AURAIT DÛ ACCÉLÉRER VOS DÉVELOPPEMENTS CORDOVA

Tristan Colombo

Le projet Apache Cordova est bien connu pour le développement rapide d'applications nomades multiplateformes. Vous l'utilisiez peut-être sans savoir qu'il vous manquait une « brique », un élément vous permettant d'accélérer encore vos développements. Cet élément pourrait-il être Ionic ? La sortie récente de sa dernière version est l'occasion de le découvrir.

Mots-clés : Ionic, Apache Cordova, HTML5, JavaScript, AngularJS, CSS, Android

Résumé

Depuis sa première version Cordova a bien évolué, mais il est vrai qu'une simplification, un regroupement des commandes seraient les bienvenus. C'est l'objectif d'Ionic en intégrant en plus AngularJS. Dans cet article, nous testerons donc ce *framework* pour voir s'il répond vraiment à nos attentes.

Ionic [1] est un *framework* regroupant plusieurs technologies pour permettre un développement rapide d'applications web multiplateformes. Le *front-end* se base sur **AngularJS [2] [3]** et le *back-end* sur **Apache Cordova [4] [5]**. On peut ainsi s'appuyer sur la puissance de deux poids lourds du domaine et bénéficier de nombreuses extensions (*plugins*) et thèmes.

La dernière version de Ionic est parue le 2 janvier 2016 (au moment où ces lignes sont écrites). Il s'agit de la version 1.2.3 dite « Copenhagen » (les versions de la branche 1.2 portent des noms de villes). Il faut noter que de nouvelles versions sont publiées très régulièrement. Ainsi, la version 1.2.1 « Amsterdam » a-t-elle été publiée le 17 décembre 2015 et la version 1.2.2 « Barcelona » le 31 décembre 2015. Il y a donc fort à parier qu'au moment où vous lirez ces lignes une nouvelle version aura été mise en ligne.

Pour tester un *framework* rien ne vaut le développement d'un petit projet, une sorte de « hello world » un peu amélioré. Nous commencerons par installer Ionic et découvrir son fonctionnement puis nous nous lancerons dans la création d'une mini application.

1 Installation et démarrage d'un projet

L'installation se fait très simplement à l'aide de **NPM** le gestionnaire de paquets de **Node.js**. Il ne faut pas oublier d'installer également Cordova si ce n'est déjà fait :

```
$ sudo npm install cordova ionic
```

Attention toutefois : vous installerez bien Cordova 5.4.1 qui est la dernière version d'Apache Cordova, mais Ionic sera en version 1.7.12. Ceci est normal : la version donnée pour Ionic est la version de l'outil Ionic CLI pour la ligne de

commandes (de toute façon il n'existe pas de version 1.7.12 du *framework*...).

Nous pouvons maintenant démarrer notre mini projet de test. Pour initier un projet, on utilise la commande **ionic start** suivie du nom du projet et éventuellement du nom du modèle à utiliser. Ionic propose six types de modèles (contrairement à ce qui est dit dans la documentation qui n'en liste que trois) :

- **tabs** qui est le modèle par défaut et propose un en-tête et un pied de page de navigation (voir figure 1) ;
- **sidemenu** qui affiche un menu sous forme de liste sur la gauche de la page lorsque l'on clique sur l'icône de menu (voir figure 2) ;
- **complex-list** pour un projet comprenant une liste « complexe »... mais qui n'est pas opérationnel au moment où ces lignes sont écrites et produit le même résultat que **blank** (voir figure 3) ;
- **maps** qui affiche un menu et utilise Google Maps (voir figure 4);

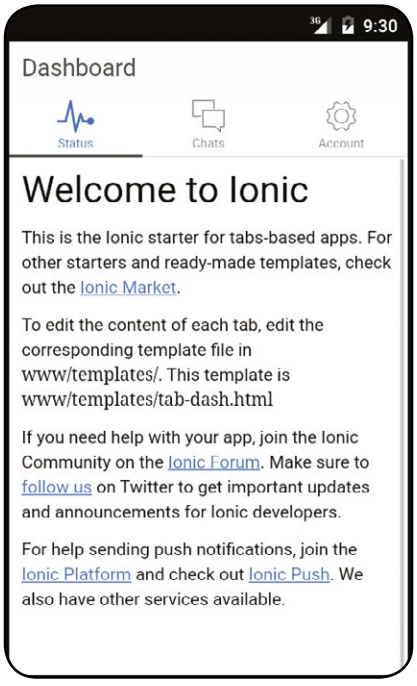


Fig. 1 : Modèle tabs.

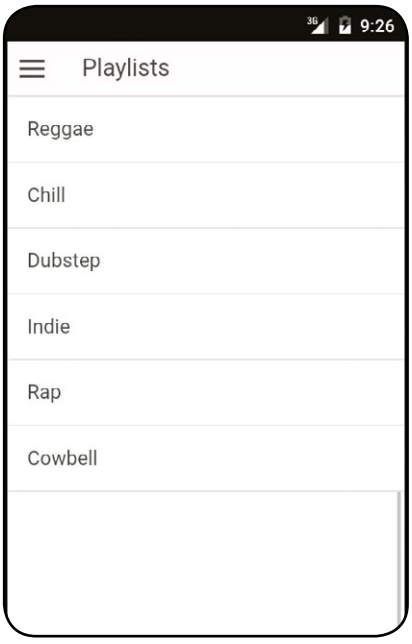


Fig. 2 : Modèle sidemenu.

- **salesforce** pour un projet associant Ionic et **Salesforce** (CRM propriétaire) ;
- **blank** qui est une application vide (voir figure 3).

Ces informations, plus à jour que la documentation ont été obtenues grâce à la commande **ionic --help** puis **ionic templates**. Pour revenir à notre projet, nous choisirons ici la forme la plus fréquemment employée dans les applications sur smartphone, le **sidemenu** :

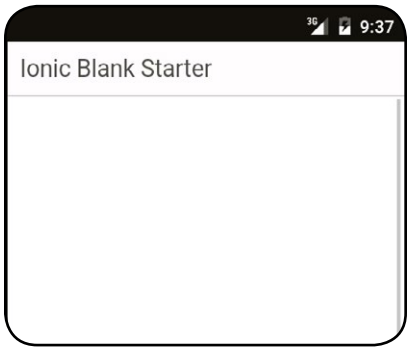


Fig. 3 : Modèle complex-list ou blank.

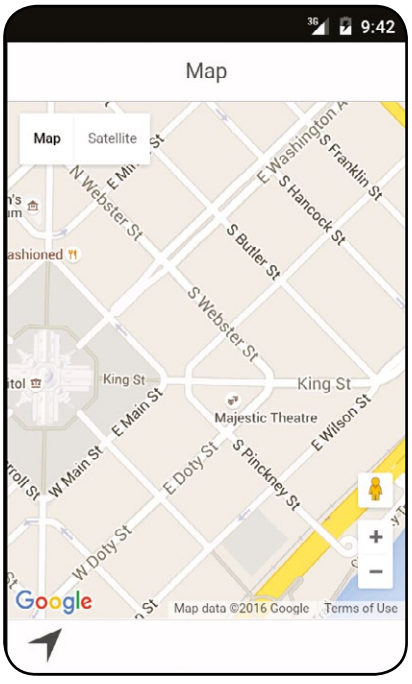


Fig. 4 : Modèle maps.

```
$ ionic start hello sidemenu
Creating Ionic app in folder hello based on sidemenu project
Downloading: https://github.com/driftyco/ionic-app-base/archive/master.zip
[============================]  100%  0.0s
Downloading: https://github.com/driftyco/ionic-starter-sidemenu/archive/master.zip
[============================]  100%  0.0s
Updated the hooks directory to have execute permissions
Update Config.xml
Initializing cordova project
```

La commande ne se contente pas de créer le projet, elle vous rappelle également l'ensemble des commandes essentielles que je résume ici :

- Pour se placer dans le répertoire du projet (au cas où vous l'auriez oublié...) :

```
$ cd hello
```

- Pour utiliser **Sass [6]** (*Syntactically Awesome Style Sheets*) pour l'écriture de vos CSS. Cette commande nécessite l'installation de **Gulp [7]** et doit être exécutée en *root*, car elle installe des dépendances :

```
$ sudo npm install -g gulp
$ sudo ionic setup sass
...
Successful sass build
Ionic setup complete
```

- Pour obtenir un aperçu de votre développement dans un navigateur, vous pouvez lancer un serveur qui affichera les pages de l'application et les rafraîchira automatiquement à chaque changement dans un fichier (ça c'est une bonne idée !) :

```
$ ionic serve

Multiple addresses available.
Please select which address to use by entering its number from the
list below:
 1) 192.168.0.12 (eth1)
 2) 192.168.42.18 (tap2)
 3) localhost
Address Selection:  3
Selected address: localhost
Running live reload server: undefined
Watching: 0=www/**/*, 1=!www/lib/**/*
Running dev server:  http://localhost:8100
Ionic server commands, enter:
  restart or r to restart the client app from the root
  goto or g and a url to have the app navigate to the given url
  consolelogs or c to enable/disable console log output
  serverlogs or s to enable/disable server log output
  quit or q to shutdown the server and exit

ionic $
```

Après avoir sélectionné l'adresse de votre serveur, ionic démarrera automatiquement le navigateur sur la bonne page (ici, il s'agira de http://localhost:8100). Il vous indiquera ensuite les différentes commandes disponibles. Notez qu'à la fin de l'exécution de cette commande le prompt a changé : il s'agit du prompt du serveur Ionic (noté **ionic $**). Ainsi, par exemple, pour activer la sortie des logs console, il vous suffira de taper :

```
ionic $ c
Console log output: enabled
Loading: /?restart=422289
```

- Pour ajouter le support d'une plateforme, il faut utiliser la commande **ionic platfrom add** suivie du nom de la plateforme (**android**, **ios**, etc.). Je supposerai ici que vous avez déjà réalisé une installation fonctionnelle du SDK Android (sinon c'est le moment !) :

```
$ ionic platform add android
Updated the hooks directory to have execute permissions
Downloading Default Ionic Resources
Downloading: https://github.com/driftyco/ionic-default-resources/
archive/master.zip
[============================]  100%  0.0s
Done adding default Ionic resources
Adding icons for platform: android
Adding android project...
...
```

Là encore, vous obtiendrez un rappel des commandes principales relatives à l'utilisation des différentes plateformes. Je prendrai ici l'exemple de la plateforme **android**, pour d'autres cibles il suffira de changer le nom de la plateforme :

→ Pour construire l'application :

```
$ ionic build android
```

Même si vous avez installé le SDK, il est possible que vous obteniez ici un message d'erreur :

```
[Error: Please install Android target: "android-22".
...
```

Utilisez simplement le gestionnaire du SDK Android pour sélectionner la plateforme Android-22 utilisée par défaut lors de la création d'un projet. Pour rappel, l'*Android SDK Manager* se lance par la commande **android**. L'autre solution consiste à éditer le fichier **platforms/Android/Android-Manifest.xml** et modifier le numéro de version du SDK en ligne 15 :

```
01: <?xml version='1.0' encoding='utf-8'?>
...
15:     <uses-sdk android:minSdkVersion="16"
android:targetSdkVersion="21" />
16: </manifest>
```

Dans le fichier **platforms/Android/project.properties**, il faut effectuer la même manipulation, mais cette fois en ligne 13 :

```
...
13: target=android-21
14: android.library.reference.1=CordovaLib
```

Attention toutefois : Cordova fait référence à **android.webkit.ClientCertRequest** qui a été ajouté dans l'API 21. N'indiquez donc pas une cible inférieure à 21 !

→ Pour lancer l'application dans un émulateur :

```
$ ionic emulate android
```

La figure 5 montre l'application lancée dans l'émulateur par défaut. Si vous avez déjà créé un émulateur (ou AVD pour *Android Virtual Device*), vous pouvez indiquer que vous souhaitez l'employer en donnant son nom après le paramètre **--target**. Par exemple, pour un émulateur **monEmulateur** il faudrait lancer :

```
$ ionic emulate android --target=monEmulateur
```

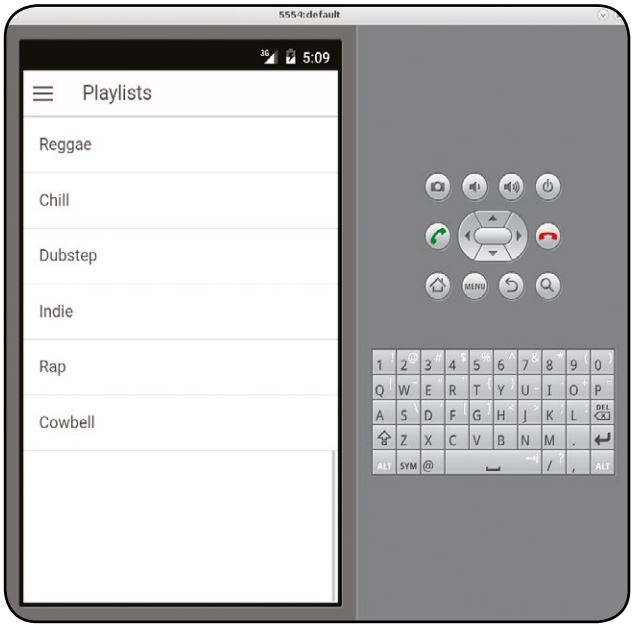


Fig. 5 : Affichage de l'application dans un émulateur.

- Démarrer l'application sur un périphérique :

```
$ ionic run android
```

Si votre smartphone est relié à l'ordinateur alors l'application sera installée et lancée sur ce dernier, sinon ce sera l'émulateur par défaut qui sera appelé. Vous pouvez spécifier sur quel périphérique lancer l'application grâce à l'option **--target** :

```
$ adb devices
List of devices attached
f669b259        device

$ ionic run android --target=f669b259
```

- Créer un paquetage de développement ou prêt à être déployé sur le marché de la plateforme ciblée (en l'occurrence ici le Google Play). L'option **--release** permet de créer le paquet de distribution au public. Pour utiliser cette commande, il faut posséder un compte Ionic.io :

```
$ ionic package build android
```

Notez qu'en cas d'erreur vous signalant l'absence d'un « Ionic App ID », vous pouvez définir manuellement cet identifiant par :

```
$ ionic link 1
Your Ionic App ID was set
```

- Finalement, une question vous sera posée pour savoir si vous souhaitez utiliser les services de ionic.io et créer un compte utilisateur. Pour nous, la réponse sera non :

```
Create an ionic.io account to send Push Notifications and use the
Ionic View app?
(Y/n): n
+-----------------------------------------------------+
+ New Ionic Updates for January 2016
+
+ The View App just landed. Preview your apps on any device
+ http://view.ionic.io
+
+ Invite anyone to preview and test your app
+ ionic share EMAIL
+
+ Generate splash screens and icons with ionic resource
+ http://ionicframework.com/blog/automating-icons-and-splash-
screens/
+
+-----------------------------------------------------+
```

Nous avons fini notre premier aperçu d'Ionic. La gestion est très proche de celle de Cordova (ce qui est normal puisqu'Ionic fait appel à ce dernier). Par contre, la création de projet consistant en fait à copier un petit projet contenant déjà des données ne m'a pas plus convaincu qu'avec Cordova. On obtient certes un projet fonctionnel sans taper une ligne de code, mais ce n'est pas notre projet, il va falloir y apporter des modifications et éventuellement nous oublierons de supprimer des éléments, ce qui sera source de bugs. De plus, nous serons influencés par l'interface qui

nous sera présentée et nous aurons tendance à essayer de nous y conformer pour minimiser le nombre de modifications. Je préfère partir d'un projet vide si je ne peux pas construire moi-même les pages de départ.

2 Ionic Lab : pour ceux qui n'aiment pas la ligne de commandes

Pour ceux d'entre vous qui préfèrent travailler avec une interface graphique, l'IDE **Ionic Lab** intègre toutes les fonctionnalités d'Ionic CLI. Voici la procédure d'installation à suivre en faisant très attention, les développeurs de chez Ionic ne sachant visiblement pas construire correctement une archive pour ne pas polluer le répertoire client :

```
$ wget http://bit.ly/ionic-lab-linux-x64 -O ionic-lab-linux.x64.
tar.gz
$ mkdir ionic-lab
$ tar -xvf ionic-lab-linux.x64.tar.gz -C ionic-lab
```

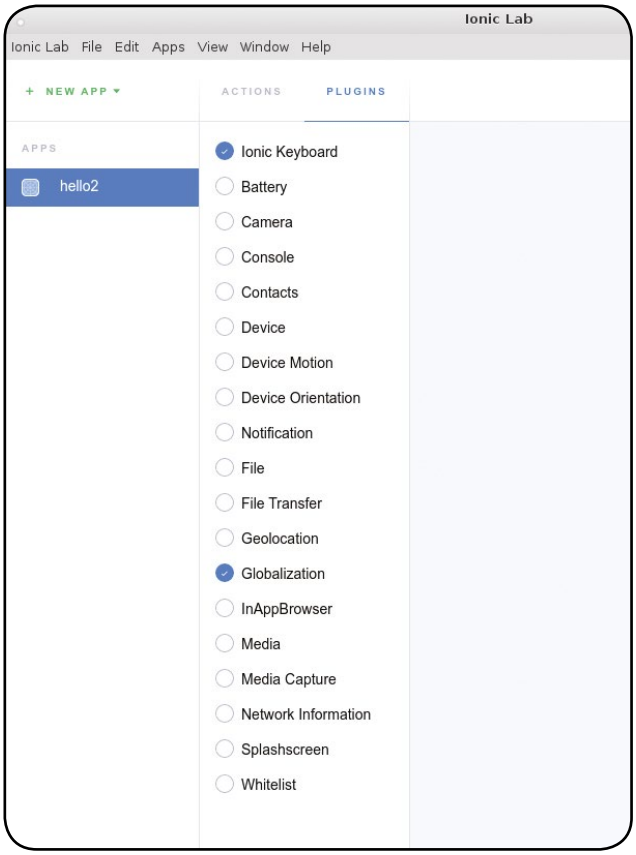


Fig. 6 : Sélection des plugins Cordova dans Ionic Lab.

Pour lancer l'IDE, il faudra exécuter le fichier **`IonicLab`** :

```
$ cd ionic-lab
$ IonicLab
```

Vous retrouverez toutes les commandes précédentes accessibles par simple clic. L'intérêt réside surtout dans le fait que les différentes options et *plugins* Cordova disponibles sont directement visibles (voir figure 6).

3 Ionic Creator : la création d'écrans simplifiée

Cette fois, vous n'y couperez pas : si vous voulez utiliser **Ionic Creator** il va falloir créer un compte chez Ionic.io. Ionic Creator est une application en ligne vous permettant de créer les prototypes de vos applications.

Rendez-vous sur la page https://creator.ionic.io/app/signup pour créer votre compte. En cliquant sur le bouton ***New Project*** en haut et à droite, vous pourrez créer un projet en renseignant les mêmes informations qu'en ligne de commandes ou dans Ionic Lab (voir figure 7).

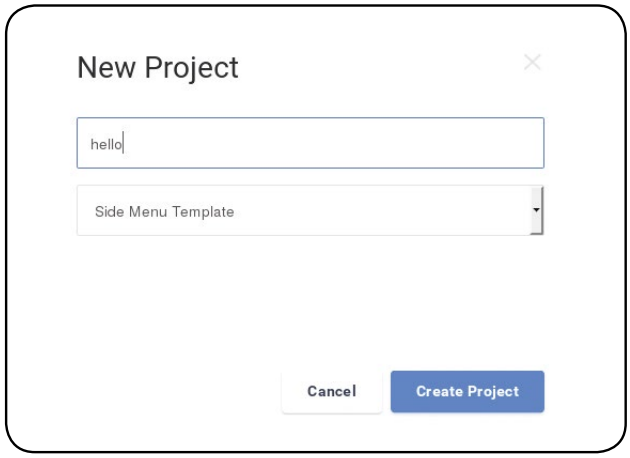


Fig. 7 : Création d'une application dans Ionic Creator.

Vous aurez accès à divers composants visibles sur la gauche de la page (voir figure 8) et vous pourrez modifier les titres des champs du menu dans l'onglet ***Content*** sur la droite. Pour que le menu soit fonctionnel, il faudra indiquer une action dans le champ ***Link*** (toujours sur la droite). Pour pouvoir le faire pointer vers une page, il faudra en créer une en cliquant sur l'icône en forme de feuille avec un symbole « + » dans l'onglet ***Pages*** sur la gauche (vous retrouverez les choix de pages préconstruites classiques).

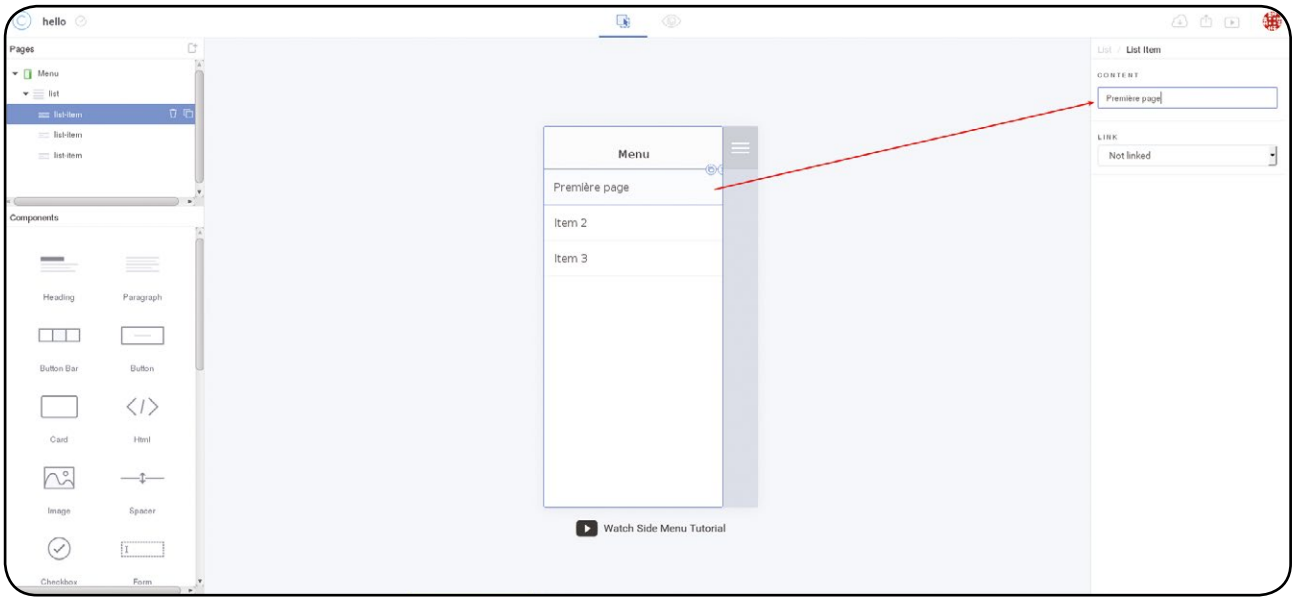


Fig. 8 : Modification d'un champ du menu.

On peut ainsi créer à l'aide des composants mis à disposition une page « Recherche » (voir figure 9) et la lier depuis le menu (voir figure 10).

Le bouton ***Preview*** en forme d'œil en haut et au milieu de la page permet de passer en mode visualisation et de tester la navigation dans l'application (le bouton ***Build*** permet de revenir à la page de construction).

Tout ceci nous permet de créer un prototype de notre application... c'est bien joli, mais peu utile si nous ne pouvons

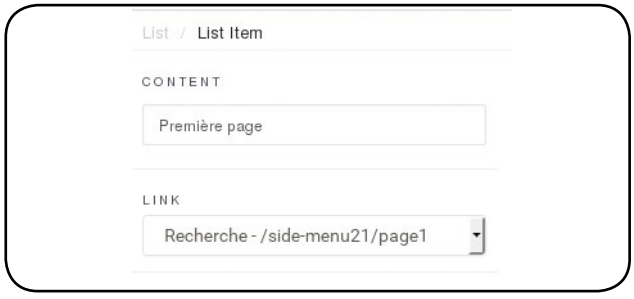


Fig. 10 : Rattachement d'une page au menu.

Fig. 9 : Création d'une page.

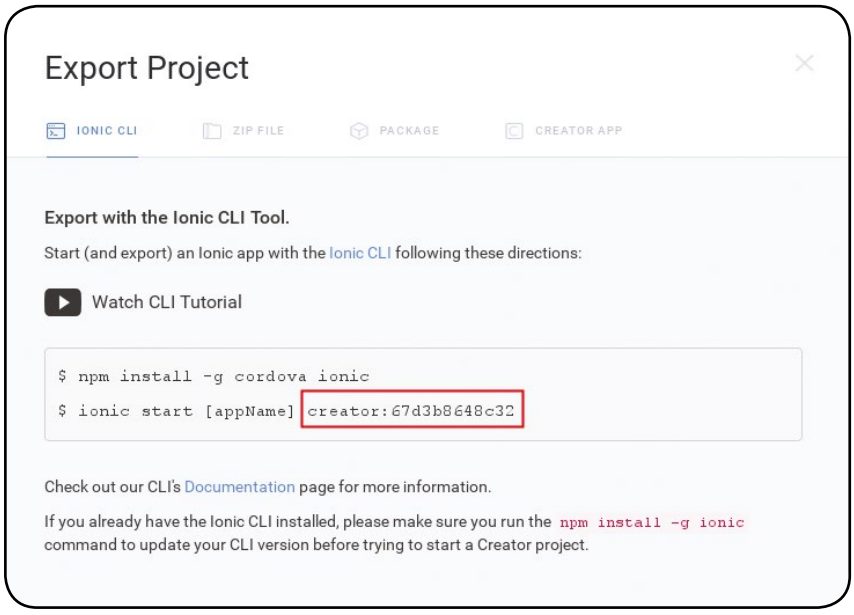


Fig. 11 : Obtention de l'identifiant du projet.

pas récupérer notre travail pour la phase de développement. En cliquant sur le bouton ***Export*** se situant en haut et à droite (l'icône en forme de nuage avec une flèche vers le bas), comme le montre la figure 11, vous obtiendrez l'identifiant de votre projet dans le *cloud* (souvenez-vous que vous avez créé un compte et que l'application est en ligne...). Il n'y a plus qu'à se connecter ensuite et créer un projet en important les données du *cloud* :

```
$ ionic login

To continue, please login to your Ionic account.
Don't have one? Create a one at: https://apps.ionic.io/signup

Email: votre_adresse@mail
Password: XXXXXXX
Logged in! :)
$ ionic start hello2 creator:67d3b8648c32
$ cd hello2
$ ionic platform add android
```

4 Faire évoluer l'application ?

L'Ionic Market, disponible sur http://market.ionic.io, permet de télécharger des *starters* (les applications de base servant de modèle), des *plugins* (extensions) ou des thèmes. Beaucoup sont payants, mais une fois l'un des trois groupes choisi vous pourrez classer les éléments par prix (***Price(low)***).

Prenons l'exemple d'un thème : **Material Design for Ionic [8]** permet d'ajouter le support de Material Design à un projet Ionic. Avouez que Material Design **[9][10]** vous manquait (on se demande d'ailleurs pourquoi il n'est pas intégré par défaut) ! Malheureusement il va falloir oublier le **`ionic theme add ionic-material-design`**... ça n'existe pas et on va tout faire à la main. Ce n'est même pas une fonctionnalité disponible uniquement pour les possesseurs d'un compte Ionic.io. Quand on a goûté à Ionic CLI, on est très, très déçu ! Dans le cas de Material Design for Ionic, il faut donc télécharger le thème, placer les fichiers **`dist/css/ionic.material-design-lite.min.css`** et **`dist/js/ionic.material-design-lite.bundle.min.js`** dans le projet, y faire référence dans nos documents HTML et modifier les attributs de classes de nos divers éléments. J'ai peut-être une vision erronée de ce que l'on appelle un « thème », mais pour moi, dans une application, un thème est défini par une ligne dans un fichier de configuration (et donc il suffit de changer une chaîne de caractères pour changer de thème). Par exemple, puisqu'Ionic utilise un fichier **`package.json`** pour ses informations de configuration, j'y aurais intégré la gestion des thèmes :

```
01: {
02:   "name": "hello",
03:   "version": "1.1.1",
04:   "description": "hello: An Ionic project",
05:   "theme": "ionic-material-design",
...
```

Il en va de même pour les *starters* et les *plugins*... Mais alors à quoi servent ces commandes alléchantes visibles depuis l'aide ?

```
$ ionic --help
...
add [name] ..................................  Add an Ion, bower
component, or addon to the project
                                                [name] The name
of the ion, bower component, or addon you wish to install

remove [name] ...............................  Remove an Ion,
bower component, or addon from the project
                                                [name] The name
of the Ion, bower component, or addon you wish to remove
...
```

En fait, vous allez pouvoir installer ou retirer des *plugins* Cordova et des ions (des composants bower). Ne vous emballez pas, voici les ions proposés par Ionic :

```
$ ionic ions
Ionic Ions - ions power your app to be awesome.
A curated collection of useful addons, components, and ux
interactions for extending ionic.


Header Shrink ........ 'ionic add ionic-ion-header-shrink'
A shrinking header effect like Facebook's

Android Drawer ....... 'ionic add ionic-ion-drawer'
Android-style drawer menu

iOS Rounded Buttons .. 'ionic add ionic-ion-ios-buttons'
iOS "Squircle" style icons

Swipeable Cards ...... 'ionic add ionic-ion-swipe-cards'
Swiping interaction as seen in Jelly

Tinder Cards ......... 'ionic add ionic-ion-tinder-cards'
Tinder style card swiping interaction
```

Pour en trouver d'autres, il faudra parcourir https://libraries.io/search?q=ionic. Certains ions sont bien documentés alors que d'autres ne le sont pas du tout. Prenons l'exemple de **ionic-toast** permettant d'afficher des *toasts* (au sens Android bien sûr, pas des tartines !).

Il faut bien entendu commencer par ajouter cet ion :

```
$ iconic add iconic-toast
Bower component installed - ionic-toast
```

Vous pouvez vérifier les éléments installés par :

```
$ ionic list
Ions, bower components, or addons installed:
ionic
ionic-toast
```

Et là tout est automatique ! Vous n'avez plus qu'à insérer le code permettant l'affichage à l'endroit où vous voulez utiliser votre toast... Mais non, bien sûr ! Il faut éditer quatre fichiers :

- Dans **www/index.html**, on ajoute le chargement du fichier Javascript ionic-toast :

```
01: <!DOCTYPE html>
02: <html>
03:   <head>
...
15:     <!-- ionic/angularjs js -->
16:     <script src="lib/ionic/js/ionic.bundle.js"></script>
17:     <script src="lib/ionic-toast/dist/ionic-toast.bundle.min.
js"></script>
...
```

- Dans **www/js/app.js**, on injecte la dépendance ionic-toast :

```
01: // Ionic Starter App
...
07: angular.module('starter', ['ionic', 'starter.controllers',
'ionic-toast'])
...
```

- Dans **www/js/controllers.js**, on définit la fonction permettant d'afficher le toast dans la page souhaitée :

```
01: angular.module('starter.controllers', [])
02:
03: .controller('AppCtrl', function($scope, $ionicModal,
$timeout) {
...
44: .controller('PlaylistsCtrl', ['$scope', 'ionicToast',
function($scope, ionicToast) {
```

```
45:   $scope.playlists = [
46:     { title: 'Reggae', id: 1 },
47:     { title: 'Chill', id: 2 },
48:     { title: 'Dubstep', id: 3 },
49:     { title: 'Indie', id: 4 },
50:     { title: 'Rap', id: 5 },
51:     { title: 'Cowbell', id: 6 }
52:   ];
53:
54:   $scope.showToast = function(){
55:     ionicToast.show('Affichage du toast', 'top', false, 2500);
56:   };
57: }])
...
```

Les paramètres de la fonction **ionicToast.show()** sont les suivants : le message à afficher, la position du toast sur la page (**top**, **middle** ou **bottom**), la fermeture par action de l'utilisateur (**true**) ou automatique (**false**), la durée d'affichage du toast (si le paramètre précédent est à **false**).

- Dans **www/templates/playlists.html**, on ajoute un bouton permettant de faire apparaître le toast :

```
01: <ion-view view-title="Playlists">
02:   <ion-content>
...
09:     <button class="button button-block" ng-click="showToast()">Affichage du toast</button>
10:
11:   </ion-content>
12: </ion-view>
```

Le résultat obtenu est présenté en figure 12.

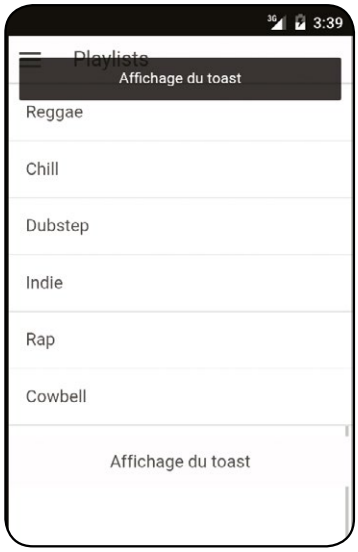


Fig. 12 : Utilisation d'ionic-toast dans un projet.

Nous avons donc pu voir que l'on peut aller piocher des éléments un peu n'importe où alors que la notion de Market pourrait faire croire à une centralisation des composants. De plus, je vous laisse imaginer le temps qu'il faut passer pour intégrer un composant qui n'est pas documenté.

Conclusion

Vous l'aurez compris en lisant cet article, je n'ai vraiment pas été convaincu par Ionic. J'aime beaucoup Cordova et la façon dont le projet évolue. Je trouvais qu'Ionic était un petit peu comme dans une vieille publicité que vous devez connaître : « j'en ai rêvé, Ionic l'a fait ». Mais le problème c'est que mon rêve allait beaucoup plus loin ! Ionic a de très bonnes bases, mais on se sent rapidement frustré : la ligne de commandes ne va pas au bout de ses promesses (je ne suis que très peu sensible aux IDE graphiques). Partir d'un projet existant (les *starters*) peut paraître formidable, mais c'est plus un gadget qu'autre chose : j'ai une application fonctionnelle qui ne correspond pas à mes besoins, mais je suis content, car ça marche. Ces *starters* devraient être construits sous la forme de *templates* de manière à être vraiment utilisables. Enfin, ne pas intégrer d'office Material Design à l'heure actuelle me paraît être une erreur.

Beaucoup d'espoir et surtout de frustrations avec ce *framework*. Désormais, je rêve d'un *Polymeric* : un Ionic qui irait au bout des idées ébauchées et qui utiliserait Polymer et Material Design... ■

Références

[1] Site officiel du *framework* Ionic : http://ionicframework.com/

[2] Site officiel de AngularJS : https://angularjs.org/

[3] COLOMBO T., « *Une application Web avec AngularJS en quelques minutes* », *GNU/Linux Magazine n°176*, novembre 2014, p. 58 à 68.

[4] Site officiel d'Apache Cordova : https://cordova.apache.org/

[5] COLOMBO T., « *Cordova : quoi de neuf ?* », *GNU/Linux Magazine n°181*, avril 2015, p. 54 à 59.

[6] Site officiel de Sass : http://sass-lang.com/

[7] Site officiel de Gulp : http://gulpjs.com/

[8] Thème Material Design for Ionic : http://market.ionic.io/themes/ionic-material-design

[9] Spécifications de Material Design : https://www.google.com/design/spec/material-design/introduction.html

[10] ADJIBI R., « *Le Material Design et son implémentation sous Android* », *GNU/Linux Magazine n°187*, novembre 2015, p.76 à 82.

APPS CLOUD READY!*

CLOUD SPECTATOR
NUMÉRO 1
Meilleur rapport qualité/prix
Test comparatif 01/2016

intel Powered by Cloud Technology

1&1 SERVEUR CLOUD
1 MOIS GRATUIT !*

1&1 Cloud App Center est le meilleur moyen de lancer vos applications ! Choisissez parmi plus de **100 applications ultra-modernes** et associez-les à la vitesse et aux performances du serveur Cloud 1&1, **numéro 1 du test comparatif Cloud Spectator !**

- ✓ **Plateforme puissante et sécurisée**
- ✓ **Pas besoin de connaissance en serveurs**
- ✓ **Facturation à la minute**

1 MOIS POUR ESSAYER
1 CLIC POUR CHANGER DE PACK
1 APPEL UN EXPERT VOUS RÉPOND

0970 808 911
(appel non surtaxé)

1&1
1and1.fr

*Prêt pour le Cloud. 1&1 Serveur Cloud : 1 mois d'essai gratuit, puis à partir de 4,99 € HT/mois (5,99 € TTC) (pour la configuration du serveur Cloud S). Facturation mensuelle en fonction de la configuration choisie. Pas de durée minimum d'engagement. Des frais de mise en service de 9,99 € HT (11,99 € TTC) s'appliquent. Conditions détaillées sur 1and1.fr. Intel® et le logo Intel® sont des marques commerciales d'Intel Corporation aux États-Unis et/ou dans d'autres pays. 1&1 Internet SARL, RCS Sarreguemines B 431 303 775.

PETIT TOUR DU FOSDEM 2016

François Revol [Guide touristique, fait des Haiku dans un garage]

Beaucoup trop de choses à voir au FOSDEM, STOP. Rapport non exhaustif ci-dessous, STOP.

Mots-clés : FOSDEM, Événement, Communauté, Libre, Hacker

Résumé

Pourquoi me suis-je proposé pour le compte-rendu d'un événement aussi complexe en moins de deux pages ? Il n'y a guère que le générique de *The Big Bang Theory* pour réussir cet exercice de style !

Certains entendront que la star du FOSDEM était **XMPP**, ou **Diaspora***. C'est peut-être ça l'effet FOSDEM : on peut passer un week-end entier dans le bâtiment AW pensant qu'il s'agit uniquement d'embarqué, de **KiCAD** et de **Coreboot** ou une journée entière dans la *devroom* BSD, croyant que **Haiku** est la seule alternative crédible aux vrais systèmes Unix ou encore flâner sur les stands pour admirer la diversité du libre. Car finalement, la star de cet événement bruxellois, c'est le libre.

Chacun des 5 à 8000 hackers présents en cette fin de janvier avait sa propre sélection dans ce programme si dense (600 événements, 12 pages par jour sur papier). Si les applications mobiles facilitent un premier choix, on se rend vite compte du nombre de collisions en rouge. Compte-rendu forcément partiel donc, à compléter par DLFP **[1]** ou Makery **[2]**.

1 Socialisation

La veille du premier jour, le traditionnel « Friday Beer Event » au centre-ville a permis à ceux n'ayant pas de conférence à préparer d'expérimenter la version stable 3.3 du « Beer Distribution Algorithm » sur un large choix de breuvages.

Durant le week-end également, à la pause repas on discute, que ce soit dans la cafétéria, ou en faisant la queue devant les différents camions, pour une pizza, une saucisse-frites ou des gaufres, sous un parapluie bien sûr !

En marge du FOSDEM, de nombreux « Fringe events » **[3]** étaient organisés indépendamment, dont le « XMPP Standards Foundation Summit 19 », « Brussels Erlang Factory Lite 2016 », et même un « MINIXCon » pour les fans de Tanenbaum.

2 Main troll^W tracks

Une dizaine de thèmes se partagent les deux amphis réservés aux *main tracks*, en plus des *keynotes*.

L'accueil dans l'amphi Janson (figure 1) fut l'occasion de rappeler la tradition : quand c'est plein, c'est plein ! Et c'est indiqué en gros sur la porte. Par contre, la capture vidéo s'est rationalisée, avec de beaux boîtiers découpés au laser pour des machines standardisées plus faciles à gérer. La conférence d'ouverture a affolé les *trollomètres*, détaillant la *roadmap* de **Systemd** pour 2016, surtout en fait l'implémentation de DNSSEC dans Systemd, même si « ça ne sert à rien, mais autant être les premiers à le faire ». De nombreux autres conférenciers se sont ensuite succédé, abordant des sujets aussi divers que

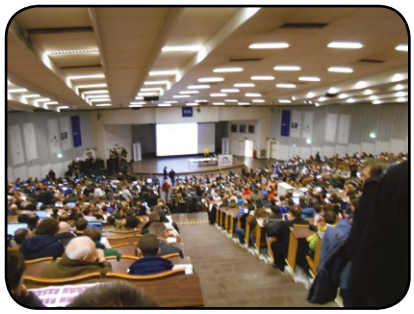


Fig. 1 : L'amphi Janson, déjà plein dès l'ouverture.

« l'architecture E/S dans Linux à l'ère du pétaoctet » ou « une entreprise de télécom avec du libre ». L'après-midi se terminant par un discours en mémoire de **Ian Murdock**, initiateur du projet Debian.

Le dimanche fut également très studieux, avec des sujets comme les *reproducible builds*, « comment concevoir une API kernel Linux » [NDLA : premier sur ma *watchlist*, j'ai déjà le pop-corn!], ou le futur du format OpenDocument. La conférence de clôture rendait compte de l'usage des outils de gestion de crise par cartographie ouverte, qui ont par exemple permis à 7000 volontaires de réaliser 13 millions d'éditions sur **OpenStreetMap** lors des deux premières semaines suivant le séisme au Népal, facilitant grandement le travail des secours.

3 Passage au stand

Deux jours durant, des bénévoles se sont relayés sur les nombreux stands, étalés sur trois bâtiments et plusieurs étages, pour expliquer leur projet, distribuer autocollants, *flyers* et *goodies*, voire vendre des bouteilles de *cuvée Perl*. Entre les classiques distributions GNU/Linux, **Mozilla**, **LibreOffice** (et **OpenOffice**, mais à bonne distance), **OSM**, **GNOME** et **KDE**, d'autres projets moins connus se sont fait une place, comme **PicoTCP** ou **ReactOS**.

Le FOSDEM permet également de renouveler ses t-shirts, par exemple au stand FSFE (*Free Software Foundation Europe*). Une occasion aussi pour les membres francophones de la FSFE de se rencontrer de visu et discuter traduction et migration de mairies au libre, assis dans une cage d'escalier.

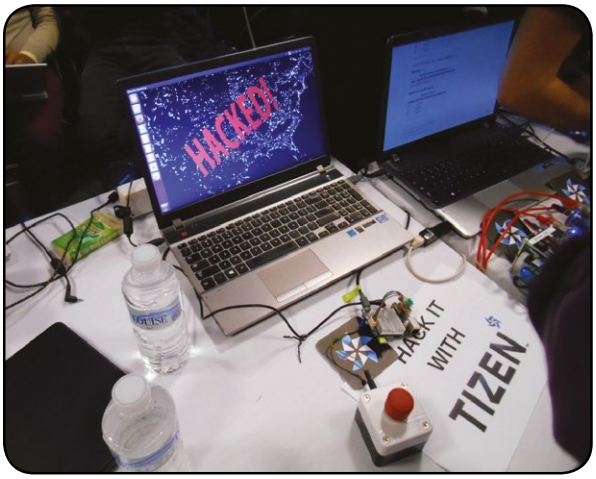


Fig. 2 : Simuler la frappe sur un clavier USB ? Aucun problème !

4 Des vrooms ?

Chaque *devroom* mériterait un article complet, chacune abordant de nombreux aspects de thèmes aussi variés que les langages comme **Ada**, **Python**, **Perl**, **Go**, **PHP**, **Lua**, **Java** ou **Guile**, ou bien l'embarqué, l'Internet des objets, les distributions, micronoyaux, Mozilla, virtualisation, temps réel, sécurité, LLVM, BSD... et même les questions légales. Ce fut d'ailleurs l'occasion de troller les licences Open Core, de raconter le libre dans les institutions européennes, ou de se plaindre que l'open source ne marche pas parce qu'il y manque l'éthique du libre.

Ainsi, chez les BSD des sujets tels que **ZFS**, **Xen** ou les pilotes graphiques (émulation linux *inside*) se succédèrent, pour finir par deux conférences sur la compilation reproductible.

En *Desktops*, on a pas seulement parlé des nouveautés de KDE, mais aussi d'**Enlightenment** sur **Wayland**, de GNOME, de **Cmake**...

Pour l'embarqué, il a été question de MIPS, de *toolchains* FPGA libres, et de fabriquer soi-même des objets USB (voir figure 2).

Concernant les distributions, **Docker** bien sûr, mais aussi les *Guest Additions* **VirtualBox** étaient au programme, ainsi que la compilation reproductible (tiens, c'est à la mode). Une standardisation du boot sur ARMv7 a aussi été proposée [NDLA : pas seulement pour Linux j'espère], automatisant la génération du menu de démarrage.

Une présentation des nouveautés de **VLC** (qui fête ses 15 ans !) a eu lieu dans la *devroom Open Media*.

Pas plus d'énumération, de nombreux comptes-rendus sont déjà indexés par vos moteurs de recherche, et les vidéos arrivent sur le site du FOSDEM, à vos claviers !

Conclusion

Le FOSDEM confirme son statut d'événement incontournable pour tous les codeurs de logiciel libre en Europe. Même s'il est frustrant de ne pouvoir se cloner quinze fois pour tout voir, on en revient toujours avec de nouvelles idées, des logiciels à tester, et l'envie renouvelée de promouvoir toujours plus le libre. ■

Références

[1] Sur DLFP, « *Retour sur le FOSDEM 2016 à Bruxelles* » : https://linuxfr.org/news/retour-sur-le-fosdem-2016-a-bruxelles

[2] DCALK sur Makery, « *À Bruxelles, le Fosdem joue le jeu open source en communauté* » : http://www.makery.info/2016/02/05/a-bruxelles-le-fosdem-joue-le-jeu-open-source-en-communaute/

[3] Liste des « *Fringe events* » : https://fosdem.org/2016/fringe/

C'EST TON BUG ! DÉBROUILLE-TOI AVEC !

The Cheshire Cat [Everyone here is mad]

Et si ce mois-ci nous discutions d'empathie et d'entraide entre développeurs ? Après tout, peu de projets informatiques peuvent être conduits de bout en bout par une seule personne. Savoir communiquer avec le reste de son équipe est tout de même parfois nécessaire.

Mots-clés : Humeur, Développeur, Empathie, Équipe, Junior, Senior

Résumé

Travailler en équipe, que ce soit dans le cadre de son travail ou dans le cadre d'une participation à un projet libre est une chose que les développeurs devraient être capables de faire. Et le travail en équipe, cela ne se limite pas à savoir s'exprimer de manière à être compris par les autres, même si cela semble la base de tout.

Je me souviens quand j'étais un tout jeune chaton. Testant alors Linux avec une Red Hat Linux 5.0, je suis allé poster tout fier un message sur un des groupes de *newsgroups* linux (merci le fait d'avoir un PC à côté sous Windows pour aller sur le Net). Bien entendu, c'était une question. Et pour le coup, j'ai reçu plusieurs RTFM. C'était comme cela les années 1990. Bien entendu, cela ne m'a pas aidé à résoudre mon problème d'alors... parce que je ne savais même pas quel manuel je devais lire. Et mon problème de carte graphique Cirux Logic qui ne fonctionnait pas avec mon serveur X, je ne l'ai jamais résolu. J'aurais pu alors abandonner Linux en me disant que les linuxiens étaient tous des rustres. Mais je ne baisse pas les pattes aussi rapidement. Et quelque temps (années) plus tard, j'étais de retour avec une Slackware sur mon nouvel ordi.

Le pire, c'est que j'avais intégré que le RTFM était normal. Un passage obligé, une sorte de rituel d'initiation. Ça voulait dire que voilà, on y était. Et on pouvait même en parler entre personnes de bonne compagnie. « C'était quand ton premier RTFM ? Rah la la, tu savais pas utiliser VI, trop Noob quoi ». Quelle honte en y repensant. Mais que celui ou celle qui n'a jamais été dans cette situation me jette le premier seau d'eau glacé.

1 Rugosité et individualisme au sein d'une équipe

1.1 Rugosité

Je sais, on vous a vendu le mythe du développeur seul dans sa grotte. Celui qui code seul en mode ours mal léché. Mais malheureusement, l'époque où l'on pouvait coder vraiment seul dans sa grotte n'est pas loin d'être totalement terminée. Et dans un contexte professionnel, c'est également vrai. Il va donc falloir apprendre à travailler en équipe et à travailler **correctement** en équipe. Qu'est-ce que je veux dire par là ? Qu'il va falloir se rendre compte que travailler en équipe veut dire travailler avec d'autres êtres humains. Et que cela implique donc de communiquer avec eux. Et que non, l'insulte, la vanne, le cynisme, le mépris ne sont pas vraiment de bonnes manières de communiquer avec autrui. Je lis bien trop souvent la prose de personnes qui arrêtent de contribuer à un projet libre, dégoûtées de l'ambiance

qui y règne. Il me semble d'ailleurs que le parfait exemple de ce que, je pense, il ne faut pas faire, se trouve dans les *mailings lists* (et autres moyens de communication) de développement du noyau linux. Démolir les gens que ce soit directement ou à travers le code est à mon sens une attitude contre-productive et qui devrait être considérée comme ce qu'elle est, un non-sens complet. Au lieu de cela, je vois fleurir « non, mais cela a toujours été comme ça » ou des «'faut savoir encaisser ». Billevesées au plus haut point. Pour être plus clair encore, je pense que votre avis technique sur le code de ceux et celles qui codent avec vous n'est pas, n'est jamais, plus important que l'état émotionnel dans lequel vont se trouver les membres de votre équipe après l'avoir entendu. Et que la phrase « ce code c'est de la merde ! », avec la mauvaise imitation de monsieur Coffe, ne devrait jamais être ni prononcée ni entendue au sein d'une équipe de développement. Même si c'est pour parler de votre propre code. J'ai bien trop souvent croisé des gens qui utilisaient l'argument « je dis que mon propre code, c'est de la merde » uniquement pour pouvoir après se défouler sur le code des autres. Le code peut être sous optimal, présenter une dette technique (parfois d'ailleurs choisie en conscience), être trop naïf, trop compliqué, etc.. Mais ce n'est pas de la merde.

1.2 Individualisme

Combien de fois ai-je entendu l'exclamation que j'ai utilisée pour le titre de ce papier ? Bien trop à mon goût. Et cela dénote à la fois d'un manque total de séniorité et d'une dose bien trop importante d'individualisme. Après tout, même si oui, celui ou celle qui a écrit le code contenant un bug est responsable, au final est-ce que ce n'est pas plutôt à cause d'une erreur collective que ce bug a pu se retrouver en production sans être détecté. Parce que le bug a été écrit oui, mais il est ensuite passé à travers les étapes de la *review* de code et de celle des tests automatisés lancés par l'intégration continue. Si un bug se retrouve en production, c'est donc « à cause » de l'équipe tout entière et non pas seulement de celui qui au départ l'a introduit dans le code. Rejeter toute la faute sur ses épaules n'est possible que dans une vision simpliste, une vision de junior donc, du développement en équipe. On se focalise sur son *backlog* personnel, sa liste de tâches. Ce qui compte c'est de livrer à l'heure. Et il ne faut surtout pas perdre de temps à aider les autres. Après tout, il y a bien trop de risques que cela nuise à notre vélocité. Que d'enfantillages...

2 Interaction avec les mainteneurs d'un projet libre

Il m'arrive d'avoir les moustaches qui frisent à voir la façon dont certains utilisateurs interagissent avec les développeurs/euses de projets libres. À lire les descriptions des issues qu'ils laissent, je me demande s'ils ont bien compris ce qu'était le développement libre. Je les vois ainsi exiger plus de documentation, plus de fonctionnalités, arguant que « c'est tout de même une fonctionnalité basique et qu'elle est absolument nécessaire, alors est-ce qu'elle pourrait être implémentée d'ici deux ou trois jours ? ». Bien entendu, il ne faut pas alors leur expliquer que c'est un projet libre, fait en fonction du temps et des besoins de ceux qui y participent et que donc leur demande est prise en compte, mais ne sera pas forcément implémentée tout de suite ou même implémentée tout court. Alors oui, on peut pester intérieurement contre le fait que la documentation d'une bibliothèque n'est pas à jour ou pas complète, que les tests ne sont pas suffisants, voire inexistants. Mais encore une fois, il ne faut pas oublier que personne ne nous force à utiliser le code d'autrui. Et que personne ne nous interdit non plus d'améliorer les choses. Et qu'une contribution bien faite n'est a priori pas rejetée. Bon, bien entendu, si vous faites une *pull request*, comme je l'ai vu une fois, qui ajoute une fonctionnalité, casse tous les tests et dont le message est « Les tests ne passent plus, mais je n'ai pas envie de regarder pourquoi, j'ai codé une nouvelle fonctionnalité, je te laisse modifier les tests avant de merger », cela ne va pas.

Conclusion

Le développement informatique c'est pratiquement tout le temps une histoire d'équipe et de communication. Une histoire d'échanges entre vous, le reste de votre équipe, votre client. Et pour que les choses se passent bien, il faut que la communication soit possible et apaisée. Parce que comme vous le diront nos ami.e.s des RH, le management par l'insulte ou le dénigrement, cela ne fonctionne pas bien. Et que d'entendre un matin sur deux que « mais c'est de la merde », cela ne donne pas envie de participer à des *codes reviews*. Cela ne donne pas envie de montrer son code, de lire le code d'autrui et de poser des questions. Autant de pratiques qui sont pourtant fondamentales pour obtenir, à la fin, un logiciel de qualité. Je reste convaincu que la mise en place d'un contrat social (lire ou relire « *Du contrat social* », de Rousseau, ne fait jamais de mal) est quelque chose de primordial pour le bon fonctionnement d'une équipe et pour permettra à chacun de non seulement travailler dans de bonnes conditions, mais aussi de s'épanouir et de s'améliorer en partageant. ■

BASH, LES BONNES PRATIQUES

Romain Pelisse [Sustain Engineer @Red Hat]

Relecture Aurélie Garnier

À l'heure du DevOps et de l'« infrastructure as code », la conception de scripts Bash est loin d'avoir disparu, comme certains le pensaient il y a quelques années. Avec des besoins d'automatisation de plus en plus présents, mais aussi de plus en plus souvent placés entre les mains de développeurs plutôt que d'administrateurs système, de nombreux scripts sont produits, et jouent des rôles toujours plus critiques dans la maintenance des systèmes. Mais, paradoxalement, si ces développeurs sont souvent des experts sur leur langage de programmation, qu'il s'agisse de Java, C# ou autre Python, ils semblent souvent comme frappés d'amnésie quand ils implémentent leurs scripts Bash ! En fait, si la syntaxe de ce dernier est très permissive, et ses mécanismes d'interprétation très différents d'un langage de programmation dit « traditionnel », il n'en offre pas moins de nombreuses techniques et pratiques pour en assurer une exécution propre, saine et fiable – pourtant peu utilisées. Démonstration.

Mots-clés : Bash, Shell, DevOps, Cloud, Programmation, Linux

Résumé

Cet article couvre un ensemble de bonnes pratiques et de techniques à utiliser lors de la conception de scripts Bash, pour leur assurer une exécution fiable et robuste.

Les scripts shell (Bash ou autres d'ailleurs) sont partout. On en trouve un peu à toutes les sauces, et dans l'ensemble, ils ne sont que rarement irréprochables, même du point de vue strictement technique. Cependant, ils font la plupart du temps le travail qu'on leur demande, et, s'ils n'y parviennent pas, on se contente généralement de les exécuter une nouvelle fois plutôt que de chercher à les rendre plus « imperméables » aux erreurs.

Leur manque de robustesse, de propreté ou même de certaines fonctionnalités n'est jamais réellement un problème, car ils ne sont que rarement exécutés. Généralement uniquement au démarrage d'une machine ou d'un serveur, ou

même parfois encore plus rarement – par exemple lors d'une installation.

Dans ce genre de contexte, s'ils ne s'exécutent pas correctement, ils sont simplement relancés et/ou rapidement corrigés à la main. On y perd un peu de temps, mais étant donné leur relative faible fréquence d'utilisation, peu de personnes reviennent dessus.

Avec l'émergence des environnements « cloud » et l'automatisation des déploiements de systèmes comme d'applications, ces mêmes scripts d'installation qui étaient exécutés « une fois de temps en temps » ou simplement qui « plantent parfois sans explication », vont désormais être exécutés presque tout le temps - lors du démarrage d'une nouvelle instance ou de la construction d'un environnement de tests par exemple.

Bref, dans ce contexte, ce comportement aléatoire va devenir une source de souffrance constante. Il est donc essentiel de désormais concevoir ces scripts de manière robuste, propre et structurée, pour leur permettre de supporter plus aisément aléas et cas limites.

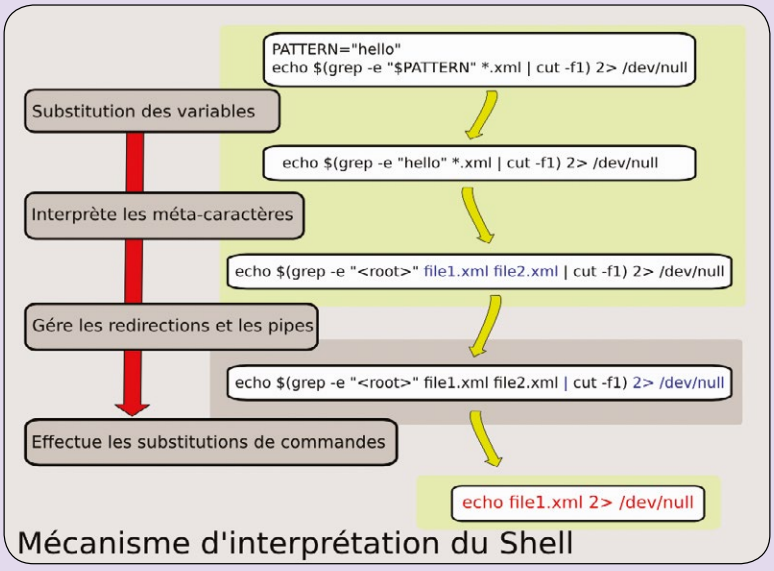
La suite de cet article va donc décrire un ensemble de bonnes pratiques et de conseils techniques visant à structurer le mieux possible nos scripts shell. Ce premier article se concentrera sur l'organisation des scripts et l'utilisation des fonctions. Un autre article sera dédié à des techniques de programmation.

1 En-tête du script

Tout script doit commencer par une ligne de commentaire indiquant au système d'exploitation quel interpréteur ce dernier doit utiliser pour l'exécuter. Ouvrons, à titre d'exemple, un script issu du répertoire **/etc/init.d/** :

Le mécanisme d'interprétation du Bash

Pour être sûr de bien comprendre l'exécution des nombreux extraits de scripts illustrant cet article, une petite piqûre de rappel – sous forme graphique, sur la manière dont le Bash interprète les lignes de commandes qui lui sont passées :



Mécanisme d'interprétation du Shell

```
$ head -3 /etc/init.d/ntpd
#!/bin/bash
#
# ntpd          This shell script takes care of starting and stopping
```

Lors de l'exécution de ce script, le système d'exploitation sait donc quel interpréteur invoquer (en l'occurrence **bash**). Notez aussi que cette pratique ne se limite bien évidemment pas au simple script Shell, puisqu'on la retrouve pour les scripts Ruby ou Python.

Attention, c'est une bonne pratique, mais comme toutes les pratiques, elle n'est en aucun cas obligatoire, et un script shell sans cet en-tête peut être exécuté sans erreur.

Ceci reste néanmoins très pertinent, car si, sur la plupart des systèmes, le raccourci **/bin/sh** pointe vers **/bin/bash**, comme on le voit ci-dessous, il est bien plus prudent d'utiliser directement **/bin/bash** et d'expliciter ainsi de manière très claire de quel interpréteur - et de quelles fonctionnalités associées, votre script a besoin pour s'exécuter sans erreur.

```
$ ls -l /bin/sh
lrwxrwxrwx. 1 root root 4 Aug 21 14:22 /bin/sh -> bash
```

En effet, en cas d'absence de bash sur le système, notre script ne sera pas exécuté, alors que si le raccourci pointe vers un autre interpréteur, il risque de s'exécuter partiellement ou, encore pire, sans problème - sauf dans certains cas limites, qui (loi de Murphy **[1]** oblige) se feront un plaisir de surgir au pire des moments.

Bref, ne jouons pas avec le feu, et prenons toujours soin de spécifier exactement l'interpréteur à utiliser. Cette information servira autant au système qui l'exécute, qu'au mainteneur du script, qui saura ainsi quelles fonctionnalités utiliser (lui permettant de savoir s'il peut, par exemple, utiliser ou non les tableaux associatifs introduits, depuis la version 4, dans la syntaxe du Bash **[2]**).

2 Fonction en Bash

Si un script shell est par nature un code procédural **[3]**, il n'en reste pas moins que l'on peut factoriser son code à l'aide de **fonctions**, comme tout autre langage de programmation. Ces dernières ne sont malheureusement que rarement utilisées par de nombreux concepteurs de scripts, qui se contentent généralement d'un « bon vieux » copier/coller de derrière les fagots, aboutissant rapidement à une kyrielle de scripts au contenu plus ou moins similaire...

Ce qu'il est intéressant de noter, c'est que ce genre de pratiques est fortement prohibé dans les autres domaines de la programmation. Néanmoins, cela ne semble poser aucun problème, par exemple, à un développeur Java chevronné, qui traque la moindre ligne de copier/coller dans son code à l'aide de CPD **[4]**, de créer trois scripts distincts - tous sur la même base, sans chercher à en factoriser quoi que ce soit...

Sans continuer plus longtemps sur les dangers liés au copier/coller de code (bug à corriger dans plusieurs endroits, code plus long à lire, etc.), qui ne sont plus à démontrer, on peut noter qu'en se privant de fonctions, le concepteur d'un script se prive non seulement d'une facilité certaine de réutilisation, mais aussi d'un mécanisme d'encapsulation fort pratique.

En effet, il est possible de déclarer au sein d'une fonction des variables qui lui seront locales. C'est non seulement très pratique, et améliore la lisibilité, mais cela permet aussi de réduire le nombre de variables utilisées et déclarées lors de l'exécution du script, évitant, par exemple, de réutiliser par inadvertance une variable dans une autre partie du script.

Par ailleurs, il est assez élégant, pour accroître la lisibilité et la propreté du script, d'utiliser ces préfixes pour renommer les arguments d'entrée de la fonction :

```
my_function() {
    local first_arg="${1}"
    local second_arg="${2}"
}
```

Vous noterez aussi, au passage, la syntaxe utilisée pour déclarer une fonction. En effet, bash en reconnaît plusieurs, dont la suivante, plus « verbeuse » :

```
function my_function() {
    local first_arg="${1}"
    local second_arg="${2}"
}
```

Cette dernière permet d'omettre les parenthèses lors de la déclaration de la fonction :

```
function my_function {
    local first_arg="${1}"
    local second_arg="${2}"
}
```

Néanmoins, la première syntaxe est à la fois plus expressive - la présence des parenthèses désignant clairement le nom qui les précède comme un nom de fonction, mais aussi plus succincte, et nous la privilégierons donc pour la suite.

Un dernier point sur les fonctions, avant de passer à la suite. Bien qu'il soit possible de déclarer une fonction à peu près n'importe où dans un script, il est de loin préférable de les regrouper au début du fichier, juste après les imports de fichiers annexes (voir plus loin) :

```
#!/bin/bash

./lib/commons.sh

my_function() {
    local first_arg="${1}"
    local second_arg="${2}"
}

...
# début du script en tant que tel
```

2.1 Variables internes dans les fonctions

Il est souvent méconnu que Bash dispose de nombreuses variables internes **[5]** qui sont fort pratiques dans la conception des scripts. Spécialement dans le cas des fonctions, car

elles permettent, entre autres, de récupérer le nom de la fonction en cours d'exécution :

```
complex_fonction() {
    # ...
    local status="${?}"
    if [ ${status} -ne 0 ] ; then
        log "a command failed in $FUNCNAME with status
${status}"
    fi
    #...
}
```

3 Fonction usage()

La toute première chose à écrire dans un script shell est la fonction **usage()** qui décrira comment utiliser le script en question. C'est élémentaire, mais, dans les faits, si vous regardez la plupart des scripts shell qui vous sont livrés avec votre dernière distribution Linux (par exemple) vous constaterez, à votre grande surprise, que même les « grands noms » de notre industrie ne se tiennent que rarement à cette convention.

Loin de se limiter à constituer une bonne pratique, cette fonction va tout d'abord nous permettre de réfléchir, avant de rentrer tête baissée dans la conception, à l'interface que nous souhaitons exposer pour notre script. Quand nous parlons ici d'interface, il ne s'agit pas d'interface homme/machine, mais plutôt de l'API, en quelque sorte, que notre script va offrir.

Ce point est d'autant plus crucial que nos scripts, devenant plus robustes et plus utiles, vont rapidement se multiplier et, surtout, s'appeler mutuellement les uns les autres.

```
usage() {
  echo "$(basename ${0}) [-v] -n <name-of-env>
  echo ""
  echo "-n name of the environment"
  echo "-v verbose mode"
}
```

Lorsque votre script est invoqué sans aucun argument, il devrait, par défaut, appeler la fonction **usage()** et s'arrêter avec un statut « zéro ». Ce comportement est très appréciable car, tout d'abord, si, lors de l'invocation manuelle du script, une erreur de saisie aboutit à l'exécution du script sans argument, on est sûr de ne pas exécuter tout de même le script. C'est peu de chose, mais c'est déjà appréciable, car sur un système en production, ceci peut avoir de graves conséquences.

Ensuite, quand vous n'avez pas exécuté un script depuis longtemps, vous aurez vraisemblablement oublié quel paramètre lui passer et comment le valoriser. Il suffit alors d'exécuter le script sans argument pour obtenir la syntaxe appropriée. Si vous ajoutez systématiquement ce comportement à tous vos scripts, vous pourrez ainsi aisément obtenir de l'aide, sans prendre le risque de déclencher de manière involontaire une quelconque opération.

Cette pratique est nettement plus efficace et utile que la rédaction de documentation technique ou de guides d'utilisation ou d'exploitation, que beaucoup d'entreprises ou d'organisations semblent tant apprécier. En effet, ces documents, vivant dans un espace généralement complètement différent de celui des scripts qu'ils décrivent, ont tendance très très rapidement à se désynchroniser de l'évolution naturelle du script, et à se retrouver complètement obsolètes ou, pire, erronés.

À l'inverse, la fonction **usage()**, quant à elle, sera plus aisément mise à jour, vraisemblablement au moment même où le mainteneur du script optera pour ajouter ou modifier des arguments. En outre, les utilisateurs du script préféreront se référer plutôt à cette aide, directement accessible depuis la console dans laquelle ils travaillent qu'à un document, qu'il faudra d'abord localiser, avant de le télécharger pour, enfin, l'ouvrir avec un autre logiciel...

Tous ces détails peuvent sembler triviaux, mais vous verrez que, dans la pratique, on apprécie vite non seulement le confort qu'ils apportent, mais surtout la structure qu'ils confèrent. Nul besoin d'un document de plus de dix pages pour s'assurer qu'un processus est bien documenté si l'ensemble des scripts dispose d'une méthode **usage()**...

4 Gérer des librairies de fonctions

Comme évoqué dans la section précédente, le bash permet, comme tout autre langage de script, d'importer des fonctions, et même des variables situées dans un autre fichier. Cette capacité est fort pratique, et pourtant, malheureusement, fort peu utilisée.

4.1 La fonction unify_args()

Comme avec tout autre de langage de programmation, nous développons aussi en Bash des fonctions réutilisables qui peuvent donc être réutilisées dans l'ensemble de nos

scripts. Prenons l'exemple d'une fonction conçue par un collègue, il y a quelques années, que j'ai souvent réutilisée, **unify_args()** :

```
unify_args () {
    local tmp=$(mktemp)

    for i in "$@"; do
            echo "$i" >>"${tmp}"
    done

    RC=$(cat "${tmp}" | sort | uniq)
    clean "${tmp}"
}
```

Comme son nom l'indique, cette fonction permet d'unifier une série d'arguments en une seule chaîne. C'est un traitement relativement simple, mais que vous aurez probablement à réutiliser dans de nombreux endroits. La logique qu'elle encapsule n'est pas très élaborée, mais en la plaçant dans une fonction, on peut non seulement la réutiliser facilement, mais aussi réduire la taille du code là où elle est invoquée.

En outre, cette implémentation se charge aussi du « nettoyage », en effaçant le fichier temporaire qu'elle utilise. Cette plomberie n'ayant pas grand intérêt en tant que telle, il est appréciable de la voir « dissimulée » dans notre fonction.

On peut remarquer aussi que l'implémentation de la fonction présentée ici n'est pas très optimale. Elle utilise un fichier temporaire - soit une entrée/sortie, ce qui est peu performant. Mais l'avantage d'avoir regroupé ce code dans une fonction est que, si jamais quelqu'un venait à réimplémenter la fonction sous une forme plus efficace, l'ensemble de scripts l'utilisant bénéficierait immédiatement du gain de performances.

4.2 La fonction remote_open()

Étudions maintenant une autre fonction, **remote_open**, qui permet d'établir une connexion SSH vers un hôte distant :

```
remote_open () {
    if [ -z "${IDENTITY}" -o -z "${HOST}" ]; then
            log "remote_open needs IDENTITY (${IDENTITY}) and
HOST (${HOST}) to be set"
        exit 1
    fi
   log "Setting up shared ssh connection"
    if [ -z "${DRYRUN}" ]; then
            if [ ! -z "${CONTROLDIR}" -o ! -z "${CONTROLOPTS}"
]; then
            log "SSH MASTER CONNECTION ALREADY UP: controldir:
'${CONTROLDIR}' \
                controlopts: '${CONTROLOPTS}'"
                return
            fi

            log "ssh [...] -C -i ${IDENTITY} root@${HOST} $@"
            CONTROLDIR="$(mktemp -d)"
        ssh ${SSH_OPTIONS} -o ControlMaster=auto -o
"ControlPath=${CONTROLDIR}/%r-%h:%p"
            \-i "${IDENTITY}" "root@${HOST}" -Nf
            rc="$?"

            CONTROLOPTS="-o ControlMaster=auto -o
ControlPath=${CONTROLDIR}/%r-%h:%p"
            log "SSH MASTER CONNECTION UP: controldir:
'${CONTROLDIR}' controlopts: '${CONTROLOPTS}'"

            # are /not/ readonly by intend, otherwise we can not
unset them
            export CONTROLOPTS
            export CONTROLDIR

            return "$rc"
    else
        log "DRYRUN: ssh -q -o ControlMaster=auto -o
ControlPath=<tmp>/%r-%h:%p \
            -o ConnectTimeout=10 -o UserKnownHostsFile=/dev/null
-o StrictHostKeyChecking=no -C -i ${IDENTITY} root@${HOST} $@"
    fi
}
```

Si vous étudiez attentivement cette fonction, vous réaliserez qu'elle est assez longue, essentiellement pour deux raisons. Tout d'abord, à l'aide d'une variable globale, on peut la rendre extrêmement verbeuse, ce qui est très précieux pour analyser tout problème lors de l'exécution des scripts. En outre, elle prend aussi en charge la mise en place d'une connexion SSH « maître » **[6]** qui pourra être réutilisée par les commandes ultérieures.

Ainsi, si un script exécute plusieurs commandes sur un hôte distant, il va réutiliser la même connexion SSH, plutôt que d'en créer une nouvelle. Ceci va grandement améliorer les performances, en supprimant de multiples - et inutiles, établissements de connexion. Ce travail, qui fut assez pénible et un peu laborieux à réaliser, il faut le reconnaître, va donc profiter à l'ensemble des scripts et leur apporter un gain de performances réellement non négligeable.

Comme évoqué dans la précédente section, l'utilisation de fonctions a donc permis non seulement de clarifier le code, mais aussi de rendre nos scripts plus efficaces.

5 Import des librairies

Comment importer un script ? De façon évidente, importer un script à l'aide d'un chemin absolu est lié très étroitement à la manière dont il est installé sur le système, mais,

à l'inverse, utiliser un chemin relatif, bien que plus portable pose aussi d'autres complications, si le script n'est pas invoqué dans le contexte attendu.

Il faut donc faire un choix sagement mûri entre l'utilisation de chemins relatifs ou absolus lors de l'import des librairies de fonctions. Soit vous optez pour utiliser un chemin absolu, il est fortement recommandé de réaliser un packaging de vos scripts sous forme de RPM, de paquet Debian, ou autre, de manière à garantir le chemin, lors de l'installation des scripts.

Soit vous optez pour l'utilisation d'un chemin relatif, qui sera plus souple, mais vous contraindra néanmoins à toujours exécuter les scripts depuis le même répertoire. L'extrait de code ci-dessous utilise ainsi un chemin relatif pour l'import, mais impose donc à l'utilisateur de ce dernier de l'exécuter depuis la racine du répertoire abritant les scripts dont il dépend :

```
#!/bin/sh

. "$(dirname "${0}")/common/base"
```

Conclusion : patron d'un script complet

Faisons un premier bilan des différentes bonnes pratiques évoquées dans cet article, sous la forme d'un script d'exemple qui en fera une première synthèse. Supposons, à titre d'exemple, que nous souhaitions réaliser un script chargé de détruire un environnement complet déployé Amazon.

Par environnement « complet », on entend ici un ensemble d'instances, sur lesquelles sont déployés nos applicatifs, associées aussi à un *bucket S3* et à une *elastic IP*, qui permettent de rendre le système accessible depuis le monde extérieur.

Ainsi, notre script doit s'occuper non seulement de détruire les instances, mais aussi de nettoyer le *bucket*, et enfin de libérer l'*elastic IP*.

```
#! /bin/sh

. "$(dirname $0)/common/ec2"

usage () {
    echo "${BASENAME} [-h] -B <bucketname> <instance-id>
[<instance-id>...]"
        log "- a script to terminate all provided instances and
the delete
        log "the associated bucket."
}

delete_instances () {
...
}

empty_bucket () {
...
}

free_ip() {
...
}
```

Comme décrit en amont, notre script commence par indiquer l'interpréteur à utiliser (dans notre cas **/bin/sh**, car on n'utilise ici aucune fonctionnalité spécifique à bash), suivi de l'import des librairies nécessaires (ici, la librairie **ec2**), puis la fonction **usage** qui documente enfin l'utilisation du script.

À la suite de la fonction **usage**, on trouve les différentes fonctions spécifiques à ce script. Par souci de concision, leur code n'est pas détaillé ici. On pourra noter que vraisemblablement ces fonctions utiliseront les primitives fournies par la librairie **ec2**.

La plupart des pratiques ou considérations évoquées tout au long de ce premier article tiennent plus de la convention ou de l'organisation, et ne sont pas, en tant que telles, spécifiques au Bash ou à d'autres langages de programmation, néanmoins, elles forment, de notre analyse, un prérequis nécessaire à la conception de scripts robustes, fiables et faciles à maintenir.

La suite au prochain article, c'est-à-dire dans les pages suivantes... ■

Références

[1] La Loi de Murphy : http://en.wikipedia.org/wiki/Murphy's_law

[2] Tableau associatif en Bash : http://www.linuxjournal.com/content/bash-associative-arrays

[3] Programmation procédurale : http://fr.wikipedia.org/wiki/Programmation_proc%C3%A9durale

[4] CPD : http://pmd.sourceforge.net/cpd.html

[5] Variables internes du Bash : http://tldp.org/LDP/abs/html/internalvariables.html

[6] Connexion SSH partagée : https://puppetlabs.com/blog/speed-up-ssh-by-reusing-connections

BASH : ALLER ENCORE PLUS LOIN AVEC LES BONNES PRATIQUES

Romain Pelisse [Sustain Engineer @Red Hat]

Dans le précédent article, nous avons déjà présenté un ensemble de conventions et d'astuces permettant d'améliorer grandement la lisibilité et la maintenance de scripts Bash. Dans ce second article, nous allons aller encore plus loin et discuter de nombreux aspects de l'interpréteur, et de la syntaxe qui lui est associée, qui assureront la robustesse de vos scripts.

Mots-clés : Bash, Shell, DevOps, Cloud, Programmation, Linux

Résumé

Cet article couvre un ensemble de techniques et de mécanismes spécifiques à Bash qui, utilisés systématiquement, apportent beaucoup de robustesse et de fiabilité à l'exécution des scripts.

Sans surprise, le premier point que nous allons évoquer dans cet article concerne la plus primitive des primitives de la programmation : les variables. Nous étudierons ensuite un ensemble de techniques permettant de gérer proprement les erreurs lors de l'exécution du script, avant de finir par une section « rubrique à brac », contenant une série d'astuces et de techniques en tout genre.

1 Protéger ses variables

1.1 Overquoting...

Vous en avez probablement déjà fait l'expérience, les scripts shell sont relativement fragiles. Il suffit parfois de passer un mauvais argument pour obtenir un comportement totalement différent de celui qu'on attendait. Un exemple rapide à l'aide de la commande **cp** :

```
function copy() {
    local src=${1}
    local dest=${2}

    cp ${src} ${dest}
}

$ copy a b
$ copy a "b d" # problème...
cp: target `d' is not a directory
```

À l'exécution de la dernière commande, une erreur se produit, car **cp** tente de copier le fichier **a** et le fichier **b** dans un répertoire **d** qui n'existe pas. Le comportement souhaité était en fait la copie du fichier **a** dans le fichier « **b d** ».

Cet exemple, profondément inutile, a pour seul mérite de bien illustrer le problème. Il est très aisé de se retrouver avec un espace ou un caractère spécial, inattendu, au sein d'une variable. C'est pour ceci qu'il est fondamental de bien

protéger ses variables, afin de se prémunir de ce genre de dysfonctionnement.

Pour ce faire, vous avez déjà dû noter que nous privilégions sciemment l'utilisation d'accolades autour du nom des variables. C'est déjà une bonne pratique en soi, mais ce n'est pas suffisant.

Il est aussi important de systématiquement - sauf si c'est contraire au comportement désiré, utiliser des guillemets pour encadrer les variables. On s'assure ainsi que lors de l'interprétation de la variable, et donc de son remplacement par sa valeur, l'intégralité du contenu de cette dernière sera interprété ensuite correctement par le shell :

```
function copy() {
    local src="${1}"
    local dest="${2}"

    cp "${src}" "${dest}"
}
```

Cette pratique est qualifiée par l'un de mes collègues, qui en est pourtant un ardent défenseur, d' *overquoting*. Elle peut sembler fastidieuse pour un gain assez réduit, mais, dans les faits, elle vous prémunira de nombreuses erreurs lors de l'exécution, et, surtout, vous permettra d'obtenir des messages d'erreur beaucoup plus explicites et cohérents.

1.2 Utiliser des variables globales en lecture seule

Les variables globales sont un mécanisme généralement fort critiqué. Pour s'en convaincre, il suffit de se rappeler qu'un des objectifs avoués de la programmation orientée objet était de rendre ce genre de variables inaccessibles depuis l'extérieur en les plaçant dans des classes, et de n'autoriser que les seules fonctions qui nécessitent d'y accéder à les manipuler.

Leur usage est beaucoup plus admis dans la conception de scripts et, dans ce contexte, il n'est pas non plus facile de pouvoir entièrement s'en passer. Néanmoins, il faut s'assurer qu'elles ne soient pas détournées de leur usage initial, ou pire encore, surchargées à notre insu lors de l'import d'autres scripts, par exemple.

Heureusement, le Bash dispose de mots-clés permettant de s'en assurer. Ainsi, il est donc prudent de déclarer les variables globales qui ne sont pas destinées à être modifiées, ce qui est le cas de la plupart de ces dernières, comme étant en lecture seule :

```
$ readonly var="true"
$ var="false"
bash: var: readonly variable
```

Ainsi, dès la première modification maladroite ou involontaire, vous serez notifié très clairement de l'existence d'un problème ou d'un conflit. Sans l'utilisation de ce mot-clé, votre variable globale verra sa valeur modifiée, ce qui entraînera vraisemblablement, en amont du script, d'autres problèmes difficilement explicables (« Mais pourquoi le répertoire créé porte-t-il le nom de l'utilisateur exécutant le script ??? »).

1.3 Tester la nullité des variables et des arguments

Au sein d'un script, il est assez courant, à la suite d'une erreur d'exécution, de se retrouver avec une variable non définie - alors que le script attend d'elle d'avoir été associé à une valeur. Ce genre de problème est notamment très fréquent lors de l'utilisation de substitution de commandes :

```
$ files_to_edit=$(ls -1 "${folder}")
# si 'folder n'a pas été défini, la liste contient le contenu du
répertoire courant !
```

En conséquence, il est prudent de toujours veiller à tester une variable, pour vérifier si elle est définie, avant de s'en servir. Cette remarque s'applique bien évidemment encore plus aux arguments des fonctions.

À titre d'illustration, reprenons notre exemple, vide de sens, mais très didactique, de la fonction **copy** :

```
copy() {
    local src="${1}"
    local dest="${2}"

    if [ -z ${src} ] ; then
        echo "le premier argument, le fichier source n'a pas été
fourni".
        exit 1
    fi

    if [ -z ${dest} ] ; then
        echo "le second argument, le répertoire ou fichier de
destination,
            n'a pas été fourni."
        exit 2
    fi

    cp "${src}" "${dest}"
}
```

Syntaxe des substitutions de commande

Une remarque importe sur les substitutions de commandes : il faut souligner que la syntaxe utilisant des *backquotes* se voit régulièrement préférer celle utilisée dans cet article et employant le symbole **$**, comme pour l'interprétation des variables, associée à des parenthèses.

```
# syntaxe 'historique'
$ hostname=`hostname'
# 'nouvelle' syntaxe:
$ hostname=$(hostname)
```

On notera au passage, toujours sur le sujet des substitutions de commandes, qu'il est rarement nécessaire d'entourer ces dernières de guillemets, puisque la nature même de leur exécution garantit que le résultat soit proprement associé à la variable.

2 Gérer les arguments : getopt

Comme nous l'avons décrit précédemment, il est crucial de doter ses scripts d'arguments appropriés à leur usage et les plus explicites possible. Mais, évidemment une fois les arguments transmis, il reste à les traiter. Ceci inclut, dans le cas du Bash, d'associer leurs valeurs aux bonnes variables, et de vérifier que ces valeurs soient cohérentes avec les attentes du script.

Ce genre de travail est fastidieux, et, heureusement, il existe différents mécanismes pour aider le développeur dans cette tâche. Nous avons retenu ici l'utilisation de la commande **getopt** qui, sans être la plus souple, ni la plus puissante des options, reste néanmoins un bon outil, aussi très structurant.

La commande **getopt** permet de déclarer, par une série de lettres, les options supportées par le script. Une option étant ici un caractère préfixé d'un **-**. Ainsi, en déclarant la chaîne **pde** à **getopt**, vous indiquez à ce dernier que votre script supporte l'utilisation d'options **-p**, **-d** et **-e**.

En outre, on peut indiquer à **getopt**, par l'ajout d'un **:** après la lettre, que l'argument est associé à une valeur. Ainsi **p:** indique que le script reçoit un argument de type **-p valeur**.

La commande **getopt** s'imbrique naturellement dans les structures **while** et **case** du Bash, et permet donc d'analyser des arguments aussi aisément que le montre le code suivant :

```
while getopts hB: OPT; do
    case "$OPT" in
        h)
            usage
            exit 0
            ;;
        b)
            readonly BATCH_NAME=${OPTARG}
            ;;
        *)
            echo "Unrecognized options:${OPTARG}"
            ;;
    esac
done
```

Ainsi, cette structure nous permet de gérer un premier cas d'erreur possible : l'argument non reconnu. En outre, **getopt** lors de son analyse des arguments reçus, va non seulement définir le nom de l'argument analysé dans la variable **OPT**, mais aussi placer la valeur associée, le cas échéant, dans la variable **OPTARG**.

Et il ne reste donc plus qu'à définir les variables associées aux options qui sont nécessaires pour le script. À la sortie de cette boucle **while**, on peut aisément tester si toutes les valeurs nécessaires ont été définies, et si ces dernières semblent pertinentes :

```
if [ -z "${BATCH_NAME}" ]; then
    echo "The batch name is not optionnal."
    usage
    exit 1
fi
```

Enfin, dernier avantage, **getopt** ne nous interdit pas d'ajouter une série d'arguments supplémentaires qu'il ne gérera donc pas lui-même. Supposons, par exemple, que nous souhaitions passer aux scripts toute une série de noms d'instance de machines virtuelles, on peut aisément extraire ces arguments, après le traitement effectué par **getopt**.

Et, comme Bash supporte désormais la notion de tableau **[1]**, la récupération de ces arguments peut se faire en quelques lignes :

```
shift $(expr $OPTIND - 1)
declare -a INSTANCES="$@"

if [ -z "${BATCH_NAME}" -a -z "${INSTANCES}" ]; then
    usage
    exit 1
fi
```

Maintenant, il faut aussi reconnaître que **getopt** a également quelques limites claires. Par exemple, il ne peut pas analyser d'arguments plus explicites tels que **--help**. Comme sous-entendu au début de cette section, ce n'est donc certainement pas le *framework* absolu pour l'analyse d'arguments, mais il offre un moyen rapide et simple aux développeurs de scripts pour analyser leurs arguments d'entrée.

3 Gestion des erreurs

Un autre point important, lors de la conception de scripts, spécialement de scripts d'infrastructure, est la **gestion des erreurs**. Malheureusement, le Bash a plutôt mauvaise réputation dans ce domaine. La plupart des scripts ne vérifient que rarement que le statut d'une commande est bien égal à **0** après son exécution, et, à l'inverse, bien souvent, des scripts continuent à s'exécuter « malgré tout ».

Heureusement, il ne s'agit que de mauvaises pratiques et, en aucun cas, de défaut irrémédiable lié à l'interpréteur en lui-même. En effet, il est possible de configurer, lors de l'exécution, ce dernier pour s'interrompre immédiatement à la première commande exécutée retournant une erreur (statut différent de zéro) :

```
set -e
```

Utiliser systématiquement cette option, au début de chaque script, dès les premières heures de sa conception, en facilite le développement, puisque son exécution s'arrête dès que la première commande retourne une valeur de statut différente de zéro, et non plusieurs lignes plus loin.

En effet, il est fréquent que les scripts continuent de s'exécuter malgré l'apparition d'une erreur, et que le « crash », en tant que tel, n'intervienne donc que bien plus tard. Ce phénomène, fort désagréable, rend souvent difficile l'identification de la cause racine du problème, en plus d'entraîner des effets de bords (ex : création d'un répertoire utilisateur alors que l'utilisateur n'a pas été créé).

Ainsi, utiliser systématiquement ce mécanisme évite que vos systèmes ne se retrouvent dans un état incohérent, à la suite d'une exécution d'une série d'opérations qui n'auraient jamais dû avoir lieu, puisqu'une commande exécutée en amont a échoué.

Notez aussi que cette commande est réversible au cours de l'exécution du script, ce qui peut parfois se révéler fort pratique - voire nécessaire, si votre script doit néanmoins continuer à s'exécuter malgré une erreur, ou s'il est simplement en mesure de gérer l'erreur (par exemple, en réessayant simplement d'exécuter la commande une nouvelle fois).

La fonction suivante illustre ce dernier point :

```
transfer_file() {
    local source="${1}"
    local target="${2}"

    local timeout=120
    local pace=5
    local wait_since=0

    while [ ${wait_since} -le ${timeout} ] ; do
        set +e
        scp "${source}" "${target}"
        if [ ${?} -ne 0 ] ;
            let wait_since=${wait_since}+${pace}
        else
            return
        fi
        set -e
    done
    exit 1
}
```

3.1 Gérer les erreurs dans les commandes imbriquées

Une des fonctionnalités internes du Bash qui gagne réellement à être connue est très certainement la variable **$PIPESTATUS**. Cette variable interne permet en effet de vérifier que chacune des commandes d'une série de commandes imbriquées a été exécutée sans erreur !

Pour s'en convaincre, étudions l'exemple ci-dessous :

```
$ ls -1 | mail | cat | cut -f1
No mail for rpelisse
$ echo "${?}"
0
$ ls -1 | mail | cat | cut -f1
No mail for rpelisse
$ echo "0:${PIPESTATUS[0]} 1:${PIPESTATUS[1]} 2:${PIPESTATUS[2]}
3:${PIPESTATUS[3]}"
0:0 1:1 2:0 3:0
```

Dans cet extrait de code, l'appel à la commande **mail**, seconde commande invoquée au sein de cette série de commandes imbriquées, échoue, car l'utilisateur ne dispose pas de compte de messagerie électronique locale. Néanmoins, si l'on se contente de tester la valeur de la variable **${?}**, en bout de chaîne d'exécution, cette anomalie n'apparaît pas, puisqu'elle est, pour ainsi dire, masquée par l'exécution avec succès de la dernière commande, **cut** en l'occurrence, qui, elle, s'est exécutée avec succès.

```
var=$(ls -1 | mail | cat | cut -f1=
for status in "${PIPESTATUS[@]}"
do
  if [  ${status} -ne 0 ] ; then
    exit ${status}
  fi
done
```

Heureusement, à l'aide de cette variable interne, **PIPESTATUS**, ou plutôt en fait de ce tableau, il est possible de vérifier, a posteriori, que, à chaque étape de l'exécution d'une série de commandes imbriquées, tout s'est déroulé sans erreur.

PIPESTATUS permet donc de rendre l'utilisation des commandes imbriquées beaucoup plus sûres, ce qui n'est pas négligeable, car l'utilisation de *pipes* apporte de nombreux avantages dans un script Bash.

Non seulement elle réduit la taille du code shell à une seule ligne, au lieu de plusieurs, mais elle permet surtout de passer les informations d'une commande à l'autre, directement en mémoire sans passer par des fichiers temporaires.

Ce dernier mécanisme conclut donc l'ensemble des éléments évoqués pour rendre nos scripts plus robustes. Une fois toutes ces techniques mises en place dans vos propres scripts, vous constaterez avec plaisir qu'ils sont désormais plus fiables, et que, par conséquent, vous vous en servirez non seulement plus souvent, mais que vous n'hésiterez plus à leur confier même des tâches plus critiques, car vous aurez désormais les bonnes pratiques pour leur garantir un comportement correct.

4 Rubrique à brac

4.1 Écrire un fichier temporaire

Lors de la conception d'un script Bash, on se retrouve, tôt ou tard, à devoir utiliser des fichiers temporaires. Et comme avec tous les langages de programmation du monde, on se retrouve dans la position désagréable de devoir « choisir » où placer ce fichier dans l'arborescence du système.

Ceci est toujours désagréable, car on ne sait jamais vraiment où placer ce fichier. Après tout, peut-on vraiment garantir qu'un emplacement existera toujours sur l'ensemble des systèmes où le script s'exécutera ? On peut opter pour un choix raisonnable, en prenant par exemple, le répertoire **/tmp**. Il est vraisemblable que ce dernier existera toujours, quel que soit le système, néanmoins ceci ne résout pas tout le problème pour autant.

Reste le nom du fichier. Comment s'assurer qu'il n'existe pas de fichier portant le même nom dans le répertoire ? Et si c'est le cas, comment faire ? Ajouter un suffixe ? Effacer le fichier existant ? Rapidement, cette simple opération d'écriture dans un fichier pose un grand nombre de questions...

Heureusement, il existe une commande fort utile nommée **mktemp**. Cette dernière permet de créer un fichier (ou un répertoire) dans **/tmp**. La commande garantit donc que le nom généré ne corresponde à aucun autre fichier déjà existant dans ce répertoire, vous libérant de facto de la gestion de potentiels conflits de noms.

L'exemple ci-dessous, bien que présentant très peu d'intérêt en termes fonctionnels, illustre bien l'utilisation de **mktemp** :

```
copy_init_scripts_without_comments() {
    local target_host="${1}"

    # copy script in temporay directory, and removes comments
    directory=$(mktemp -d)
    for file in /etc/init.d/
    do
        sed -e '/#/d' "${file}" > \
            "${directory}/$(basename ${file})"
    done

    # create an archive and push it to the targeted host
    archive=$(mktemp)
    tar czf "${archive}" "${directory}"
    scp "${archive}" \
        "${target_host}":$(date +%Y%m%d)-initscripts-backup.tgz
    rm -rf "${archive}" "${directory}"
}
```

4.2 Processus et PID

Au sein d'un script, on a souvent besoin d'une sorte d'identifiant unique pour indiquer, par exemple, comment

aisément retrouver, au sein des fichiers de journalisation, l'instance du script qui a été responsable de telle ou telle action. Cet identifiant unique existe en fait déjà naturellement au sein du système, puisque tout script Bash qui s'exécute est, en fait, un processus, et qu'il possède donc, en toute logique, un PID.

Et ce PID est, de manière fort pratique, très aisé à récupérer au sein d'un script :

```
$ echo $$
14092
```

Mais ceci va plus loin, car on peut aussi lancer, au sein d'un script, des commandes en tâches de fond. Pour ce faire, il suffit simplement d'ajouter le symbole **&** à la fin de la ligne de commandes. Ce mécanisme est en général bien connu de la plupart des utilisateurs du shell, mais le fait qu'on puisse récupérer le PID du processus mis en tâche l'est souvent moins :

```
$ dd if=/dev/zero of=/dev/null &
$ pid=$(${!})
$ sleep 120
$ kill "${pid}"
```

Ce mécanisme se révèle très utile pour coordonner un ensemble de scripts s'exécutant en parallèle. Le script ci-dessous illustre bien ceci.

```
...
run() {
  local client="${1}"
  local nb_threads="${2}"
  local logfile="${3}"

  echo -n "starting ${nb_threads} with ${client}..."
  ./run.sh -n 10 ${client} -u "${url}" -t "${topic}" &>
"${logfile}" &
  export LAST_PID="$(echo $!)"
  echo "started (pid:${LAST_PID})"
}


export LAST_PID=""


run -c 10 "${log_dir}/10_consumers.log"
readonly CONSUMERS_10_PID="${LAST_PID}"
run -p 60 "${log_dir}/60_producers.log"
readonly PRODUCERS_60_PID="${LAST_PID}"
run -c 60 "${log_dir}/60_consumers.log"
readonly CONSUMERS_69_PID="${LAST_PID}"

echo "logs redirected into ${log_dir}"
echo -n "waits for 3 minutes before killing 10 consumers..."
if [ "${log}" = "true" ]; then
        tail -f ${log_dir}/*.log &
fi


sleep 180
echo "kill 10 consumers (${CONSUMERS_10_PID})"
kill "${CONSUMERS_10_PID}"
echo  "Done."
```

Note

Ce script a été conçu pour reproduire un problème d'accès concurrent par des processus « consommateurs » et « producteurs » **[2]**. Ce script utilise donc jusqu'à trois sous-processus pour démarrer deux groupes de consommateurs et un groupe de producteurs.

Pour reproduire le problème, il était nécessaire, après un certain temps de démarrage, d'interrompre abruptement un groupe de consommateurs. Ceci a aussi été automatisé dans le script, en utilisant le PID du processus lancé en tâche de fond précédemment.

Conclusion

Ces quelques pratiques et astuces sont loin d'être exhaustives, mais vous auront, espérons-le, donné déjà quelques éléments concrets pour améliorer vos scripts, changer vos habitudes et explorer plus en profondeur les nombreuses options mises à votre disposition par Bash. ■

Références

[1] Tableau associatif en Bash : http://www.linuxjournal.com/content/bash-associative-arrays

[2] Producteur / Consommateur : http://en.wikipedia.org/wiki/Producer-consumer_problem

DIX FAÇONS DE TESTER L'OUVERTURE D'UN PORT

Frédéric Le Roy [Ingénieur informaticien, touche à tout et plus encore]

Lorsqu'un service local ou distant ne fonctionne pas, une action qui est souvent réalisée pour le debuggage consiste à vérifier si le port correspondant au service est ouvert... Mais comment réaliser ce simple test ?

L'OBJECTIF

Tester l'ouverture d'un port peut se faire de différentes manières. Suivant l'environnement, les outils à disposition et vos préférences, il y a plein de manières différentes de faire ce simple test. Nous allons aujourd'hui faire un tour des différentes possibilités en commençant, la politesse l'exige, par la partie adverse, à savoir Windows (car oui il arrive de devoir vérifier si un port sur un serveur Unix/Linux est accessible depuis un poste bureautique sous Windows).

Chaque test sera réalisé en double : sur un port accessible et sur un port fermé.

LES OUTILS

Aujourd'hui, il n'y aura pas de prérequis sur les outils. Nous partons du postulat qu'il va falloir faire avec ce qui est disponible sur la machine !

Manière n°1 – telnet sous Windows (et les autres aussi)

Telnet est un reliquat du passé, d'une époque révolue où tout passait en clair sur le réseau, même votre mot de passe (cette époque est-elle vraiment révolue au final ?). Ce reliquat du passé, qu'on peut trouver encore sur quasiment tous les systèmes d'exploitation aujourd'hui peut servir encore à plein de choses (tester un serveur SMTP, vérifier un serveur Web, etc.). Voici comment faire pour tester un port depuis n'importe quel système d'exploitation.

Port ouvert :

```
C:\Users\stephanie>telnet 192.168.1.10 40406
```

Ici, comme le port est ouvert, tout le contenu de la fenêtre va disparaître. Pour sortir, il faut faire **<Ctrl>** + **<$>**. Un prompt va s'afficher et il suffira de taper **quit** pour sortir :

```
Bienvenue dans le client Telnet Microsoft
Le caractère d'échappement est 'CTRL+$'

Microsoft Telnet> quit
```

Port fermé :

```
C:\Users\stephanie>telnet 192.168.1.10 666
Connexion à 192.168.1.10...Impossible d'ouvrir une connexion à l'hôte, sur le
port 666: Échec lors de la connexion
```

Comparons sur un serveur Debian :

▪ Port ouvert :

```
$ telnet 192.168.1.10 40406
Trying 192.168.1.10...
Connected to 192.168.1.10.
Escape character is '^]'.
```

▪ Port fermé :

```
$ telnet 192.168.1.10 666
Trying 192.168.1.10...
telnet: Unable to connect to remote host: Connection refused
```

À noter pour finir que sous certaines versions de Windows, il faut activer **telnet** dans les paramètres systèmes pour pouvoir l'utiliser.

Manière n°2 - netstat sous Windows

Netstat est un outil qui va permettre d'avoir des informations sur les flux réseaux sur la machine locale.

En utilisant la commande suivante dans une invite de commandes **PowerShell**, il vous permet de vérifier si le port **1099** est ouvert sur votre machine.

Port ouvert :

```
C:\Users\IUSR_METRO> netstat -ao | select-string 1099
  TCP    10.22.55.44:1099     uytltip27:1918     ESTABLISHED   2072
```

Notez le **| select-string** qui est le cousin de notre **| grep**.

Port fermé :

```
C:\Users\IUSR_METRO> netstat -ao | select-string 1099
```

Manière n°3 – netcat, premier acte

Netcat est le couteau suisse des outils réseaux. Il permet de faire énormément de choses. La première méthode pour tester un port avec **nc** est la suivante : **nc -v -z -w <timemout_seconds> <host> <port>; echo $?**. L'option **-z** permet de scanner des services en écoute.

Port ouvert :

```
$ nc -v -z -w 5 192.168.1.10 40406; echo $?
Connection to 192.168.1.10 40406 port [tcp/*] succeeded!
0
```

Port fermé :

```
$ nc -v -z -w 5 192.168.1.10 666; echo $?
nc: connect to 192.168.1.10 port 666 (tcp) failed: Connection refused
1
```

Notez combien cette méthode peut être facilement intégrable dans un script : une jolie ligne d'affichage très claire et un code retour **$?** tout aussi clair.

Manière n°4 – netcat, second acte

Voici une deuxième manière de réaliser le même test avec **netcat**. Ici j'ai en plus enlevé le flag **-v**, ce qui permet d'éviter l'affichage de la sortie du port pour une intégration plus poussée dans un script. Ici nous n'utilisons pas d'option spécifique, nous nous contentons de ne rien écrire sur un port distant ;)

Port ouvert :

```
$ nc 192.168.1.10 40406 < /dev/null ; echo $?
0
```

Port fermé :

```
$ nc 192.168.1.10 666 < /dev/null ; echo $?
1
```

Manière n°5 - wget verbeux

Wget est un outil qui permet de télécharger des ressources en ligne de commandes. Une première manière de l'utiliser est de faire comme si on souhaitait récupérer la ressource.

Port ouvert sur un serveur web :

```
$ wget -qO- 192.168.1.10:40406 | head
<!DOCTYPE html>
<html>
[...]
</html>
```

Port fermé :

```
$ wget -qO- 192.168.1.10:666
```

Il n'y a aucun retour, l'affichage du code retour est différent de **0** :

```
$ echo $?
4
```

Manière n°6 – wget en plus propre

Wget combiné avec l'option **--spider**, permet de simplement vérifier la présence des ressources sans les télécharger.

Port ouvert :

```
$ wget -q --spider 192.168.1.10:40406 ; echo $?
0
```

Port fermé :

```
$ wget -q --spider 192.168.1.10:666 ; echo $?
4
```

Manière n°7 – curl

Curl est le cousin de **wget**. Il permet donc de la même manière de tester simplement un port.

Port ouvert :

```
$ curl telnet://192.168.1.10:40406
^C
```

Ici la commande reste « bloquée » si le port est ouvert, car elle attend qu'on saisisse quelque chose, comme dans une invite **telnet** classique. Il faut simplement faire **<Ctrl>** + **<c>** pour sortir.

Port fermé :

```
$ curl telnet://192.168.1.10:666
curl: (7) Failed to connect to 192.168.1.10 port 666: Connexion refusée
```

Manière n°8 – bash

Bash et le système de fichiers permettent de tester un port.

Port ouvert :

```
$ timeout 1 bash -c 'cat < /dev/null > /dev/
tcp/192.168.1.10/40406' ; echo $?
0
```

Port fermé :

```
$ timeout 1 bash -c 'cat < /dev/null > /dev/tcp/192.168.1.10/666'
; echo $?
bash: connect: Connexion refusée
bash: /dev/tcp/192.168.1.10/666: Connexion refusée
1
```

Une commande idéale pour passer pour un vrai pro ;)

Manière n°9 – nmap

Nmap est un outil qui permet de scanner un réseau de différentes manières. Il permet du coup aussi de vérifier si un port est ouvert.

Port ouvert :

```
$ nmap -p 40406 192.168.1.10

Starting Nmap 6.00 ( http://nmap.org ) at 2015-10-04 22:21 CEST
Nmap scan report for 192.168.1.10
Host is up (0.000048s latency).
PORT      STATE SERVICE
40406/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
```

Port fermé :

```
$ nmap -p 666 192.168.1.10

Starting Nmap 6.00 ( http://nmap.org ) at 2015-10-04 22:23 CEST
Nmap scan report for 192.168.1.10
Host is up (0.000048s latency).
PORT    STATE  SERVICE
666/tcp closed doom

Nmap done: 1 IP address (1 host up) scanned in 0.06 seconds
```

Test des deux ports en une fois :

```
$ nmap -p 40406,666 192.168.1.10

Starting Nmap 6.00 ( http://nmap.org ) at 2015-10-04 22:24 CEST
Nmap scan report for 192.168.1.10
Host is up (0.000066s latency).
PORT      STATE  SERVICE
666/tcp   closed doom
40406/tcp open   unknown

Nmap done: 1 IP address (1 host up) scanned in 0.06 seconds
```

Manière n°10 – en Python

Il peut être parfois nécessaire d'intégrer le test des ports dans des scripts ou une application en Python. Voici comment faire.

Port ouvert :

```
$ python
Python 2.7.10 (default, Jul  1 2015, 10:54:53)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> clientsocket.connect(('192.168.1.10', 40406))
>>> clientsocket.send('\n')
1
```

Port fermé :

```
$ python
Python 2.7.10 (default, Jul  1 2015, 10:54:53)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> clientsocket.connect(('192.168.1.10', 666))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/socket.py", line 228, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 111] Connection refused
```

Manière n°11 – bonus : netstat et la mnémotechnique

Pour tester l'état d'un port en local sur un serveur, il est possible d'utiliser sous Linux la commande **netstat**. Il existe un ensemble d'options qui permet de vérifier l'état des différents ports : **-lapute**. Ce n'est certes pas glamour, mais ça a l'avantage de marquer les esprits !

Port ouvert :

```
$ netstat -lapute | grep 40406
(Tous les processus ne peuvent être identifiés, les infos sur les processus
non possédés ne seront pas affichées, vous devez être root pour les voir
toutes.)
tcp        0      0 darkstar.local:40406    *:*                     LISTEN
fritz      276517     19270/python
tcp        0      0 localhost:40406         *:*                     LISTEN
fritz      276516     19270/python
tcp        0      0 darkstar.local:40421    darkstar.local:40406    ESTABLISHED
fritz      286344     4981/iceweasel
tcp        0      0 darkstar.local:40406    darkstar.local:40421    ESTABLISHED
fritz      286345     19270/python
tcp        0      0 darkstar.local:40407    darkstar.local:40406    ESTABLISHED
fritz      281458     4981/iceweasel
tcp        0      0 darkstar.local:40406    darkstar.local:40407    ESTABLISHED
fritz      281459     19270/python
```

Port fermé :

```
$ netstat -lapute | grep 666
(Tous les processus ne peuvent être identifiés, les infos sur les processus
non possédés ne seront pas affichées, vous devez être root pour les voir
toutes.)
```

Si d'aventure en clientèle vous aviez honte de sortir cette commande, l'équivalent en plus passe-partout est **netstat -taupe**.

LE RÉSULTAT

Et voilà, nous avons vu, comme promis plus de 10 façons de tester l'ouverture d'un port. Vous êtes outillé pour toutes les situations ! ■

LE CLUSTER H-A MYSQL SANS RAMER AVEC GALERA CLUSTER !

Jérôme Delamarche [Consultant indépendant en technologies Open Source – pas toutes !]

De MySQL vous connaissez la réplication ? Mais qui souhaite déployer des technologies qui ne sont pas facilement « dimensionnables » ? Depuis quelques années, toute solution qui se respecte se doit d'apporter ces fonctionnalités. Pour MySQL/MariaDB, nous allons explorer la solution proposée par Galera Cluster.

Mots-clés : MySQL, MariaDB, Réplication, Galera Cluster, Haute Disponibilité

Résumé

Depuis maintenant des « lustres », les administrateurs sont habitués à déployer des solutions à base de SGDB-R implémentant une forme de réplication souvent asynchrone souvent nommée « maître/esclave ». Mais ce type de solution est obsolète, car elle ne permet pas facilement de former un véritable « cluster » où il serait possible de lire et d'écrire sur chaque nœud. Ces restrictions sont levées par le Cluster Galera que nous vous présentons ici. Cette technologie de réplication synchrone et multimaître est intégrée dans le *fork* de MySQL développé par la société MariaDB [1] (notons que nous aurions aussi pu aussi mettre en œuvre le *fork* développé par la société Percona [2]).

Le cluster Galera est composé de nœuds qui exécutent chacun un processus `mysqld`. Chaque processus communique avec les autres nœuds du cluster et dispose de la totalité des données. Le partitionnement n'est pas directement géré par le cluster. Il y a là une différence avec le cluster NDB de **MySQL**.

Quels sont les avantages du Cluster **Galera** ?

- tous les nœuds étant équivalents, lectures et écritures peuvent être effectuées sur chacun d'eux ;
- il n'y a pas de nœud « maître » ou d'administration ou de « *management* », donc le cluster continue à fonctionner, quel que soit le nœud qui est arrêté.

Et quels sont les inconvénients ?

- comme chaque nœud dispose de l'intégralité des données, l'ajout d'un nœud provoque un *overhead* d'initialisation plus important selon le volume de données à recopier localement ;
- comme les écritures doivent être reproduites sur tous les nœuds, le cluster Galera n'apporte pas d'amélioration significative à la tenue en charge lors d'écritures nombreuses ;
- le cluster Galera impose quelques restrictions que nous évoquerons plus loin. La principale étant que seul le moteur de stockage **InnoDB**/**XtraDB** est supporté, toutefois la base « `mysql` » est correctement répliquée.

Dans cet article, notre objectif sera de construire un cluster « haute disponibilité » composé de 3 nœuds, sur lequel chacun d'eux pourra accepter les lectures et les écritures. Les nœuds seront nommés **node1**, **node2** et **node3** !

Nous verrons ensuite qu'à l'aide d'un composant additionnel, nous pourrons ajouter l'équilibrage de charge à notre cluster !

1 Installation

On trouve facilement les paquets binaires sur le site de MariaDB **[1]**, ou bien on peut ajouter un dépôt pour **yum** ou **aptitude** sur notre système. Dans notre cas, nous utilisons une distribution CentOS 7 et nous ajoutons le dépôt de MariaDB en créant le fichier **/etc/yum.repos.d/mariadb.repo** qui contient les directives suivantes :

```
[mariadb]
name = MariaDB
baseurl = http://yum.mariadb.org/10.0/centos7-amd64
gpgkey = https://yum.mariadb.org/RPM-GPG-KEY-MariaDB
gpgcheck=1
enabled = 1
```

Supprimez un éventuel paquet de MariaDB qui serait déjà présent sur le serveur, puis installez le paquet Galera Server de la manière suivante :

```
node1# yum update
node1# yum install MariaDB-Galera-server
```

Outre le paquet du cluster, le paquet **galera** contenant le protocole de synchronisation (WSREP) entre les nœuds du cluster, sera installé.

Le serveur MariaDB n'est pas lancé automatiquement, nous le faisons manuellement :

```
node1# systemctl start mysql
```

La connexion au serveur montre ceci :

```
node# mysql
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 8
Server version: 10.0.17-MariaDB-wsrep MariaDB Server,
wsrep_25.10.r4144

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and
others.

Type 'help;' or '\h' for help. Type '\c' to clear the current
input statement.

MariaDB [(none)]>
```

Vérifiez bien que c'est la version 10 qui vous est proposée et non pas la version 5.5 livrée avec la distribution CentOS 7. Notez aussi la présence de l'extension **-wsrep** qui montre que le protocole WSREP est bien disponible. La commande suivante doit aussi afficher une liste de variables d'état du cluster, en particulier, la valeur de **wsrep_connected** est **OFF** : c'est normal, nous n'avons pas encore configuré le cluster !

```
MariaDB> SHOW STATUS LIKE 'wsrep%' ;
```

Comme nous ne disposons pas de serveur DNS, le fichier **/etc/hosts** des nœuds du cluster contiendra les adresses IP associées aux noms logiques **node1**, **node2** et **node3**.

Le cluster utilise plusieurs ports TCP : le **3306** pour les clients MySQL, le **4567** pour la communication entre les nœuds (en TCP et UDP), **4568** pour les transferts de données incrémentaux (IST – voir section 2) et enfin **4444** pour les transferts complets (SST – voir également en section 2), aussi pour éviter ici les problèmes réseau, nous désactivons le pare-feu :

```
node1# systemctl disable firewalld
node1# systemctl stop firewalld
```

De même, pour des raisons de simplicité, nous désactivons **SELinux** et **AppArmor**. Sur la CentOS-7, nous avons dû rendre SELinux permissif avec la commande :

```
node1# setenforce 0
```

Nous aurons besoin des outils de *backup* et restauration **Xtrabackup** de Percona Software : pour cela nous ajoutons le dépôt de Percona ainsi :

```
node1# yum install http://www.percona.com/downloads/percona-
release/redhat/0.1-3/percona-release-0.1-3.noarch.rpm
```

Puis nous installons les outils manquants (**xtrabackup** et **innobackupex**) ainsi :

```
node1# yum update
node1# yum install percona-xtrabackup
```

Ces étapes d'installation doivent être reproduites sur les serveurs **node2** et **node3** bien sûr.

2 Comment dupliquer les données entre les nœuds du cluster ?

Avant de configurer notre cluster, nous devons nous poser la question suivante : puisque chaque nœud dispose de la totalité des données, comment transférer les données d'un nœud qui les possède vers un nouveau nœud du cluster ?

Trois solutions sont possibles : l'utilisation de **mysqldump**, de **rsync** ou de **xtrabackup**. Nous éliminons les solutions utilisant **mysqldump** et **rsync**, car elles verrouillent les bases en *READ-ONLY*. Nous choisissons donc **xtrabackup** pour réaliser les transferts initiaux et complets des bases. Ce transfert complet est appelé SST (*State Snapshot Transfer*) par opposition à IST (*Incremental State Transfer*) qui est un transfert incrémental des données permettant à un nœud temporairement sorti du cluster de « rattraper son retard ».

Pour avoir accès aux données, la méthode « **xtrabackup** » utilise un script nommé **innobackupex** qui a besoin d'un compte et d'un mot de passe créés pour l'occasion. Les commandes suivantes, jouées sur **node1**, permettent de créer le compte **xtrauser** avec le mot de passe **wspass** et les privilèges nécessaires :

```
MariaDB [None]> CREATE USER 'xtrauser'@'localhost' IDENTIFIED BY
'wspass';
MariaDB [None]> GRANT RELOAD, LOCK TABLES, REPLICATION CLIENT ON
*.* TO 'xtrauser'@'localhost';
MariaDB [None]> FLUSH PRIVILEGES;
```

Quand un nœud sera désynchronisé, la méthode la plus efficace sera automatiquement sélectionnée : soit SST, soit IST.

3 Configuration du premier nœud

Nous supposons que nous avons réalisé les mêmes opérations sur les machines **node2** et **node3** : il est grand temps de créer notre premier cluster ! Sur le serveur **node1**, modifiez le fichier **/etc/my.cnf.d/server.cnf** afin que la rubrique **[galera]** contienne les paramètres suivants :

```
[galera]
# Mandatory settings
binlog_format=row
default_storage_engine=InnoDB
#innodb_autoinc_lock_mode=2
#bind-address=0.0.0.0
#
# Optional setting
#wsrep_slave_threads=1
#innodb_flush_log_at_trx_commit=0
wsrep_provider=/usr/lib64/galera/libgalera_smm.so
wsrep_cluster_address=gcomm://
wsrep_cluster_name='clu1'
wsrep_sst_method=xtrabackup-v2
wsrep_sst_auth=xtrauser:wspass
```

Le paramètre **wsrep_cluster_name** donne le nom logique du cluster (car vous pouvez avoir plusieurs clusters sur le même réseau !). Pour l'instant, aucun autre nœud n'appartient au cluster, donc le paramètre **wsrep_cluster_address** vaut **gcomm://**. C'est par cette valeur qu'on signifie à cette instance qu'il faut initialiser le cluster !

Le paramètre **binlog_format** vaut obligatoirement **row** et non pas **statement**, enfin comme le cluster fonctionne avec les tables au format InnoDB/XtraDB, nous fixons ce moteur comme moteur de stockage par défaut.

Comme nous avons choisi le mode « xtrabackup-v2 » (une version améliorée de xtrabackup) pour effectuer les transferts de données SST, nous devons configurer les paramètres **wsrep_sst_method** et **wsrep_sst_auth**.

Relancez le service **mysql**. En cas de problème, consultez les logs et en particulier, le fichier **/var/lib/mysql/node1.err**. Pour vérifier que le cluster est initialisé, connectez-vous avec le client **mysql** et relancez la commande **SHOW STATUS LIKE 'wsrep%'**. Maintenant près de 60 variables sont affichées, dont :

```
...
| wsrep_cluster_conf_id      | 1                                    |
| wsrep_cluster_size         | 1                                    |
| wsrep_cluster_state_uuid   | c2fa9a92-f0c6-11e4-8a7e-
7fe96a16d8ba |
| wsrep_cluster_status       | Primary                              |
| wsrep_connected            | ON                                   |
...
```

Le serveur **node1** est bien le premier nœud du cluster (**wsrep_cluster_conf_id = 1**), il peut délivrer des données, car il fait partie du groupe primaire (**wsrep_cluster_status = Primary**) et le cluster est opérationnel (**wsrep_connected = ON**).

En cas de problème, pensez à regarder les logs de sécurité, en particulier ceux liés à SELinux et AppArmor.

VOICI TOUTES LES OFFRES COUPLÉES AVEC GNU/LINUX MAGAZINE !

POUR LE PARTICULIER ET LE PROFESSIONNEL ...

Prix TTC en Euros / France Métropolitaine

CHOISISSEZ VOTRE OFFRE !

Offre	SUPPORT (Prix en Euros / France Métropolitaine)	PAPIER Réf	PAPIER Tarif TTC	PAPIER + PDF (PDF 1 lecteur) Réf	PAPIER + PDF Tarif TTC	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Réf	PAPIER + BASE DOCUMENTAIRE Tarif TTC	PAPIER + PDF + BASE DOCUMENTAIRE (PDF 1 lecteur + 1 connexion BD) Réf	PAPIER + PDF + BASE DOCUMENTAIRE Tarif TTC
	ABONNEMENT								
LM	11n° GLMF	LM1	65,-	LM12	95,-	LM13	149,-	LM123	174,-
LM+	11n° GLMF + 6n° HS	LM+1	118,-	LM+12	177,-	LM+13	197,-	LM+123	256,-
	LES COUPLAGES « LINUX »								
A	11n° GLMF + 6n° LP	A1	95,-	A12	140,-	A13	218,-	A123	263,-
A+	11n° GLMF + 6n° HS + 6n° LP + 3n° HS	A+1	182,-	A+12	263,-	A+13	300,-	A+123	386,-
B	11n° GLMF + 6n° MISC	B1	100,-	B12	147,-	B13	233,-	B123	280,-
B+	11n° GLMF + 6n° HS + 6n° MISC + 2n° HS	B+1	172,-	B+12	248,-	B+13	300,-	B+123	381,-
C	11n° GLMF + 6n° LP + 6n° MISC	C1	135,-	C12	197,-	C13	312,-	C123	374,-
C+	11n° GLMF + 6n° HS + 6n° LP + 3n° HS + 6n° MISC + 2n° HS	C+1	236,-	C+12	339,-	C+13	403,-	C+123	516,-
	LES COUPLAGES « EMBARQUÉ »								
F	11n° GLMF + 6n° HK* + 4n° OS	F1	125,-	F12	188,-	F13	229,-*	F123	292,-*
F+	11n° GLMF + 6n° HS + 6n° HK* + 4n° OS	F+1	183,-	F+12	275,-	F+13	287,-*	F+123	379,-*
	LES COUPLAGES « GÉNÉRAUX »								
H	11n° GLMF + 6n° HK* + 6n° LP + 6n° MISC + 4n° OS	H1	200,-	H12	300,-	H13	402,-*	H123	499,-*
H+	11n° GLMF + 6n° HS + 6n° HK* + 6n° LP + 3n° HS + 6n° MISC + 2n° HS + 4n° OS	H+1	301,-	H+12	452,-	H+13	493,-*	H+123	639,-*

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | OS = Open Silicium | HC = Hackable

* HK : Attention : La base Documentaire de Hackable n'est pas incluse dans l'offre.

N'hésitez pas à consulter les détails des offres ci-dessus sur : www.ed-diamond.com !

De même, vérifiez bien que les fichiers **/etc/hosts** sont corrects et que vous avez désactivé le pare-feu sur chaque nœud.

Les logs du cluster sont stockés sous **$datadir/<FQDN>.err**, c'est-à-dire ici, sous **/var/lib/mysql/node1.err**. Si l'initialisation du premier nœud réussit, le fichier de log contient la ligne suivante :

```
[Note] WSREP: Synchronized with group, ready for connections
```

4 Ajout des autres nœuds

Avant d'intégrer la machine **node2** dans le cluster, créons quelques données minimalistes sur **node1** :

```
node1# mysql
MariaDB [None]> create database demo ;
Query OK, 1 row affected (0.03 sec)
MariaDB [None]> use demo
Database changed
MariaDB [demo]> create table t1 (nom char(64), prenom char(64));
Query OK, 0 rows affected (0.07 sec)
MariaDB [demo]> insert into t1 values ('dupond', 'jean');
Query OK, 1 row affected (0.02 sec)
```

Recopiez ensuite le fichier **/etc/my.cnf.d/server.cnf** de **node1** sur **node2**, puis apportez-y les modifications suivantes :

```
wsrep_cluster_address=gcomm://node1
wsrep_node_address='192.168.0.26'
wsrep_node_name='node2'
```

Nous indiquons ainsi qu'il y a déjà un nœud dans le cluster (**node1**) et donc qu'il faudra s'y connecter pour obtenir les données initiales. Lançons alors le service :

```
node2# service mysql restart
Starting MySQL......SST in progress, setting sleep higher... SUCCESS!
```

Connectons-nous sur cette instance de MySQL et vérifions que les données sont bien là !

```
node2# mysql
MariaDB [None]> use demo
MariaDB [demo]> select * from t1 ;
+--------+--------+
| nom    | prenom |
+--------+--------+
| dupond | jean   |
+--------+--------+
1 row in set (0.00 sec)
```

Les données sont là ! Sur **node1**, on peut constater que l'état du cluster a changé :

```
MariaDB [None]> show status like 'wsrep%';
| wsrep_incoming_addresses     | 192.168.0.25:3306,192.168.0.26:3306 |
| wsrep_cluster_size           | 2                                   |
```

Reproduisez les mêmes opérations pour intégrer le serveur **node3** dans le cluster : copie et modification du fichier **/etc/my.cnf.d/server.cnf**, puis lancement du service.

Nous pouvons maintenant donner la configuration définitive du cluster aux 3 nœuds en modifiant le paramètre **wsrep_cluster_address** du fichier **/etc/my.cnf.d/server.cnf** ainsi :

```
wsrep_cluster_address = gcomm://node1,node2,node3
```

Si vous ne voulez pas redémarrer tous les serveurs pour tester que le cluster est fonctionnel, vous pouvez modifier ce paramètre dynamiquement par la commande :

```
MariaDB [(none)]> set global wsrep_cluster_address="gcomm://
node1,node2,node3";
```

Attention ! Si vous relancez tous les serveurs en même temps, le cluster ne remontera pas, faute de serveur « primaire » disponible. Il vous faudra lancer un des serveurs ainsi :

```
# service mysql start --wsrep_cluster_address= 'gcomm://'
```

Ce serveur servira de primaire. Vous pouvez alors lancer les autres nœuds, puis, quand le cluster sera opérationnel, vous pouvez à nouveau rejouer la commande **set global...** précédente sur le nœud primaire.

5 Administration et vie du Cluster

5.1 Perte d'un nœud

Un cluster à 3 nœuds nous permet de supporter la perte d'un ou plusieurs nœuds sans interruption de service. Par exemple, arrêtons le serveur **node3** :

```
node3# service mysql stop
```

Continuons ensuite à lire et écrire des données sur **node1** et **node2** : aucune erreur n'est détectée.

```
(node1) MariaDB [demo]> insert into t1 values ('durand', 'paul') ;
(node1) MariaDB [demo]> insert into t1 values ('martin', 'pierre') ;
(node2) MariaDB [demo]> insert into t1 values ('leblanc', 'bernard') ;
```

Que se passe-t-il lorsque **node3** voudra réintégrer le cluster ? Et bien, espérons qu'il pourra simplement « rattraper son retard » par rapport aux autres nœuds.

Redémarrons le service :

```
node3# service mysql start
```

Après quelques secondes, le fichier de log **/var/lib/mysql/node3.err** montre la ligne suivante :

```
[Note] WSREP: IST received: c2fa9a92-f0c6-11e4-8a7e-7fe96a16d8ba:8
```

Les données ont été recopiées par la méthode incrémentale IST.

5.2 Sauvegarde et restauration

Concernant la sauvegarde, si vous utilisez des tables au format InnoDB/XtraDB, l'outil xtrabackup ne pose aucune difficulté d'utilisation et ne place aucun verrou transactionnel. Il peut donc s'utiliser sur n'importe quel nœud du cluster, par exemple, via le script **innobackupex** qui va créer une sauvegarde dans un sous-répertoire de **/tmp** :

```
node3# innobackupex --galera-info --user=xtrabackup -
password=wspass /tmp
```

Effaçons les données de **node3** :

```
node3# service mysql stop
node3# rm -rf /var/lib/mysql/*
```

Puis effectuons une restauration :

```
node3# cp -r /tmp/<répertoire_de_sauvegarde>/* /var/lib/mysql/
node3# chown -R mysql:mysql /var/lib/mysql/
```

Avant de relancer le nœud, ajoutez une nouvelle entrée, à partir de **node1** ou **node2**, dans la table **demo.t1** !

Nous pouvons relancer **node3**, mais nous devons indiquer un point de resynchronisation : la valeur à passer est donnée par la commande :

```
node3# cat /tmp/<répertoire_de_sauvegarde>/xtrabackup_galera_info
c2fa9a92-f0c6-11e4-8a7e-7fe96a16d8ba:8
```

La commande de lancement devient alors :

```
node3# service mysql start -wsrep_start_position="c2fa9a92-f0c6-
11e4-8a7e-7fe96a16d8ba:8"
```

Vérifiez alors que la table **demo.t1** a un contenu correct qui contient les anciennes données, mais aussi la ligne ajoutée depuis le *backup*.

5.3 Quelques considérations

Comme toutes les technologies de type « Cluster », le nombre de nœuds qui participent au cluster doit être impair, afin de dégager une majorité : dans le cas normal, les nœuds qui appartiennent au cluster sain, forment un *Primary Component* et le cluster continue de fonctionner tant qu'il y a une majorité de nœuds qui se voient et qui donc peuvent former un *Primary Component* ! Par exemple, si un cluster est composé de 5 nœuds et qu'un nœud devient indisponible, il suffit de 3 nœuds actifs pour continuer à former un *Primary Component*, car 3 nœuds sur 5 forment une majorité ! Mais imaginons maintenant un Cluster à 4 nœuds : deux des quatre nœuds sont situés dans une salle machine (ou une baie) et les deux autres dans une autre salle machine (ou une autre baie). Si le réseau se coupe entre les deux salles (ou les deux baies), nous obtenons deux groupes (ou composants) isolés de deux nœuds chacun. Aucun de ces groupes n'obtient de majorité et aucun *Primary Component* ne peut être créé. Le cluster est indisponible.

Si vous ne voulez pas déployer de nœuds supplémentaires, vous aurez besoin d'un arbitre qui assurera la non-parité de votre Cluster. Cet arbitre se nomme **garbd**.

Concernant la supervision, il existe des plugins pour **Nagios [3]** qui permettent de vérifier l'état du cluster : en particulier ils vérifient qu'il existe un *Primary Component* et qu'il existe un nombre de nœuds actifs suffisant dans le cluster.

Jusqu'à présent, nous avons vu que la configuration et la supervision du cluster mettaient en œuvre des paramètres et des variables d'état dont le nom débute par le préfixe **wsrep_**. Mais parmi ces paramètres, il en existe un, un peu particulier, qui permet de régler les paramètres internes du *plugin* de réplication **galera**. Ce paramètre se nomme **wsrep_provider_options**. Il permet de régler des valeurs fines concernant les *timeouts* mis en œuvre pour la bonne gestion du Cluster, ou bien pour affiner le rôle des nœuds du Cluster, en particulier en leur donnant un « poids » qui impacte la constitution du *Primary Component*.

6 Cluster et Load-Balancer

À ce stade, nous avons un beau cluster multimaître qui assure la haute disponibilité, mais du point de vue du client MySQL, comment faire pour toujours choisir un nœud actif, comment faire pour équilibrer les requêtes sur les nœuds actifs du cluster ?

Soit vous pouvez modifier les applications et leur faire prendre en compte la nouvelle topologie du cluster (par exemple avec une URL de ce type utilisée avec le driver JDBC **`jdbc:mysql:loadbalance://node1,node2,node3/mydb`**), soit vous ne pouvez pas ou vous souhaitez une solution plus générique : auquel cas, il va falloir installer et configurer un *Load-Balancer*.

Un logiciel spécialisé comme **HA-Proxy [4]** fera parfaitement l'affaire. Mais ce composant, s'il est unique, devient lui-même un SPOF (*Single Point of Failure*) et il faut donc créer un cluster d'au moins deux machines HA-Proxy, avec une adresse IP virtuelle afin de rendre ce cluster totalement transparent vis-à-vis des applicatifs.

Le schéma de la figure 1 est issu du site web de la société **FromDual** qui donne la procédure pour installer ces deux instances de HA-Proxy en cluster.

Petit retour sur le transfert des données entre nœuds : même si la solution proposée par xtrabackup limite les blocages au niveau de la machine qui émet les données, il peut être intéressant de fixer définitivement le nom de cette machine, qu'on appellera alors un « *donor* ». Cette opération s'effectue à l'aide du paramètre **`wsrep_sst_donor`** du fichier de configuration :

```
wsrep_sst_donor=node1
```

Quel est l'avantage dans le cadre de l'équilibrage de charge ? Et bien, si le *donor* ne fait pas partie du *pool* de serveurs défini au niveau du *Load-Balancer*, les resynchronisations ne bloqueront pas un nœud susceptible d'être appelé par ces mêmes *Load-Balancers* avec le risque de bloquer des clients.

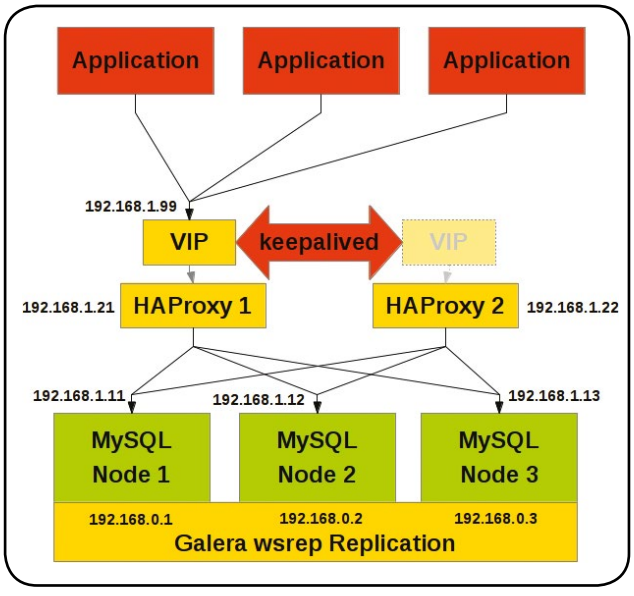


Fig. 1 : Procédure pour installer deux instances de HA-Proxy en cluster (source : http://www.fromdual.com/making-haproxy-high-available-for-mysql-galera-cluster).

Conclusion

Le cluster Galera est plutôt facile à mettre en œuvre, il est le complément idéal de MySQL/MariaDB pour absorber une montée en charge tant qu'une solution de partitionnement n'est pas nécessaire. Par contre, il impose l'utilisation des moteurs InnoDB/XtraDB et possède quelques restrictions mineures sur l'utilisation des requêtes SQL (voir **[5]**). En outre, dès lors que la réplication se veut synchrone, les modifications de schéma peuvent être bloquantes et lentes. Nous vous conseillons de lire l'article relatif à ce sujet **[6]**.

Nous regrettons aussi l'absence de processus de gestion et configuration centralisés qui pourraient être consultés par les applications clientes pour découvrir la topologie du cluster. Cependant, le cluster Galera est une excellente alternative à la réplication traditionnelle de MySQL/MariaDB. ■

Références

[1] MariaDB : http://www.mariadb.org

[2] Percona : http://www.percona.com

[3] Plugin pour Nagios : http://www.fromdual.com/galera-cluster-nagios-plugin-en

[4] Assmann B., « *Les nouveautés d'Haproxy 1.5* », *GNU/Linux Magazine n°179*, février 2015, p.6 à 15.

[5] *MariaDB Galera Cluster - Known Limitations* : https://mariadb.com/kb/en/mariadb/mariadb-galera-cluster-known-limitations/

[6] *Online Schema Upgrade in MySQL Galera Cluster using TOI Method* : http://www.severalnines.com/blog/online-schema-upgrade-mysql-galera-cluster-using-toi-method

DÉCOUVERTE DES DRIVERS VIRTUELS : LES DRIVERS VIDÉO

Thierry Gayet [CTO AMA SA Rennes]

Que faire lorsque l'on a besoin de lire un flux distant, comme dans le cas d'une camera IP, qu'il soit audio ou vidéo, et ce, via les API standard video4linux ou ALSA/OSS, tel que le ferait une vraie webcam ou carte son locale ?

Mots-clés : Multimédia, Webcam, Drivers virtuels, V4L2, Vidéo, Loopback, FFmpeg, Drivers

Résumé

Au-delà de l'intérêt d'un driver vidéo factice, cet article va présenter sa mise en œuvre. Pour ce faire, le driver v4L2loopback sera mis en avant en couvrant les différentes facettes de ses possibilités.

La première question qui se pose, porte sur l'utilité des drivers virtuels. Pourquoi en a-t-on besoin, quels sont leurs usages ?

1 Quels usages ?

Dans la finalité, certaines applications de visioconférence peuvent refuser de démarrer s'il manque un microphone même si seule l'image de la caméra aurait suffi. L'installation d'un périphérique virtuel permet, dans ce cas, de combler un manque. Un autre cas permet également d'avoir une caméra distante comme si elle était connectée en local.

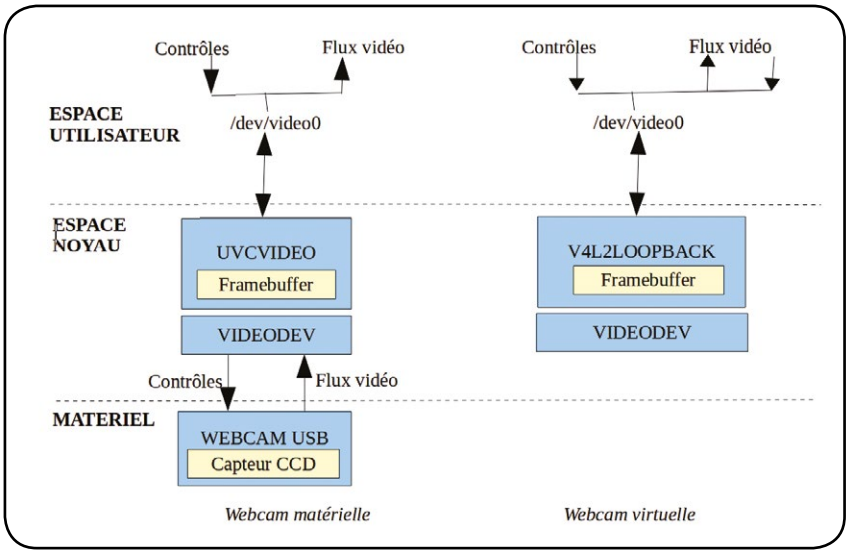


Fig. 1 : Comparaison entre drivers virtuels et non virtuels.

À titre de comparaison, nous pouvons commencer par lister les différences entre un driver d'une webcam physique et un driver virtuel (voir le schéma de la figure 1). Une webcam physique est associée à un module noyau gérant le *framebuffer* des images provenant de la caméra mais aussi certaines métriques comme la résolution, le nombre d'images par seconde ou *framerate*, ou bien le type de format du flux vidéo, souvent YUV ou MJPEG.

Ce genre de drivers est donc fréquemment utilisé lorsque la présence d'une webcam est nécessaire alors que l'ordinateur actuel n'en dispose pas.

Bien qu'il ait été possible de transmettre ces flux audio et vidéo par des ipc comme des sockets ou une fifo, l'usage des api html5 webrtc pour les transmissions audio et vidéo décolle de plus en plus. Pour ce faire, au lancement des navigateurs Web comme **Chrome** ou **Firefox**, la détection des ressources audio et vidéo est faite pour permettre la possibilité de les utiliser directement depuis une application Web. Cela ne peut se faire que par la présence d'un périphérique ad-hoc qui sera détectée par le programme puisqu'il y a une vérification des fichiers **`/dev/video*`** ou des *devices* oss/alsa.

Ces drivers virtuels sont également compatibles avec des logiciels de visioconférence comme **Skype** pour Linux et bien d'autres logiciels comme **Jitsi** (client Jabber gérant l'audio et la vidéo).

2 Le driver virtuel pour webcam

2.1 Présentation et architecture

Le driver **`v4l2loopback`** est un module dynamique situé en espace noyau, dépendant du module générique **`videodev`** offrant les api pour video4linux pour GNU/linux (aka v4l2). La version du noyau support doit être supérieure ou égale à 2.6.32. Il permet de simuler une webcam dans tous les sens du terme, sans pour autant disposer du matériel. Le flux vidéo dans l'espace utilisateur sera accessible en lecture et écriture pour pouvoir effectuer l'injection (écriture) ou bien un enregistrement (lecture) comme le montre la figure 2.

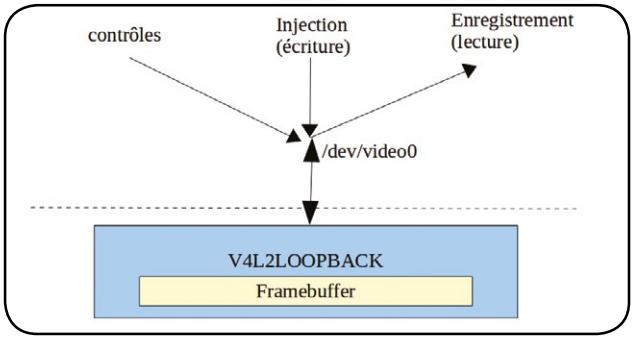


Fig. 2: Architecture du driver v4l2loopback.

La partie contrôle permet de paramétrer le nombre de *frames* par seconde ou la résolution.

2.2 Installation

Pour disposer de ce driver nous allons récupérer ses sources depuis son dépôt Github **[1]** :

```
$ git clone https://github.com/umlaeute/v4l2loopback.git
Clonage dans 'v4l2loopback'...
remote: Counting objects: 1472, done.
remote: Total 1472 (delta 0), reused 0 (delta 0), pack-reused 1472
Réception d'objets: 100% (1472/1472), 685.30 KiB | 381.00 KiB/s,
done.
Résolution des deltas: 100% (810/810), done.
Vérification de la connectivité... fait.
$ cd v4l2loopback/
$ ls
AUTHORS           doc             NEWS        utils
ChangeLog         examples        README      v4l2loopback.c
COPYING           Makefile        release.sh  v4l2loopback_
formats.h
currentversion.sh  Makefile.manual  test
```

Commençons par compiler le module dynamique :

```
$ make
  Building v4l2-loopback driver...
  make -C /lib/modules/`uname -r`/build M=/home/tgayet/Workspace/
v4l2loopback modules
  make[1]: entrant dans le répertoire " /usr/src/linux-headers-
3.16.0-33-generic "
    CC [M]  /home/tgayet/Workspace/v4l2loopback/v4l2loopback.o
    Building modules, stage 2.
    MODPOST 1 modules
    CC      /home/tgayet/Workspace/v4l2loopback/v4l2loopback.mod.o
    LD [M]  /home/tgayet/Workspace/v4l2loopback/v4l2loopback.ko
  make[1]: quittant le répertoire " /usr/src/linux-headers-3.16.0-
33-generic "
```

Et installons-le dans la foulée :

```
$ sudo make install
 [sudo] password for tgayet:
 make -C /lib/modules/`uname -r`/build M=/home/tgayet/Workspace/
v4l2loopback modules_install
 make[1]: entrant dans le répertoire " /usr/src/linux-headers-
3.16.0-33-generic "
   INSTALL /home/tgayet/Workspace/v4l2loopback/v4l2loopback.ko
   DEPMOD  3.16.0-33-generic
 make[1]: quittant le répertoire " /usr/src/linux-headers-3.16.0-
33-generic "
 depmod -a  `uname -r`
```

Inspectons les propriétés du module :

```
$ modinfo ./v4l2loopback.ko
  filename:       /home/tgayet/Workspace/v4l2loopback/
./v4l2loopback.ko
  license:        GPL
  author:         Vasily Levin, IOhannes m zmoelnig <zmoelnig@
iem.at>,Stefan Diewald,Anton Novikov
```

```
  description:    V4L2 loopback video device
  srcversion:     ACDF27939D8E0689986D1209
  depends:        videodev
  vermagic:       3.16.0-33-generic SMP mod_unload modversions 686
  parm:           debug:debugging level (higher values == more verbose) (int)
  parm:           max_buffers:how many buffers should be allocated (int)
  parm:           max_openers:how many users can open loopback device (int)
  parm:           devices:how many devices should be created (int)
  parm:           video_nr:video device numbers (-1=auto, 0=/dev/video0,
etc.) (array of int)
  parm:           card_label:card labels for every device (array of charp)
  parm:           exclusive_caps:whether to announce OUTPUT/CAPTURE
capabilities exclusively or not (array of bool)
  parm:           max_width:maximum frame width (int)
  parm:           max_height:maximum frame height (int)
```

Les paramètres utilisables au moment du chargement du driver v4l2loopback sont résumés dans le tableau suivant :

Paramètre	Description
debug	Définit le niveau de debug.
max_buffers	Définit le nombre de *buffers* à allouer.
max_openers	Définit le nombre d'instances d'applications pouvant utiliser le *device* par instance.
devices	Spécifie le nombre d'instances séparées auxquelles sera associé un fichier différent **/dev/video***.
video_nr	Permet de spécifier le numéro du fichier qui sera créée dans **/dev/video*** ; La numérotation commence à **0**. Cela est utile pour disposer d'un numéro fixe.
cards_label	Nommage de chaque instance.
exclusive_caps	Spécifie l'exclusivité d'une instance.
max_width	Largeur maximale de la *frame*.
max_height	Hauteur maximale de la *frame*.

Le chargement du driver se fait normalement : soit manuellement avec la commande **insmod**, soit automatiquement avec **modprobe** (vu que le **depmod** a été fait au moment de l'installation). Dans l'exemple suivant, nous allons charger le driver en le fixant sur **/dev/video1** (**video_nr = 1**) et en nommant le *hardware* (**card_label = video_linuxmag**) :

```
$ sudo modprobe v4l2loopback devices=1 video_nr=1 card_label=video_linuxmag

$ lsmod|grep v4l2
v4l2loopback           28897  0
 v4l2_common           15132  1 videobuf2_core
 videodev             131265  4 uvcvideo,v4l2loopback,v4l2_
common,videobuf2_core
```

Au chargement, le driver virtuel **v4l2loopback** apparaît bien dans le *listing* :

```
$ v4l2-ctl --list-devices
 Hercules Optical Glass (usb-0000:00:1d.7-3):
  /dev/video0
 video_linuxmag (v4l2loopback:0):
  /dev/video1
```

Ceci est cohérent car on vérifie bien que le driver s'est également bien déclaré sur **/dev/video1** :

```
$ ls -al /dev/video*
 crw-rw----+ 1 root video 81, 0 avril  6 09:36 /dev/video0
 crw-rw----+ 1 root video 81, 1 avril  6 16:28 /dev/video1
```

On peut ensuite vérifier ses caractéristiques :

```
$ v4l2-dbg --info --device=/dev/video1
 Driver info:
  Driver name   : v4l2 loopback
  Card type     : video_linuxmag
  Bus info      : v4l2loopback:0
  Driver version: 0.8.0
  Capabilities  : 0x05000002
    Video Output
    Read/Write
    Streaming
```

Pour que la configuration soit maintenue au prochain *reboot*, deux fichiers sont nécessaires. Le premier, **/etc/modules-load.d/v4l2loopback.conf**, sert à indiquer quel *driver* charger :

```
v4l2loopback
```

Le second fichier, **/etc/modprobe.d/v4l2loopback.conf**, sert à préciser les paramètres à passer au moment du chargement. Il est de coutume de séparer la configuration par fonctionnalité :

```
v4l2loopback devices=1 video_nr=1 card_label=video_linuxmag
```

On pourra aussi vérifier après coup la valeur des paramètres dans le *sysfs* **/sys/module/v4l2loopback/parameters/**.

Il est à noter que ce driver étant assez bien rodé, il dispose d'une configuration pour dkms lui permettant d'être toujours à jour même après un changement de noyau.

Bien que nous ayons recompilé le driver depuis ses sources, il est à noter que certaines distributions GNU/Linux intègrent ce *driver* dans leur dépôt :

```
$ apt-cache search v4l2loopback
v4l2loopback-dkms - Source for the v4l2loopback driver (DKMS)
v4l2loopback-source - Source for the v4l2loopback driver
v4l2loopback-utils - Commandline utilities for the for the v4l2-
loopback module
gem-plugin-v4l2 - Environnement graphique pour le multimédia -
prise en charge de la sortie V4L2
```

Notez le support dkms qui lui permet d'être recompilé soit à un changement du code source de ce driver, soit à un changement de version de noyau GNU/Linux.

2.3 Paramétrage

Pour changer le paramétrage du module, un script bash **v4l2loopback-ctl** est fourni avec le code source.

Modifions pour l'exemple le nombre de *frames* par seconde (ou FPS) :

```
$ cd utils/
$ ./v4l2loopback-ctl set-fps 25 /dev/video1
 OK
```

En plus direct, il est possible de modifier le niveau des fps :

```
$ sudo echo '@100' | sudo tee /sys/devices/virtual/video4linux/
video1/format
```

Le driver **v4l2loopback** étant spécifique, c'est-à-dire non lié à un capteur ccd, ce dernier s'adapte au format YUV ou MJPEG fournit en entrée. Par conséquent, cela explique pourquoi aucun format spécifique n'est retourné :

```
$ v4l2-ctl --list-formats --device=/dev/video1
 ioctl: VIDIOC_ENUM_FMT
```

Par comparaison, le driver standard **uvcvideo** retourne, quant à lui, les spécificités suivantes :

```
$ v4l2-ctl --list-formats
 ioctl: VIDIOC_ENUM_FMT
  Index       : 0
  Type        : Video Capture
  Pixel Format: 'YUYV'
  Name        : YUV 4:2:2 (YUYV)
```

2.4 Injection et lecture

Pour injecter en écriture un flux vidéo, cela peut se faire avec pléthore d'outils comme **ffmpeg**, **gstreamer** etc.

Dans un premier shell, injectons un flux mjpeg en exécutant une première instance de l'outil ffmpeg. Le flux MJPEG d'entrée est issu d'une webcam en ligne sur Internet à Saint-Malo. À noter que pour ce faire nous avons dû spécifier le format de couleur de sortie via la commande **ffmpeg** à savoir **yuv420p**, ce qui correspond à **YUV 4:2:2 planar** attendu par le driver virtuel **[2]** :

```
$ ffmpeg -f mjpeg -i http://webcam.st-malo.com/axis-cgi/mjpg/video.
cgi?resolution=352x288 -pix_fmt yuv420p  -f v4l2 /dev/video1
ffmpeg version 2.4.3-1ubuntu1~trusty6 Copyright (c) 2000-2014 the
FFmpeg developers
  built on Nov 22 2014 17:07:19 with gcc 4.8 (Ubuntu 4.8.2-19ubuntu1)
  configuration: --prefix=/usr --extra-version='1ubuntu1~trusty6'
--build-suffix=-ffmpeg --toolchain=hardened --extra-cflags= --extra-
cxxflags= --libdir=/usr/lib/i386-linux-gnu ... --enable-libsoxr
--enable-openal --enable-libopencv
...
Input #0, mjpeg, from 'http://webcam.st-malo.com/axis-cgi/mjpg/video.
cgi?resolution=352x288':
  Duration: N/A, bitrate: N/A
    Stream #0:0: Video: mjpeg, yuvj420p(pc, bt470bg), 352x288 [SAR 1:1
DAR 11:9], 25 tbr, 1200k tbn, 25 tbc
[swscaler @ 0x8e33bc0] deprecated pixel format used, make sure you did
set range correctly
Output #0, v4l2, to '/dev/video1':
  Metadata:
    encoder         : Lavf56.4.101
    Stream #0:0: Video: rawvideo (I420 / 0x30323449), yuv420p, 352x288
[SAR 1:1 DAR 11:9], q=2-31, 200 kb/s, 25 fps, 25 tbn, 25 tbc
    Metadata:
      encoder         : Lavc56.1.100 rawvideo
Stream mapping:
  Stream #0:0 -> #0:0 (mjpeg (native) -> rawvideo (native))
Press [q] to stop, [?] for help
frame=    7 fps=1.1 q=0.0 Lsize=N/A time=00:00:00.28 bitrate=N/A
```

Une fois l'injection lancée, il est possible d'ouvrir le programme vlc pour effectuer une capture sur le fichier video **/dev/video1** (voir la figure 3).

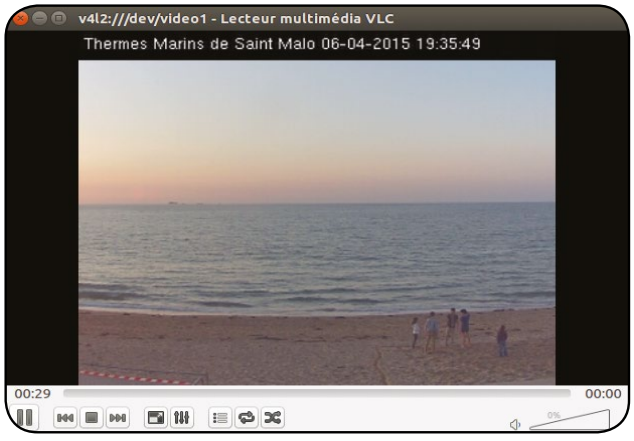


Fig. 3 : Screenshot du flux vidéo sous vlc.

À noter que depuis le *device* il est possible de contrôler qui en fait l'usage. Grâce à la commande **lsof** qui liste les ressources, on peut retrouver quel processus utilise quel fichier et comme sous Unix, le dogme est que « tout est fichier », il est possible de remonter à la racine de tout :

```
$ lsof |grep /dev/video1
 ffmpeg    3991        tgayet   5u   CHR   81,1   0t0    56526 /dev/video1
 ffmpeg    3991 3999   tgayet   5u   CHR   81,1   0t0    56526 /dev/video1
 ffmpeg    3991 4000   tgayet   5u   CHR   81,1   0t0    56526 /dev/video1
 ffmpeg    3991 4001   tgayet   5u   CHR   81,1   0t0    56526 /dev/video1
```

La commande **lsof** détaillera les quatre *threads* alors que la commande **fuse** globalisera le tout autour du même PID rattaché au processus père :

```
$ fuser /dev/video1
 /dev/video1:         3991
```

Depuis une autre console, il est également possible de lancer une seconde instance de **ffmpeg** mais cette fois-ci en capture (lecture) :

```
$ ffmpeg -f v4l2 -i /dev/video1 -f mjpeg ./video-capture
  ffmpeg version 2.4.3-1ubuntu1~trusty6 Copyright (c) 2000-2014
the FFmpeg developers
   built on Nov 22 2014 17:07:19 with gcc 4.8 (Ubuntu
4.8.2-19ubuntu1)
   configuration: --prefix=/usr --extra-version='1ubuntu1~trusty6'
--build-suffix=-ffmpeg
...
frame=  355 fps= 30 q=24.8 Lsize=    1563kB time=00:00:11.83
bitrate=1082.3kbits/s dup=328 drop=0
video:1563kB audio:0kB subtitle:0kB other streams:0kB global
headers:0kB muxing overhead: 0.000000%
```

À noter qu'au lieu de **ffmpeg** qui aura servi à enregistrer le flux, il aurait été possible de le visualiser avec son petit frère **ffplay** :

```
$ ffplay -f v4l2 -i /dev/video1
```

Voilà pour ce qu'il en est du driver virtuel permettant la simulation d'une webcam.

2.5 Restrictions de sécurité

Il est à noter que pour éviter d'avoir des flux vidéo issus de webcam *fake*, certains navigateurs Web comme Chromium (ou Google Chrome) filtrent les *devices* vidéo en retirant ceux qui ne font pas uniquement de la capture. En d'autres termes, habituellement depuis l'espace utilisateur un fichier **/dev/video1** n'est accessible qu'en lecture (capture) pour une webcam physique. Le filtrage s'effectue donc sur ceux étant également accessibles en écriture. Il y a une première vérification sur les capacités du *device* mais également sur le fait qu'un flux est fourni.

Il est à noter que, depuis la version 0.7.0, le driver **v4l2loopback** est de nouveau compatible avec Chromium en suivant scrupuleusement la séquence suivante :

- Chargement du driver :

```
$ modprobe v4l2loopback device=1 video_nr=1 exclusive_cap=1
```

- Lancer une injection avec cette fois-ci **gstreamer** :

```
$ gst-launch-0.10 -v videotestsrc ! "video/x-raw-yuv, width=640,
height=360, framerate=30/1, format=(fourcc)I420" ! v4l2sink
device=/dev/video1
```

- Démarrer une capture avec Google Chrome et l'url suivante : https://opentokrtc.com/ ou bien https://apprtc.appspot.com/ (voir figure 4).

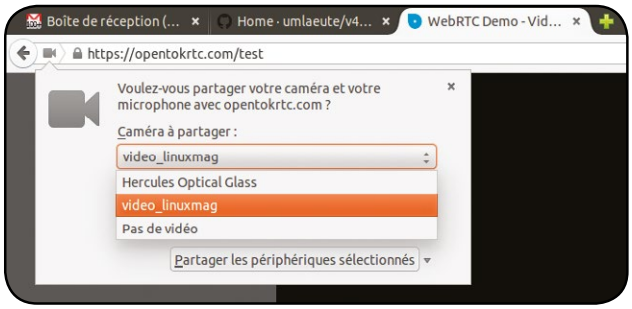


Fig. 4 : Visibilité de la caméra virtuelle depuis chrome.

Pour des utilisations avec skype la commande **gst-launch** de gstreamer peut servir :

```
$ gst-launch videotestsrc ! ffmpegcolorspace ! "video/x-raw-
yuv,format=(fourcc)YUY2" ! v4l2sink device=/dev/video1
```

Pour d'autres informations, le wiki du site Web peut être de bon conseil **[3]**.

Conclusion

Bon, vous voilà équipés pour gérer des webcams. Dans l'article suivant, nous aborderons le cas des cartes sons virtuelles.

Pour les usages, les flux d'entrée peuvent servir à diffuser un flux déjà enregistré (comme un fichier vidéo, une image ou une bande sonore), d'injecter un flux issu de caméras IP ou d'autres objets high-tech comme des lunettes connectées. ■

Références

[1] Site officiel pour v4l2loopback : https://github.com/umlaeute/v4l2loopback

[2] Wiki sur le format YUV : https://fr.wikipedia.org/wiki/YUV

[3] Wiki de v4l2loopback : https://github.com/umlaeute/v4l2loopback/wiki

DÉCOUVERTE DES DRIVERS VIRTUELS : LES DRIVERS AUDIO

Thierry Gayet [CTO AMA SA Rennes]

Que faire lorsque l'on a besoin de lire un flux en ligne, comme dans le cas d'une caméra IP, qu'il soit audio ou vidéo ? Cette question, on se l'est déjà posée pour la vidéo mais reposons-nous-la pour l'audio.

Mots-clés : Multimédia, Microphone, Drivers virtuels, ALSA, OSS, FFmpeg, Loopback

Résumé

Dans le précédent article, nous avons abordé le cas d'une webcam virtuelle via le driver `v4l2loopback`.

Dans ce nouvel article, nous allons aborder le cas d'un microphone virtuel via le drivers `aloop` d'ALSA. Nous avons démarré par la webcam car pour moi `v4l2loopback` est beaucoup mieux documenté que `aloop`.

Comme l'audio va souvent de pair avec la vidéo, le cas d'usage donné précédemment pourra être repris ici même, le but étant identique, à savoir pouvoir disposer d'une carte son et ce sans matériel.

Certes c'est limité mais nous allons montrer comment injecter un son comme s'il émanait d'un vrai microphone puis d'aller relire ce son en sortie.

Passons maintenant à la mise en place du driver virtuel pour le son, ce qui sera une autre paire de manches.

1 Présentation et architecture

Pour le son, la famille ALSA (**A**dvanced **L**inux **S**ound **A**rchitecture) propose un driver *loopback* équivalent, d'où son nom **`snd-aloop`**. Nul besoin de chercher longtemps pour le trouver car il est intégré de base dans les sources des noyaux GNU/linux récents.

2 Installation

On l'aura compris, étant partie intégrante du noyau, il n'y a pas de compilation, ni même d'installation à proprement parler. Dans les distributions courantes, ce driver est déjà compilé comme module dynamique même s'il n'est pas préchargé, ni même inclus dans la partie statique du noyau.

Examinons la carte d'identité du driver :

```
$ modinfo snd-aloop
filename:       /lib/modules/3.16.0-33-generic/kernel/sound/drivers/snd-aloop.ko
license:        GPL
description:    A loopback soundcard
author:         Jaroslav Kysela <perex@perex.cz>
srcversion:     EE6EE96BF30B357E6187865
depends:        snd-pcm,snd
intree:         Y
vermagic:       3.16.0-33-generic SMP mod_unload modversions 686
signer:         Magrathea: Glacier signing key
sig_key:        46:40:5A:A7:A3:EB:AC:E5:F4:E8:24:CF:08:6C:34:D2:00:33:B9:B5
sig_hashalgo:   sha512
```

```
  parm:           index:Index value for loopback soundcard. (array of
int)
  parm:           id:ID string for loopback soundcard. (array of charp)
  parm:           enable:Enable this loopback soundcard. (array of
bool)
  parm:           pcm_substreams:PCM substreams # (1-8) for loopback
driver. (array of int)
  parm:           pcm_notify:Break capture when PCM format/rate/
channels changes. (array of int)
```

Les paramètres utilisables au moment du chargement sont donnés dans le tableau ci-dessous :

Paramètre	Description
index	Index associé à la carte son.
id	Identifiant de la carte son.
enable	Active la fonctionnalité *loopback*.
pcm_substreams	Définis un nom de substreams (8 par défaut).
pcm_notify	Stoppe une capture si le flux PCM change selon son format, son débit ou encore selon le nombre de c anneaux.

Tout comme pour le driver précédent, le chargement se contente de lier le driver au noyau via la commande **modprobe** :

```
$ sudo modprobe snd-aloop enable=1 index=1
```

Pour que la configuration soit persistante, même après un *boot*, deux fichiers sont nécessaires. Le premier va servir à indiquer quel driver charger. Il s'agit du fichier **/etc/modules-load.d/alsa-loopback.conf** :

```
snd-aloop
```

Le second fichier, **/etc/modprobe.d/alsa-loopback.conf**, sert à préciser les paramètres à passer au moment du chargement. Traditionnellement on sépare la configuration par fonctionnalités :

```
options snd-aloop enable=1 index=1
```

3 Paramétrage

Pour le paramétrage, créons un fichier **~/.asound.rc** dans le répertoire de l'utilisateur. Pour ce faire, utilisons le script bash **setup-asoundrc.sh** suivant :

```
#!/bin/bash
# setup-asoundrc.sh v0.1.0 20090101 markc@renta.net GPLv3
# setup-asoundrc.sh v0.2.0 20090320 quatro_por_quatro@yahoo.es
GPLv3
#
# A simple script to create a particular default audio device
regardless
# of what cards are loaded or in what order. It could be used
anytime or
# placed in a ~/.bashrc script for a persistent setup every
login.
#
# Usage: setup-asoundrc.sh [DEFAULT_CARD] > ~/.asoundrc
# use the first parameter as the card name, or else
# look for the sound card, discarding those that are only
microphones
# when there are multiple cards, use the first one
if default_card="${1:-$(cat "$(for f in $(ls -1 /proc/asound/
card[0-9]*/{midi,codec}* 2>/dev/null); do echo "${f%/*}"; done \
| sed -e '\|^[\[:blank:]\]$|d' -e 'q')/id" 2>/dev/null)}"; then
   echo "Using sound card: ${default_card}" >&2
   cat /proc/asound/card[0-9]*/id | \
   gawk --assign default_card="${default_card}" \
'{print "pcm."$1" { type hw; card "$1"; }\nctl."$1" { type hw;
card "$1"; }" }
END {print "pcm.!default pcm."default_card"\nctl.!default
ctl."default_card}'
else
   echo "Warning: No sound cards found." >&2
fi
```

Pour l'utiliser, il suffit de spécifier en paramètre, le nom donné à la carte par défaut (à savoir **Loopback**) et de rediriger le tout vers le fichier **$HOME/.asoundrc** !

```
$ ./setup-asoundrc.sh Loopback > $HOME/.asoundrc
```

Cela permettra d'obtenir le fichier de configuration **~/.asoundrc** suivant :

```
pcm.Loopback { type hw; card Loopback; }
ctl.Loopback { type hw; card Loopback; }
pcm.Dummy { type hw; card Dummy; }
ctl.Dummy { type hw; card Dummy; }
pcm.PCH { type hw; card PCH; }
ctl.PCH { type hw; card PCH; }
pcm.!default pcm.Loopback
ctl.!default ctl.Loopback
```

Ce fichier peut soit être défini dans chaque répertoire utilisateur, soit être centralisé dans **/etc/asound.conf**.

De plus, la majorité des applications fonctionnent très bien sans. Cela sert à rajouter des fonctionnalités supplémentaires comme des routages ou bien des échantillonnages.

4 Détection du device ALSA

Avec ALSA, c'est beaucoup moins tranquille qu'avec **video4linux** : ici il faut son permis ! En effet, il n'y a pas de *device* bien nommé. Il est donc nécessaire de connaître le nom de la carte, ce que l'on peut obtenir avec **procfs** :

```
$ cat /proc/asound/modules
0 snd_hda_intel
1 snd_usb_audio
2 snd_aloop
```

Notre module **snd_aloop** se trouve ici en position **2**.

Pour être utilisé avec des outils comme **vlc**, **ffmpeg** et compagnie, les modules ALSA sont identifiés suivant la désignation suivante : **hw:X,Y,Z**. **X** représente l'identifiant de la carte son, **Y** celui du périphérique et enfin **Z** le sous-périphérique. **Y** et **Z** sont optionnels.

Comme les identifiants peuvent changer, il est souvent préconisé de rendre la détection dynamique :

```
OUT_ALSA_MAJOR_ID=`cat /proc/asound/pcm|grep Loopback|head -n
1|awk -F : '{print $1}'|awk -F - '{print $1}'`
OUT_ALSA_MAJOR=`echo ${OUT_ALSA_MAJOR_ID:1}`
OUT_ALSA_MINOR_ID=`cat /proc/asound/pcm|grep Loopback|head -n
1|awk -F : '{print $1}'|awk -F - '{print $2}'`
OUT_ALSA_MINOR=`echo ${OUT_ALSA_MINOR_ID:1}`
OUT_ALSA_ALOOP="hw:${OUT_ALSA_MAJOR},${OUT_ALSA_MINOR}"
```

Ce qui peut ensuite servir dans une commande **ffmpeg** :

```
$ ffmpeg -loglevel 0 -f v4l2 -i /dev/video2 -f alsa -i ${OUT_ALSA_
ALOOP}  -async 1 /tmp/enregistrementVideo > /dev/null 2>&1
```

Ces identifiants s'obtiennent pour le *playback* avec la commande issue des alsatools :

```
$ aplay -l
  **** Liste des Périphériques Matériels PLAYBACK ****
  carte 0: Intel [HDA Intel], périphérique 0: AD1984 Analog
[AD1984 Analog]
    Sous-périphériques: 1/1
    Sous-périphérique #0: subdevice #0
  carte 0: Intel [HDA Intel], périphérique 2: AD1984 Alt Analog
[AD1984 Alt Analog]
    Sous-périphériques: 1/1
    Sous-périphérique #0: subdevice #0
  carte 2: Loopback [Loopback], périphérique 0: Loopback PCM
[Loopback PCM]
    Sous-périphériques: 8/8
    Sous-périphérique #0: subdevice #0
...
    Sous-périphérique #7: subdevice #7
  carte 2: Loopback [Loopback], périphérique 1: Loopback PCM
[Loopback PCM]
    Sous-périphériques: 8/8
    Sous-périphérique #0: subdevice #0
...
    Sous-périphérique #7: subdevice #7
```

Le périphérique en entrée pour **snd-aloop** sera identifié par la carte numéro **2** et le périphérique numéro **0**, soit : **hw:2,0**. Quant à la « capture » il a son équivalent lui aussi des alsatools :

```
$ arecord -l
**** Liste des Périphériques Matériels CAPTURE ****
carte 0: Intel [HDA Intel], périphérique 0: AD1984 Analog [AD1984
Analog]
  Sous-périphériques: 1/1
  Sous-périphérique #0: subdevice #0
carte 0: Intel [HDA Intel], périphérique 2: AD1984 Alt Analog
[AD1984 Alt Analog]
  Sous-périphériques: 1/1
  Sous-périphérique #0: subdevice #0
carte 1: Glass [Hercules Optical Glass], périphérique 0: USB Audio
[USB Audio]
  Sous-périphériques: 1/1
  Sous-périphérique #0: subdevice #0
carte 2: Loopback [Loopback], périphérique 0: Loopback PCM
[Loopback PCM]
  Sous-périphériques: 8/8
  Sous-périphérique #0: subdevice #0
...
  Sous-périphérique #7: subdevice #7
carte 2: Loopback [Loopback], périphérique 1: Loopback PCM
[Loopback PCM]
  Sous-périphériques: 8/8
  Sous-périphérique #0: subdevice #0
...
  Sous-périphérique #7: subdevice #7
```

Le périphérique en sortie pour **snd-aloop** sera identifié par la carte numéro **2** et le périphérique numéro **1**, soit : **hw:2,1**.

Pour une carte donnée, si le périphérique **0** est utilisé en entrée, le périphérique **1** sera utilisé en sortie et vice-versa (voir figure 1). Certains logiciels comme Chromium

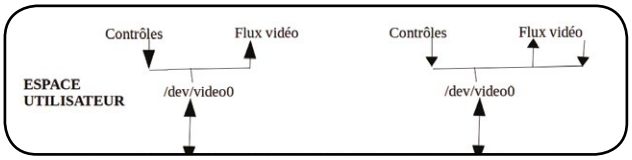


Fig. 1 : utilisation du driver aloop d'alsa avec ffmpeg.

ont câblé l'entrée **1** et ne prennent pas en compte l'entrée 0 même si elle est fonctionnelle.

5 Injection et lecture

Pour l'injection en écriture, nous allons rediriger un flux PCM avec **ffmpeg** vers le module **loopback** en utilisant la carte par défaut :

```
$ ffmpeg -f s16le -acodec pcm_s16le -i tcp://localhost:4001 -f
alsa hw:2,0
  ffmpeg version N-70567-gb3a56e6 Copyright (c) 2000-2015 the
FFmpeg developers
  built with gcc 4.8 (Ubuntu 4.8.2-19ubuntu1)
   configuration: --enable-ffplay --enable-libx264 --enable-
libfaac --enable-gpl --enable-nonfree --extra-libs=-lasound
--prefix=/home/fabien/git/
  libavutil      54. 20.100 / 54. 20.100
...
  libpostproc    53.  3.100 / 53.  3.100
  Guessed Channel Layout for  Input Stream #0.0 : mono
  Input #0, s16le, from 'tcp://localhost:4001':
    Duration: N/A, bitrate: 705 kb/s
    Stream #0:0: Audio: pcm_s16le, 44100 Hz, 1 channels, s16,
705 kb/s
    Output #0, alsa, to 'default':
    Metadata:
    encoder         : Lavf56.25.101
  Stream #0:0: Audio: pcm_s16le, 44100 Hz, mono, s16, 705 kb/s
    Metadata:
    encoder         : Lavc56.26.100 pcm_s16le
    Stream mapping:
    Stream #0:0 -> #0:0 (pcm_s16le (native) -> pcm_s16le (native))
Press [q] to stop, [?] for help
    size=N/A time=00:00:07.10 bitrate=N/A
    video:0kB audio:612kB subtitle:0kB other streams:0kB global
headers:0kB muxing overhead: unknown
```

Pour utiliser **ffmpeg** avec alsa, ce dernier pourra nécessiter une re-compilation avec le support **--extra-libs=-lasound**.

Quant à la lecture, voici un exemple d'injection d'un flux audio incluant une webcam :

```
$ ffmpeg -f v4l2 -i /dev/video2 -f alsa -i hw:2,1 -ss 00:00:05
-async 1 videorecord
  ffmpeg version N-70567-gb3a56e6 Copyright (c) 2000-2015 the
FFmpeg developers
  built with gcc 4.8 (Ubuntu 4.8.2-19ubuntu1)
  configuration: --enable-ffplay --enable-libx264 --enable-libfaac
--enable-gpl --enable-nonfree --extra-libs=-lasound --prefix=/
home/fabien/git/
  libavutil      54. 20.100 / 54. 20.100
...
Output #0, mp4, to '/tmp/amahealth/record/live-
  Metadata:
    encoder         : Lavf56.25.101
    Stream #0:0: Video: h264 (libx264) ([33][0][0][0] / 0x0021),
yuv420p, 720x480, q=-1--1, 25 fps, 12800 tbn, 25 tbc
    Metadata:
      encoder         : Lavc56.26.100 libx264
    Stream #0:1: Audio: aac (libfaac) ([64][0][0][0] / 0x0040),
48000 Hz, stereo, s16, 128 kb/s
    Metadata:
      encoder         : Lavc56.26.100 libfaac
Stream mapping:
  Stream #0:0 -> #0:0 (rawvideo (native) -> h264 (libx264))
  Stream #1:0 -> #0:1 (pcm_s16le (native) -> aac (libfaac))
Press [q] to stop, [?] for help
[alsa @ 0x2b164e0] Thread message queue blocking; consider raising
the thread_queue_size option (current value: 8)
frame=    0 fps=0.0 q=0.0 Lsize=       0kB time=00:00:00.00
bitrate=N/A
video:0kB audio:0kB subtitle:0kB other streams:0kB global
headers:0kB muxing overhead: unknown
```

Conclusion

Voilà le second volet terminé. J'espère vous avoir donné les clefs pour maîtriser les drivers vidéo mais aussi audio. En y pensant, il y a pléthore d'application possible en incluant des prés ou post traitements en injectant ou en récupérant des flux de drivers v4l2... ■

Pour aller plus loin

- Page sur le module **snd-aloop** : http://www.alsa-project.org/main/index.php/Matrix:Module-aloop
- Détail sur le fichier **~/.asoundrc** : http://alsa.opensrc.org/Asoundrc/
- Détail sur l'utilisation du module **aloop** : http://people.redhat.com/~jkysela/RHEL5/loop/BACKGROUND
- Explication d'**aloop** sur le site d'ALSA : http://www.alsa-project.org/main/index.php/Matrix:Module-aloop
- Détail des procfs pour ALSA : https://www.kernel.org/doc/Documentation/sound/alsa/Procfile.txt
- Détail de la configuration d'ALSA : https://www.kernel.org/doc/Documentation/sound/alsa/ALSA-Configuration.txt

UTILISEZ SWIFTMAILER POUR ENVOYER DES MAILS

Stéphane Mourey [Mousse sur le Seeraiwer]

La fonction mail de PHP, malgré sa simplicité apparente, est souvent l'occasion de quelques déconvenues. Voyons comment utiliser SwiftMailer, son parfait remplaçant, écrit en pur PHP.

Mail PHP SMTP SwiftMailer

L'OBJECTIF

Pour découvrir l'utilisation de **SwiftMailer**, nous allons créer un formulaire de contact simple dont le contenu sera envoyé par mail au destinataire de notre choix.

LES OUTILS

- PHP 5

Phase 1

Commencez par vérifier les prérequis à l'aide de la commande suivante :

```
$ php -a
Interactive mode enabled

php > echo function_exists('proc_open')
? "OK" : "proc_open() desactive, vous ne
pouvez pas utiliser SwiftMailer";
OK
```

Si vous obtenez une autre réponse que « OK », c'est que la fonction `proc_open` est désactivée. Cela peut être le cas par exemple si vous êtes en *Safe Mode*. Vous ne pourrez pas utiliser SwiftMailer avant d'avoir résolu ce point.

Phase 2

Créez votre répertoire de travail à un emplacement accessible dans l'arborescence de votre serveur web. Si vous n'avez pas de serveur web, créez un répertoire où vous voudrez, puis exécutez la commande suivante dans ce dossier :

```
# php -S 0.0.0.0:80
```

Ceci aura pour effet de démarrer le serveur web intégré à PHP en servant de dossier courant.

Phase 3

Installez maintenant SwiftMailer. La méthode recommandée s'appuie sur **Composer** :

```
$ php composer.phar require swiftmailer/
swiftmailer @stable
```

Ou bien, si Composer est installé globalement :

```
$ composer require swiftmailer/
swiftmailer @stable
```

Une troisième façon de faire est de s'appuyer sur Git :

```
$ git clone git://github.com/swiftmailer/
swiftmailer.git
```

Phase 4

Mettez en place votre formulaire de contact dans le fichier `contact.html` dans votre dossier de travail :

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Contact</title>
  </head>

  <body>
    <h1>Contact</h1>
    <form action="process.php">
      <input name="name"
placeholder="Votre nom"><br>
      <input name="mail"
placeholder="Votre adresse email"><br>
      <input name="subject"
placeholder="Objet"><br>
      <textarea name="content"
placeholder="Votre message"></textarea>
      <input type="submit"><br>
    </form>
  </body>
</html>
```

Phase 5

Voyons maintenant comment créer notre message.

Si vous utilisez Composer et que votre projet est déjà paramétré pour, l'inclusion de SwiftMailer sera automatique. Pour le cas où ce ne serait pas le cas, vous devrez commencer votre script par l'inclusion de l'*autoloader* de Composer :

```
01: <?php
02:   require_once('vendor/autoload.php');
```

Si vous avez opté pour l'installation par Git, vous devrez ajouter l'*autoloader* propre de SwiftMailer :

```
01: <?php
02:   require_once('swift-mailer/lib/swift_
required.php');
```

Cela fait, vous pouvez commencer la création de votre message. En premier lieu, définissez-en le destinataire. Ici, j'utilise une variable de remplacement. Vous devrez naturellement adapter ce code à votre propre situation

Il est possible d'indiquer plusieurs destinataires en utilisant un tableau :

```
04:   $config['contactDest'] =
        array('mailto:example@example.com',
'mailto:example@example.com');
```

Ajoutez maintenant au message les données recueillies depuis le formulaire :

```
06: $msg = Swift_Message::newInstance()
07:       ->setTo($config['contactDest'])
08:          ->setFrom($_GET['mail'])
09:          ->setSubject($_GET['subject'])
10:          ->setBody($_GET['content'])
```

Nous n'avons ici indiqué que l'essentiel, mais les possibilités de SwiftMailer quant au traitement du message sont beaucoup plus vastes et vont depuis la génération d'un corps en HTML, l'ajout de pièces jointes, à la demande d'un accusé de réception et même la signature numérique. Pour plus d'informations, consultez la documentation officielle sur http://swiftmailer.org/docs/messages.html.

Phase 6

Maintenant que votre message est prêt, il ne reste plus qu'à l'envoyer. SwiftMailer est susceptible d'utiliser plusieurs méthodes de transport : le protocole SMTP, une commande locale telle que **sendmail** ou encore, **à éviter**, la fonction PHP **mail**. À nouveau ce genre d'éléments sont à définir dans la configuration. Voici des exemples de remplacement :

Le plus modulaire est d'utiliser SMTP :

```
12:   $config['transportMethod'] = 'SMTP';
13:   $config['smtpHost'] = 'smtp.example.org';
14:   $config['smtpPort'] = 25;
```

Des paramètres de configuration avancés de SMTP sont possibles, permettant d'utiliser une connexion chiffrée ou d'effectuer l'authentification de l'utilisateur. Consultez la documentation officielle sur ce point : http://swiftmailer.org/docs/sending.html#the-smtp-transport.

Pour utiliser Sendmail, les choses sont plus simples, mais il faut que votre système soit configuré correctement pour envoyer lui-même les mails :

```
12:   $config['transportMethod'] = 'SENDMAIL';
13:   $config['transportCommand'] = '/usr/sbin/
sendmail -bs'; // ce paramètre est optionnel et
ne requiert de modification que s'il diverge de
la valeur par défaut indiquée ici.
```

Enfin, la configuration la plus simple qui utilise la fonction PHP **mail** :

```
12:   $config['transportMethod'] = 'PHPMAIL';
```

Naturellement, dans ce cas, tout reposera sur votre **php.ini** qui devra être configuré correctement. Sous Unix, il s'appuiera sur Sendmail ou un autre programme d'envoi de messages. Sous Windows, il s'appuiera sur SMTP.

Si votre code doit être portable, je ne saurais trop vous recommander de vous appuyer en priorité sur la méthode SMTP de SwiftMailer.

Phase 7

Il est temps maintenant de créer l'objet qui va prendre en charge l'envoi du message. La classe à utiliser dépend de la méthode d'envoi.

```
16:   switch ($config['transportMethod']) {
17:     case 'SMTP':
18:       $transport = Swift_SmtpTransport::
newInstance($config['smtpHost'],
$config['smtpPort']);
19:       break;
20:     case 'SENDMAIL':
21:       $transport = Swift_SendmailTransport::
newInstance($config['transportCommand']);
22:       break;
23:     default:
24:       $transport = Swift_
MailTransport::newInstance();
25: }
```

Phase 8

Un objet intermédiaire est nécessaire pour utiliser le transport . Il vous faudra l'utiliser pour effectuer l'envoi du message. Il est appelé *mailer* et est de la classe **Swift_Mailer** :

```
27:   $mailer = Swift_Mailer::newInstance
($transport);
```

Phase 9

Tout est prêt maintenant pour envoyer votre message :

```
29:   $result = $mailer->send($message,$failures);
```

Phase 10

Il convient maintenant d'avertir l'utilisateur du résultat de cet envoi. La valeur de retour de la méthode **send** indique si des erreurs se sont produites et son second paramètre, passé par référence, reçoit un tableau contenant la liste des destinataires pour lesquels l'envoi a échoué.

```
31:   if ($result){
32:     print '<p>Votre message a été envoyé
avec succès !</p>';
33:   }else{
34:     print '<p>Votre message n\' pas pu être
envoyé à certains destinataires. En voici la
liste :</p><ul>';
35:     foreach ($failures as $failure){
36:       print '<li>'.$failure.'</li>';
37:     }
38:     print '</ul>';
39:   }
```

LE RÉSULTAT

Nous avons maintenant un formulaire de contact utilisant SwiftMailer, dont le fonctionnement est bien moins aléatoire que celui de la fonction **mail**. ■

CRÉER DES DOCUMENTS SIMPLEMENT AVEC PILLAR

Damien Cassou [Enseignant chercheur, auteur et contributeur sur de nombreux projets libres dont Pillar]

Pillar est un format textuel simple et concis. Un de plus ? Oui. Les développeurs de Pillar (dont moi) ont réutilisé le parser d'un vieux CMS appelé Pier et l'ont modifié jusqu'à en faire une plateforme idéale pour tous leurs besoins de documentation : des livres (dont 3 terminés ou quasiment terminés), des manuels techniques, des slides de cours (dont un MOOC sur Pharo) et des sites web. Le résultat de ce travail est un outil incluant des fonctions puissantes comme les éléments référençables (les titres, les figures, les scripts), la génération automatique de texte, la numérotation finement paramétrable des titres, et la génération de slides et de colonnes de texte. Cet article présente un tutoriel d'utilisation et les bases de la syntaxe

Mots-clés : Format textuel, PDF, HTML, Slides, Markdown, Pharo

Résumé

Pillar [1] est un format textuel simple et concis, c'est aussi un logiciel libre (licence MIT) et multiplateforme, écrit en Pharo [2], permettant de convertir un texte Pillar vers un fichier PDF avec LaTeX (voir figure 1), HTML (voir figure 2) ou Markdown. D'autres formats de sortie sont en développement ou prévus comme Asciidoc et un format XML propre à une plateforme de cours en ligne.

Pillar a déjà été utilisé à de nombreuses occasions dans la communauté Pharo afin de produire des articles, des livres **[3][4]**, le contenu des CMS **Pier** et **Marina**, le contenu du générateur de sites web statiques **Ecstatic [5]** et même le MOOC (cours en ligne) Pharo **[6]** pour la plateforme France Université Numérique (60 cours avec enseignant filmé, des dizaines de *screencasts*, des exercices de conception, des quiz, etc.). Pillar peut générer non seulement des documents textuels (livres, sites), mais aussi des *slides* (avec **DeckJS** ou **Beamer**, voir figure 3) pour illustrer des présentations orales ou des cours.

Au-delà des plugins pour quelques éditeurs de texte (**Emacs**, **VIM**, **Textmate**, **ATOM**), Pillar dispose également de sa plateforme d'édition en ligne **PillarHub [7]**.

1 Tutoriel en 5 minutes

Commençons par écrire un premier document en Pillar pour sortir du HTML. Il vous faut d'abord créer un dossier dans lequel mettre votre prose, Pillar lui-même et son fichier de configuration.

```
$ mkdir mydocument
$ cd mydocument
```

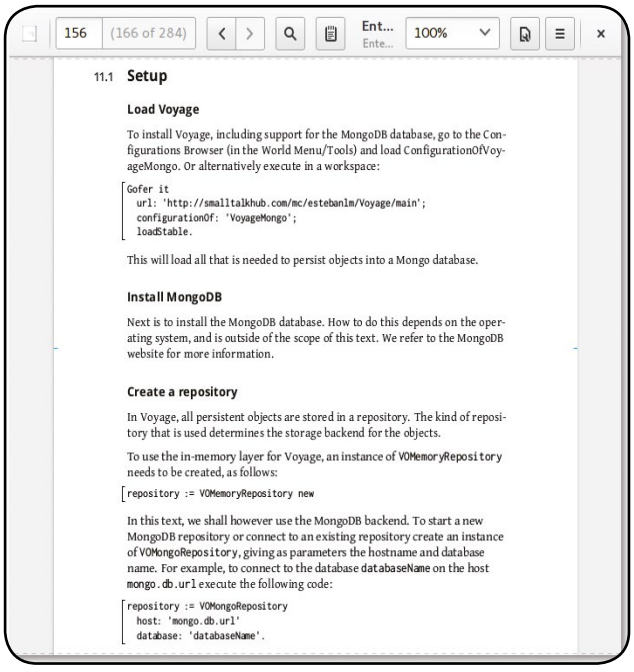


Fig. 1 : Extrait PDF d'un chapitre du livre Enterprise Pharo **[3]**, *écrit en Pillar.*

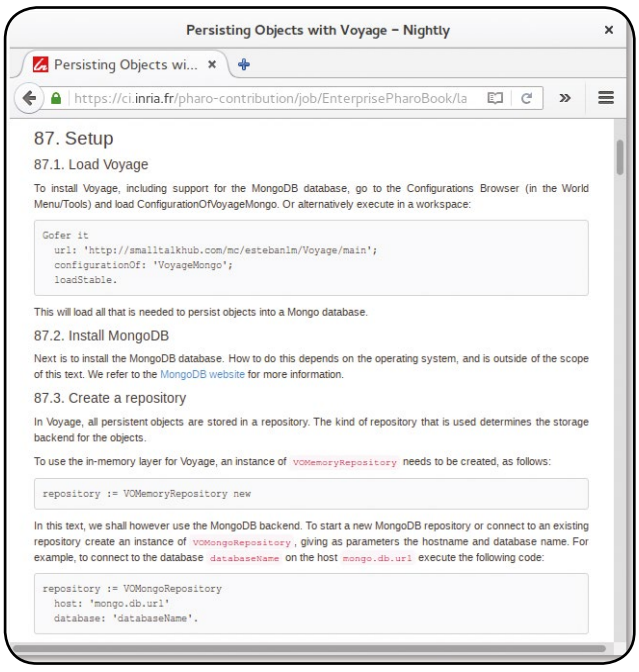


Fig. 2 : Extrait HTML d'un chapitre du livre Enterprise Pharo **[3]**, *écrit en Pillar.*

1.1 Installation et génération d'un premier document

Téléchargez Pillar grâce au script **download.sh** :

```
$ wget https://raw.githubusercontent.com/pillar-markup/book-skeleton/master/download.sh
$ bash ./download.sh
```

Créez maintenant un fichier **first.pillar** avec le contenu suivant :

```
!Section
!!Sous-section
```

Un point d'exclamation en début de ligne spécifie un titre. Exportez ce document vers un fichier HTML :

```
$ ./pillar export --to=html first.pillar
```

Ceci va générer un fichier **output.html** que vous pouvez ouvrir dans un navigateur web et qui ressemble à l'extrait ci-dessous :

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <title></title> [...]
   </head>
   <body>
      <div class="container"> [...]
         <h1>I. Section</h1> [...]
         <h2>I.1 Sous-section</h2> [...]
      </div> [...]
   </body>
</html>
```

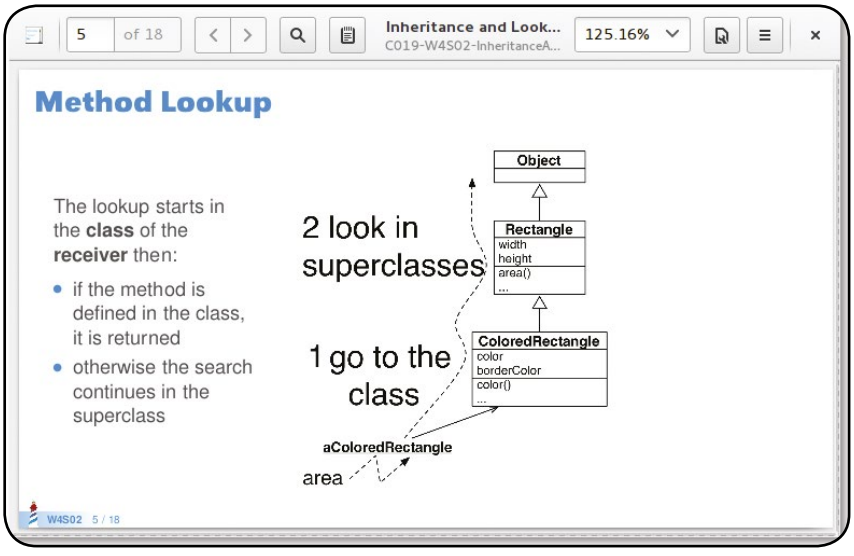


Fig. 3 : Extrait PDF d'un diaporama du MOOC Pharo **[6]**, *écrit en Pillar.*

Les titres de section que vous avez indiqués ont été placés dans les balises **<h1>** et **<h2>** et ont été numérotés automatiquement. Nous allons maintenant voir comment configurer Pillar pour changer la sortie.

1.2 Configurer Pillar pour ce document

Comme vous pouvez le voir dans la sortie HTML ci-dessus, les sections possèdent un titre (dans **<h1>** et **<h2>**), mais le document lui-même n'en possède pas (rien dans **<title>**). Pour ajouter des métadonnées à un document, vous devez les spécifier au début du fichier :

```
{
   "title": "Tutorial Pillar en 5 minutes"
}

!Hello World
```

Après compilation, votre document doit maintenant avoir un titre :

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <title>Tutorial Pillar en 5 minutes</title>
   </head>[...]
```

Une autre méthode permettant d'arriver au même résultat est d'éditer un fichier de configuration. Ce fichier s'appelle habituellement **pillar.conf**. Il se trouve dans le même dossier que les fichiers Pillar et il doit être écrit au format **STON [8]** (proche du **JSON**). Créez votre premier fichier **pillar.conf** avec ce contenu :

```
{
   "title" : "Tutorial Pillar en 5 minutes avec pillar.conf"
}
```

Les métadonnées des fichiers Pillar sont prioritaires par rapport aux métadonnées des fichiers de configuration : il va donc falloir effacer les métadonnées de **first.pillar** pour voir le titre spécifié dans **pillar.conf**.

Le fichier **pillar.conf** supporte une myriade de paramètres. Par exemple, le paramètre **headingLevelOffset** permet de décaler le niveau de titre Pillar et le niveau de titre du format de sortie : un titre niveau 1 en Pillar (un point d'exclamation) peut ainsi être transformé en un titre HTML de niveau 2 (**<h2>**). Grâce aux paramètres **level<X>** (**<X>** est un entier positif), vous pouvez changer la numérotation automatiquement. Le fichier **pillar.conf** ci-dessous montre une configuration possible :

```
{
   "title" : "Tutorial Pillar en 5 minutes avec pillar.conf",
   "headingLevelOffset" : 1,
   "level1" : {
      "size" : 0,
      "renderAs" : "letter"
   },
   "level2" : {
      "size" : 1,
      "renderAs" : "upperLetter"
   }
}
```

Grâce à la même commande que précédemment, on va obtenir le fichier HTML ci-dessous :

```
<h2>Section</h2> [...]
   <h3>A. Sous-section</h3>
```

Le paramètre **level1** permet de contrôler les titres de section de niveau 1. Le sous paramètre **size** contrôle le nombre d'éléments de numérotation à afficher (par exemple, **2** affiche un numéro pour la section courante précédée d'un numéro pour la section parente). Le sous-paramètre **renderAs** permet de choisir la numérotation utilisée parmi une liste de styles (**number**, **roman**, **letter** ou **upperLetter**).

1.3 Choix entre sortie unique et sorties multiples

Pillar a une notion de projet dans laquelle plusieurs fichiers Pillar peuvent être exportés soit au sein d'un même fichier (par exemple, un fichier LaTeX pour l'ensemble d'un livre), soit au sein de plusieurs fichiers (par exemple un fichier HTML par chapitre).

Si vous créez un fichier **second.pillar** qui contient **I am a second file**, vous pouvez référencer vos deux fichiers sources dans le fichier **pillar.conf** de cette façon :

```
{
   "title" : "Tutorial Pillar en 5 minutes avec pillar.conf",
   "inputFiles" : [ "first.pillar", "second.pillar" ]
}
```

Par défaut, Pillar exporte tous les fichiers sources vers un fichier unique :

```
$ ./pillar export --to=html
```

Ceci devrait construire un fichier **output.html** qui contient :

```
[...]
<h1>I.  Hello World</h1>[...]
   <h2>I.1. Sous-section</h2>[...]
   <p>I am a second file</p>
```

Le paramètre **outputFile** peut-être utilisé pour que Pillar génère un fichier nommé autrement que **output.html**.

Pour séparer les fichiers générés, utilisez le paramètre **separateOutputFiles** dans **pillar.conf** :

```
{
   [...]
   "inputFiles" : [ "first.pillar", "second.pillar" ],
   "separateOutputFiles" : true
}
```

La commande précédente devrait générer deux fichiers HTML : **first.html** et **second.html**. Voici un extrait de **second.html** :

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <title>Tutorial Pillar en 5 minutes avec pillar.conf
</title>[...]
   </head>
   <body>
      <div class="container">
         <p>I am a second file</p>
      </div>[...]
</html>
```

Pour éviter de mélanger fichiers générés et fichiers sources, vous pouvez spécifier le dossier de sortie dans le paramètre **outputDirectory** :

```
{
   [...]
   "inputFiles" : [ "first.pillar", "second.pillar" ],
   "separateOutputFiles" : true,
   "outputDirectory" : "result"
}
```

1.4 Configurer le patron de génération

Comme vous avez pu le constater, Pillar génère non seulement le contenu du document dans **<body>**, mais aussi ce qui l'entoure (notamment l'en-tête du document). Ce qui est généré autour du document peut être modifié en spécifiant un patron de génération différent du patron par défaut. C'est utile par exemple pour référencer votre propre feuille de styles CSS ou pour ajouter un pied de page. **Mustache [9]** est utilisé pour définir les patrons. Par exemple, placez le patron HTML ci-dessous dans un fichier **myhtml.template** :

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <title>{{{title}}}</title>
   </head>
   <body>
      <div class="container">
         {{{content}}}
      </div>
      <footer>
         <p>{{author}}, {{year}}</p>
      </footer>
   </body>
</html>
```

Il vous faut ensuite référencer ce patron depuis votre fichier de configuration :

```
{
   [...]
   "year" : 2016,
   "author" : "Vous Même",
   "template" : "myhtml.template"
}
```

Exportez ce fichier et vous devriez obtenir un fichier HTML tout à fait similaire à votre patron dans lequel les termes entre triples accolades **{{{** ont été remplacés.

Le fichier **pillar.conf** peut gérer plusieurs sorties avec différents paramètres. Par exemple, pour un livre, un fichier PDF complet peut être généré pour être envoyé à un imprimeur tandis que les chapitres sont exportés dans des fichiers HTML séparés pour être visionnés sur le site du livre et dans des fichiers Markdown séparés pour être visualisés sur **GitBook**. Tous les paramètres de **pillar.conf** (par exemple ceux gérant la numérotation des titres et le patron) peuvent avoir des valeurs différentes en fonction de la sortie. Vous pouvez aller voir les fichiers sources du livre « Enterprise Pharo » **[3]** pour vous rendre compte des nombreuses possibilités.

2 La syntaxe

Dans cette section, nous développons les éléments de base de la syntaxe Pillar, puis nous nous attardons sur quelques éléments différenciants. Un résumé de la syntaxe se trouve figure 4, page suivante.

Fig. 4 : Résumé de la syntaxe Pillar **[10]**.

Les titres sont indiqués par un ou plusieurs points d'exclamation en début de ligne. Le nombre de points d'exclamation indique la profondeur du titre. Pour pouvoir référencer un titre depuis un autre endroit du document, il est commode d'ajouter une ancre à la ligne suivant l'endroit marqué. Par exemple, l'exemple suivant crée un titre et une ancre qui pourront ensuite être référencés par un lien **`*@syntaxe*`**.

```
! La syntaxe
@syntaxe
```

Les paragraphes sont séparés par une ligne vide. Il est possible de créer des types de paragraphes particuliers pour les représenter différemment. Par exemple, un paragraphe **`todo`** peut encourager le lecteur à faire quelque chose grâce à un cadre autour du paragraphe et une icône de boîte à outils placée au début. Pour créer un tel paragraphe, il faut le préfixer de **`@@<paragraphClass>`** dans lequel **`<paragraphClass>`** est n'importe quel identifiant choisi par l'auteur (**`todo`**, **`idea`**, **`references`**, **`summary`**...). Pour un tel paragraphe, Pillar va générer du *markup* spécifique (e.g., un attribut **`class=<paragraphClass>`** en HTML, un environnement **`\begin{<paragraphClass>}`** en LaTeX) qu'il faudra ensuite représenter différemment (grâce à CSS pour HTML et en définissant l'environnement pour LaTeX).

Il existe trois **types de listes** qui peuvent être arbitrairement imbriquées les unes dans les autres. Les éléments d'une liste ordonnée commencent par un dièse, ceux d'une liste non ordonnée commencent par un tiret, et ceux d'une liste de description commencent soit par un point-virgule pour les termes, soit par un deux-points pour les descriptions. Le document Pillar ci-dessous est un exemple d'utilisation d'une liste avec des éléments imbriqués :

```
; listes ordonnées
: les éléments de ces listes commencent
par ~#~ ce qui génère un nombre qui
augmente à chaque nouvel élément
; listes non ordonnées
: les éléments de ces listes commencent
par ~-~ ce qui génère une puce à chaque
élément
; listes de description
: les éléments sont normalement une
alternance de termes commençant par un
~;~ et de descriptions commençant par
un ~:~
; imbrication de listes
:- les listes peuvent être imbriquées
:- il suffit d'indiquer tous les
marqueurs d'éléments de liste en début
de ligne
```

À partir de cet exemple, Pillar va générer le HTML suivant :

```
<dl>
   <dt>listes ordonnées</dt>
   <dd>les éléments...<dd>
   <dt>listes non ordonnées</dt>
   <dd>les éléments...</dd>
   <dt>listes de description</dt>
   <dd>les éléments...</dd>
   <dt>imbrication de listes</dt>
   <dd>
      <ul>
         <li>les listes peuvent être
imbriquées</li>
         <li>il suffit...</li>
      </ul>
   </dd>
</dl>
```

Différents **formats de texte** sont gérés : le **gras** avec deux guillemets (**`""gras""`**), l'**italique** avec deux apostrophes (**`''italique''`**), le **monospace** avec deux signes égal (**`==monospace==`**),

le **barré** avec deux signes moins (**--barré--**), le **subscript** avec deux arobases (**@@subscript@@**), le **superscript** avec deux accents circonflexes (**^^superscript^^**) et le **souligné** avec deux tirets bas (**__souligné__**).

Les **tableaux** se font en séparant les cellules par des barres verticales. Un point d'exclamation en début de cellule permet d'indiquer une cellule titre. Les cellules peuvent être alignées à gauche avec **{**, à droite avec **}** ou centrées avec **|**. Le texte Pillar ci-dessous est un exemple de tableau.

```
||  cellule centrée  !!|  titre centré     || cellule centrée
|{  cellule à gauche |}   cellule à droite || cellule centrée
```

À partir de cet exemple, Pillar va générer le HTML suivant :

```
<table style="border: solid thin">
   <tr>
      <td style="text-align: center">cellule centrée</td>
      <th style="text-align: center">titre centré</th>
      <td style="text-align: center">cellule centrée</td>
   </tr>
   <tr>
      <td style="text-align: left">cellule à gauche</td>
      <td style="text-align: right">cellule à droite</td>
      <td style="text-align: center">cellule centrée</td>
   </tr>
</table>
```

En Pillar, on peut spécifier **des liens** externes (vers des sites web, des adresses e-mail, des images), mais aussi des liens internes vers d'autres emplacements du même document (sections, figures, scripts). Tous les liens s'écrivent sur le même modèle ***alias>reference@anchorName*** où **alias** représente le texte à afficher à la place du chemin vers la destination, **reference** représente le chemin vers la destination et **anchorName** spécifie un endroit particulier dans le document (i.e., une ancre). Voici quelques exemples :

- ***http://www.pharo.org*** est un lien externe vers le site de Pharo ;
- ***Pharo>http://www.pharo.org*** affiche **Pharo** plutôt que l'adresse du lien ;
- ***https://fr.wikipedia.org/wiki/Pharo#Introduction*** est un lien externe vers la section **Introduction** de la page Wikipédia de Pharo ;
- ***@bla*** est un lien interne vers l'ancre **bla** du document en cours ;
- ***damien@cassou.me*** est un lien **mailto** vers mon adresse e-mail.

Les figures sont des liens externes vers un fichier image pour lesquels l'étoile a été remplacée par un signe plus. La partie **alias** du lien représente la légende. Il est possible de spécifier des paramètres à une figure pour, par exemple, indiquer une ancre (avec le paramètre **label**) ou la taille de l'image (avec **width**). Le code suivant affiche une image que l'on peut référencer avec le lien ***@pharoLogo*** :

```
+Ceci est la légende>file://figures/pharo-logo.
png|width=50|label=pharoLogo+
```

Les scripts permettent d'insérer des lignes de code au sein du document. Chaque script doit être entouré par une série de trois crochets. Voici un exemple de script :

```
[[[label=script1|caption=My script that works|language=smalltalk
self foo bar
]]]
```

La première ligne donne des informations sur le script. Le script est référençable par un lien vers l'ancre **script1** grâce à ***@script1***. Il a pour légende « *My script that works* » et le contenu doit être considéré comme étant du Smalltalk pour la coloration syntaxique.

Il est possible de générer du contenu automatiquement à partir d'un code Smalltalk. Ceci doit se faire dans un script avec le paramètre **eval** ayant la valeur **true** :

```
[[[eval=true
stream
    nextPutAll: 'GNU/Linux Magazine a "';
    print: ((Date today - '1998/09/01' asDate) days / 365.25) floor;
    nextPutAll: ' ans".'
]]]
```

Ce script va afficher la phrase « *GNU/Linux Magazine a 17 ans* », avec la valeur **17** provenant d'une soustraction entre la date du jour et la date de parution du premier numéro (septembre 1998). L'ensemble de l'API Pharo est disponible. Dans tous les cas, il faut envoyer à l'objet **stream** le texte à afficher. Cerise sur le gâteau, n'importe quel texte au format Pillar peut être envoyé ! Dans l'exemple ci-dessus, *« 17 ans »* sera en gras grâce aux deux guillemets après *« a »* et *« ans »*.

Pillar permet de créer **des slides** comme vous pouvez le voir figure 3. Par exemple :

```
${slide:title=Method Lookup}$
The lookup starts in the ""class"" of the ""receiver"" then:
- if the method is defined in the class, it is returned
- otherwise the search continues in the superclass

${slide:title=Some Lookup Cases}$
[...]
```

La première et la dernière ligne du code ci-dessus définissent chacune un *slide* dont le titre est passé en paramètre. Tout ce qui se trouve entre la ligne de définition d'un *slide* et la ligne de définition du *slide* suivant fait partie du premier *slide* défini.

Il est possible de séparer le contenu en colonnes. C'est particulièrement pratique dans les *slides* pour mettre du texte à côté d'une figure. Par exemple, le code suivant crée un *slide* à 2 colonnes :

```
${slide:title=Method Lookup}$
${columns}$

${column:width=38}$
The lookup starts in the ""class"" of the ""receiver"" then:
- if the method is defined in the class, it is returned
- otherwise the search continues in the superclass

${column:width=62}$
+file://figures/InheritanceDiagram-lookupTwoStages.png|width=100+

${endColumns}$
```

La première colonne, de taille 38% contient un peu de texte tandis que la colonne de droite contient une figure.

Conclusion

Pillar est en développement actif, car les auteurs continuent à écrire des documents, ils ont donc besoin de toujours plus de fonctions. Grâce à du code objet de très grande qualité (petites méthodes, design patterns simples, commentaires, beaucoup de tests automatiques), le développement de Pillar peut être pris en main facilement.

Pillar se différencie de certains concurrents grâce à sa notion de projet qui lui permet de faire de la publication assistée par ordinateur (PAO) pour par exemple écrire des livres techniques complets avec plusieurs sorties (par exemple PDF pour l'impression et HTML pour la lecture en ligne). Pillar peut aussi être utilisé comme logiciel de présentation assistée par ordinateur (PréAO) pour créer des diaporamas de cours. ■

Références

[1] Site de Pillar : http://www.smalltalkhub.com/#!/~Pier/Pillar

[2] Site de Pharo : http://www.pharo.org

[3] Le livre « Enterprise Pharo », entièrement écrit en Pillar : http://books.pharo.org/enterprisepharo/

[4] Le livre « Updated Pharo by Example », entièrement écrit en Pillar : https://github.com/SquareBracketAssociates/UpdatedPharoByExample

[5] Ecstatic pour générer des sites web statiques : http://guillep.github.io/ecstatic/

[6] Le MOOC Pharo : https://www.france-universite-numerique-mooc.fr/courses/inria/41010/session01/about

[7] PillarHub pour écrire du Pillar en ligne : http://pillarhub.pharocloud.com/

[8] STON pour décrire les métadonnées et configurer Pillar : http://smalltalkhub.com/#!/~SvenVanCaekenberghe/STON

[9] Mustache pour décrire des *templates* : http://mustache.github.io/

[10] Résumé de la syntaxe Pillar originellement créée par Benjamin Van Ryseghem : http://www.cheatography.com/benjaminvanryseghem/cheat-sheets/pillar/

PROFESSIONNELS DES TICE, COLLECTIVITÉS, ÉCOLES D'INGÉNIEURS, UNIVERSITÉS, R & D, ENSEIGNANTS, ...

LINUX MAGAZINE / FRANCE

PROFESSIONNELS !

DÉCOUVREZ NOS OFFRES D'ABONNEMENTS ...

...EN VOUS CONNECTANT À L'ESPACE DÉDIÉ AUX PROFESSIONNELS SUR :

www.ed-diamond.com

PDF COLLECTIFS PRO

OFFRE	ABONNEMENT	1 - 5 lecteurs Réf	1 - 5 lecteurs Tarif TTC	6 - 10 lecteurs Réf	6 - 10 lecteurs Tarif TTC	11 - 25 lecteurs Réf	11 - 25 lecteurs Tarif TTC
PROLM2	11n° GLMF	☐ PRO LM2/5	260,-	☐ PRO LM2/10	520,-	☐ PRO LM2/25	1040,-
PROLM+2	11n° GLMF + 6n° HS	☐ PRO LM+2/5	472,-	☐ PRO LM+2/10	944,-	☐ PRO LM+2/25	1888,-

Prix TTC en Euros / France Métropolitaine

PROFESSIONNELS :
N'HÉSITEZ PAS À NOUS CONTACTER POUR UN DEVIS PERSONNALISÉ PAR E-MAIL :
abopro@ed-diamond.com
OU PAR TÉLÉPHONE :
03 67 10 00 20

ACCÈS COLLECTIFS BASE DOCUMENTAIRE PRO

OFFRE	ABONNEMENT	1 - 5 connexion(s) Réf	1 - 5 connexion(s) Tarif TTC	6 - 10 connexions Réf	6 - 10 connexions Tarif TTC	11 - 25 connexions Réf	11 - 25 connexions Tarif TTC
PROLM+3	GLMF + HS	☐ PRO LM+3/5	267,-	☐ PRO LM+3/10	534,-	☐ PRO LM+3/25	1068,-
PROA+3	GLMF + HS + LP + HS	☐ PRO A+3/5	297,-	☐ PRO A+3/10	594,-	☐ PRO A+3/25	1188,-
PROH+3	GLMF + HS + LP + HS + MISC + HS + OS	☐ PRO H+3/5	447,-	☐ PRO H+3/10	894,-	☐ PRO H+3/25	1788,-

Prix TTC en Euros / France Métropolitaine

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France HS = Hors-Série LP = Linux Pratique OS = Open Silicium

SÉLECTIONNEZ VOTRE OFFRE DANS LA GRILLE CI-DESSUS ET RENVOYEZ CE DOCUMENT COMPLET À L'ADRESSE CI-DESSOUS !

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
E-mail :	

☐ Je souhaite recevoir les offres promotionnelles et newsletters des Éditions Diamond.
☐ Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : **boutique.ed-diamond.com/content/3-conditions-generales-de-ventes** et reconnais que ces conditions de vente me sont opposables.

LINUX MAGAZINE / FRANCE

Les Éditions Diamond
Service des Abonnements
10, Place de la Cathédrale
68000 Colmar – France
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

.....

.....

ATTEINDRE LE GRAAL : ÉCRIRE SON PROPRE LANGAGE DE PROGRAMMATION

Guillaume Saupin [Responsable R&D QosEnergy, Data Scientist]

Il est des quêtes dont on revient forcément changé. Celle que je vous propose est la quête ultime, celle que doit mener tout programmeur pour pouvoir un jour asséner avec force et sérénité cette affirmation : j'ai créé mon propre langage de programmation !

Mots-clés : C++, Lisp, LLVM, Compilation, Parseur, Langage

Résumé

Mettre au point son propre langage de programmation, voilà ce que nous proposons dans cette série d'articles. Dans ce premier, nous décrirons rapidement le Lisp, qui nous servira de modèle, puis nous écrirons en C++ un parseur récursif descendant pour passer du code à sa représentation informatique. Les articles suivants traiteront de l'interprétation, puis de la compilation avec LLVM.

Le langage **Lisp**, sans conteste le plus remarquable langage de programmation jamais mis au point, se distingue, entre autres, par sa capacité à générer lui-même du code. Dans le même esprit, nous allons dans cet article, vous armer pour que vous deveniez un programmeur qui écrit des programmes qui permettent d'écrire des programmes.

Et tant qu'à faire, nous n'allons pas nous contenter d'un petit langage lointainement dérivé du Lisp : nous allons directement taper dans le lourd, et implémenter un dialecte du Lisp.

Pour rendre la chose encore plus intéressante, et frapper un grand coup, nous allons essayer d'atteindre les objectifs suivants :

- Nous n'allons pas écrire notre Lisp dans un autre Lisp. Ce serait chose facile, et cela a déjà été fait, et même très bien fait, entre autres par Christian Quenneic, SICP, etc. Le **C++x14**, avec son cortège de nouvelles fonctionnalités et ses excellentes performances est un excellent candidat.
- Nous allons tenter d'obtenir des performances dignes de ce nom, et comparables à celles de langages compilés. Pour cela, halte à la pusillanimité, notre dialecte du Lisp n'aura rien à envier à son auguste géniteur, et devra pouvoir être aussi bien compilé qu'interprété.
- Nous allons nous efforcer d'organiser notre code de manière particulièrement modulaire, de manière à pouvoir étendre facilement notre langage. Pour cela, on organisera chaque extension de langage en autant de librairies dont le chargement sera dynamique.

Ce menu étant particulièrement copieux, afin d'éviter toute indigestion, nous allons aborder les différents points dans deux, voire trois articles.

Dans le premier, que vous tenez entre vos mains, nous rappellerons les particularités de Lisp, puis nous écrirons en C++ la couche d'analyse syntaxique.

Le second traitera de la partie interprétation, et le ou les articles suivants se pencheront sur la délicate question des performances, et montreront comment le projet LLVM peut venir à notre secours.

1 Lisp

1.1 Syntaxe

La syntaxe des dialectes du Lisp est simple et élégante. Un programme est un ensemble de S-expressions. Ces dernières sont définies récursivement comment étant soit un atome, soit une liste de S-expressions.

Un atome, est une chaîne de caractères, qui peut entre autres identifier une fonction, une variable, une forme spéciale, etc.

Les listes sont pour leur part constituées d'atomes ou de S-expressions entourées de deux parenthèses :

```
(+ 1 2) ;;; retourne 3
t ;;; retourne le booléen vrai
(defun add (a b)
   (+ a b)) ;;; définit la fonction add ajoutant deux nombres
(add 6 3) ;;;  retourne 9
```

1.2 Sémantique

La sémantique est elle aussi simple : le premier élément d'une liste est une fonction ou une macro, et les S-expression suivantes sont ses arguments.

Dans le cas d'un appel de fonction, les arguments sont évalués avant que la fonction ne leur soit appliquée. L'ordre d'évaluation des arguments peut varier d'une implémentation à une autre.

Dans le cas d'une macro, les arguments ne sont pas évalués, mais remplacés tels quels dans le code au moment de l'expansion de la macro.

1.3 Particularités du Lisp

Le lisp offre quelques fonctionnalités qui lui confèrent une souplesse et une élégance que peu de langages offrent, même si on observe actuellement une convergence de beaucoup de langages vers le Lisp. C'est le cas notamment du C++ avec l'introduction récente des fonctions anonymes par exemple.

1.3.1 Macros

La plus remarquable des fonctionnalités offertes par les langages de type Lisp est le mécanisme de macro. Ce mécanisme confère aux dialectes Lisp leur caractère homoiconique, c'est-à-dire que la structure de représentation d'un programme est aussi une structure de base du langage. Dit simplement, il est possible d'appliquer toutes les fonctions du langage sur la représentation du code. Ce sont ces fonctions que l'on nomme **macro**.

La structure de base du Lisp étant bien sûr la liste, le code est représenté, comme nous l'avons vu, sous forme de liste. Pour travailler sur cette représentation de code, Timothy Hart a introduit dès 1963 les macros. L'idée est de pouvoir modifier du code pour l'exécuter ensuite.

Pour illustrer les macros, nous allons en écrire une modérément complexe, avec pour objectif de transformer un bout de code en suivant les règles suivantes :

- les **+** sont transformés en **–** et inversement ;
- les ***** sont transformés en **/** et inversement ;
- toute autre expression est laissée telle quelle ;
- enfin, nous appliquons récursivement ces transformations aux sous-listes de code.

Ainsi, le bout de code suivant :

```
(+ (- 12 5) (* 4 5))
```

Doit devenir :

```
(- (+ 12 5) (/ 4 5))
```

Voici notre macro :

```
(defmacro exchange (exp)
  (labels ((swap (x)
       (cond
         ((eq x '+) '-)
         ((eq x '-) '+)
         ((eq x '*) '/)
         ((eq x '/) '*)
         ((listp x)
    (mapcar #'swap x))
         (t x)))) ;;; fin de la déclaration des fonctions locales
   `,(mapcar #'swap
        exp)))
```

Analysons rapidement ce bout de code, un peu complexe il est vrai pour une première macro, mais qui montre bien qu'on peut appliquer tous les outils du Lisp pour modifier du code.

Tout d'abord, notons le premier symbole, **defmacro**, qui indique que nous allons définir une macro nommée **exchange** qui prendra un unique paramètre **exp**.

Le corps de cette macro contient ensuite un bloc commençant par **label**. Ce bloc permet de définir une liste de fonctions locales, qui pourront être appelées par la suite dans les blocs de code suivants, mais uniquement au sein du bloc **label**.

Nous n'avons défini dans ce bloc que la fonction **swap**, qui prend en argument **x**, et qui le compare ensuite à **+**, **-**, *****, **/**, et change le sens selon les règles énoncées ci-dessus. Notez aussi la présence du prédicat **listp**, qui retourne vrai si le paramètre **x**, est une liste. Dans ce cas, la fonction **swap** est appliquée récursivement à la liste **x**, à l'aide de **mapcar** qui est une fonction qui applique une fonction à tous les éléments d'une liste et retourne cette nouvelle liste.

Enfin, faites attention à la présence du *backquote* ` (**<Alt>** + **7**), qui indique que nous rentrons dans le cœur de la macro. Tout ce qui est après cette *backquote* sera retourné tel quel, sans être exécuté, à l'exception notable près du code situé après une virgule.

Dans le cas de la virgule, le code qui la suit est exécuté et le résultat de l'exécution se substitue à ce qu'il y avait après la virgule.

Appliquons maintenant notre macro à un bout de code :

```
CL-USER> (exchange (+ (- 12 5) (* 4 5)))
81/5
```

Un rapide calcul nous confirme que notre code a bien été modifié selon nos règles puis exécuté. On peut néanmoins vérifier plus finement, en utilisant **macroexpand**, qui permet d'afficher le code une fois transformé par la macro :

```
CL-USER> (macroexpand '(exchange (+ (- 12 5) (* 4 5))))
(- (+ 12 5) (/ 4 5))
```

1.3.2 Let Over Lambda

Un autre mécanisme offert par le Lisp, et que l'on retrouve de plus en plus dans les langages *dits modernes*, mais aussi récemment en C++, est la fermeture, ou *closure* en bon anglais. En Lisp, on la nomme généralement *Let Over Lambda*, en raison de la forme sous laquelle elle apparaît généralement dans le code :

```
(let ((counter 0))
  (defun inc-counter ()
    (incf counter))
  (defun get-counter ()
     counter)
  (defunc dec-counter ()
    (decf counter)))
```

L'idée du *Let Over Lambda*, que l'on retrouve dans l'exemple ci-dessus, est de capturer une variable dans le corps d'une fonction. Ce mécanisme permet alors de disposer d'une variable accessible uniquement dans la portée du **let**, mais dont la valeur perdure en dehors des appels à la fonction créée.

L'exemple ci-dessous montre ainsi comment ce mécanisme peut être utilisé pour construire un compteur et fournir de quoi l'incrémenter et le décrémenter.

Il est intéressant de remarquer qu'avec ces quelques lignes de code, on arrive à recréer une sorte d'objet, dans le sens d'orienté objet, avec un état interne, **counter**, un accesseur, **get-counter**, et deux méthodes : **dec-counter** et **inc-counter**.

1.3.3 Variables dynamiques

Enfin, dernière fonctionnalité du Lisp, qui s'avère à l'usage très pratique : les variables dynamiques, ou variables spéciales. Ce type de variables s'oppose à celui des variables dites lexicales.

Les variables lexicales sont celles que l'on retrouve dans la quasi-totalité des langages. Elles sont liées à une unique valeur, au sein de leur portée.

Les variables dynamiques, pour leur part, peuvent voir la valeur à laquelle elles sont liées changer dynamiquement, au gré de liaisons qui peuvent être réalisées en dehors de leur portée.

La distinction entre ces deux modes de fonctionnement apparaît très clairement à travers un simple exemple :

```
(defun my-lexical-val ()
  (let ((var 3))
    var))
(defvar *var* 3 "A dynamical variable")
(defun my-dynamic-var ()
  *var*)
```

La variable dynamique, entourée par convention par deux étoiles, est déclarée ici à l'aide de la forme spéciale **defvar**.

Si nous exécutons maintenant ces deux fonctions, en liant leurs variables respectives avec de nouvelles valeurs, on a alors :

```
CL-USER> (let ((var 42)) (my-lexical-val))
3
CL-USER> (let ((*var* 42)) (my-dynamic-var))
42
```

La variable ***var*** a bien été dynamiquement liée à une nouvelle valeur. Ce mécanisme nous offre donc une sorte de variable globale, mais dont la liaison peut être modifiée dynamiquement, avec un effet uniquement au sein de la portée. De plus, ce mécanisme est *thread-safe* !

2 Notre Lisp : Llisp

2.1 Fonctionnalités

Afin d'être un Lisp digne de ce nom, notre dialecte va devoir supporter bon nombre des spécificités du Lisp citées ci-dessus. Nous allons donc assurer :

- le support des macros : c'est le minimum syndical ;
- le *Let Over Lambda* ;
- le support de quatre types (dans l'esprit de pico lisp) : les flottants, les chaînes de caractères, les symboles, et bien sûr les listes !
- la possibilité d'appeler du code natif, c'est-à-dire du C++ compilé.

À cela nous allons ajouter la contrainte suivante : le code doit être le plus modulaire possible, et notamment, il devra être possible de charger à chaud des extensions de langage. Nous allons donc supporter un mécanisme de *plugins*.

Enfin, et cela sera l'objet d'un troisième article, nous allons intégrer un compilateur afin de pouvoir compiler du code depuis le code !

2.2 Organisation du code

Notre code va s'articuler autour de deux ensembles :

- Le cœur de notre langage, à savoir le *parseur*, la structure de données représentant le code, et l'environnement.
- Une série d'extensions, qui nous permettront d'augmenter progressivement les capacités de notre langage : les formes spéciales, les fonctions de base, le support des fonctions, des chaînes de caractères, des listes, etc.

Le chargement se fera depuis notre langage, et aura la forme suivante :

```
(load "./core.so" "registerSpecialHandlers")
```

Cette expression permet par exemple de charger le cœur du langage, en appelant la méthode **registerSpecialHandlers** du module **core.so**.

Les modules présents seront les suivants :

- **special** : ce module contiendra ce que l'on nomme les formes spéciales du langage : **if**, **when**, **cond**, **quote**, **backquote**, etc.
- **core**, le cœur du système : l'arithmétique de base, les symboles spéciaux (**t**, **nil**), etc.

- **functional** : de quoi définir des fonctions. Très utile ;)
- **string** : de quoi manipuler des chaînes de caractères.
- **list** : la boite à outils pour jouer avec des chaînes.
- **compiler** : tout ce qu'il faut pour apporter de la performance à notre code, grâce à LLVM.

2.3 Les listes, cœur du système

Nous savons maintenant ce vers quoi nous voulons tendre. Nous pouvons attaquer les développements, en commençant par décrire les structures de données qui contiendront notre code.

2.3.1 Principes

Entrons maintenant dans le cœur du sujet, en détaillant la structure de données que nous allons utiliser pour représenter notre code : la cellule. Cette dernière peut contenir deux choses : soit un **Atom**, soit une **S-exp**, c'est-à-dire une liste de *cells*.

Cela peut sembler un point anecdotique d'avoir choisi la liste comme représentation du code, mais cela ne l'est absolument pas. C'est même à mon sens, et je m'en suis rendu compte en écrivant ce langage, ce qui fait la force du Lisp. En effet, le Lisp est un langage écrit pour manipuler des listes : rappelez-vous, Lisp signifie *Lists Processing*. Or, ce langage étant lui-même écrit sous la forme d'une liste, la *S-expression*, et représenté en interne par une liste, il devient alors naturel de manipuler le code depuis le code.

Et c'est là l'une des grandes forces du Lisp : son homoiconicité, c'est-à-dire que sa représentation est aussi la structure de donnée de base du langage.

2.3.2 Contenu des Cells

En terme de code, cela donne :

```
struct Cell
{
  virtual ~Cell() {};
};
struct Sexp : public Cell
{
  std::vector<std::shared_ptr<Cell> > cells;
  virtual ~Sexp() {};
  static std::shared_ptr<Sexp> New();
};

struct Atom : public Cell
{
  virtual ~Atom() {};
  protected:
  Atom(const std::string& val) {this->val = val;};
  Atom() {};
};

struct RealAtom : public Atom
{
  static std::shared_ptr<RealAtom> New();
  protected:
  RealAtom() {};
};

struct StringAtom : public Atom
{
  static std::shared_ptr<StringAtom> New();
  protected:
  StringAtom() {};
};

struct SymbolAtom : public Atom
{
  static std::shared_ptr<SymbolAtom> New();
  static std::shared_ptr<Atom> New(Cell::CellEnv& env, const
std::string& name);
 protected:
 SymbolAtom() {};
};
```

Nous avons donc un type parent, **Cell**, dont dérive le type **Atom**, dont dérivent ensuite les types **RealAtom**, **StringAtom**, et enfin **SymbolAtom**. Le type **Sexp**, dérive lui directement de **Cell**, et contient une liste de **Cells**.

À noter que pour des raisons de performances, nous avons implémenté nos listes à l'aide de **std::vector**.

J'attire aussi votre attention sur l'usage de **shared_ptr** qui a été choisi pour permettre une gestion simple de la mémoire. Ainsi, les classes terminales ne peuvent être créées qu'à l'aide de la fonction statique **New**. Le mécanisme de **share_ptr** s'assure qu'elles soient bien détruites quand plus personne ne les utilise.

Le code ci-dessus, a été expurgé d'une bonne partie de son contenu, afin de ne montrer que l'articulation entre les différents objets présents au sein d'une **Sexp**. L'ensemble du code est bien sûr disponible sur le GitHub associé à l'article. Il manque notamment le code permettant d'afficher nos structures avec un simple **std::cout**.

2.3.3 Parsing

Maintenant que nous avons notre structure de données, nous allons pouvoir la remplir en *parsant* du code. Pour cela, nous allons écrire un *parseur* récursif descendant, qui révèle avec quelle simplicité se *parse* du code Lisp. L'idée du *parseur* récursif descendant est d'utiliser des fonctions qui s'appellent récursivement, et qui collent de près à la grammaire du langage.

En Lisp, nous l'avons vu, la grammaire est simple : des listes de *cells* dont le premier élément est une fonction, et le reste des arguments. Nous allons donc avoir uniquement besoin de quatre fonctions :

- **parseCell(std::istream& is, Cell::CellEnv& env)** ;
- **parseSexp(std::istream& is, Cell::CellEnv& env)** ;
- **parseAtom(std::istream& is, Cell::CellEnv& env)** ;
- **parseString(std::istream& is, Cell::CellEnv& env)**.

Ces quatre fonctions prennent en entrée un *stream*, ainsi qu'un environnement, dont nous parlerons par la suite.

Le point d'entrée est bien sûr la fonction **parseCell**, qui appelle selon le contenu du *stream*, soit **parseSexp**, soit **parseAtom**.

ParseAtom analyse le contenu du *stream*, et suivant le type de l'atome courant, crée soit un **RealAtom**, soit un **StringAtom**, ou soit un **SymbolAtom**.

ParseSexp fait elle appel à **parseCell**, qui à nouveau, suivant le contenu du *stream*, fait appel à **parseSexp** ou **parseAtom**.

Le code résultant est assez simple. Le voici :

```
std::shared_ptr<Cell> parseCell(std::istream& is, Cell::CellEnv& env) {
  char ch;
  is >> ch;

  std::shared_ptr<Sexp> quote;
  std::shared_ptr<Cell> cell;

  if (ch == '\'' || ch == '`' || ch == ',') {
    quote = Sexp::New();
    std::string atomName = ch == '\'' ? "quote" : (ch == '`' ?
"backquote" : "comma");
    std::shared_ptr<Atom> atom = SymbolAtom::New(env, atomName);

    atom->computeVal(atomName);
    quote->cells.push_back(atom);
    is >> ch;
  }

  if (ch == '(') {
    cell = parseSexp(is, env);
  } else {
    is.putback(ch);
    cell = parseAtom(is, env);
  }

  if (quote) {
    quote->cells.push_back(cell);
    cell = quote;
  }

  return cell;
}
```

Il n'y a pas grand-chose à dire sur cette première méthode, hormis peut-être une petite explication sur l'appel à **computeVal** sur l'atome. Il s'agit simplement de calculer la valeur de notre atome. Nous effectuons donc une part de l'analyse syntaxique lors de cette phase d'analyse lexicale.

```
std::shared_ptr<Cell> parseAtom(std::istream& is, Cell::CellEnv& env) {
  std::shared_ptr<Cell> at;
  char ch;
  is >> ch;
  is.putback(ch);
  if (ch == '"') {
    at = parseString(is, env);
  } else {
    std::string atom;
    is >> atom;
    Atom::Type type = Atom::computeType(atom);

    if(type == Atom::Symbol)
      at = SymbolAtom::New(env, atom);
    if(type == Atom::Real)
      at = RealAtom::New();
    std::dynamic_pointer_cast<Atom>(at)->computeVal(atom);
  }
  return at;
}
```

Nous retrouvons dans cette seconde fonction la méthode **computeVal**, qui associe encore sa valeur à l'atome, s'il en a une.

```
std::shared_ptr<Cell> parseSexp(std::istream& is, Cell::CellEnv& env) {
  char ch;
  std::shared_ptr<Sexp> sx = Sexp::New();
  while(is >> ch) {
    if (ch == ')') {
      return sx;
    } else {
      is.putback(ch);
      std::shared_ptr<Cell> cell = parseCell(is, env);
      sx->cells.push_back(cell);
    }
  }
}
```

Le *parsing* des **Sexp** est très simple. Si on est dans cette fonction, c'est qu'une parenthèse ouvrante a été détectée dans **parseCell**. Tant que nous ne trouvons pas de parenthèse fermante, nous appelons récursivement **parseCell**. Une fois que nous la rencontrons, le *parsing* de la **Sexp** est terminé, et nous la retournons.

```
std::shared_ptr<Cell> parseString(std::istream& is, Cell::CellEnv& env) {
  char ch;
  std::string atom;
  std::stringstream ss;
  is >> ch;
  //assert(ch == '"')
  while(is >> std::noskipws >> ch) {
    if(ch == '"')
      break;
    ss << ch;
  }
  is >> std::skipws;
  atom = ss.str();
  Atom::Type type = Atom::computeType(atom);
  //assert(type == Atom::String);
  std::shared_ptr<Cell> at = StringAtom::New();
  std::dynamic_pointer_cast<Atom>(at)->computeVal(atom);
  return at;
}
```

On arrive ainsi en moins de cent lignes, à écrire un *parseur*. Le code ne présente pas de difficultés particulières. On peut juste noter le traitement des caractères spéciaux *comma*, *quote*, et *backquote*, qui sont des raccourcis syntaxiques utilisés dans les macros.

Il nous reste à tester le tout :

```
#include "parse.h"
#include <sstream>
#include <iostream>

int main(int argc, char* argv[])
{
  if(argc != 1) {
    std::cout << "Usage : " << argv[0] << std::endl;
    return 0;
  }

  std::stringstream cl("(+ (* 3 4) 30)");
  Cell::CellEnv env;
  std::vector<std::shared_ptr<Cell> > cells = parse(cl, env);
  std::shared_ptr<Sexp> sexp = std::dynamic_pointer_cast<Sexp>(cells[0]);
  if(sexp) {
    std::cout << "Hourrah" << std::endl;
    std::cout << *sexp << std::endl;
  } else {
    std::cout << "we failed :(" << std::endl;
  }
  return 1;
}
```

Depuis le code du projet, vous pouvez compiler et tester simplement ce code :

```
$ make testParser
$ ./testParser "(+ 1 2)"
Hourrah
(+ 1 3)
```

Pour aller plus loin

Il existe de remarquables ouvrages sur le Lisp, qui brillent non seulement parce qu'ils traitent du Lisp, mais aussi parce qu'ils exposent des concepts de programmation réutilisables dans d'autres langages. On peut citer notamment :

- « *Structure and Interpretation of Computer Programs (SICP)* » de Hal Abelson, Jerry Sussman et Julie Sussman ;
- « *Lisp In Small Pieces* » de Christian Queinnec ;
- « *On Lisp* » de Paul Graham ;
- « *Practical Common Lisp* » de Peter Seibel.

La plupart ont le bon goût d'être accessibles en ligne librement, y compris au format ePub ou PDF.

Il existe aussi quelques implémentations de Lisp minimaux, comme celui qu'on se propose de développer :

- PicoLisp : http://picolisp.com/
- FemtoLisp : https://github.com/JeffBezanson/femtolisp.

Conclusion

Ce premier article nous a permis de redécouvrir le Lisp, et d'écrire en quelques lignes un *parseur* qui nous permette de représenter notre langage sous forme d'un arbre syntaxique abstrait, lui-même codé à l'aide de listes. Nous verrons dans notre prochain article comment faire parler ce langage, c'est-à-dire comment le faire s'exécuter. ■

Note

Le code de cet article a été développé sous Ubuntu, et est disponible sur GitHub : https://github.com/kayhman/llisp.

Pour en savoir plus...

Vous vous sentez un peu perdu en C++ ? *GNU/Linux Magazine Hors-série n°83* fournit un guide complet pour débuter en C++.

On y traite de tous les principes clés du langage en se basant sur un projet qui servira de fil rouge : la conception d'un casse-briques. Un numéro à découvrir en kiosque et sur www.ed-diamond.com/.

OUTILLEZ VOS PROCESSUS DE DÉVELOPPEMENT AVEC APHRODITE

Romain Pelisse [Sustain Developer @Red Hat]

Relecture Aurélie Garnier

La communauté de projet open source JBoss [1] s'est construite autour de son serveur d'application, Wildfly [2], mais a aussi été le berceau de librairies et de frameworks très utiles aux développements logiciels, comme par exemple Hibernate [3]. Pour le besoin de tous ces projets, mais aussi pour assurer un développement rapide, efficace et sûr, la communauté est aussi à l'origine de nombreux frameworks et outils de développement, malheureusement moins connus. Certains de ces outils seront peut-être tout aussi utiles aux lecteurs qu'aux développeurs JBoss. Je vous présente aujourd'hui Aphrodite [4], une couche d'abstraction développée par mon équipe, la JBoss Sustain Engineering Team, qui va vous permettre de facilement développer vos propres outils pour contrôler et outiller vos propres processus de développement.

Mots-clés : Java/JEE, Maven, Junit, Tests, CDI, JIRA, Bugzilla, GitHub

Résumé

Aphrodite est un *framework* open source qui offre une API abstraite et unifiée pour accéder aux informations exposées par JIRA [5] et Bugzilla [6], mais aussi aux métadonnées liées aux dépôts de code source sur GitHub. À l'aide de celle-ci, on peut aisément automatiser de nombreuses tâches courantes du développeur, mais aussi outiller les processus mis en place par son organisation.

1 Outils et processus de développement

Presque tous les projets informatiques utilisent le même genre d'outil (un dépôt pour le code source, tel que Git ou SVN, un ou des gestionnaires de tâches ou de « bugs » tels que JIRA ou Bugzilla). À l'aide de ces outils, chaque équipe ou compagnie construit généralement son propre processus de développement, à l'aide de certaines fonctionnalités proposées par ces derniers.

Par exemple, une équipe peut utiliser les *pull requests* sur GitHub comme seule et unique manière d'intégrer des changements à une base de code, ou tout changement peut être associé à une entrée au sein d'un gestionnaire de tâches. Dans le dernier cas, chaque équipe définit généralement son propre *workflow* changeant l'état de la tâche selon l'avancement du travail, suivant une manière qui est propre à ses besoins (et donc assez spécifique).

Dans les grandes lignes, tout le monde fait pareil, mais, comme le dit très bien l'expression « le diable se cache dans les détails », en étudiant les processus des différentes équipes, on se rend rapidement compte que chacune a sa propre manière de travailler, et donc aussi d'utiliser ces outils.

Certaines équipes complètent l'état (« nouveau », « assigné »...) de drapeaux (*flags*) supplémentaires, d'autres n'utilisent pas certains champs ou fonctionnalités du logiciel de suivi de tâches, ou assignent les tâches à une équipe plutôt qu'à un individu. Les exemples de variations de ce genre sont nombreux. Ce qu'il faut juste retenir ici c'est que chaque équipe développe en fin de compte son propre processus de développement à l'aide des fonctionnalités proposées par les outils de gestion de suivi et les dépôts de révision de code.

Voilà, avec un peu de contexte en tête, nous allons voir, dans la section suivante, comment ce genre de processus se traduit, dans la pratique, en des tâches facilement automatisables à l'aide d'Aphrodite.

Pourquoi « Aphrodite » ?

Sans raison particulière, si ce n'est que les autres outils internes, développés par notre équipe, portent le nom de divinités de l'antiquité (Zeus) ou y sont reliés (un autre projet se nomme Mjolnir, d'après le nom du marteau de Thor).

2 Cas pratique : assignation d'une tâche

2.1 Contexte

Passons immédiatement à un cas pratique très simple et concret : l'assignation d'une tâche. Très souvent dans le développement, mais aussi dans la conduite d'un projet, on associe une tâche à effectuer à un développeur de l'équipe. Encore une fois, ce que cette attribution implique varie selon les équipes, donc nous allons juste nous appuyer sur le processus de mon équipe pour illustrer ceci.

Dans notre cas, lorsque l'un de nous s'assigne à une tâche sur laquelle il souhaite travailler, il doit effectuer les actions suivantes sur Bugzilla – car, oui, nous utilisons Bugzilla et non JIRA ;) – enfin du moins pas encore :

1. assigner la tâche à son adresse mail Red Hat (ex : rpelisse@redhat.com) ;
2. passer l'état de la tâche de nouveau (`NEW`) à assigné (`ASSIGNED`) ;
3. ajouter une estimation de temps ;
4. ajouter une série de drapeaux (*flags*) demandant la validation de la tâche par au moins un autre développeur, un responsable des tests et de la qualité (« QA ») et enfin un chargé de produits ;
5. et pour finir, j'ajoute généralement un commentaire pour indiquer quand je pense pouvoir travailler sur la tâche.

Avec cinq opérations distinctes pour effectuer la tâche, qui ne peuvent pas être concurrentes en passant par l'interface web de Bugzilla, cette attribution est non seulement une tâche un peu barbante, mais surtout prompte à l'erreur. Il m'arrive souvent, comme à mes collègues, d'oublier un élément (l'estimation, ou l'ajout des drapeaux, ou simplement de laisser la tâche à l'état « nouveau »).

Nous allons donc procéder à la réalisation d'un simple script, qui, à l'aide de l'API offerte par Aphrodite nous permettra d'implémenter ce processus non seulement facilement, mais de manière complètement transparente. Ainsi, si pour le moment il est développé pour fonctionner avec Bugzilla, il pourra aussi servir pour JIRA !

2.2 Script Scala

Pour concevoir notre script, nous allons utiliser le langage **Scala**, qui va nous permettre non seulement d'écrire un seul fichier (et non des classes Java à compiler), mais aussi de disposer d'une syntaxe beaucoup moins verbeuse et donc de ne pas noircir inutilement les pages de cet article. Un peu d'économie n'a jamais fait de mal :)

La première chose à faire est bien sûr de télécharger le JAR, l'archive binaire associée à l'API d'Aphrodite, et de le placer dans le chemin d'exécution de Scala lorsque nous exécuterons ce script. En plus des APIs offertes par Aphrodite, nous allons aussi utiliser **Jcommander [7]**, qui va nous simplifier grandement l'interprétation des arguments. Nous

devons donc ajouter aussi ce dernier à la liste des éléments du **CLASSPATH** :

```
# scala -classpath aphrodite.jar:jcommander.jar assign-task.scala ${@}
```

Note

Si vous utilisez **Maven** pour gérer vos dépendances, il est probablement plus élégant d'ajouter un fichier **pom.xml** vers ces deux dépendances et de modifier le script d'exécution pour utiliser les jars situés dans votre dépôt local.

Pour ne pas avoir à saisir cette ligne de commandes à chaque fois, plaçons cette dernière dans un fichier nommé **run-scala.sh** que nous invoquerons par la suite, à chaque fois que l'on souhaitera exécuter notre script.

2.3 Gestion des arguments

Regardons rapidement la mise en place de la gestion des arguments d'entrée du script, à l'aide de JCommander :

```
import com.beust.jcommander.JCommander
import com.beust.jcommander.Parameter
import com.beust.jcommander.ParameterException

object Args {

    @Parameter(names = Array( "-u", "--username" ), description =
"Bugzilla username", required = true)
    var username = ""

    @Parameter(names = Array( "-p", "--password" ), description =
"BugZilla password", required = true)
    var password = ""

    @Parameter(names = Array( "-i", "--bug-id" ), description =
"bug id", required = true)
    var bugId = ""

    @Parameter(names = Array( "-e", "--estimate" ), description =
"estimate", required = false)
    var estimate = 24.0

    @Parameter(names = Array("-a", "--assigned-to" ), description
= "", required = true)
    var assignedTo = ""

    @Parameter(names = Array("-c", "--comment" ), description =
"", required = true)
    var comment = "I'll take a look at this issue asap."
}

new JCommander(Args, args.toArray: _*)
```

Le code est vraiment simple et se passe presque de commentaires : en quelques lignes, on déclare un nouvel objet, nommé **Args**, unique en mémoire (singleton). Il possède un attribut par argument d'entrée de notre script. Pour chaque attribut, on trouve une annotation, décrivant l'option associée à ce dernier, ainsi qu'un champ indiquant que le paramètre est requis ou (*required*). S'il est requis et non utilisé lors de l'invocation du script, JCommander lèvera une exception pour signaler le problème.

On passe ensuite le tableau d'arguments d'entrée de notre script au *framework* JCommander qui va se charger de vérifier que les arguments sont présents et valides et les associer aux attributs de l'objet. Voilà, comment on en a fini avec la gestion des arguments ! Pratique, n'est-ce pas ? :)

2.4 Initialisation d'Aphrodite

Une fois ce travail de mise en place effectué, nous allons (enfin) pouvoir commencer à utiliser Aphrodite en tant que telle. La première chose à faire est d'initialiser le *framework* à l'aide de la configuration nécessaire. Dans notre cas, nous allons juste utiliser le *back end* Bugzilla, donc le code se réduit à :

```
def getServerUrl(s: String) = {
  s.substring(0,s.indexOf('/', "https://".length))
}

val tracker = getServerUrl(Args.bugId)
val issueTrackerConfigs: List[IssueTrackerConfig] = new
ArrayList[IssueTrackerConfig];
issueTrackerConfigs.add(
  new IssueTrackerConfig(tracker, Args.username, Args.password,
TrackerType.BUGZILLA, 1))
val aphrodite = Aphrodite.instance(AphroditeConfig.issueTrackersOnly(is
sueTrackerConfigs))
println("Aphrodite configured - retrieving data from server:" + tracker)
```

L'extrait est encore une fois relativement simple, si ce n'est la définition des types associés aux listes qui le rend un peu plus cryptique que nécessaire. Petite astuce à noter, la fonction **getServerUrl**, qui permet d'extraire l'URL du serveur (soit http://bugzilla.redhat.com/) à partir de l'identifiant de l'issue fournie en paramètre (soit de la forme bugzilla.redhat.com/show_bug.cgi?id=99999999). Ceci permet de ne pas avoir à passer inutilement un paramètre supplémentaire au script.

2.5 Récupérer la tâche associée

La prochaine étape consiste donc à récupérer les données associées à la tâche que nous souhaitons assigner, ce qui ne requiert qu'un seul appel à l'API d'Aphrodite :

```
val issue = aphrodite.getIssue(new java.net.URL(Args.bugId))
println("Retrieved Issue:" + issue.getSummary.get())
```

À ce stade, nous pouvons déjà exécuter notre script pour vérifier que tout fonctionne sans problème :

```
$ ./run-scala.sh assign-bz.scala -u 'rpelisse@redhat.com' -p '*******' \
            -i 'https://bugzilla.redhat.com/show_bug.cgi?id=1268185'
-e 3 \
            -a rpelisse@redhat.com -c "Hello"
Aphrodite configured - retrieving data from server:https://bugzilla.
redhat.com/
Retrieved Issue:[GSS](6.4.z) Custom socket factory for JGroups subsystem
not set correctly
```

Comment débugger ?

Une petite remarque ici pour signaler que même si nous développons le script Scala en dehors de tout IDE, il est bien évidemment possible d'utiliser un *debugger* pour exécuter le code au pas à pas, et vérifier l'état des variables. Il suffit pour ceci d'effectuer un débogage distant en passant les paramètres suivants à la machine virtuelle Java de Scala :

```
$ export JAVA_OPTS=-agentlib:jdwp=transport=dt_socket,
server=y,address=8888,suspend=y
```

Comme la plupart des programmes Java, Scala utilise la variable d'environnement **JAVA_OPTS** pour configurer la JVM qui l'exécute. Ces paramètres permettent à un IDE (ou juste un *debugger*) de se connecter sur un port TCP (ici **8888**) pour effectuer une session de *debug* à distance.

On notera que le paramètre **suspend=y** implique que le script s'arrêtera au démarrage, pour vous laisser le temps de vous connecter au programme et définir vos points d'arrêts. Une fois la session de debug finie, vous pouvez désactiver ce mode d'exécution en supprimant la variable d'environnement :

```
$ unset JAVA_OPTS
```

2.6 Assigner la tâche et la mettre à jour

Une fois la tâche récupérée, on peut aisément mettre à jour les différents champs avant de la mettre à jour sur le serveur, toujours à l'aide des APIs d'Aphrodite :

```
issue.setAssignee(Args.assignedTo)
issue.setStatus(IssueStatus.ASSIGNED)
issue.setEstimation(new IssueEstimation(Args.estimate))
val stage = new Stage()
stage.getStateMap().put(Flag.DEV,FlagStatus.SET)
stage.getStateMap().put(Flag.PM,FlagStatus.SET)
stage.getStateMap().put(Flag.QE,FlagStatus.SET)
issue.setStage(stage)
aphrodite.updateIssue(issue)
```

Encore une fois, le code est très simple et transparent. À noter aussi qu'il n'y aurait aucun changement à effectuer sur cette partie du code si la tâche était enregistrée sur une instance de JIRA plutôt que Bugzilla. Aphrodite permet de limiter le code à l'expression métier, soit la définition d'une action attachée à notre processus de développement, sans se soucier des problématiques de bas niveau (connexion au tracker, récupération des données et transfert vers un modèle objet Java, etc.).

Avant d'aller plus loin, n'oublions pas aussi d'ajouter un commentaire à la tâche – comme je le fais habituellement pour indiquer quand je pense m'attaquer à cette dernière :

```
val comment = new Comment(Args.comment, false)
aphrodite.postCommentOnIssue(issue,comment)
println("Task " + issue.getTrackerId.get + " has been assigned to
" + issue.getAssignee.get)
```

3 Cas pratique : affichage de l'état de la prochaine livraison mineure

3.1 Contexte

Passons maintenant à un second cas pratique : la gestion des correctifs et améliorations à intégrer à une nouvelle livraison du projet. En effet, il est courant, pour définir le contenu d'une nouvelle livraison, d'utiliser aussi le gestionnaire de suivi de tâches, et de simplement créer une tâche, nommée comme la livraison à effectuer (par exemple, « mon logiciel 4.3.2 ») et simplement d'ajouter toutes les tâches associées comme dépendances.

C'est fort pratique, mais il n'est pas très simple, tout du moins à l'aide de l'interface graphique de Bugzilla, d'avoir une idée claire de l'état de la livraison, sans consulter tous les correctifs un par un. Le principal problème ici est que Bugzilla n'affiche pas les drapeaux supplémentaires définis et utilisés par le processus de développement.

Donc, par exemple, pour un développeur, il n'est pas simple de savoir laquelle parmi ses tâches n'a pas reçu le vote justement

d'un développeur (dans notre cas, le drapeau **DEV**, associé à l'état **ACCEPTED**). Nous allons maintenant réaliser un second petit script, récupérant ces informations et affichant uniquement les tâches associées à une tâche de livraison logicielle, et n'ayant pas le drapeau **DEV** associé à l'état **ACCEPTED**.

3.2 Récupérer les dépendances

Le début du script est pratiquement le même, avec l'initialisation d'Aphrodite, donc on saute directement à la partie intéressante :

```
def printIssueIfDevAckMissing(issue: Issue) = {
  print(issue.getTrackerId.get + ":" + issue.getSummary.get)
  if ( issue.getStage.getStatus(Flag.DEV) != FlagStatus.ACCEPTED )
    println(" - is ACCEPTED")
   else
    println(" - is NOT ACCEPTED:" + issue.getStage.getStatus(Flag.DEV).
toString)
}
...

val issue = aphrodite.getIssue(new java.net.URL(Args.bugId))
println("Retrieved Issue:" + issue.getSummary.get())
issue.getDependsOn.foreach { url => printIssueIfDevAckMissing(aphrodite.
getIssue(url)) }
```

En quelques lignes, on retrouve l'ensemble des tâches qui dépendent de notre tâche de livraison, et on affiche l'identifiant et le résumé de chacune d'elle qui ne dispose pas d'un drapeau **DEV** associé à l'état **ACCEPTED** :

```
...
Aphrodite configured - retrieving data from server:https://bugzilla.redhat.
com/
Retrieved Issue:EAP 6.4.6 (CP06) Payload Tracker
1204300:[GSS](6.4.z) Security ModuleClassLoaderLocator.CombinedClassLoader
does not implement getResources - is NOT ACCEPTED:REJECTED
1241799:[PST] (EAP 6.4.z) CVE-2015-0254 web: jakarta-taglibs-standard: XXE
and RCE via XSL extension in JSTL XML tags - is NOT ACCEPTED: NOT_SET
...
```

Note

Problème de performance

On notera bien évidemment que cette approche n'est pas très performante. Le code est simple et très lisible, mais le script est amené à faire un appel au service *back end*, en l'occurrence Bugzilla, par dépendance. Ce n'est pas très efficace ni économe en ressource. Les prochaines versions d'Aphrodite proposeront des méthodes pour récupérer directement un ensemble de tâches – mais pour le moment, et aussi pour des raisons didactiques, nous garderons cette approche sommaire.

Conclusion

Pour rester simple et didactique, cet article ne s'est concentré que sur quelques cas pratiques, volontairement limités à l'utilisation de Bugzilla, mais il devrait être évident à la lecture de cet article que les possibilités d'automatisation et d'outillage du processus de développement à l'aide d'Aphrodite sont très larges. Si nous n'avons, ici, qu'abordé l'intégration avec les gestionnaires de tâches, il ne faut pas non plus oublier qu'Aphrodite permet aussi de récupérer l'ensemble des métadonnées mises à disposition par un gestionnaire de sources tel que GitHub.

En outre, si les exemples ci-dessus se sont concentrés sur des problématiques de simple utilisateur, il est évident qu'intégrer des scripts construits autour d'Aphrodite à, par exemple, des environnements d'intégration continue – comme **Jenkins CI [8]**, ouvre un univers de possibles très riche pour l'outillage de ses processus de développement.

Un exemple des possibilités qu'apporte Aphrodite est le projet **BugClerk [9]** – déjà évoqué dans un précédent article, qui permet de vérifier que le descriptif de la tâche à effectuer soit toujours bien renseigné et en cohérence avec les règles des processus de développement définis. La démonstration est faite, espérons-le ; la communauté attend donc avec impatience que vous ajoutiez vos scripts **[10]**, et autres extensions, à ceux déjà présentés dans cet article... ■

Références

[1] Communauté « open source » JBoss : http://www.jboss.org/

[2] Serveur d'applications JEE « Open Source » Wildfly (anciennement « JBoss AS ») : http://wildfly.org//

[3] Hibernate ORM : http://hibernate.org/

[4] Aphrodite : https://github.com/jboss-set/aphrodite/

[5] JIRA : https://www.atlassian.com/software/jira

[6] Bugzilla : https://www.bugzilla.org/

[7] JCommander : http://jcommander.org/

[8] Jenkins CI : https://jenkins-ci.org/

[9] BugClerk : https://github.com/jboss-set/bug-clerk/

[10] In Bed With Aphrodite : https://github.com/rpelisse/in-bed-with-aphrodite/

Note

GitHub : https://github.com/jboss-set/aphrodite

INTÉGREZ JAVASCRIPT DANS PHP

Stéphane Mourey [Seeraiwer]

L'extension V8Js permet d'utiliser JavaScript depuis PHP. Elle a le statut de curiosité, ce qui explique que, bien qu'elle ne soit pas récente, elle soit toujours en bêta. Pourtant elle peut rendre quelques menus services, et, qui sait, ouvrir de nouvelles perspectives.

Mots-clés : JavaScript, PHP, V8JS, extension, validation, formulaire

Résumé

Nous allons voir comment installer et utiliser l'extension V8Js dans PHP, ainsi qu'un exemple simple pour effectuer une validation de champs de formulaire utilisant le même code côté client et côté serveur.

Lorsque j'ai découvert le module **V8Js** pour **PHP**, j'étais dubitatif : je ne voyais pas l'intérêt d'un tel projet. Comme beaucoup de développeurs PHP, j'étais assez hostile à **JavaScript**, le considérant comme un mal nécessaire, indispensable pour certaines tâches, mais rendant ses services de façon si laborieuse que je n'y avais recours qu'en cas d'absolue nécessité. Depuis, les choses avaient évolué, les *frameworks* JavaScript ont rendu les choses plus supportables et **JQuery** fut un émerveillement. Pour autant, l'utilisation de JavaScript dans un environnement serveur ne me paraissait pas intéressante : j'avais PHP pour cela, alors pourquoi passer à **Node.js** ? L'argument selon lequel cela permettait de n'avoir qu'un seul langage à apprendre pour programmer tant le client que le serveur ne me touchait pas : je connaissais déjà PHP, je savais me débrouiller avec JavaScript et l'une des choses que j'apprécie toujours avec PHP est qu'il m'oblige à avoir une ouverture grandissante vers d'autres langages.

Depuis, les choses ont bien évolué encore. JavaScript a connu de tels développements qu'on en arrive à imaginer des applications autonomes et communicantes fonctionnant dans le seul navigateur, mais pouvant également être hébergées par un serveur, ou même fonctionner en ligne de commandes. Cette capacité qu'a JavaScript de brouiller les frontières est troublante, mais doit être perçue comme une invitation à créer de nouvelles architectures et explorer de nouvelles possibilités.

Dans cette perspective, l'idée d'intégrer JavaScript dans PHP paraît intéressante, sans que pour autant des cas d'utilisation pertinents viennent immédiatement à l'esprit. J'ai fini par en concevoir deux.

Le premier consiste à réutiliser le code client de vérification de formulaire web depuis PHP. Une des tâches laborieuses pour le développeur web est de programmer une double vérification des saisies des utilisateurs. En effet, la bonne pratique consiste à effectuer une vérification en JavaScript du côté du client avant la soumission des données, puis une vérification soignée du côté serveur. Cette double vérification a pour but d'éviter à l'utilisateur d'attendre la réponse du serveur pour découvrir, par exemple, qu'il a oublié l'arobase dans son adresse e-mail. Le plus souvent, la vérification du côté client est plus simple : le client ne dispose pas forcément de toutes les informations nécessaires à toutes les vérifications, certaines nécessitant de

consulter la base de données. Par ailleurs, cette vérification peut aisément être contournée par un utilisateur malintentionné – quand il n'a pas tout simplement désactivé JavaScript. La vérification côté serveur se doit d'être absolue, car il s'agit de la dernière étape avant la mise en œuvre des données. Du coup, le développeur peut être tenté de se contenter d'une vérification JavaScript trop sommaire, voire de « l'oublier ». Une solution serait d'avoir recours au module V8Js de PHP : ainsi, le code utilisé pour la vérification côté client peut être réutilisé côté serveur. Le temps de développement en sera raccourci, et la maintenance du code à l'avenir en sera allégée.

Le deuxième cas que j'ai imaginé, que nous n'étudierons pas ici, mais qui montre la pertinence de la démarche, concerne l'utilisation de bibliothèques de fonctions partagées entre le serveur et le client ou pourquoi pas, un serveur PHP et un autre Node.Js. La pensée m'en est venue en travaillant sur la cryptographie en JavaScript. Il existe plusieurs bibliothèques permettant de faire de la cryptographie côté client qui reposent sur ce langage. Dans les discussions qui animent les communautés autour de ces bibliothèques revient régulièrement la question de comment les faire dialoguer avec les bibliothèques équivalentes, implémentées pour d'autres langages côté serveur, en particulier PHP. La possibilité d'utiliser JavaScript depuis PHP permettrait de résoudre ces problèmes à moindres frais.

1 Installation

L'extension V8Js est accessible depuis **PECL**. Il faut donc l'installer séparément. Mais avant cela, il convient d'installer ses dépendances à savoir la librairie du moteur V8, ainsi que ses en-têtes. Sous Ubuntu, vous pouvez réaliser cela à l'aide de la commande suivante :

```
# apt-get install libv8-3.14.5 libv8-3.14-dev libv8-3.14-dbg
```

Puis lancez l'installation de l'extension :

```
# pecl install v8js-0.1.3
```

Enfin, ajoutez la ligne suivante à votre **php.ini**, et redémarrez votre serveur web :

```
extension=v8js.so
```

Pour terminer, vous pouvez valider rapidement votre installation en ligne de commandes :

```
$ php -r 'var_dump(class_exists("V8Js"));'
bool(true)
```

Voilà, vous êtes prêt à vous lancer.

2 Utilisation

2.1 Hello world !

Commençons par prendre en main notre nouvelle extension en utilisant JavaScript pour afficher un texte. L'outil est démesuré par rapport à l'objectif, mais il s'agit d'abord de faire connaissance.

Pour exécuter du JavaScript depuis PHP, nous aurons toujours besoin d'un objet de classe **V8Js**. Cette classe propose la méthode **executeString** destinée justement à lancer l'exécution du code JavaScript. On obtient le code suivant :

```
$js = <<<EOJS
  print('Hello world !');
EOJS;
$v8 = new V8Js;
$v8->executeString($js);
```

Même si le code JavaScript ne comprend qu'une seule ligne ici, nous avons utilisé la syntaxe *HEREDOC*, qui permet de définir une chaîne de caractères sur une seule ligne : cela nous permet de mieux séparer les deux langages et de ne pas avoir à jongler avec les guillemets. JavaScript et PHP ayant une syntaxe très proche, ne pas marquer suffisamment la séparation rendrait le code rapidement illisible.

2.2 Recevoir une valeur en retour

La valeur de la dernière ligne de JavaScript est retournée par la méthode **executeString**. Il est ainsi possible de recevoir des résultats du traitement JavaScript en PHP :

Ainsi, le code suivant :

```
$js = <<<EOJS
  1+1;
EOJS;
$v8 = new V8Js;
var_dump($v8->executeString($js));
```

affichera :

```
int(2)
```

2.3 Envoyer une valeur depuis PHP

Il serait bien sûr possible d'injecter une valeur directement dans le code JavaScript par remplacement de variable ou par concaténation :

```
$myInt = 3;
$js = <<<EOJS
  $myInt+1;
EOJS;
$v8 = new V8Js;
var_dump($v8->executeString($js));
```

affichera :

```
int(4)
```

Mais vous avouerez que de devoir réécrire le code à chaque changement de valeur à traiter le rend nettement moins réutilisable. Heureusement, l'environnement JavaScript est conscient de certaines variables PHP.

Par défaut, l'environnement JavaScript reçoit un objet appelé **PHP**. Ce dernier porte les propriétés qui auront été affectées depuis PHP à l'objet **V8Js** porteur de l'environnement JavaScript. Prenons l'exemple du code suivant :

```
$v8 = new V8Js;
$v8->myInt = 3;
$js = <<<EOJS
  PHP.myInt+1;
EOJS;
var_dump($v8->executeString($js));
$v8->myInt = 5;
var_dump($v8->executeString($js));
```

Il affichera :

```
int(4)
int(6)
```

Notez que les opérateurs permettant d'accéder aux propriétés d'un objet ne sont pas les mêmes en PHP et en JavaScript. Là où en PHP on utilise la flèche **->**, en JavaScript, on utilisera le **.**.

Il est possible de changer le nom de la variable JavaScript qui porte les propriétés reçues depuis PHP. Ce nom est en effet le premier argument du constructeur de la classe V8Js.

```
$v8  = new V8Js('HHVM'); // Ici, l'objet porteur de variables
s'appellera HHVM en JavaScript
```

Le second argument est un tableau associatif contenant la liste des variables qui seront passées en propriétés à cet objet JavaScript. Ce tableau utilise comme clé le nom de la propriété à utiliser en JavaScript et comme valeur, le nom de la variable en PHP. Voyons ce que cela donne :

```
$myIntPHP = 10;
$js = <<<EOJS
  PHP.myIntJs+1;
EOJS;
$v8 = new V8Js('PHP', array('myIntJs'=>'myIntPHP'));
var_dump($v8->executeString($js));
```

Ce code affiche :

```
int(11)
```

Cette syntaxe est tout de même plus lourde que celle qui consiste à utiliser les propriétés de l'objet **V8Js** comme nous l'avons vu plus haut.

Le troisième argument, par défaut un tableau vide, contient la liste des extensions JavaScript à activer au moment de la création de l'objet. Nous y reviendrons lorsque nous traiterons des extensions.

Le quatrième argument concerne le comportement à adopter lorsque des exceptions sont levées depuis l'environnement JavaScript. Nous y reviendrons à la fin de cet article.

2.4 Passage d'objets et de tableaux ?

La différence de syntaxe pour adresser les propriétés d'objets n'est pas la seule entre PHP et JavaScript sur la façon dont les objets, comme d'ailleurs les tableaux, sont conçus. Si rien ne vous empêche *a priori* de passer objets et tableaux de PHP à JavaScript et inversement, vous pourriez le regretter *a posteriori*. En tout cas, avant de le faire en production, prenez un soin particulier à tester le comportement de votre code. En effet, les implémentations des concepts d'objet et de tableau sont si différentes dans les deux environnements que vous risquez non seulement de perdre des informations, mais également de voir la structure même de vos données grandement modifiée.

Prudence, donc. Si vous le pouvez, contentez-vous des types scalaires. Entre le transtypage automatique de PHP et celui de JavaScript, vous risquez déjà quelques surprises épineuses.

2.5 Utilisation d'extensions

Au point où nous en sommes, il est déjà possible d'utiliser des fragments de code JavaScript comme de véritables fonctions depuis PHP. Le code JavaScript produit en PHP est

Maker Faire® Paris

30 avril & 1er mai 2016

Paris expo porte de Versailles - Foire de Paris

BIENVENUE DANS LE FUTUR !

réutilisable depuis PHP, mais isolé dans le contexte d'objet **V8Js**. Mais ne serait-il pas mieux de le rendre réutilisable également depuis JavaScript ? Sans partager de variables, nous disposons pour cela des extensions JavaScript, persistantes, globales et indépendantes du contexte ; elles permettent, par exemple, de partager une bibliothèque de fonctions entre différents objets **V8Js**.

Il y a une précaution méthodologique à prendre lorsqu'on utilise de telles extensions : celles-ci doivent être enregistrées à l'aide de la méthode statique **V8Js::registerExtension**, et cet enregistrement doit impérativement avoir lieu avant toute instanciation d'un objet **V8Js**. En effet, pour des raisons de performance, l'environnement **V8** est initialisé une seule fois, lors de la création du premier objet **V8Js**. C'est lors de cette initialisation que l'ajout de ces extensions est effectif, et il le sera pour tous les objets **V8Js** à venir.

Examinons la méthode **V8Js::registerExtension**. Celle-ci attend au moins deux paramètres et en accepte jusqu'à quatre. Le premier est le nom sous lequel l'extension est enregistrée. Il convient qu'il soit unique dans votre environnement d'exécution PHP. Le second est une chaîne de caractères, le code JavaScript, l'extension proprement dite. Le troisième est un tableau contenant les noms des extensions JavaScript dont dépend l'extension en cours d'enregistrement. L'activation de l'extension en cours d'enregistrement déclenchera celle de toutes les extensions dont elle dépend. En effet, ce n'est pas parce qu'une extension JavaScript est enregistrée et donc disponible pour tous les objets **V8Js** qu'elle y est activée. Nous avons vu plus haut comment un objet **V8Js** active une extension lors de sa construction. La relation de dépendances définie lors de l'activation d'une extension permet d'activer ses dépendances en même temps qu'elle sans avoir à s'en préoccuper et ainsi de s'assurer de son bon fonctionnement.

Cela peut être utile naturellement pour partager des fonctions dans plusieurs contextes, mais cela vaut également pour tout le contexte créé dans l'extension. Pour mieux comprendre, étudions cet exemple de code :

```
01: $jsExt = <<<EOJSEXT
02:  var maVar ="Hello World !\\n";
03: EOJSEXT;
04: $js = <<< EOJS
05:  print(maVar);
06: EOJS;
07: V8Js::registerExtension('jsExt',$jsExt);
08: $v8 = new V8Js();
09: $v8->executeString($js);
10:
11: $v8 = new V8Js('PHP',array(),array('jsExt'));
12: $v8->executeString($js);
```

Lors de son exécution, il affichera :

```
PHP Warning:  Uncaught exception 'V8JsException' with message
'V8Js::executeString():1: ReferenceError: maVar is not defined'
in php shell code:1
Stack trace:
#0 php shell code(1): V8Js->executeString('print(maVar);')
#1 {main}
  thrown in php shell code on line 1
Hello World !
```

Le script commence, des lignes 1 à 6, par définir deux variables **$jsExt** et **$js**, la première étant destinée à recevoir une extension, la seconde une liste d'instructions JavaScript à exécuter. J'ai écrit **$js** pour que son code ne puisse s'exécuter correctement si le code de **$jsExt** ne l'avait pas d'abord été, et dans le même contexte. Ligne 7, j'enregistre l'extension. Lignes 8 et 9, je tente l'exécution de **$js**, mais sans activer l'extension. C'est la cause des six premières lignes de la sortie, un *PHP Warning* détaillé qui fait remonter l'erreur JavaScript à sa source. Ligne 11 à 12 du script, je fais une nouvelle tentative, mais en activant l'extension enregistrée. Là, tout se passe comme attendu, mon « Hello World ! » s'affiche sur la dernière ligne de la sortie.

Si vous comptez utiliser massivement V8Js, vous comprendrez vite l'intérêt de n'activer que les extensions nécessaires, et cette gestion des dépendances vous facilite grandement le travail. À l'inverse, si vous n'utilisez V8Js que très ponctuellement, la gestion d'une telle architecture vous paraîtra lourde et inutile. C'est là qu'intervient le dernier paramètre de **V8Js::registerExtension()**. Appelé **$auto_enable**, il s'agit d'un booléen faux par défaut. Si vous le passez à vrai, alors votre extension sera automatiquement activée dans tous les contextes JavaScript.

Reprenons notre script précédent en modifiant la fin :

```
07: V8Js::registerExtension('jsExt',$jsExt,array(),true);
08: $v8 = new V8Js();
09: $v8->executeString($js);
```

Cette fois-ci, pas d'avertissement, le code s'exécute correctement, et « Hello World ! » s'affiche comme attendu.

Ainsi, ce système de gestion de dépendances est en mesure de satisfaire à la fois ceux qui sont à la recherche de performances et ont besoin d'une utilisation optimale de la mémoire, et ceux qui sont moins pointilleux et qui recherchent avant tout la rapidité de développement.

Petite astuce : utiliser **V8Js::registerExtension()** conjointement à **file_get_contents()** vous permettra de charger facilement dans votre projet toutes ces merveilleuses bibliothèques JavaScript qui vous tendent les bras.

2.6 Traitement des erreurs

Une erreur produite dans votre code JavaScript provoquera en PHP la levée d'une exception de classe **V8JsException**. À vous de l'intercepter et de l'analyser. Cette classe vous propose un certain nombre de méthodes pour vous y aider, permettant en principe l'analyse du contexte dans lequel cette erreur s'est produite. Malheureusement, ces méthodes ne sont pas documentées. Aussi, à moins de vous plonger dans le code source de l'extension, ou encore de les tester jusqu'à en comprendre le fonctionnement, elles ne vous seront pas de grande utilité. Toutefois, le message associé à l'exception, comme nous avons pu le voir en section 2.5 est assez explicite pour permettre un débogage précis de votre code JavaScript.

2.7 Et l'accès au DOM ?

Nous avons évoqué au début de cet article l'idée de partager le même code JavaScript entre le navigateur et le serveur. Certains auront sans doute eu l'idée d'utiliser certains scripts qui leur rendent bien des services sur leurs pages web en les manipulant pour réaliser tels ou tels effets de mise en forme. De cette façon, ils espèrent que l'exécution de leurs scripts ne reposerait plus sur le bon vouloir de certains navigateurs trop vieux ou trop divergents.

Pour l'essentiel, ils se trompent. En effet, PHP n'a pas d'accès direct au *Document Object Model*, le fameux DOM, qui est la représentation interne que se fait un navigateur de votre modèle, et il ne peut donc le passer à JavaScript. PHP manipule des chaînes de caractères pour produire le document HTML. Il existe bien une extension qui permet de manipuler *un* DOM **[1]**, mais celle-ci travaillera uniquement à partir d'une chaîne de caractères que vous lui aurez fournie. Pour pouvoir travailler sur l'intégralité du document que vous souhaitez communiquer au navigateur, il faudra alors utiliser les fonctions de *bufferisation* de sortie **[2]** comme **ob_start()**, **ob_get_contents()** et **ob_clean()** pour pouvoir retenir votre document jusqu'à ce qu'il soit finalisé, en récupérer le contenu, analyser son DOM et le modifier avant d'en envoyer la nouvelle version. La démarche est déjà bien laborieuse en considérant seulement PHP. Pour utiliser V8Js, vous pouvez vous passer de l'analyse du DOM par PHP, mais pas du mécanisme de rétention et de libération que je viens de décrire. Il vous faudra alors passer tout votre document comme une chaîne de caractères à votre objet **V8Js** et l'utiliser pour reconstruire votre DOM en JavaScript, avant de pouvoir le manipuler. Et cette tâche n'est pas triviale, le DOM étant normalement fourni par le navigateur à JavaScript.

L'enjeu en vaut-il la chandelle ? Pour ma part, je crois que non. Mais ne soyez pas déçu pour autant, l'extension V8Js peut rendre bien d'autres services.

3 Un cas pratique

Pour illustrer ce long exposé théorique, rien de tel qu'un exemple pratique. Celui-ci sera réduit au minimum nécessaire à cette fin. Nous aurons donc trois fichiers placés dans le même dossier :

- **form.html** qui contiendra un formulaire simple avec deux champs texte destinés à recevoir une adresse e-mail et sa confirmation ; ce formulaire devra valider les données saisies avant de les soumettre au serveur ;
- **process.php**, un script dont la tâche est de valider les données saisies ;
- **validLib.js**, un script contenant les fonctions de validation à utiliser par les deux précédents.

Commençons par **validLib.js** :

```
function validEmail(email) {
  return (email.indexOf('@')===-1)?false:true;
}

function validIdentical(email,confirm) {
  return email === confirm;
}
```

Deux fonctions, rien de plus. La première, **validEmail()**, valide une adresse e-mail par la présence du signe arobase **@**. C'est insuffisant pour cette tâche, mais cela suffit à notre propos. La seconde vérifie si les deux paramètres qu'elle reçoit sont identiques. Ce sont ces deux fonctions qu'il va nous falloir utiliser dans notre formulaire et dans notre script PHP.

Voyons comment procéder dans le formulaire :

```
01: <!doctype html>
02: <html>
03:   <head>
04:     <script src="validLib.js"></script>
05:     <script>
06:       function validForm(){
07:         var valid = true;
08:         var email = document.getElementById('email').value;
09:         var emailConfirm = document.
getElementById('emailConfirm').value;
10:         if (!validEmail(email)) {
11:           valid = false;
12:           alert("L'email indiqué n'est pas valide !");
13:         }
```

```
14:         if (!validIdentical(email,emailConfirm)) {
15:           valid = false;
16:           alert("L'email et sa confirmation ne sont pas
identiques !");
17:         }
18:         return valid;
19:     }
20:     </script>
21:   </head>
22:   <body>
23:     <form action="process.php" onSubmit="Javascript:return
validForm();">
24:       <input type="text" name="email" id="email"><br>
25:       <input type="text" name="emailConfirm"
id="emailConfirm"><br>
26:       <input type="submit">
27:     </form>
28:   </body>
29: </html>
```

Le traitement est très classique. On importe notre librairie JavaScript à la ligne 4 et on utilise ses fonctions aux lignes 10 et 14. L'essentiel du travail consiste à récupérer les valeurs à transmettre à ces deux fonctions et à afficher les messages d'erreur à l'utilisateur. Tout cela aurait pu être également traité dans un fichier externe pour être réutilisé, mais n'aurait pas de sens dans le contexte de PHP : il faudrait donc alors avoir recours à un autre fichier **js** pour bien séparer le code à importer dans les deux cas.

Voyons enfin ce qui se passe dans **process.php** :

```
01: <?php
02: V8Js::registerExtension('validLib',file_get_contents('validLib.
js'),array(),true);
03: $v8 = new V8Js;
04: $v8->email = $_GET['email'];
05: $v8->emailConfirm = $_GET['emailConfirm'];
06:
07: $jsValidEmail = <<<EOJS1
08:   validEmail(PHP.email);
09: EOJS1;
10:
11: $jsValidIdentical = <<<EOJS1
12:   validIdentical(PHP.email,PHP.emailConfirm);
13: EOJS1;
14:
15: if (!$v8->executeString($jsValidEmail)){
16:   print "L'email indiqué n'est pas valide !<br>";
17: }
18:
19: if (!$v8->executeString($jsValidIdentical)){
20:   print "L'email et sa confirmation ne sont pas identiques !<br>";
21: }
```

À la ligne 2, nous chargeons notre bibliothèque JavaScript comme une extension de V8Js de telle façon à ce que nous n'ayons pas besoin de l'activer. Ligne 4 et 5, nous récupérons les valeurs du formulaire en propriétés de notre nouvel objet **V8Js**. Lignes 7 à 13, nous préparons le supplément de JavaScript nécessaire pour utiliser nos fonctions de validation avec les données reçues. Puis, lignes 15 à 21, nous effectuons le traitement proprement dit en validant les données en JavaScript et en affichant les messages d'erreur s'il y a lieu.

Qu'avons-nous gagné à faire tout cela ? Certes, un double travail d'écriture est nécessaire, une fois dans le formulaire et une fois dans le traitement PHP, pour envoyer les valeurs saisies par l'utilisateur aux fonctions JavaScript de validation. Mais ce double travail est trivial et peut être systématisé facilement. La complexité réelle s'est déplacée dans la bibliothèque JavaScript dans laquelle peut alors se concentrer l'essentiel du développement, sans double travail.

Conclusion

L'extension V8Js pour PHP fait figure de curiosité. Pourtant, elle peut se révéler utile dans des situations où le traitement attendu par le serveur et le client sont identiques ou font appel aux mêmes méthodes. Une seule librairie peut ainsi servir dans deux cas, limitant ainsi la complexité et le risque de bogue. Il est regrettable que son utilisation soit freinée par le manque de support dans les environnements partagés ou encore en raison des préjugés des développeurs PHP à l'égard de JavaScript. Mais rien ne vous interdit de vous en servir dans vos propres projets. ■

Références

[1] Voir la documentation officielle de l'extension DOM : http://php.net/manual/fr/book.dom.php

[2] Fonctions de *bufferisation* de sortie : http://php.net/manual/fr/ref.outcontrol.php

Pour aller plus loin

Le code source de l'extension V8Js pour PHP 7 est disponible sur GitHub : https://github.com/phpv8/v8js. Vous y trouverez également une documentation un peu plus explicite que celle fournie sur le site officiel de PHP http://php.net/manual/fr/book.v8js.php.

Note

Code PHP 5 et JavaScript avec **libv8-3.14.5** testé sous Ubuntu.

LE CLOUD GAULOIS, UNE RÉALITÉ ! VENEZ TESTER SA PUISSANCE

EXPRESS HOSTING

Cloud Public
Serveur Virtuel
Serveur Dédié
Nom de domaine
Hébergement Web

sales@ikoula.com
01 84 01 02 66
express.ikoula.com

ENTERPRISE SERVICES

Cloud Privé
Infogérance
PRA/PCA
Haute disponibilité
Datacenter

sales-ies@ikoula.com
01 78 76 35 58
ies.ikoula.com

EX10

Cloud Hybride
Exchange
Lync
Sharepoint
Plateforme Collaborative

sales@ex10.biz
01 84 01 02 53
www.ex10.biz

Kévin et Céline, libristes depuis plus de 6 ans*

LINAGORA

REJOIGNEZ-NOUS

JOB.LINAGORA.COM

*campagne de publicité parodique proposée et réalisée par de vrais collaborateurs de Linagora.

DÉCOUVREZ NOS LOGICIELS LIBRES !

OBM.org	LinShare	LinID	LinPKi	petals ESB	OpenPaaS
Messagerie collaborative	Partage et transfert sécurisé de fichiers	Gestion et fédération des identités	PKI et signature électronique	ESB et orchestration	Plateforme collaborative

www.linagora.com

@Linagora