GNU LINUX MAGAZINE / FRANCE

DÉVELOPPEMENT SUR SYSTÈMES UNIX, OPEN SOURCE & EMBARQUÉ

N°207
SEPTEMBRE 2017

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

L 19275 - 207 - F: 7,90 € - RD

IA / Reconnaissance d'images

## DEEP LEARNING EN PRATIQUE : RECHERCHEZ DES OBJETS DANS UNE COLLECTION D'IMAGES ! p.20

- Compilez manuellement (et correctement) Python
- Faites vos premiers pas avec le framework de deep learning PyTorch
- Associez des mots-clés à des images
- Recherchez le contenu d'un lot d'images

Embarqué / STM32F410

## FREERTOS : APPLICATION À LA RÉALISATION D'UN ANALYSEUR DE RÉSEAU NUMÉRIQUE SUR STM32 p.44

Bidouille / SARPI

## INSTALLEZ UNE SLACKWARE SUR RASPBERRY PI 3 HEADLESS p.62

Architecture / Métier

## COMPRENEZ LE TRAVAIL MAL CONNU D'ARCHITECTE EN SYSTÈMES D'INFORMATION p.06

OAuth2 / Java

## SÉCURISEZ TOUTE VOTRE CHAÎNE DE PRODUCTION p.92

DevOps / Amazon <3 Linux

## ACCÉDEZ AU CLOUD AWS AVEC DES OUTILS OPEN SOURCE p.32

*EN PLUS : EXEMPLE PRATIQUE D'UNE APPLICATION WEB AVEC DJANGO - CRÉEZ UN PAQUET POUR EMACS...*

ikoula
HÉBERGEUR CLOUD

*PRÉSENTE*

# CLOUD**IKOULA**ONE


## Le succès est votre prochaine destination

MIAMI SINGAPOUR PARIS
AMSTERDAM FRANCFORT ___

*CLOUD**IKOULA**ONE* est une solution de Cloud public, privé et hybride qui vous permet de déployer **en 1 clic et en moins de 30 secondes** des machines virtuelles à travers le monde sur des infrastructures SSD haute performance.

www.ikoula.com sales@ikoula.com 01 84 01 02 50

ikoula
HÉBERGEUR CLOUD

NOM DE DOMAINE | HÉBERGEMENT WEB | SERVEUR VPS | SERVEUR DÉDIÉ | CLOUD PUBLIC | MESSAGERIE | STOCKAGE | CERTIFICATS SSL

**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
**Résponsable service infographie :** Kathrin Scali
**Réalisation graphique :** Thomas Pichon
**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 €





# ÉDITO

Ça y est les vacances sont terminées et vous voilà lancé, plein d'énergie, à la conquête d'un nouveau projet et pour cela vous aurez besoin de lui… ce *framework* qui devrait remplir tout ce dont vous avez besoin dans ce projet, ce *framework* qui vous fera gagner un temps précieux, celui pour lequel vous vous êtes mis à baver devant votre écran en vous disant « il me le faut ! ». Alors vous vous lancez, plein de fougue, dans l'installation de cet outil… mais bien entendu, il y a toujours un léger décalage entre la présentation du site web et la mise en pratique…

Souvent la page web est en totale adéquation avec la difficulté d'installation et de mise en œuvre : une seule page html contenant quatre lignes pour l'installation et au grand maximum une cinquantaine de lignes de documentation de l'API (quand on ne vous renvoie pas directement sur les sources). Dans ce cas, on sait à quoi s'attendre et généralement on ne se fait pas trop d'illusions quant au temps qu'il faudra passer/perdre pour effectuer quelques tests qui généralement conduiront à l'abandon du *framework* puisque l'absence de documentation le rend très difficilement utilisable.

Malheureusement, dans certains cas assez rares, on peut être en présence d'un projet particulièrement bien documenté, donnant accès à une description précise de l'API avec liens vers le code source et contenant de nombreux exemples sous la forme de tutoriels. Là il est normal d'être optimiste… mais lorsque l'on déchante, on tombe de bien plus haut ! Des éléments pouvant paraître anodins prennent des proportions gigantesques. Par exemple, si le projet requiert la dernière version de tel ou tel langage, non disponible dans le gestionnaire de paquets de votre distribution préférée, il va falloir l'installer. La documentation se bornera à cela : « Installez la dernière version du langage ». Le problème ici est que pour compiler un langage il faut disposer de certains outils et qu'il y a parfois de petites astuces de configuration à connaître ! Vous allez donc perdre deux jours (ou plus) avant même de pouvoir ne serait-ce que tenter d'installer le *framework* lui-même. Pourtant, dans la documentation cela semblait assez simple, la phrase était concise : « Installez la dernière version du langage »… (si les développeurs avaient pu le faire, ils auraient certainement ajouté « … et démxxxez-vous ! »). Ensuite, après bien des efforts, vous allez vouloir enfin tester le super exemple trouvé dans l'un des tutoriels de la documentation… et là encore il faudra y passer énormément de temps : on utilise des fonctions, des objets sans savoir exactement ce qu'ils font ni quels sont les paramètres à leur transmettre ! Certes l'exemple fonctionne, mais à quoi bon si l'on ne comprend pas ce que l'on fait et que l'on ne peut pas l'adapter ?

Écrire une documentation en apparence fournie ne suffit pas toujours, il faut également savoir expliquer, se mettre à la portée des utilisateurs qui ne vivent pas 24h/24 avec ce *framework*. C'est donc l'objet de l'un des articles que vous trouverez dans ce magazine et d'un autre côté, tant qu'il y aura ce genre de documentation il y aura du boulot pour *GNU/Linux Magazine*…;-)

Tristan Colombo

# SOMMAIRE

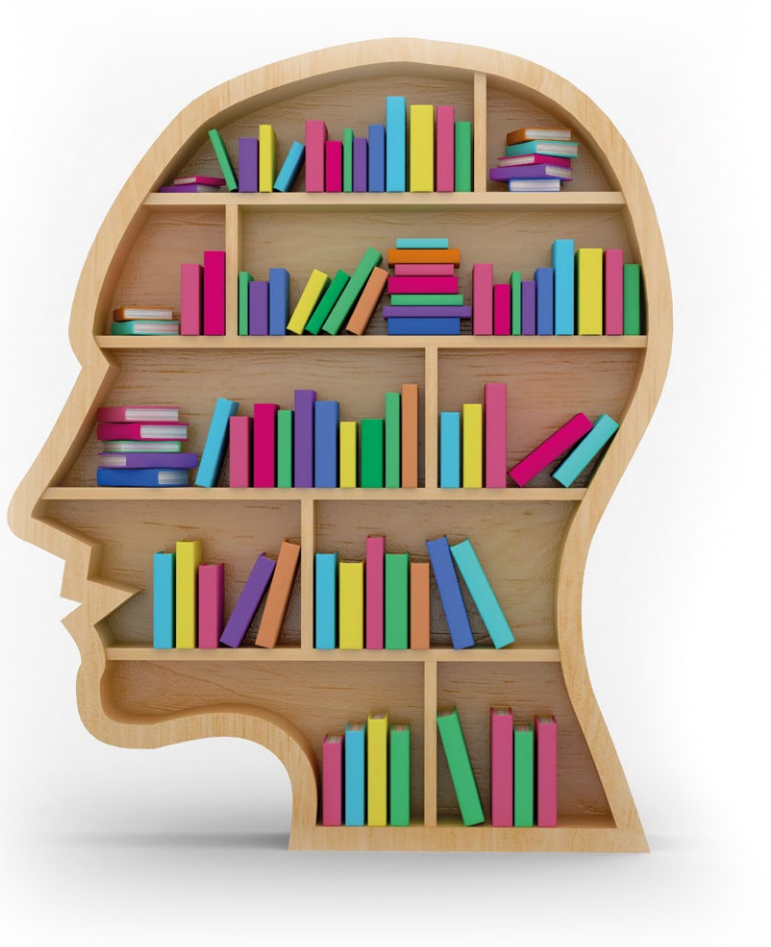
GNU/LINUX MAGAZINE FRANCE

**N°207**

# LE MÉTIER D'ARCHITECTE EN SI

**PHILIPPE PRADOS**
[Consultant Senior – OCTO Technology]

**MOTS-CLÉS : ARCHITECTURE, CONCEPTION, RÉSILIENCE, PERFORMANCE, SCALABILITÉ, MÉTIER, EVENT-SOURCING, MICRO-SERVICES**



**Le métier d'architecte en systèmes d'information est souvent mal connu. Je vous propose de vous mettre à sa place pour imaginer une architecture innovante pour un nouveau SI.**

Pour illustrer le travail d'un architecte, nous allons imaginer une situation spécifique et nous positionner dans sa tête. Quelles questions se pose-t-il ? Comment y répond-il ? Comment gère-t-il l'impact de ses choix ? Quand remet-il en cause ses décisions ?

## 1. LES GRANDES ÉTAPES POUR CONCEVOIR UNE ARCHITECTURE

Un architecte de Système d'Information doit faire des choix structurants pour implémenter le système à venir. Ce n'est pas facile, car les erreurs peuvent être difficiles à rattraper. Il a une très grande responsabilité sur la réussite ou l'échec d'un projet.

Une architecture de SI est souvent décorrélée du besoin métier direct, même si une connaissance grosse-maille du besoin est absolument nécessaire. On ne conçoit pas un site marchand comme une application devant traiter un flux boursier.

La première difficulté est de déterminer les **objectifs précis** de l'architecture, en termes de *patterns* métiers, consistance, performance, élasticité, résilience, évolutivité, etc.

Il faut ensuite sélectionner des technologies, compatibles avec les objectifs, puis s'assurer de la cohérence de l'ensemble.

Parfois, un choix ponctuel peut invalider un objectif primordial d'une architecture. Par exemple, dans une architecture qui doit être élastique, utiliser une base de données avec verrou centralisé - type **SQL** - est souvent une mauvaise idée. L'architecture peut être très élégante, mais elle ne présente pas les caractéristiques désirées en termes de scalabilité (capacité à augmenter linéairement la puissance de calcul en ajoutant des serveurs). En effet, il n'est pas possible d'ajouter indéfiniment des instances de base SQL. Si la base de données est trop sollicitée, l'intégralité du SI et de l'architecture ne fonctionnent plus. La seule solution pérenne est de revoir l'ensemble de l'application et de l'architecture.

Enfin, il faut extraire les grands principes de l'architecture que toute évolution future devra conserver. Par exemple, « un message est immuable », même si on désire annuler son effet.

Chaque évolution dans la réflexion de l'architecte doit faire adapter la combinaison des technologies, les grands principes et autres « best-practices » dans un cercle vertueux.

C'est au terme de nombreux cycles que l'architecture finale émerge. Il sera alors temps d'en faire une première implémentation succincte afin de s'assurer de la pertinence des choix et de leurs combinaisons.

Au fur et à mesure de l'analyse, des **faits** émergents, ainsi que des *interrogations*.

# 2. LE SCÉNARIO

Imaginons une entreprise qui possède un SI dont l'architecture est essentiellement basée sur des batchs et une base de données **Oracle** surpuissante. L'essentiel du code est en **PL/SQL**, exécuté dans la base de données. Régulièrement, des batchs sont exécutés pour extraire des données dans des fichiers à l'intention de prestataires, pour calculer des synthèses, la facturation, etc.

Cette entreprise doit gérer des événements venant du terrain concernant la livraison de biens, partout dans le monde. Ces événements arrivent périodiquement, pas toujours dans l'ordre. Les traitements batchs ingèrent ces événements pour déduire l'état des commandes et déclencher les facturations ou alerter les clients, dans un *workflow* bien précis.

Le DSI est inquiet, car la base de données n'est plus en capacité de répondre aux nouveaux besoins. Malgré un coût de licence approchant le million d'euros par an, et une machine de guerre dédiée, la base de données Oracle est à genoux. De plus, il souhaite avoir un traitement pour informer les agents sur le terrain directement sur leurs mobiles, pour avoir un tableau de bord capable de suivre jusqu'au chiffre d'affaires ou d'autres fonctionnalités proches du temps réel.

L'ambition de l'entreprise est très importante. Elle devra gérer bien plus d'événements qu'actuellement (positionnement GPS, nouveaux services, suivi plus fin des employés et des partenaires) et fournir des informations aux clients distributeurs ou destinataires de colis, le plus vite possible, afin de pouvoir ajuster rapidement les tournées.

La mission consiste à concevoir une architecture très rapide, qui ne sera pas limitée dans dix ans par des contraintes technologiques (vitesse des CPUs, taille des disques, vitesse des I/O, latence réseau, etc.). Au-delà de dix ans, c'est de la science-fiction.

# 3. LA CONCEPTION PAS À PAS

La première douleur constatée par le client est la limitation de la base de données Oracle. Dans l'architecture précédente, comme elle est arrivée à saturation, il n'y a plus rien à faire. Il faut revoir en profondeur l'architecture ou les applications pour que cela n'arrive plus jamais.

## 3.1 Gérer la base de données

Il faut donc choisir une base de données qui ne présente pas cet inconvénient. Les bases de données **NoSQL [1]** proposent généralement un modèle de persistance sans verrou. Le principe est « la consistance à terme » **[2]**. C'est-à-dire que les données finiront par être consistantes. Lors de l'exécution d'une requête, il est possible d'avoir des données un peu anciennes. Tous les serveurs ne voient pas exactement les mêmes valeurs au même moment.

C'est la même approche que la mise à jour d'un serveur DNS. Il faut un certain temps avant qu'un nouveau nom soit visible sur toute la planète.

Commençons par étudier ces solutions. Disons que nous envisageons d'utiliser **Cassandra [3]**. C'est une base de données bien implantée, avec une bonne réputation. Il y a également une implémentation plus rapide qui émerge : **ScyllaDB**. Elle est 10 fois plus rapide que l'implémentation Cassandra originale et nécessite donc dix fois moins de serveurs pour les mêmes capacités.

Il faut alors parcourir l'intégralité de la documentation d'architecture, de la documentation technique, des *best-practices* et chercher les questions fréquentes sur **StackOverflow**. Au fur et à mesure de la lecture, il faut rechercher les limites de la technologie. Il ne sert à rien de se focaliser sur le *core-business*, ce qui fonctionne, mais plutôt identifier les limites et les douleurs de la technologie.

C'est à l'architecte d'identifier les difficultés au plus vite, pour trouver un contournement en termes d'architecture ou de *pattern* d'utilisation. S'il loupe cela, l'architecture finale semblera pérenne, jusqu'au jour où le projet atteint les limites de la technologie qu'il a choisie.

Cassandra est une **base de données élastique**. Il suffit d'ajouter des serveurs pour augmenter la puissance de la base de données. C'est un bon point pour notre architecture.

Comment Cassandra fait pour gérer cela alors que les bases SQL n'en sont pas capables ? Il faut utiliser judicieusement des règles de distributions des données. C'est-à-dire qu'il faut identifier un attribut métier pour distribuer, le plus uniformément possible, les données sur les différents serveurs. Mais attention, **cet attribut doit être valorisé dans chaque requête**, afin d'identifier le serveur en charge d'y répondre. Cassandra sait retrouver des données si on lui indique la valeur de l'attribut utilisé pour la distribution. En gros et en simplifié, le client Cassandra applique un algorithme de hash sur la valeur de cet attribut. Il en déduit le serveur à qui s'adresser. Puis il lui envoie la requête pour retrouver les données.

Recherchons les limites de cette technologie.

*Comment gérer les cas où la clé de répartition n'est pas connue lors d'une requête ?*

Le modèle préconisé par Cassandra est « **le design by query** » **[4]**. C'est-à-dire que l'on va créer autant de tables que nécessaire, avec des clés primaires différentes, et plus ou moins les mêmes valeurs. Il faut donc dénormaliser les données. C'est-à-dire qu'il faut dupliquer les mêmes données dans différentes structures.

Par exemple, une table va utiliser le nom de l'utilisateur comme clé primaire. Une autre table va porter les mêmes données, mais en utilisant la ville de l'utilisateur comme clé primaire. Chaque table est conçue suivant la requête qu'on souhaite lui appliquer. Si l'on connaît le nom de l'utilisateur, on peut utiliser la première table. Si c'est la ville, on utilisera la deuxième table.

Le théorème CAP **[5]** indique que dans les architectures distribuées, il faut choisir entre la disponibilité et la consistance. En général, la disponibilité est alors préférée. Il n'y a donc plus de transactions ACID.

*Cela va consommer du disque !*

En effet, comme les mêmes données sont présentes dans plus d'une table, la place disque nécessaire est multipliée d'autant.

*Comment garantir la cohérence des différentes tables possédant les mêmes données ? Quelle est la table qui fait référence ?*

Cette question reste en suspens pour le moment. L'architecture devra y répondre.

Pour des raisons de résilience, Cassandra duplique chaque écriture sur plusieurs serveurs. Il n'y a donc pas de garantie que toutes les réplications soient visibles en même temps. Un serveur peut avoir des données un peu anciennes. Mais rapidement, tout rentre dans l'ordre.

Différentes stratégies permettent d'indiquer à partir de quel nombre de réplications confirmées, l'application peut sereinement continuer à travailler lors d'une écriture. Cela a un impact direct sur les performances lors des écritures. Combien faut-il attendre d'acquittements ?

Il est donc possible qu'un lecteur récupère une donnée qui n'a pas encore été modifiée, alors qu'elle l'est sur d'autres instances Cassandra.

Nous sommes en présence d'un modèle « **Éventuellement consistant** ». À terme, les modifications sont propagées, mais à un instant donné, il peut y avoir un décalage.

De même, lors des lectures, il est possible de définir un quorum de lecture cohérente pour se rassurer sur la valeur d'une donnée. Si n serveurs répondent la même valeur, alors il faut la considérer comme valide. **Suivant les cas métiers, il faudra donc ajuster correctement ce paramètre, sans trop dégrader les performances.**

Lors de la modification d'une donnée, il faut généralement lire une donnée, la modifier localement, puis écrire la version modifiée. C'est le *pattern* « Read-Modify-Write ».

*Que se passe-t-il si deux serveurs différents fonctionnent simultanément sur la même donnée de la même table ?*

Comme il n'est pas possible de poser un verrou lors de la lecture d'une donnée, deux traitements peuvent lire la même donnée, appliquer une modification comme ajouter 10€ au solde, avant d'écrire le résultat. Si deux traitements s'exécutent simultanément, le solde sera incrémenté de 10€ et non de 20€ au total.

Nous avons des données « éventuellement consistantes », répliquées dans différentes tables. Il est donc possible que cela arrive, avec parfois, les deux serveurs voyant les mêmes données, parfois des versions différentes des données.

C'est le dernier qui a écrit qui gagne.

**Il est fort probable que des modifications simultanées risquent de casser la cohérence des données.**

Cassandra propose un mécanisme de verrous optimiste pour gérer cela. C'est le *pattern* « *Compare-and-Set* » (CAS). En quelques mots, les serveurs se mettent d'accord sur une modification cohérente. L'utilisateur doit indiquer un critère qui permet de s'assurer que la modification est valide. Par exemple, « *ajuste le solde à 110€, si et seulement si, le solde est à 100€* ». En cas d'erreur, la modification n'est pas effectuée et une erreur est retournée. L'application peut relire la nouvelle version du solde et recommencer « *ajuste alors le solde à 120€, si et seulement si, le solde est à 110€* ».

À terme, la modification finira par passer. En cas de forte sollicitation, cela peut prendre pas mal de temps avant de réussir la modification. Il faut donc concevoir la base de données de telle sorte à éviter des contentions fortes sur certaines données, tel que le chiffre d'affaires total de l'entreprise.

**Le CAS dégrade les performances.**

*Peut-on éviter les situations de CAS ?*

Nous essayerons de trouver une solution via l'architecture.

Le problème est amplifié si les deux traitements écrivent sur plusieurs tables dénormalisées dans un ordre différent. Par exemple, le serveur Albert veut stocker l'information « `client:123, nom:paul` » dans une table et « `nom:paul, client:123` » dans une autre (le premier champ sert de clé de répartition). Au même moment, le serveur Bernard fait de même, mais en inversant l'ordre des tables. Il y a inconsistance entre les deux versions des mêmes données. L'utilisation d'une stratégie CAS fonctionne sur une table à la fois. Elle ne fonctionne pas entre deux tables. C'est en écrivant la deuxième table que des modifications sont rejetées. Faut-il alors annuler la première modification ? Est-on certain de pouvoir le faire ?

Il existe bien une possibilité de batch dans Cassandra, pour rendre « atomique » la sauvegarde des différentes versions. Un algorithme de type Paxos (consensus distribué) permet aux différentes modifications de se stabiliser. Mais c'est encore au prix de dégradations notables des performances (30%).

Nous voulons centraliser les mutations pour garantir que les modifications s'effectueront toujours dans le même ordre, la première table puis la deuxième. Il est possible de faire un CAS sur la première table, puis un simple Set sur les autres tables.

Si on part de cette hypothèse, on peut imaginer que toute mutation du SI soit portée par un message et traitée par un consommateur d'une file de messages. Ainsi, c'est uniquement la consommation du message qui permet de modifier les données. Les mutations sont centralisées. Cela fait émerger un type d'architecture : l'*Event-Sourcing*.

**Pattern Event-sourcing pour ne pas dépendre de l'ordre des écritures en base.**

L'idée est de produire des messages qui décrivent les mutations à appliquer aux données et de les déposer dans un bus. Modifier ou effacer un objet est décrit dans un message.

Il est alors possible de reconstituer l'état d'un objet en rejouant ce flux, en mémoire ou pour le sauver dans Cassandra par exemple. Dans ce modèle, la base de données n'est qu'un cache du flux des mutations.

L'idéal est de ne jamais effacer un message afin de pouvoir rejouer l'intégralité de l'historique.

Bon, à ce stade, nous avons deux principes forts :

- « Design by query » via une « Dénormalisation des données persistées » ;
- « Event-sourcing ».

Et un premier composant à notre *stack* technique : Cassandra.

Nous avons quelques questions importantes à régler :

- Comment gérer les modifications simultanées sur les mêmes données ?
- Comment contenir la consommation disque ?
- Comment éviter les dégradations de performances lors de l'utilisation d'algorithme CAS ou Paxos ?

## 3.2 Au plus vite

Nous ne voulons plus fonctionner en mode batch. Un batch est en effet toujours en retard et nous voulons du temps réel. C'est une notion toute relative qui dépend du besoin métier.

Pour fonctionner en « temps réel », nous avons besoin d'un bus de message très rapide. En effet, nous voulons pouvoir traiter au plus vite un flux de commandes, via un nombre de serveurs variable. Idéalement, nous souhaitons une scalabilité linéaire. C'est-à-dire que le gain de performance via l'ajout

d'un serveur est constant. Je peux ajouter autant de serveurs que nécessaire pour atteindre les performances attendues.

La technologie **JMS** avec différentes implémentations open source est une solution à envisager. Cette technologie propose conceptuellement deux modèles de file de messages : les queues et les *topics*. Les queues possèdent des messages consommés l'un après l'autre, éventuellement par plusieurs consommateurs. En cas de problème, la transaction peut échouer et le message retourne dans la file de messages comme si rien ne s'était passé. Les *topics* procèdent différemment. Le même message est envoyé à tous les processus à l'écoute du *Topic* (approche *publish/subscribe*).

Après étude, cette solution semble répondre à nos besoins au niveau fonctionnel, mais elle présente deux faiblesses :

- Comme la consommation des messages est gérée dans une transaction, le serveur JMS est le garant de la consommation du message. C'est un point de concentration de tous les traitements. Il n'est pas possible d'étendre les capacités du serveur JMS en ajoutant simplement un nouveau serveur. Si ce dernier tombe, il existe bien des solutions de type actif/passif, mais ce n'est pas suffisant. Dans les faits, les implémentations de files JMS sont limitées en nombre de messages pouvant rester dans la file et en nombre de clients à l'écoute des messages.
- La deuxième faiblesse concerne le mode Topic. Dans ce dernier, si un composant n'est pas disponible lors de la publication d'un message, le processus loupe un message et ne pourra jamais rattraper son retard. C'est très gênant pour la résilience de l'architecture. Comment mettre à jour à chaud une application qui écoute un *topic*, sans perdre des messages ? Il existe bien un mode permettant de garder les messages mêmes si les *subscribers* sont absents. Les messages sont alors gardés sur disque autant que nécessaire. Cela a un impact important en termes de capacité disque et de performance.

JMS ne semble pas être une solution viable pour notre architecture.

*Quelles sont les autres technologies disponibles, compatibles avec une architecture scalable ?*

La technologie **Kafka [6]** propose une autre approche. Elle est spécifiquement conçue pour gérer les architectures scalables, avec des centaines de serveurs en écoute des files de messages et un volume de message sans limite autre que la taille du disque.

Kafka peut être vu comme un énorme tampon circulaire dont chaque consommateur gère son curseur dans ce tampon. Après un certain temps, les anciennes données sont écrasées par les nouvelles.

Même si, à première vue, Kafka ressemble à JMS, le modèle est très différent. Lors du dépôt d'un message, il faut indiquer une clé de distribution (tiens, encore une). Celle-ci permet d'identifier la partition (le serveur kafka) en charge de sauvegarder la donnée dans la file. Ce dernier réplique dans un premier temps le message vers deux autres serveurs avant d'acquitter. En cas de perte d'un serveur, le driver client se charge d'en trouver un autre.

Côté consommateur, le modèle est également très différent. Contrairement à JMS, ce n'est pas Kafka qui gère les curseurs de lecture. C'est aux clients de le faire. Pour cela, Kafka n'autorise qu'un seul client à la fois sur une partition (pour un groupe donné). Un client peut se connecter sur plusieurs partitions, mais il ne peut pas y avoir plus de clients que de partitions.

Pourquoi donc, Kafka ne souhaite pas plus de clients sur chaque partition ? Et bien justement, pour garantir que les messages ne seront consommés qu'une seule fois, sans pour autant avoir besoin de synchronisation entre les instances Kafka. C'est une idée géniale permettant une scalabilité sans limite (voir figure 1).

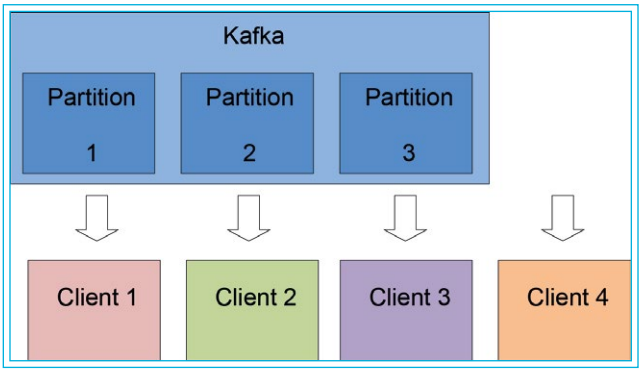



*Fig. 1 : Utilisation des partitions Kafka par les clients.*

Il est possible d'avoir plusieurs groupes différents consommant les mêmes messages. Au sein d'un groupe, il y a la garantie que les messages ne seront consommés qu'une seule fois.

Cette notion de groupe est très importante. En effet, c'est comme cela que l'on peut simuler le mode *publish/subscribe*. Chaque *subscriber* utilise un nom de groupe différent. Si aucun composant de l'application B ne fonctionne lorsque l'application A émet un événement, ce n'est pas grave. Au redémarrage de B, il pourra rattraper son retard, tant que le message est toujours dans le tampon circulaire. Cela est indispensable pour permettre la mise à jour à chaud des applications, sans devoir arrêter intégralement le SI.

L'étude de l'architecture et de l'implémentation de Kafka montre que les performances sont optimales si les instances Kafka sont sur des machines physiques non virtualisées, avec des disques dédiés. La stratégie d'implémentation est le *zero-copy*. C'est-à-dire que tout est fait pour permettre de passer du disque vers le réseau, sans sortir du kernel. Il n'y a donc aucune copie des tampons entre l'espace noyau et l'espace utilisateur.

**Il faut une clé de distribution pour chaque message dans Kafka.**

**Nous avons maintenant deux composants à notre architecture : Cassandra et Kafka.**

*Il existe une clé de distribution pour ces deux technologies. Peut-on utiliser la même ?*

La clé de distribution de Kafka permet de répartir les messages. La clé de distribution de Cassandra n'est pas sous le contrôle de l'utilisateur. C'est un calcul de Hash qui permet la répartition des objets dans les différents nœuds Cassandra.

En théorie, il est possible d'utiliser l'algorithme de Cassandra pour répartir les messages dans Kafka. Ainsi, il est possible de proposer des chaînes de traitements complètement isolées. Sur le même serveur, on place un Kafka et un Cassandra. Les communications ne s'effectueront qu'au sein du même serveur, sans avoir besoin du réseau. Dans les faits, ce n'est généralement pas une bonne idée.

La clé de distribution utilisée par Kafka doit nous permettre de gérer les problèmes de mutations simultanées des objets par plusieurs serveurs. En effet, si on utilise la clé primaire de l'objet à manipuler comme clé de répartition pour déposer les messages Kafka, on se retrouve dans une situation intéressante.

Par exemple, utilisons l'`id` d'un client comme clé de répartition pour tous les messages. Par construction, tous les messages concernant ce client seront exécutés sur le même serveur, à l'écoute d'une des partitions Kafka. Le traitement associé a la garantie d'être le seul à manipuler l'objet Cassandra de ce client. Il n'est donc pas nécessaire d'utiliser des stratégies du type *Compare-And-Set* (CAS) ou autres transactions distribuées.

Oui mais, on a bien vu qu'il fallait un *design by query*. Pour le même objet, il peut y avoir plusieurs clés primaires, une par table dénormalisée. Par exemple, une table indexée par id et une autre par ville. Le *mapping* clé de distribution / clé primaire ne fonctionne pas.

À moins de s'organiser pour apporter toutes les modifications des différentes tables dans le même traitement, distribué par une des clés primaires ! Dans ce modèle, les messages sont distribués sur les serveurs Kafka suivant l'id du client par exemple (voir figure 2). Le traitement qui consomme ce message va être chargé d'apporter toutes les modifications sur toutes les tables portant les données du client. Ainsi, il est possible de bénéficier des performances optimales de Cassandra, sans synchronisation entre les serveurs. Il n'est pratiquement plus nécessaire d'avoir des *Compare-And-Set* ou des algorithmes de type Paxos.

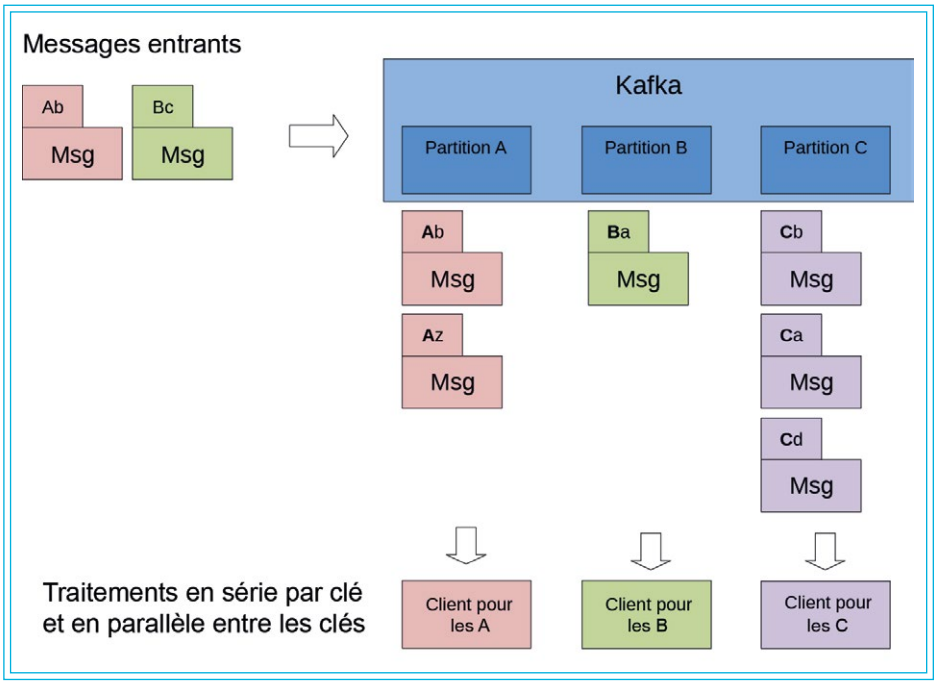



*Fig. 2 : Distribution des messages dans les partitions Kafka.*

**Il faut centraliser les modifications des données dans Cassandra en s'appuyant sur Kafka pour sérialiser les écritures du même objet.**

Toute mutation du SI est portée par un message et traitée par un consommateur Kafka.

Nous pouvons alors sortir de notre besace, le modèle d'architecture CQRS (*Command Query Responsibility Segregation*). C'est un modèle d'architecture où les écritures et les lectures en bases de données sont séparées (voir figure 3).

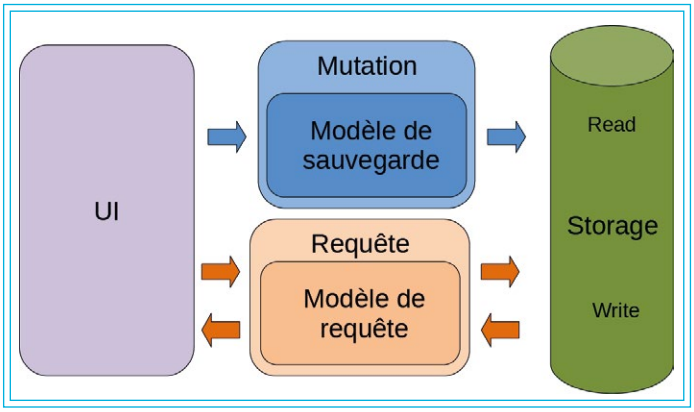



*Fig. 3 : CQRS : Séparation des modèles de sauvegardes et de requêtes.*

Les modèles de données peuvent évoluer différemment suivant les besoins (cf. *Design by query*), avec des structures et des technologies variées (base de données tabulaires, clés/valeurs, graphes, etc.). Une base de données sert de référence pour les écritures. Les modifications sont reportées vers les différents modèles de persistance.

De cette idée il semble possible de régler certains problèmes de notre liste. Comme chaque mutation est décrite par un message (*Event Sourcing*), c'est le traitement du message qui se chargera de l'écriture des données pour chaque table. Ce traitement est garant de l'ordre d'écriture dans les différentes tables dénormalisées (voir figure 4).

De plus, si une nouvelle table dénormalisée est nécessaire, il ne faudra intervenir qu'à un seul endroit : dans le traitement du message qui décrit la mutation des données. Il est même envisageable d'utiliser un compte utilisateur à la base de données ayant les droits d'écriture pour ce traitement, et un compte n'ayant que le droit de lecture pour toutes les autres applications. Ainsi, nous avons la garantie que les mutations seront cohérentes entre les différentes technologies de persistance.

Une autre approche consiste à avoir plusieurs composants différents à l'écoute des messages décrivant les mutations (voir figure 5). Chacun est libre de persister la mutation dans la table et la base qu'il souhaite.

Les règles de distributions de Kafka vont permettre de garantir qu'un même objet ne sera pas modifié dans une table de la base de données par deux serveurs différents.

Le modèle CQRS permet d'ajuster les capacités du *cluster* différemment pour les flux d'écriture et les flux de lecture. Par exemple, utiliser 3 serveurs pour les écritures et 20 pour les lectures.

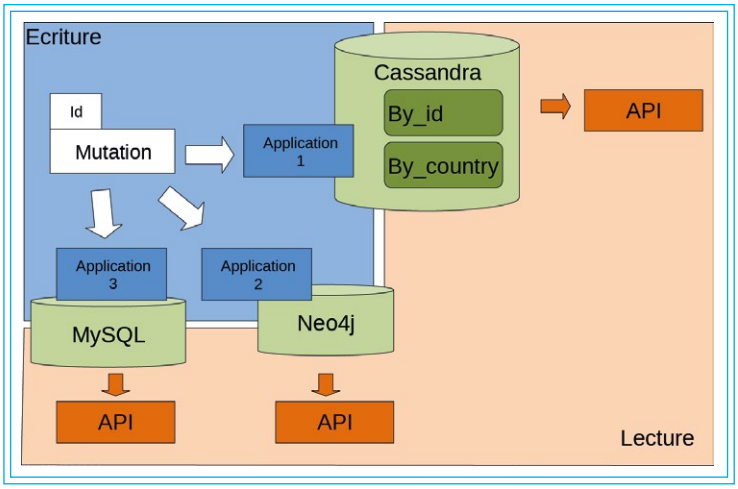



*Fig. 5 : Dénormalisation par service.*

**Pattern CQRS pour optimiser les écritures par rapport aux lectures.**

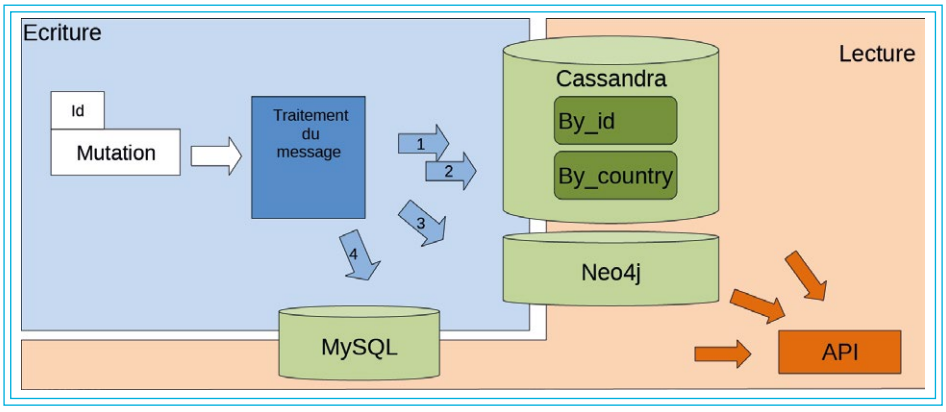



*Fig. 4 : EventSource, CQRS : Dénormalisation centralisée.*

Nous avons encore quelques questions importantes à régler. Par exemple, comment contenir la consommation disque ?

L'utilisation d'un bus de message rend les mutations asynchrones. Tant que le message n'est pas consommé, la mutation n'est pas appliquée. Cela présente des avantages en termes de résilience et des inconvénients en termes de consistance. Sur le théorème CAP, nous avons sacrifié la cohérence au bénéfice de la disponibilité.

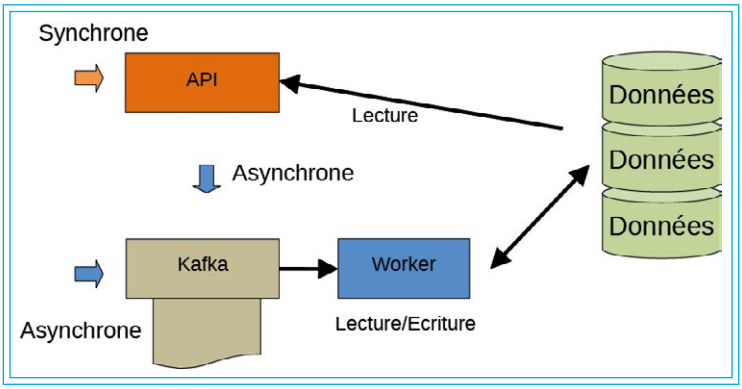



*Fig. 6 : Appel synchrone et asynchrone.*

Nous avons finalement un modèle de consistance à terme, le temps que les modifications soient propagées vers tous les middlewares de persistance.

Le modèle qui se profile ressemble à la figure 6.

Un flux de message décrit toutes les mutations du SI. Ce flux est traité par des clients Kafka via une règle de distribution garantissant qu'un seul serveur peut modifier la même donnée. Une donnée est modifiée en série, via l'écriture vers les différentes tables dénormalisées. Plusieurs données peuvent être modifiées en parallèle, mais sur des objets différents.

Avec ce modèle, il est possible de l'étendre en indiquant que les traitements des messages peuvent dénormaliser sur d'autres technologies comme un moteur d'indexation type ElasticSearch, une base de données Graph comme **Neo4j**, voire une base SQL. Néanmoins, utiliser Neo4j ou une base SQL risque de réintroduire un SPOF (*Single Point Of Failure*), car ces technologies utilisent une approche centralisée et non distribuée.

*Comment faire pour lire les données présentes dans les bases de données ?*

Nous avons deux solutions :

- La première est orientée requête : le consommateur invoque une API avec des critères de recherche pour obtenir une vue de l'information qu'il recherche.
- La deuxième approche est orientée flux. Un flux de sortie décrit les états consolidés des objets. « *Tel client a maintenant 120€ sur son compte* ». Les applications sont à l'écoute des évolutions des états.

C'est un modèle de développement en **micro-services**. Chaque service utilise une ou plusieurs bases de données privées, avec différentes technologies si besoin. Il expose des services web pour permettre l'accès à ses données. Le service se charge de choisir la meilleure technologie ou table conçue en interne pour répondre au plus vite aux requêtes des autres applications. De plus, chaque application publie des flux décrivant ses modifications d'états (les nouvelles synthèses consolidées, les alertes, etc.)

Les services communiquent entre eux via leurs services web et via le dépôt et l'émission de messages (voir figure 7). Les communications peuvent être synchrones et asynchrones.

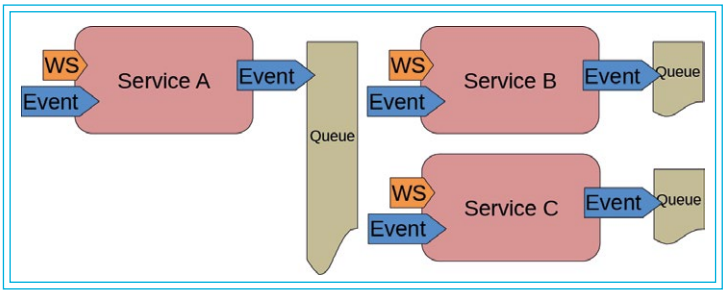



*Fig. 7 : Plusieurs services communiquent en synchrone et asynchrone.*

Nous ajoutons dans l'architecture des serveurs web qui exposent des **API WebService** pour lire les données dans les différentes tables et les différentes technologies de persistances. Ces API n'ont besoin que d'un accès en lecture, car elles ne doivent pas modifier directement les données, au risque de casser la cohérence entre les tables dénormalisées, voire les données elles-mêmes lorsqu'un objet est modifié simultanément par plusieurs *workers* de *middlewares* (voir figure 8).

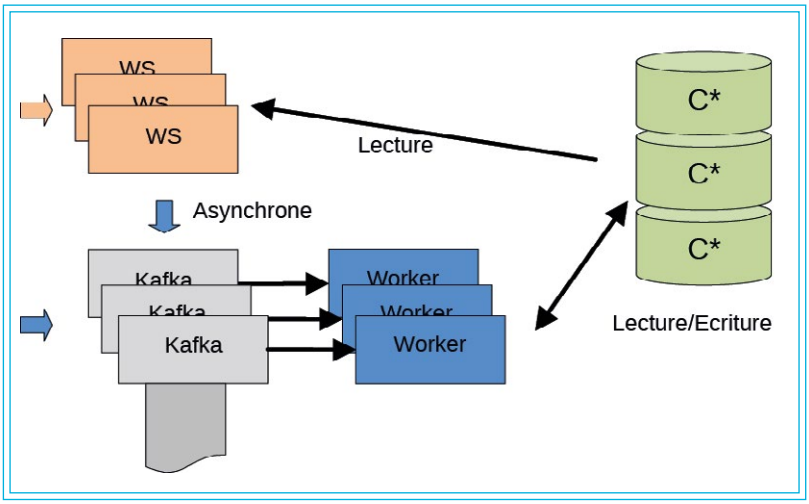



*Fig. 8 : Plusieurs instances de middlewares.*

Mais comment proposer des API pour modifier les objets ? Il faut pour cela que ces dernières construisent des messages décrivant les mutations à appliquer et les injectent dans le bus Kafka. Nous sommes conformes avec le modèle de cohérence à terme.

Les services web peuvent répondre avec un code http **200** pour confirmer l'écriture (même si elle n'est pas immédiatement appliquée) ou un code http **202** pour indiquer que la demande est prise en compte et sera appliquée dès que possible. C'est conforme avec la stratégie consistant à privilégier la disponibilité sur la cohérence.

Nous souhaitons que le serveur web utilise également une approche scalable et résiliente. Il faut donc prévoir en amont un répartiteur de charge.

**Il faut un serveur web pour les lectures et un répartiteur de charge.**

**HAProxy** est une bonne solution pour le répartiteur de charge et devrait nous aider à faire des migrations progressives (voir figure 9). Il faut au moins deux instances pour fonctionner en mode actif/passif et une VIP (*Virtual IP*).

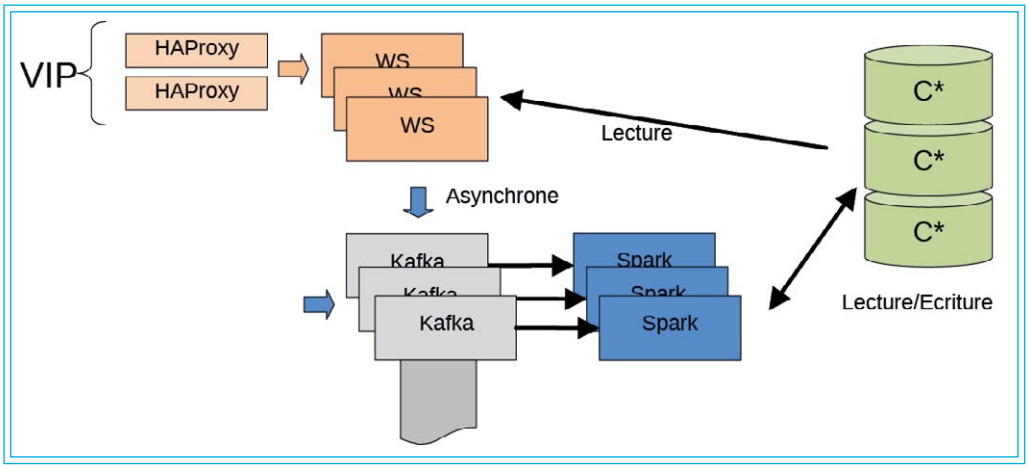



*Fig. 9 : Ajout d'une VIP et de HAProxy.*

Toujours à la recherche de performance, nous voulons choisir un serveur web le plus rapide possible afin de réduire le nombre d'instances nécessaires pour tenir la charge. Les approches réactives semblent de bons candidats (**Node.js**, **Play**, etc.). Un serveur sans état semble une piste intéressante et compatible avec notre modèle.

Que peut-on identifier comme avantage dans l'approche CQRS ? Au niveau résilience, il est possible de perdre momentanément les composants en charge des mutations, à l'écoute des partitions Kafka, sans interruption des API. Les mutations continueront à être déposées dans les serveurs Kafka. Par contre, elles seront effectives un peu plus tard. Lorsque le service sera à nouveau disponible.

Nous avons donc la possibilité de mettre à jour indépendamment la partie écriture d'une application de la partie lecture. Il n'y a, à aucun moment, une indisponibilité du service.

En utilisant un *middleware* différent pour les services web et pour les traitements Kafka, nous pouvons ajuster différemment le nombre de serveurs entre la lecture et l'écriture. Cela peut représenter une économie pour le projet.

Comme nous l'avons décrit, des flux sont émis par les services pour exposer en temps réel les nouveaux états des objets. Ainsi, il est facile de propager les impacts d'une modification sur l'ensemble du SI le plus rapidement possible. Un flux de mutation se propage de service en service. Il est même possible de propager ce flux de mutation vers les navigateurs des utilisateurs, via des **WebSocket** par exemple.

Si pour une raison ou une autre, un autre service du SI est indisponible lors de la modification d'un objet, avec Kafka, il sera capable de rattraper son retard. C'est indispensable pour avoir une dépendance lâche entre les services.

Chaque service est découpé en deux :

- une partie à l'écoute de flux, en charge des écritures et de l'émission de flux d'événements ;
- et une autre partie en charge d'exposer l'état de l'application à la demande.

Les services consomment des flux, persistent des états, émettent des flux et exposent des API. Chaque service est responsable de ses propres bases de données. Il n'est pas possible d'y accéder directement sans passer par les API proposées par le service.

Un service peut envoyer une commande de mutation à un autre service, consulter des données ou être à l'écoute de l'impact de chaque mutation.

Bon, à ce stade, nous avons plusieurs principes forts :

- *Design by query* via une dénormalisation des données persistées ;

- modèle *Event-Sourcing* (toute mutation est portée par un message immuable) et CQRS ;
- un accès en lecture seule pour les API.

Nous avons identifié quelques composants à notre *stack* technique : Cassandra, Kafka, un consommateur Kafka à identifier, un serveur web résiliant, deux HAProxy et une VIP.

## 3.3 Choisir un consommateur de flux

Nous devons choisir un *middleware* pour consommer les flux Kafka. Ce dernier doit être scalable, résiliant et riche en fonctionnalités.

Il existe plusieurs familles de *middlewares* de ce type. Certains ont été conçus dès le départ pour le temps réel comme **Flink**, d'autres sont une évolution d'un modèle batch comme **Spark Streaming**. Il existe de nombreuses solutions.

Là, l'architecte va probablement utiliser son instinct et proposer un choix subjectif suivant son histoire et ses connaissances. Néanmoins, il faut faire une étude comparative entre différentes solutions, toujours à la recherche des limites des technologies.

Spark Streaming semble une technologie en pleine croissance. Le moteur de *streaming* utilise une approche micro-batch (des batchs réguliers à périodicité courte). Les données des flux sont accumulées, puis périodiquement, les données sont traitées comme si elles étaient une seule base de données. Et le processus recommence. Ce modèle impose donc une latence minimum de la durée du micro-batch.

Est-ce gênant pour notre projet ? L'échelle de temps ne nécessite pas un temps réel très rapide. Ce n'est pas un problème pour le projet. Nous ne traitons pas de flux financier.

Spark offre une programmation à deux niveaux :

- un **modèle fonctionnel** où les objets sont immuables dans le pipeline de traitement ;
- un modèle SQL Like (qui utilise le modèle fonctionnel en interne).

Les API permettent de joindre plusieurs flux de traitement, de les trier et les distribuer en *peer-to-peer* entre les nœuds, etc.

Spark est codé avec le langage **Scala** qui porte nativement les concepts de la programmation fonctionnelle. Ce langage est alors un candidat potentiel pour le développement.

Utiliser Scala avec Spark va nous faciliter la vie. Mais si on souhaite partager du code entre Spark et les serveurs web, Play s'impose comme le *middleware* pour les API. Play est sans état, ce qui est compatible avec notre architecture. Ainsi, nous avons le langage Scala dans tous les composants.

*Comment Spark est-il capable de gérer les pics de charge ?*

Il existe un algorithme de *back-pressure* qui permet de limiter le nombre d'objets à traiter le temps d'un micro-batch. Cela évite autant que possible d'accumuler un retard dans le traitement des messages. Un ajustement dynamique est effectué pour distribuer les traitements sur les *workers* Spark le mieux possible.

Donc, en cas d'indisponibilité de l'ensemble des instances Spark pour une mise à jour du *middleware*, lors de la reprise de l'activité, il n'y aura pas d'écroulement des Workers par surcharge de travail. Automatiquement, le flux en retard sera traité convenablement par rapport à la capacité de la plateforme.

Si besoin, il est possible d'ajouter de nouveaux *Workers* dynamiquement. Lors du prochain micro-batch, ce dernier sera ajouté aux capacités à faire du *middleware* Spark.

*Comment Spark s'interface avec Kafka ?*

Le dernier modèle d'intégration de Kafka dans Spark consiste à obtenir une connexion directe entre un *Worker* Spark et une partition Kafka. Donc, avoir plus de *Workers* que de partitions Kafka n'est pas optimal pour les premiers traitements (voir figure 10).

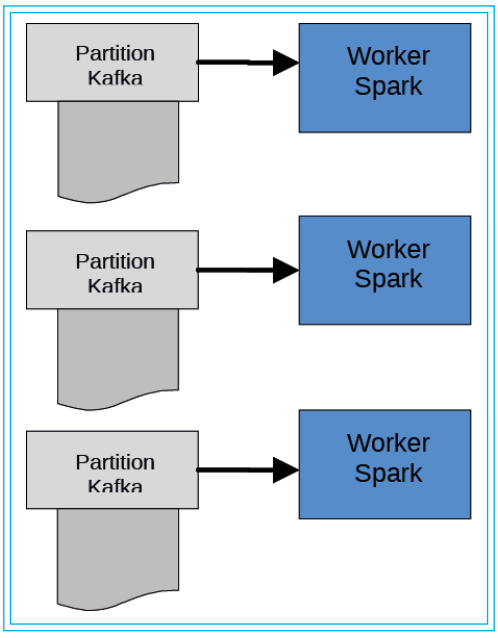



*Fig. 10 : Une partition Kafka par worker Spark.*

*Comment Spark gère les crashs ?*

Comme tout modèle distribué, il est difficile de traiter les cas de pertes de messages. La garantie que nous avons est que tous les messages seront traités au moins une fois.

**Il faut gérer les cas des messages traités plusieurs fois.**

Les messages de mutation doivent être idempotents. C'est-à-dire que le traitement du même message plusieurs fois n'a pas d'effet de bord différent que l'exécution du message une seule fois. Pour obtenir cela, il y a plusieurs approches :

- Maintenir une fenêtre temporelle permettant d'identifier le rejeu d'un message via un id unique dans chaque message. Cet id peut être injecté dans l'objet modifié.
- Concevoir des messages par nature idempotents : « *Ajoute 10 au solde qui doit être à 100 pour obtenir 110* » à la place de « *+10 au solde* ». Le traitement peut vérifier que le solde correspond bien à ce qui est prévu par le message. Si ce n'est pas le cas, il peut envoyer une erreur ou finalement se rendre compte que la demande correspond à ce qui vient d'être effectué et retourner un acquittement.

**Les traitements des messages de mutations doivent être idempotents.**

## 3.4 Choisir un format d'échange

Nous avons des micro-services qui doivent communiquer entre eux, en synchrone ou en asynchrone. Il faut choisir un format de message.

Idéalement, un schéma doit décrire le contrat d'interface entre les composants afin de s'assurer qu'il n'y aura pas de message mal formé.

Comme chaque micro-service doit pouvoir évoluer indépendamment des autres, il doit être possible de monter de version d'un service sans devoir mettre à jour l'intégralité du SI.

Nous avons deux modèles de communication : un modèle à base de service web et un modèle asynchrone à base de flux. Il faut donc envisager les montées de versions sur tous ces contrats d'interfaces.

Un service A doit pouvoir envoyer un message en version 1 ou en version 2 à un service B, sans avoir besoin de savoir si le service B est déjà en capacité de gérer la version 2. De même, lorsqu'un service émet des événements, il ne peut pas connaître les versions prises en charge par l'intégralité des *subscribers*.

Si on étudie l'architecture, on constate que, par construction, les messages sont répliqués de nombreuses fois sur le réseau et sur disque. Par exemple, déposer un message dans une instance Kafka entraîne l'utilisation de trois versions sur le réseau pour communiquer avec les autres partitions lors des réplications et trois fois sur disque pour que chaque partition possède une copie. La lecture d'une partition entraîne une nouvelle version de plus sur le réseau. Spark peut mémoriser des *Snapshots* sur un disque distribué pour être en capacité de reprendre un job complexe, etc.

Ainsi, la taille des messages a un impact très important sur la consommation des ressources disques et réseaux. C'est un point à prendre en compte dès le début du projet.

**XML** permet de décrire un schéma via **XML-Schema**, mais n'est pas en capacité de gérer les montées et les descentes de versions. Une application qui est codée pour gérer un message en version 1 n'est pas capable de gérer un message en version 2. De plus, ce format n'est absolument pas compact.

JSON est plus compact, mais porte la description du schéma dans les données. La taille des noms des champs a un impact sur la taille des messages ! Il existe bien quelques initiatives pour proposer des Schémas-Json pour valider les messages, mais ce n'est pas idéal.

Les géants du Web utilisent en interne des formats binaires, bien plus efficaces que XML ou JSON. Mais dans ce cas, les schémas permettant de décrire le flux binaire doivent être portés par les applications. Les schémas ne sont pas présents dans les messages, mais dans les applications. Comment gérer les évolutions dans les schémas binaires ?

Une rapide recherche sur Internet nous amène sur la solution proposée par **Confluent.IO**. Il s'agit d'utiliser le format binaire **AVRO** de la fondation Apache et d'utiliser un *Schema-Registry* pour centraliser les différentes versions des schémas.

Les messages binaires peuvent posséder l'identifiant du schéma présent dans le *Schema-Registry*, puis le message en binaire. Si l'application ne connaît pas le schéma, elle le récupère puis demande à l'API AVRO de faire la conversion nécessaire pour exposer les données via le schéma qu'elle connaît.

AVRO permet ainsi d'avoir des valeurs par défaut, des alias pour les noms des champs ou des *packages* et propose de nombreux formats pour structurer les données (Map, séquence, Union, Enum, Structure, etc.).

Le format binaire peut servir également pour optimiser l'invocation des services web. Il suffit d'ajouter un : `Content-Type: application/avro`.

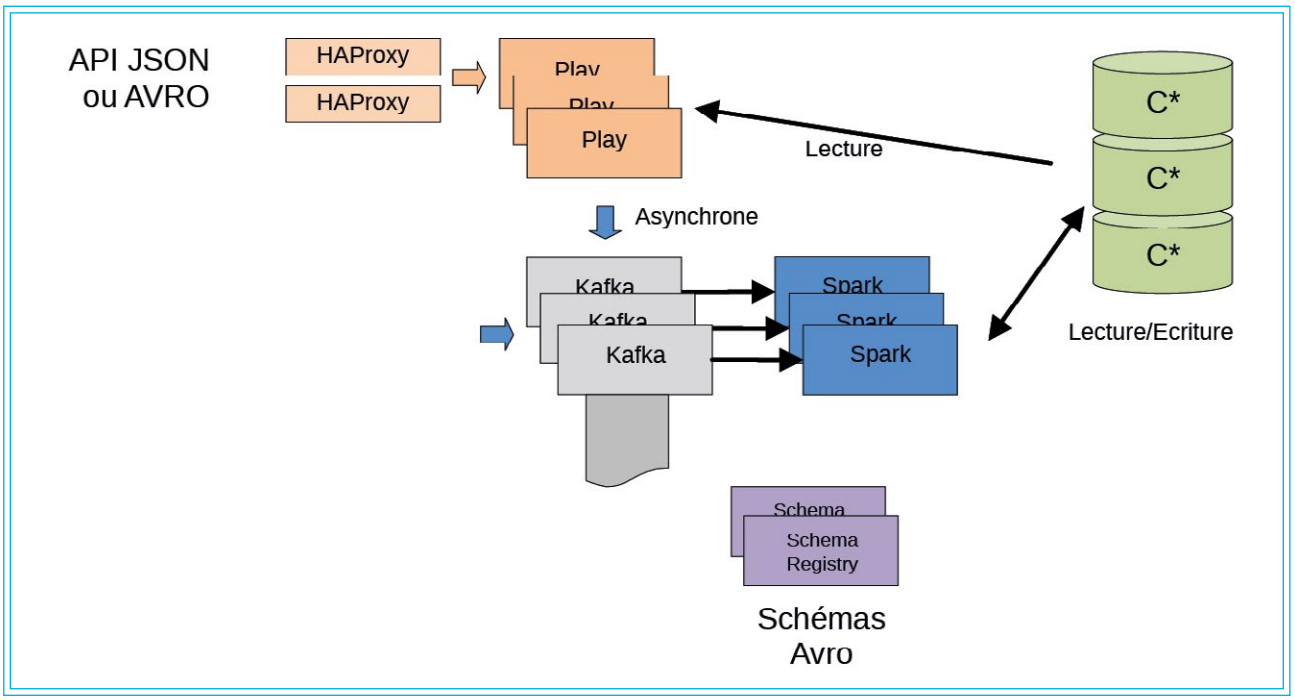



*Fig. 11 : Intégration de AVRO.*

Nous avons maintenant identifié un format d'échange et un nouveau composant (voir figure 11).

Nous avons identifié que le « design by query » proposé par Cassandra a tendance à manger du disque. En y réfléchissant, dans les structures complexes du métier, seuls quelques champs servent de critère de recherche dans la base. Au minimum la clé primaire, éventuellement un ou deux autres champs. Pour le reste, il est envisageable d'utiliser à nouveau le format AVRO pour persister les données dans Cassandra en binaire. Des colonnes servent pour les critères (*design by query*) et une colonne pour la donnée binaire complète. Lors de la lecture de la donnée binaire, AVRO se charge de reconstituer le graphe de l'objet, dans toute sa complexité.

Avec cette stratégie, on réduit la consommation disque et le trafic réseau lors de la communication avec Cassandra. Cela peut être une recommandation de l'architecture, mais pas une préconisation. À chaque micro-service d'identifier le meilleur compromis entre simplicité et efficacité dans la consommation des ressources.

À ce stade, nous avons plusieurs principes forts :

- *Design by query* ;
- Dénormalisation des données persistées, éventuellement en binaire AVRO ;
- Modèle *Event-Sourcing* et CQRS ;
- Toute mutation est portée par un message immuable ;
- Message en binaire AVRO avec un schéma dans un référentiel ;
- Un accès en lecture seule pour les API ;
- Les messages doivent être compatibles avec l'idempotence.

Nous avons identifié quelques composants à notre *stack* technique : Cassandra, Kafka, Spark Streaming, Play, Schema Registry, plusieurs HAProxy, un VIP et le langage de programmation : Scala.

## 3.5 Invocation des services web

L'approche micro-service ne permet pas d'avoir des transactions ou des jointures entre les données de différents micro-services. Tout cela doit être codé « à la main ».

Parmi les micro-services proposés, l'un est particulièrement exposé : le référentiel d'entreprise. Les serveurs Plays de ce micro-service vont subir une pression très forte de tous les autres micro-services.

Concrètement, si tous les jobs Spark Streaming invoquent le référentiel en parallèle, les piles IP vont exploser et supprimer pas mal de paquets. Des *times-outs* vont apparaître.

Pour régler cela, nous avons plusieurs pistes :

- Écrire une bibliothèque d'invocation de service web qui :
  - → exploite le protocole HTTP/1.1 pour recycler les connexions afin de limiter le nombre de connexions ;
  - → ajuste dynamiquement le nombre de connexions avec un algorithme de « back-pressure » ;
  - → est capable de rejouer une requête plusieurs fois avant d'en informer l'application (pour la tolérance à la panne d'un service web).
- Écrire une bibliothèque de cache du référentiel, branchée sur les flux de mutations du service, afin d'ajuster les valeurs des caches au plus tôt. Cette bibliothèque sera présente dans chaque service.

**Nous devons sélectionner ou écrire un algorithme d'invocation de service web avec back pressure.**

## 3.6 Intranet et Internet

Nous avons des services communiquant via des WS ou en flux. Mais il n'est pas question, pour des raisons de sécurité, de donner un accès direct à ces flux depuis Internet. Nous devons placer un serveur web en façade, qui se charge d'exposer un sous-ensemble des micro-services internes, de filtrer les requêtes et les réponses pour limiter le nombre de réponses par exemple (pour éviter le *dump* de la base client), etc. Un serveur dans la zone de sécurité Internet et un autre dans la zone de sécurité Intranet sont autorisés par le *firewall* à communiquer avec les API des services. Pour des raisons d'isolation des zones, nous devons ajouter des instances HAProxy devant chaque serveur Intranet et Internet et autant de VIP (voir figure 12).

Nous avons maintenant pratiquement l'intégralité de l'architecture d'exécution.

Notez que nous n'utilisons pas de NAS ou autres réplications de disque. Les technologies sélectionnées sont parfaitement capables de gérer la résilience des données.

## 3.7 Mise à jour à chaud

Chaque composant de l'architecture doit pouvoir évoluer indépendamment des autres, sans interruption notable de service. Vérifions cela :

- Kafka est capable de faire un *rolling-update* à chaud. Chaque partition est arrêtée et migrée l'une après l'autre. Il est donc possible de monter de version de Kafka sans interruption. Quelques flux seront un peu en retard.
- Il est également possible d'ajouter dynamiquement de nouvelles partitions Kafka.
- Pour Spark streaming, c'est plus compliqué. On peut arrêter une application Spark, la mettre à jour, et la relancer. Kafka sert de tampon. Les messages auront un peu de retard, mais ce n'est pas problématique. Les interfaces de lecture continuent à fonctionner.
- Pour Cassandra, il est également possible de faire un *rolling-update* des différents nœuds, sans interrompre la base de données.
- Il est également possible d'ajouter de nouveaux serveurs Cassandra et/ou de nouveaux disques.
- Pour Play, les instances n'ont pas d'état. À l'aide d'un répartiteur de charge comme HAProxy, il est possible de migrer les instances progressivement à chaud.

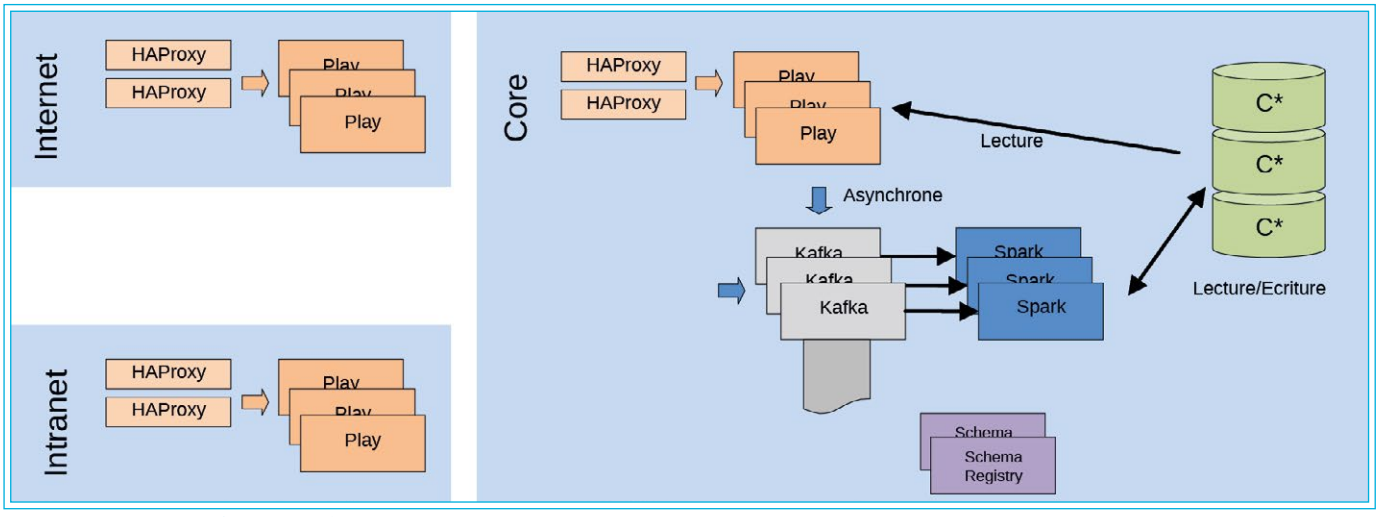



*Fig. 12 : Architecture d'exécution.*

- HAProxy peut également être mis à jour à chaud, en s'occupant du serveur passif avant le serveur actif.

### 3.8 Gestionnaire de cloud

Nous devons gérer de nombreux serveurs, pouvoir en arrêter et en ajouter à chaud. À la main, ce n'est pas raisonnable. Il faut une approche Devops complète de l'intégralité de la plateforme.

Là, intervient le côté commercial de l'architecture. En effet, il est possible de n'utiliser que les versions open source des composants ou d'utiliser des solutions commerciales. Les versions open source sont généralement moins bien équipées pour une mise en production. C'est à voir avec les budgets de chacun. Attention, il est très difficile d'automatiser la gestion d'une plateforme comme celle-ci à l'aide de scripts **Ansible** par exemple.

## 4. L'ARCHITECTURE FINALE

Nous avons un premier jet d'architecture à proposer. Elle fait apparaître des concepts forts, qu'il va falloir tenir le plus longtemps possible :

- chaque *middleware* doit être élastique et résilient ;
- il faut pouvoir mettre à jour à chaud les services et les composants ;
- les mutations doivent pouvoir traverser l'intégralité du SI le plus vite possible.

## CONCLUSION

L'architecte peut maintenant présenter son analyse et justifier ces choix. Il a monté une application moderne, orientée événement, qui utilise un modèle de transaction éventuellement consistant.

L'architecture peut apporter par elle-même des solutions à des difficultés. Dans le scénario présenté, la séparation du flux de mutation est justifiée par la volonté de garantir la cohérence de la dénormalisation et éviter autant que possible les scénarios non performants de Compare-And-Set.

Cette simulation de la réflexion d'un architecte montre qu'il s'agit d'un travail d'analyse sensible. Il est nécessaire d'avoir en permanence un regard critique et d'identifier les faiblesses de chaque choix. Il ne faut pas occulter certaines difficultés pour simplifier l'architecture. La loi de Murphy garantit que tous les problèmes identifiés arriveront. Et bien d'autres encore, inconnus à ce stade. Le rôle de l'architecte est d'en prévoir le plus possible pour proposer des solutions les réduisant.

Nous espérons vous avoir éclairé sur la complexité du métier d'architecte en SI. ■

## RÉFÉRENCES


**[1]** NoSQL : https://fr.wikipedia.org/wiki/NoSQL

**[2]** Consistance à terme : https://en.wikipedia.org/wiki/Eventual_consistency

**[3]** Cassandra : http://cassandra.apache.org/

**[4]** Desing by Query : https://docs.datastax.com/en/cql/3.1/cql/ddl/dataModelingApproach.html

**[5]** Théorème CAP : https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_CAP

**[6]** Kafka : https://kafka.apache.org/

# DES RÉSEAUX DE NEURONES POUR CLASSER DES IMAGES


TRISTAN COLOMBO

MOTS-CLÉS : TORCH, IMAGES, CLASSIFICATION, GPU, RÉSEAU DE NEURONES

**On fait énormément de choses avec les réseaux de neurones et de plus en plus de frameworks sont disponibles pour les utiliser simplement. Ce mois-ci je vous propose de classer des images en catégories (voiture, chat, chien, etc.) en utilisant PyTorch, ce qui nous permettra d'appréhender le fonctionnement des moteurs de recherche visuels.**

**Tensorflow** de **Google [1][2]**, **CNTK** (pour *CogNitive ToolKit*) de **Microsoft [3]**, **Theano [4]**, **Caffe [5]** ou encore **Torch [6]** : on assiste à une explosion du nombre de *frameworks* dédiés aux réseaux de neurones et au *deep learning*. Nous avons déjà utilisé Tensorflow dans le cadre de la reconnaissance d'une écriture manuscrite de chiffres **[2]** et nous allons donc tester un nouveau *framework* pour mettre en place une classification d'images. Ce sera Torch qui dispose d'un module **Python** nommé **PyTorch** (sinon il faut coder en **LUA**).

Avant de nous lancer dans l'installation des différents outils nécessaires et au codage, revenons un peu sur notre problématique. Nous disposons d'un ensemble de photos contenant des chats, des chiens, des chevaux, des voitures, etc. et nous souhaiterions pouvoir les classer : toutes les photos de chats ensemble, toutes les photos de chiens ensemble, etc. Pour cela, nous allons devoir mettre en place une méthode classique de classification basée sur un apprentissage supervisé. Nous utiliserons ici un ensemble d'images d'apprentissage fourni directement par le projet Torch. Le *framework* PyTorch est techniquement plus complexe

à mettre en œuvre et cet article sera donc vraiment basé sur la technique, sur les petites astuces qui vous feront gagner quelques heures pour l'utilisation de ce framework.

# 1. INSTALLATION DE PYTORCH

PyTorch est bien disponible via **pip3**, mais avant de pouvoir l'installer il va falloir satisfaire un certain nombre de prérequis :

- disposer d'un Python 3.5 ou 3.6 (il est possible d'installer PyTorch pour Python 2.7, mais nous resterons sur la branche 3 de Python) ;
- pour l'accélération graphique, avoir installé CUDA en version 7.5 ou 8.0.

Comme il y a très peu de chances pour que ces versions soient disponibles directement dans le gestionnaire de paquets de votre distribution, nous allons procéder à une installation manuelle.

**NOTE**

Les installations sont décrites pour les dernières versions disponibles au moment de l'écriture de cet article. Si de nouvelles versions des logiciels sortent entre temps, vous n'aurez vraisemblablement qu'à changer de numéro de version.

## 1.1 Python 3.6.1

Commencez par télécharger le code source mis à disposition sur python.org :

```
$ wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz
```

Vous obtenez un fichier **Python-3.6.1.tar.xz** qu'il faut décompresser :

```
$ tar -xJf Python-3.6.1.tar.xz
```

**NOTE**

Vous pouvez également utiliser l'option **-xf** de **tar** qui détecte automatiquement le format de compression.

Nous passons ensuite à la compilation (n'hésitez pas à faire une pause pendant que l'ordinateur travaille, c'est un peu long...) :

```
$ cd Python-3.6.1
$ ./configure --enable-optimizations --enable-shared
$ make
```

**NOTE**

Vous n'êtes pas obligé d'ajouter l'option **--enable-optimizations** à **configure**. Cette option va accroître le temps de compilation des sources... mais l'exécution de code Python sera 10 à 20% plus rapide (ajoute des optimisations PGO (*Profile Guided Optimization*) et LTO (*Link Time Optimization*)) **[7]**.

**NOTE**

Vous pouvez ajouter l'option **--prefix=<chemin>** si vous souhaitez que les fichiers ne soient pas stockés dans le répertoire par défaut. De même, **--exec-prefix=<chemin>** spécifie un chemin pour l'interpréteur.

Pour l'installation, trois options s'offrent à vous :

1. L'installation brutale :

```
$ sudo make install
```

Avec ce type d'installation, vous allez masquer le Python 3 (commande **python3**) déjà présent sur votre système et qui fonctionnait plutôt bien jusqu'à présent... Si vous êtes joueur, vous pouvez tenter cette commande :-)

2. L'installation alternative :

```
$ sudo make altinstall
```

Vous conserverez votre commande **python3** (qui est un lien vers le Python 3.x inclus dans votre distribution) et vous aurez accès à une commande **python3.6.1** supplémentaire.

3. Sur une distribution **Debian**, vous devriez pouvoir créer un paquetage **.deb** de votre installation et l'installer dans la foulée. Ceci vous permet théoriquement de supprimer très simplement cette version de Python avec le gestionnaire **apt**. Je n'ai malheureusement pas pu aller au bout de cette méthode avec une erreur à la toute fin de l'installation... Je vous donne toutefois la démarche, car si ça fonctionne, c'est à mon avis la meilleure solution.

On commence par installer un paquet supplémentaire :

```
$ sudo apt install checkinstall
```

Puis on appelle **checkinstall** au lieu de **make altinstall** (car **make install** est vraiment à éviter) :

```
$ sudo checkinstall
checkinstall 1.6.2, Copyright
2009 Felipe Eduardo Sanchez
Diaz Duran
            This software is
released under the GNU GPL.

**************************************
** Debian package creation selected **
**************************************

*** Warning: The package name "Python"
contains upper case
*** Warning: letters. dpkg might not
like that so I changed
*** Warning: them to lower case.

This package will be built according
to these values:

0 -  Maintainer: [ root@yggdrasil ]
1 -  Summary: [ Python 3.6.1 ]
2 -  Name:    [ python3.6.1 ]
3 -  Version: [ 3.6.1 ]
4 -  Release: [ 1 ]
5 -  License: [ GPL ]
6 -  Group:   [ checkinstall ]
7 -  Architecture: [ amd64 ]
8 -  Source location: [ Python-3.6.1 ]
9 -  Alternate source location: [  ]
10 - Requires: [  ]
11 - Provides: [ python3.6.1 ]
12 - Conflicts: [  ]
13 - Replaces: [  ]
Enter a number to change any of them
or press ENTER to continue:
```

Vous aurez éventuellement à modifier quelques informations en indiquant un numéro et le contenu de remplacement (j'ai par exemple modifié ici les entrées 2 et 11). Ensuite, normalement, ça doit marcher... chez moi j'obtiens :

```
Makefile:1448: recipe for target
'sharedinstall' failed
make: *** [sharedinstall] Error 1

****  Installation failed.
Aborting package creation.
```

Ou alors une erreur sur **libainstall**... bref, ça ne marche pas !

**NOTE**

Il existe une quatrième possibilité sur Debian qui consiste à activer le dépôt unstable, à créer un fichier **/etc/apt/preferences.d/00source** (ou ce que vous voulez, les fichiers sont lus dans l'ordre alphanumérique) :

```
Package: *
Pin: release a=stable
Pin-Priority: 900

Package: *
Pin: release o=unstable
Pin-Priority: -10
```

Ceci permet de donner par défaut la priorité aux paquets stables. Pour installer un paquet du dépôt unstable, il suffit ensuite de le préciser :

```
$ sudo apt install -t unstable python3.6
```

Mixer les sources est tout à fait possible, mais ne me paraît pas forcément être une solution très raisonnable sur une machine de travail, c'est pourquoi cette solution n'est présentée qu'en note, pour la culture générale...

**NOTE**

Si en lançant python3.6 vous vous rendez compte que vous ne pouvez pas importer **tkinter**, c'est qu'il vous manquait les paquets de développement de Tcl/Tk lors de la compilation :

```
$ python3.6
Python 3.6.1 (default, May 19 2017, 16:29:32)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>> import tkinter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.6/tkinter/__init__.py",
line 36, in <module>
    import _tkinter # If this fails your Python may not
be configured for Tk
ModuleNotFoundError: No module named '_tkinter'
```

La solution est donc très simple :

```
$ sudo apt install tcl-dev tk-dev
```

Avant de relancer une compilation, vérifiez également que vous disposez bien du module **ssl**. Si ce n'est pas le cas, **pip** risque fort de vous ennuyer avec des messages d'erreur :

```
...
pip is configured with locations that require TLS/SSL,
however the ssl module in Python is not available.
```

Pour corriger le problème, installez **libssl-dev** :

```
$ sudo apt install libssl-dev
```

Et vous êtes reparti pour une compilation...

## 1.2 CUDA 8.0

Rendez-vous sur la page du projet CUDA de **Nvidia** et téléchargez le fichier correspondant à votre distribution (pour peu bien entendu que vous possédiez une carte graphique équipée d'un GPU Nvidia...). Pour Debian, on choisira ***Linux***, ***x86_64***, ***Ubuntu***, ***16.04*** (ou la dernière version disponible) puis ***deb (local)***. Cliquez ensuite sur le bouton ***Download*** apparaissant en dessous. Il suffit ensuite d'installer le fichier **cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64.deb** obtenu. Pour cela, nous utiliserons **gdebi** qui permet de gérer automatiquement les problèmes de dépendances :

```
$ sudo apt install gdebi
$ sudo gdebi cuda-repo-ubuntu1604-8-0-
local-ga2_8.0.61-1_amd64.deb
```

## 1.3 PyTorch 0.1.12

L'installation de PyTorch se fait ensuite en une ligne (certes il a fallu installer beaucoup de choses avant...) :

```
$ sudo python3.6 -m pip install http://
download.pytorch.org/whl/cu80/torch-0.1.12.
post2-cp36-cp36m-linux_x86_64.whl
```

Mais normalement, avec **python3.6** vous avez dû compiler et installer **pip3.6**. La commande suivante doit également fonctionner :

```
$ sudo pip3.6 install http://download.
pytorch.org/whl/cu80/torch-0.1.12.post2-
cp36-cp36m-linux_x86_64.whl
```

Vous aurez besoin de numpy :

```
$ sudo pip3.6 install numpy
```

Pour tester, lancez un shell Python et tentez d'importer **torch** :

```
$ python3.6
>>> import torch
```

Il est possible que la réponse soit un message d'erreur, voire que vous n'ayez tout simplement pas pu lancer **python3.6** :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.6/site-
packages/torch/__init__.py", line 53, in
<module>
    from torch._C import *
ImportError: libpython3.6m.so.1.0: cannot
open shared object file: No such file or
directory
```

Si tel est le cas, c'est que vous avez simplement oublié l'option **--enable-shared** lors de la configuration de Python... Mais ça ne fonctionnera pas pour autant ! En effet, après compilation vous risquez fort d'obtenir une nouvelle erreur dès le lancement de **python3.6** :

```
$ python3.6
python3.6: error while loading shared libraries:
libpython3.6m.so.1.0: cannot open shared object
file: No such file or directory
```

Pour comprendre pourquoi la bibliothèque n'est pas trouvée :

```
$ which python3.6
/usr/local/bin/python3.6
$ ldd /usr/local/bin/python3.6
        linux-vdso.so.1 (0x00007fff2ddde000)
        libpython3.6m.so.1.0 => not found
        libpthread.so.0 => /lib/x86_64-linux-
gnu/libpthread.so.0 (0x00007f4f8c4a2000)
        libdl.so.2 => /lib/x86_64-linux-gnu/
libdl.so.2 (0x00007f4f8c29d000)
        libutil.so.1 => /lib/x86_64-linux-gnu/
libutil.so.1 (0x00007f4f8c09a000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.
so.6 (0x00007f4f8bd99000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.
so.6 (0x00007f4f8b9ed000)
        /lib64/ld-linux-x86-64.so.2
(0x00005609685e9000)
```

Vous pouvez donc spécifier le chemin manquant lors du lancement de **python3.6** :

```
$ LD_LIBRARY_PATH=/usr/local/lib python3.6
```

Et il faudra préfixer tous vos appels à **python3.6** et à **pip3.6** de cette déclaration... Une autre solution et de déclarer la variable dans votre **~/.bashrc** :

```
...
export LD_LIBRARY_PATH=/usr/local/lib
```

Pensez à le recharger :

```
$ source ~/.bashrc
```

Vous pouvez maintenant finir l'installation de Torch :

```
$ sudo pip3.6 install torchvision
```

Vous l'aurez constaté, la compilation de Python de manière à pouvoir utiliser Torch n'est pas évidente... Après bien des essais et de nombreuses compilations dont vous pourrez vous inspirer en cas de problème de compilation avec d'autres projets, je vous propose un résumé de la solution la plus simple que j'ai trouvée. Normalement, en la suivant, vous devriez obtenir un Python 3.6 complètement fonctionnel avec PyTorch :

1. Compilation de Python3.6 :

```
$ sudo apt install tcl-dev tk-dev libssl-dev
$ wget https://www.python.org/ftp/python/3.6.1/
Python-3.6.1.tar.xz
$ tar -xJf Python-3.6.1.tar.xz
$ cd Python-3.6.1
$ ./configure --enable-optimizations --enable-shared
$ make
$ sudo make altinstall
```

2. Configuration de **LD_LIBRARY_PATH** dans **~/.bashrc** :

```
...
export LD_LIBRARY_PATH=/usr/local/lib
```

Puis rechargez le fichier :

```
$ source ~/.bashrc
```

3. Installation de Torch :

```
$ sudo pip3.6 install numpy
$ sudo pip3.6 install http://download.pytorch.org/whl/
cu80/torch-0.1.12.post2-cp36-cp36m-linux_x86_64.whl
$ sudo pip3.6 install torchvision
```

4. Test :

```
$ python3.6
>>> import torch
```

Miracle ! Ça marche ! Si ce n'est pas le cas, essayez de préfixer vos commandes par **LD_LIBRARY_PATH=/usr/local/lib** comme dans **sudo LD_LIBRARY_PATH=/usr/local/lib pip3.6 install numpy**.

**NOTE**

Toute cette partie détaille essentiellement la configuration et la compilation de Python pour pouvoir utiliser Torch. Il m'a semblé indispensable de bien expliquer les différents problèmes que l'on peut rencontrer dans cette phase, ayant eu la « chance » de tous les tester, ou presque... J'espère que mon expérience vous fera économiser de nombreuses heures de compilation et de recherche.

# 2. PASSONS AUX CHOSES SÉRIEUSES

Nous allons utiliser PyTorch pour créer un réseau de neurones, notion déjà abordée **[2][8]** et sur laquelle je ne reviendrai pas pour me concentrer sur l'aspect pratique. L'exemple détaillé est basé sur l'un des nombreux tutoriels disponibles sur le site de PyTorch **[9]**.

## 2.1 Charger le jeu d'images d'entraînement

Il existe des jeux d'images mis à disposition pour pouvoir entraîner et tester simplement des réseaux de neurones. Parmi ces jeux d'images, on trouve notamment les jeux CIFAR-10 et CIFAR-100 :

- CIFAR-10 est un ensemble de 60 000 images en couleur de taille 32 x 32 pixels :
  - → 10 classes : *airplane* (avion), *automobile* (voiture), *bird* (oiseau), *cat* (chat), *deer* (cerf), *dog* (chien), *frog* (grenouille), *horse* (cheval), *ship* (bateau), et *truck* (camion) ;
  - → 50 000 images dans le jeu d'entraînement (donc classées) ;
  - → 10 000 images dans le jeu de test.
- CIFAR-100 est le même ensemble d'images que le précédent, mais avec une classification plus pointue en 20 superclasses raffinées en 100 classes. Il serait trop long de toutes les énumérer ici, mais on y trouve par exemple :
  - → *aquatic mammals* (mammifères aquatiques) : *beaver* (castor), *dolphin* (dauphin), *otter* (loutre), *seal* (phoque), et *whale* (baleine) ;
  - → *fish* (poissons) : *aquarium fish* (poisson d'aquarium), *flatfish* (poisson plat), *ray* (raie), *shark* (requin), et *trout* (truite).
  - → etc.

Le reste des classes est disponible sur **[10]**.

Nous allons charger le jeu CIFAR-10, le normaliser et afficher quelques images issues de l'ensemble d'entraînement ainsi que la classe qui leur est associée. Pour cela, nous aurons besoin de **matplotlib** :

```
$ sudo pip3.6 install matplotlib
```

Passons maintenant au code avec la définition d'une classe **CIFAR10** (fichier **CIFAR10.py**) permettant de lire les données CIFAR-10 :

```
01: import torch
02: import torchvision
03: import torchvision.transforms as
transforms
04:
05:
06: class CIFAR10:
07:     classes = ('Avion', 'Voiture',
'Oiseau', 'Chat',
```

```
08:             'Cerf', 'Chien', 'Crapaud', 'Cheval',
'Bateau', 'Camion')
09:
10:     def __init__(self):
11:         self.transform = transforms.Compose(
12:                             [transforms.ToTensor(),
13:                              transforms.Normalize((0.5,
0.5, 0.5), (0.5, 0.5, 0.5))])
14:
15:         self.trainset = torchvision.datasets.
CIFAR10(root='./data', train=True,
16:                                             download=True,
transform=self.transform)
17:         self.testset = torchvision.datasets.
CIFAR10(root='./data', train=False,
18:                                            download=True,
transform=self.transform)
19:
20:     def loadTrainSet(self, size=4):
21:         return torch.utils.data.DataLoader(self.
trainset, batch_size=size,
22:                           shuffle=True, num_workers=2)
23:
24:     def loadTestSet(self, size=4):
25:         return torch.utils.data.DataLoader(self.
testset, batch_size=size,
26:                           shuffle=False, num_workers=2)
```

Dans cette classe, outre les divers imports de PyTorch des lignes 1 à 3, nous définissons un attribut de classe **classes** qui contient le nom de toutes les classes associées aux images (les classes étant indicées, **0** correspond à **'airplane'**, mais nous pouvons afficher **'Avion'**). La classe **Grenouille** a été rebaptisée **Crapaud**... de manière à utiliser moins de lettres et à ne pas déborder de l'image lors de l'insertion du label sur une image, tout en conservant une police lisible.

Dans le constructeur (lignes 10 à 18), nous définissons un attribut qui est une transformation ou plus précisément la composition (**transforms.Compose()**) de deux transformations **[11]** :

- **ToTensor()** pour convertir une matrice (**h x w x c**) représentant une image dont les points se trouvent dans l'intervalle **[0, 255]** en matrice **FloatTensor (c x h x w)** où les points sont dans **[0.0, 1.0]** ;
- **Normalize(mean, std)** pour normaliser les points suivant la formule **(point - mean) / std**.

Pour y voir un petit peu plus clair, je vous propose quelques manipulations directement dans un shell Python en commençant par créer une petite matrice **3 x 3 x 3** que nous remplissons de manière aléatoire d'entiers compris entre **0** et **255** :

```
>>> import random
>>> import numpy as np
>>> a = np.zeros((3, 3, 3))
>>> for x in range(3):
...     for y in range(3):
...         for z in range(3):
...             a[x, y, z] =
random.randint(0, 255)
...
>>> a
array([[[   6.,  180.,  137.],
        [  19.,   15.,  248.],
        [ 220.,  228.,  255.]],

       [[ 155.,  136.,  109.],
        [  20.,   38.,   61.],
        [ 235.,  201.,    4.]],

       [[ 187.,  217.,  119.],
        [ 244.,   17.,  200.],
        [ 190.,   73.,  213.]]])
```

**NOTE**

Nous aurions pu utiliser la compréhension de liste pour créer notre matrice :

```
>>> a = [ random.randint(0,
255) for x in range(3) for y in
range(3) for z in range(3) ]
```

Nous allons maintenant la transformer en tenseur à l'aide de **ToTensor()** :

```
>>> import torchvision.transforms as
transforms
>>> to_tensor = transforms.ToTensor()
>>> t = to_tensor(a)
>>> t

(0 ,.,.) =
  0.0235  0.0745  0.8627
  0.6078  0.0784  0.9216
  0.7333  0.9569  0.7451

(1 ,.,.) =
  0.7059  0.0588  0.8941
  0.5333  0.1490  0.7882
  0.8510  0.0667  0.2863

(2 ,.,.) =
  0.5373  0.9725  1.0000
  0.4275  0.2392  0.0157
  0.4667  0.7843  0.8353
[torch.FloatTensor of size 3x3x3]
```

Vous noterez ici l'appel particulier à **ToTensor()**. En fait, ToTensor est défini comme une classe possédant une méthode **__call__()**, en faisant un objet *callable* ou « appelable ». Cela signifie que

l'on peut appeler une instance de l'objet sous forme de fonction, ce qui lancera la méthode **__call__()**. Ici **to_tensor** est une instance de **transforms.ToTensor()** et lorsque nous appelons **to_tensor(a)**, cela déclenche l'exécution de la méthode **__call__()** (voir encadré pour plus d'explications).

Au niveau de la création du tenseur, il ne s'agit que d'une transposition **(2, 0, 1)** avec une normalisation en divisant tous les éléments par **255** :

```
>>> aT = np.transpose(a, (2, 0, 1))
>>> aT
array([[[   6.,   19.,  220.],
        [ 155.,   20.,  235.],
        [ 187.,  244.,  190.]],

       [[ 180.,   15.,  228.],
        [ 136.,   38.,  201.],
        [ 217.,   17.,   73.]],

       [[ 137.,  248.,  255.],
        [ 109.,   61.,    4.],
        [ 119.,  200.,  213.]]])
>>> aT / 255
array([[[ 0.02352941,  0.0745098 ,  0.8627451 ],
        [ 0.60784314,  0.07843137,  0.92156863],
        [ 0.73333333,  0.95686275,  0.74509804]],

       [[ 0.70588235,  0.05882353,  0.89411765],
        [ 0.53333333,  0.14901961,  0.78823529],
        [ 0.85098039,  0.06666667,  0.28627451]],

       [[ 0.5372549 ,  0.97254902,  1.        ],
        [ 0.42745098,  0.23921569,  0.01568627],
        [ 0.46666667,  0.78431373,  0.83529412]]])
```

On voit bien que la première colonne de **a[0]** devient la première ligne de **aT[0]**, la deuxième colonne de **a[0]** devient la première ligne de **aT[1]**, etc. Après normalisation dans **[0, 1]** en divisant par **255**, on retrouve les mêmes valeurs que dans le tenseur.

La normalisation effectuée à l'aide de **Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))** fait elle aussi intervenir un *callable* :

```
>>> normal = transforms.Normalize((0.5, 0.5,
0.5), (0.5, 0.5, 0.5))
>>> normal(t)

(0 ,.,.) =
 -0.9529 -0.8510  0.7255
  0.2157 -0.8431  0.8431
  0.4667  0.9137  0.4902

(1 ,.,.) =
  0.4118 -0.8824  0.7882
  0.0667 -0.7020  0.5765
  0.7020 -0.8667 -0.4275

(2 ,.,.) =
  0.0745  0.9451  1.0000
 -0.1451 -0.5216 -0.9686
 -0.0667  0.5686  0.6706
[torch.FloatTensor of size 3x3x3]
```

## LES OBJETS « CALLABLES » EN PYTHON

**En Python, il est possible de créer des objets *callables*, c'est-à-dire des objets dont une instance peut-être appelée pour déclencher une fonction. Pour réaliser cela, on utilise une méthode spéciale appelée `__call__()`. Voici un petit exemple avec un objet `TestCallable` possédant une méthode `print()` qui affiche un message et une méthode `__call__()` qui permet d'appeler une fonction sommant deux entiers (oui, l'exemple est très simple...) :**

```
>>> class TestCallable:
...     def __init__(self):
...         print('Creation de l\'instance...')
...     def print(self):
...         print('Coucou!')
...     def __call__(self):
...         return lambda x,y : x + y
...
>>> t = TestCallable()
Creation de l'instance...
>>> t.print()
Coucou!
>>> fct = t()
>>> fct(1, 2)
3
>>> t()(1, 2)
3
```

**Les deux derniers appels sont identiques, mais le premier est quand même plus lisible que le second...**

Nous avons vu que **Normalize** appliquait la formule **(point - mean) / std** où **mean** et **std** sont les paramètres de la fonction (attention, le tenseur est modifié après appel de la fonction). Effectuons manuellement le calcul pour **t[0, 0, 0]** (**0.0235294122248888**) :

```
>>> (t[0, 0, 0] - 0.5) / 0.5
-0.9529411755502224
```

On retrouve bien la valeur de **t[0, 0, 0]** après normalisation.

Revenons à la classe **CIFAR10** et aux attributs **trainSet** et **testSet** des lignes 15 à 18. On peut voir que les données seront téléchargées si elles ne sont pas présentes (**download=True**), qu'elles seront placées dans **./data** (paramètre **root**), et que nous appliquerons la transformation déclarée précédemment dans les lignes 11 à 13 (paramètre **transform**). Le paramètre **train** permet d'indiquer quel type de données on souhaite utiliser : pour **True**, il s'agit des données d'apprentissage et pour **False** des données de test **[12]**.

Les méthodes **loadTrainSet()** et **loadTestSet()** des lignes 20 à 26 permettent de véritablement charger les données et de les placer dans les attributs créés précédemment. On peut spécifier le nombre d'images à charger dans le paramètre **size**. Les valeurs des autres paramètres sont fixées directement dans le corps des méthodes (données mélangées avec **shuffle**, nombre de processus utilisés avec **num_workers**, etc. **[13]**).

Nous utilisons ensuite la classe **CIFAR10** pour lire les données et afficher 16 images de manière aléatoire sous la forme d'une mosaïque. Nous collons le label associé aux images pardessus chacune des images :

```
01: from CIFAR10 import CIFAR10
02: import torchvision
03: import matplotlib.pyplot as plt
04: import numpy as np
05:
06:
07: def imshow(img):
08:     img = img / 2 + 0.5
09:     npimg = img.numpy()
10:     plt.imshow(np.transpose(npimg,
(1, 2, 0)))
11:
12:
13: if __name__ == '__main__':
14:     data = CIFAR10()
15:
16:     # Récupération des données
17:     dataiter = iter(data.
loadTrainSet(size=16))
18:     images, labels = dataiter.next()
19:
20:     # Affichage de la mosaïque
21:     plt.axis('off')
22:     imshow(torchvision.utils.make_
grid(images, nrow=4))
23:     for row in range(4):
24:         for col in range(4):
25:             plt.text(2 + 34 * col, 8
+ 34 * row, CIFAR10.classes[labels[row *
4 + col]], fontsize=15, color='yellow')
26:     plt.show()
```

La fonction **imshow()** des lignes 7 à 10 permet de « corriger » les couleurs (en ligne 8) pour un affichage visible par un être humain puis on convertit et on transpose les données pour les afficher (lignes 9 et 10). Le chargement des données d'entraînement se fait par la classe **CIFAR10** (lignes 14 et 17) et l'on place les images dans la variable **images** et les étiquettes dans **labels** (ligne 18). Au niveau de l'affichage de la mosaïque, on désactive les axes (ligne 21), on affiche les images avec **imshow()** (ligne 22) puis on place les étiquettes sur chaque image (lignes 23 à 25) avant de rendre la fenêtre visible (ligne 26).

**NOTE**

Les images faisant 32 x 32 pixels, il est facile de déduire l'emplacement des étiquettes pour chaque image. Les coordonnées de **plt.text()** en ligne 25 tiennent compte des marges d'où le calcul en **(2 + 34 * col, 8 + 34 * row)**.

À l'exécution, ce programme génère une fenêtre semblable à celle présentée en figure 1.


*Fig. 1 : Mosaïque de 16 images de l'ensemble d'entraînement sur lesquelles sont superposés (en jaune) les labels qui leur sont associés. La définition des images étant de 32 x 32 pixels, le rendu est forcément pixelisé.*

## 2.2 Création du réseau de neurones, apprentissage et prédiction !

Nous allons maintenant créer notre réseau de neurones grâce à **pytorch.nn**. Pour cela, il faut écrire une classe (appelée ici **Net**) dans laquelle nous spécifierons les paramètres du réseau et nous définirons une méthode **forward()**. Je

partirai ici de la classe donnée dans le tutoriel de PyTorch que nous utiliserons tel quel... mais que nous allons expliquer, contrairement au dit tutoriel !

```
01: from torch.autograd import Variable
02: import torch.nn as nn
03: import torch.nn.functional as F
04:
05:
06: class Net(nn.Module):
07:     def __init__(self):
08:         super(Net, self).__init__()
09:         self.conv1 = nn.Conv2d(3, 6, 5)
10:         self.pool = nn.MaxPool2d(2, 2)
11:         self.conv2 = nn.Conv2d(6, 16, 5)
12:         self.fc1 = nn.Linear(16 * 5 * 5, 120)
13:         self.fc2 = nn.Linear(120, 84)
14:         self.fc3 = nn.Linear(84, 10)
15:
16:     def forward(self, x):
17:         x = self.pool(F.relu(self.conv1(x)))
18:         x = self.pool(F.relu(self.conv2(x)))
19:         x = x.view(-1, 16 * 5 * 5)
20:         x = F.relu(self.fc1(x))
21:         x = F.relu(self.fc2(x))
22:         x = self.fc3(x)
23:         return x
```

Nous avons ici une classe **Net** qui hérite de **nn.Module** et qui contient un constructeur et une méthode **forward()** qui sera exécutée à chaque appel pour effectuer les calculs.

Commençons par le constructeur. Comme la classe **Net** est une classe fille, elle appelle le constructeur de sa classe mère en ligne 8. Ensuite, dans les lignes 9 à 14 nous trouvons la définition de six paramètres à propos desquels nous ignorons tout... En fait, il s'agit simplement du mécanisme mis en place pour ajouter des sous-modules à un modèle : on définit un attribut et celui-ci sera enregistré en tant que sous-module **[15]**. Voyons donc maintenant quels sous-modules ont été ajoutés au modèle :

- **nn.Conv2d(3, 6, 5)** : applique une convolution 2d avec **3** canaux en entrée (notre image 2d), **6** canaux en sortie et un noyau de convolution de taille **5** **[16]**. Il n'y a rien d'étonnant ici à utiliser un réseau de neurones à convolution (perceptron multicouche inspiré du comportement du cortex visuel des vertébrés) puisque ce type de réseau est particulièrement performant en reconnaissance d'images... et c'est un peu ce que nous cherchons à faire ! La taille du noyau permet de définir la profondeur de la couche de neurones (le nombre de neurones liés à la même entrée) ;
- **nn.MaxPool2d(2, 2)** : applique une compression 2d avec un noyau de **2** et un pas de **2** **[17]**. Ce sous-module permet de réduire la taille de l'image intermédiaire en effectuant un sous-échantillonnage (l'image est découpée en de multiples tuiles qui ne se chevauchent pas comme le montre la figure 2) ;
- **nn.Conv2d(6, 16, 5)** : encore une convolution 2d, mais avec des paramètres différents ; les entrées seront issues de la première convolution d'où le **6**) ;
- **nn.Linear(16 * 5 * 5, 120)**, **nn.Linear(120, 84)** et **nn.Linear(84, 10)** : enchaînement de transformations linéaires ; on part d'une taille de données de **16 * 5 * 5** en entrée pour ressortir en **120** puis **84** et enfin **10**. Il s'agit du travail de la couche de correction souvent appelée ReLU (pour *Rectified Linear Unit*).

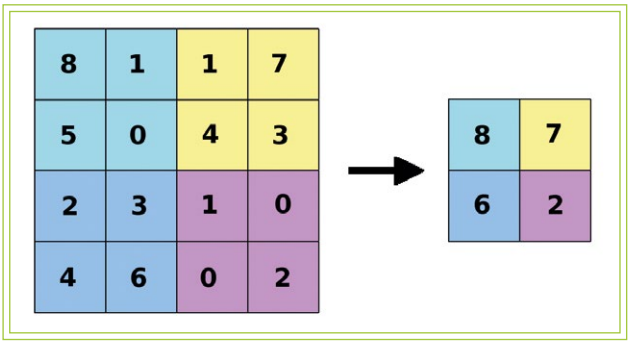



*Fig. 2 : Pooling avec un filtre 2 x 2 et un pas de 2.*

Tous ces sous-modules référencés dans le constructeur sont ensuite employés dans la fonction **forward()**. Les appels à **self.pool(F.relu())** font référence à **self.pool**, c'est-à-dire **nn.MaxPool2d(2, 2)**, à laquelle on transmet en paramètre **F.relu()** qui est la fonction d'activation non linéaire **relu()** de **torch.nn.functional** (la documentation est assez exceptionnelle sur cette fonction puisque bien que référencée, il n'y a absolument aucune description **[16]**). Cette fonction est appelée une première fois sur le résultat de **self.conv1()** appliqué à la donnée **x** puis sur **self.conv2()** appliqué au résultat précédent.

La fonction **view()** de la ligne 19 permet simplement de modifier la structure du tenseur (comme un **reshape()** numpy). Voici un exemple d'application dans un script shell :

```
>>> import torch
>>> a = torch.arange(0, 16)
>>> a

  0
  1
  2
  3
  4
```

```
 5
 6
 7
 8
 9
10
11
12
13
14
15
[torch.FloatTensor of size 16]

>>> a = a.view(4, 4)
>>> a

 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
[torch.FloatTensor of size 4x4]
```

Si l'on désire fixer le nombre de colonnes et que le nombre de lignes s'adapte automatiquement, on peut spécifier la valeur **-1** (la réciproque fonctionne également). En reprenant notre exemple et en fixant le nombre de colonnes à **2**, nous obtiendrons automatiquement 8 lignes :

```
>>> a = a.view(-1, 2)
>>> a

 0  1
 2  3
 4  5
 6  7
 8  9
10 11
12 13
14 15
[torch.FloatTensor of
size 8x2]
```

Le **view()** de la méthode **forward()** fixe le nombre de colonnes à **16 x 5 x 5**, ce qui correspond à la taille en entrée de **fc1()**. Dans les lignes suivantes, on enchaîne ensuite les appels à **fc1()**, **fc2()** puis **fc3()** (lignes 20 à 22).

On va ensuite utiliser la classe **Net** et modifier notre programme **display_images.py** pour ajouter l'apprentissage et mettre en évidence dans notre mosaïque (prise cette fois sur les données de test) les étiquettes prédites erronées :

```
01: from CIFAR10 import CIFAR10
02: import torchvision
03: import matplotlib.pyplot as plt
04: import numpy as np
05: import torch.optim as optim
06: import torch.nn as nn
07: from torch.autograd import Variable
08: from Net import Net
09: import torch
10:
11:
12: def imshow(img):
13:     img = img / 2 + 0.5
14:     npimg = img.numpy()
15:     plt.imshow(np.transpose(npimg, (1, 2, 0)))
16:
17:
18: if __name__ == '__main__':
19:     # Chargement des données CIFAR10
20:     data = CIFAR10()
21:     trainSet = data.loadTrainSet(size=4)
22:     testSet = data.loadTestSet(size=16)
23:
24:     # Création du réseau de neurones
25:     net = Net()
26:     criterion = nn.CrossEntropyLoss()
27:     optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
28:
29:     # Apprentissage
30:     for epoch in range(2):
31:         running_loss = 0.0
32:         for i, Data in enumerate(trainSet, 0):
33:             inputs, labels = Data
34:             inputs, labels = Variable(inputs), Variable(labels)
35:             optimizer.zero_grad()
36:
37:             outputs = net(inputs)
38:             loss = criterion(outputs, labels)
39:             loss.backward()
40:             optimizer.step()
41:
42:             running_loss += loss.data[0]
43:             if i % 2000 == 1999:
44:                 print('[%d, %5d] loss: %.3f' %
45:                       (epoch + 1, i + 1, running_loss / 2000))
46:                 running_loss = 0.0
47:     print('Fin de l\'apprentissage')
48:
49:
50:     # Récupération des données de test
51:     dataiter = iter(testSet)
52:     images, labels = dataiter.next()
53:
54:     # Affichage de la mosaïque
55:     plt.axis('off')
56:     imshow(torchvision.utils.make_grid(images, nrow=4))
57:     for row in range(4):
58:         for col in range(4):
59:             plt.text(2 + 34 * col, 8 + 34 * row, CIFAR10.
classes[labels[row * 4 + col]], fontsize=15, color='yellow')
```

```
60:
61:     # Calcul des prédictions et affichage des
erreurs
62:     outputs = net(Variable(images))
63:     _, predicted = torch.max(outputs.data, 1)
64:     for row in range(4):
65:         for col in range(4):
66:             if CIFAR10.classes[labels[row * 4 +
col]] != CIFAR10.classes[predicted[row * 4 + col][0]]:
67:                 plt.text(2 + 34 * col, 24 + 34 *
row, CIFAR10.classes[predicted[row * 4 + col][0]],
fontsize=15, color='red')
68:
69:     plt.show()
```

Nous avons ajouté ici la création du réseau de neurones dans les lignes 24 à 27, l'apprentissage dans les lignes 29 à 47 et la récupération des données de tests (lignes 50 à 52) puis la prédiction (lignes 54 à 69).

Pour la création du réseau, on utilise la classe **Net** définie précédemment (ligne 24) ainsi qu'un critère de classification (ligne 25) **[20]**, et un algorithme d'optimisation (ligne 26) pour lequel vous obtiendrez plus d'informations sur **[21]**. L'apprentissage va permettre de parcourir les données de **trainSet** (ligne 32) et de les associer au réseau (lignes 37 à 40). Enfin, l'affichage des lignes 54 à 69 reste classique avec toutefois la récupération des prédictions du réseau en ligne 63.

À l'exécution nous obtenons une fenêtre similaire à celle présentée en figure 3. Sur ce jeu d'exemple, nous obtenons 4 erreurs sur 16 images.


*Fig. 3 : Affichage d'une mosaïque de 16 images avec les étiquettes associées (en jaune l'étiquette du jeu de test et, lorsqu'elle est différente, en rouge l'étiquette prédite).*

# 3. ACCÉLÉRATION AVEC CUDA

Et si l'on veut utiliser la puissance du GPU pour accélérer le traitement ? Vous vous dites qu'il va falloir sans doute réécrire une bonne partie du code... Eh bien non, et c'est là l'une des forces de PyTorch ! Nous allons simplement indiquer que notre réseau doit utiliser CUDA et le tour sera joué. Pour cela, nous ne modifions que deux lignes :

```
...
24:     # Création du réseau de neurones
25:     net = Net()
26:     net.cuda()
...
30:     # Apprentissage
31:     for epoch in range(2):
32:         running_loss = 0.0
33:         for i, Data in enumerate(trainSet, 0):
34:             inputs, labels = Data
35:             inputs, labels = Variable(inputs.
cuda()), Variable(labels.cuda())
...
```

Bien entendu, il est préférable de disposer d'un driver NVIDIA :

```
$ LD_LIBRARY_PATH=/usr/local/lib python3.6
display_images.py
Files already downloaded and verified
Files already downloaded and verified
...
AssertionError:
Found no NVIDIA driver on your system. Please
check that you
have an NVIDIA GPU and installed a driver from
http://www.nvidia.com/Download/index.aspx
```

En effet, l'installation précédente de CUDA n'a pas installé automatiquement les drivers NVIDIA ! S'ils n'étaient pas présents, c'est à vous de les installer (trop de paramètres entrent en jeu pour pouvoir décrire ici une installation « standard »).

# 4. ET POUR UTILISER SON PROPRE JEU DE DONNÉES ?

N'ayant pas de jeux de données suffisamment importants pour réaliser une réelle expérimentation, je ne donnerai ici que des pistes permettant d'utiliser vos données.

Tout d'abord, comme il y a de grandes chances pour que vos données se trouvent dans des fichiers structurés (ou soient référencées par de tels fichiers), n'oubliez pas qu'il existe des modules pour vous faciliter la vie. Si vous devez lire un fichier xml, **xml.etree.ElementTree [22]** sera votre allié, s'il s'agit d'un fichier csv, pensez à **pandas** et sa fonction **read_csv()**, etc.

**ATTENTION !**

Si vous installez un nouveau module, rappelez-vous qu'il faut utiliser **pip3.6**...

Pour créer un jeu de données (un *dataset*), il va falloir créer une nouvelle classe qui hérite justement de **Dataset** :

```
import torch.utils.data

class DatasetPerso(torch.utils.
data.Dataset):
    def __init__(self, images,
labels):
        self.images = images
        self.labels = labels
        assert len(images) ==
len(labels)

    def __getitem__(self, index):
        return self.
images[index], labels[index]

    def __len__(self):
        return len(self.labels)
```

Vous aurez certainement d'autres problèmes à régler par la suite (sinon ça ne serait pas drôle...), mais vous avez déjà la façon de charger les données (qu'il faudra convertir en tenseurs, etc.). N'oubliez pas également que plus vous aurez de données, plus l'apprentissage permettra une prédiction pertinente. Inutile donc de vous lancer avec quelques centaines d'images par exemple.

## CONCLUSION

Dans cet article, nombre d'explications théoriques ont été volontairement passées sous silence. La masse d'informations pratiques permettant d'utiliser PyTorch et même simplement de pouvoir faire tourner l'un des exemples (certes modifié) de la documentation s'est avéré tellement importante qu'il aurait pratiquement fallu écrire un hors-série pour pouvoir tout traiter. J'espère toutefois avoir bien dégrossi le travail avec cet article et que cela vous permettra d'employer PyTorch avec plus de facilité ! ■

## RÉFÉRENCES


**[1]** Tensorflow : https://www.tensorflow.org/

**[2]** COLOMBO T., *« Apprentissage supervisé à l'aide de réseaux de neurones »*, *GNU/Linux Magazine n°198*, novembre 2016, p. 22 à 32 : http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-198/Apprentissage-supervise-a-l-aide-de-reseaux-de-neurones

**[3]** CNTK : https://github.com/Microsoft/CNTK

**[4]** Theano : http://www.deeplearning.net/software/theano/

**[5]** Caffe : http://caffe.berkeleyvision.org/

**[6]** Torch : http://torch.ch/

**[7]** README.rst de Python 3.6.1 : https://github.com/python/cpython/blob/3.6/README.rst

**[8]** MATTE C., *« Transfert de style : et si Van Gogh peignait Tux ? »*, *GNU/Linux Magazine n°202*, mars 2017, p. 12 à 22 : http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-202/Transfert-de-style-et-si-Van-Gogh-peignait-Tux

**[9]** Tutoriaux PyTorch : http://pytorch.org/tutorials

**[10]** Classes de CIFAR-10 et CIFAR-100 : https://www.cs.toronto.edu/~kriz/cifar.html

**[11]** Documentation de torchision.transforms : http://pytorch.org/docs/torchvision/transforms.html

**[12]** Chargement des données CIFAR : http://pytorch.org/docs/torchvision/datasets.html#cifar

**[13]** Chargement des données avec DataLoader() : http://pytorch.org/docs/data.html#torch.utils.data.DataLoader

**[14]** Création d'une mosaïque avec make_grid() : http://pytorch.org/docs/torchvision/utils.html#torchvision.utils.make_grid

**[15]** Documentation sur la création d'un réseau de neurones par nn.Module : http://pytorch.org/docs/nn.html#torch.nn.Module

**[16]** Convolution 2d : http://pytorch.org/docs/nn.html#torch.nn.Conv2d

**[17]** MaxPool2d : http://pytorch.org/docs/nn.html#torch.nn.MaxPool2d

**[18]** Linear layers : http://pytorch.org/docs/nn.html#torch.nn.Linear

**[19]** Fonction d'activation non linéaire relu() : http://pytorch.org/docs/nn.html#torch.nn.functional.relu

**[20]** Critère de classification CrossEntropyLoss : http://pytorch.org/docs/nn.html#torch.nn.CrossEntropyLoss

**[21]** Algorithme d'optimisation SGD : http://pytorch.org/docs/optim.html#torch.optim.SGD

**[22]** Lecture de fichiers xml : https://docs.python.org/3.6/library/xml.etree.elementtree.html

# AMAZON <3 LINUX

ÉMILE 'IMIL' HEITOR
[Pour GCU, le jardin magique depuis fouyayaaa]

MOTS-CLÉS : API AMAZON, AWSCLI, AWS, PYTHON, BOTO3

**Qui d'entre vous n'a jamais entendu parler de AWS (Amazon Web Services), LE cloud public qui domine le marché, la panacée du DevOps, le destructeur de toute une industrie, l'hébergeur qui soutient désormais un bon 40% du Web mondial ? Si tu réponds « moi » à cette question, tu devrais visiter autre chose que des sites en HTML v2 et cesser d'éditer tes fichiers de conf à la main.**

La triple pique d'entrée de jeu c'est gratos, ça me fait plaisir. AWS, c'est le rouleau compresseur, et cette position de leader n'est pas liée à la chance, puisque comme nombre de solutions disruptives, les analystes les plus pointus proféraient à son lancement que « ça marchera jamais » ©®™ ; non, le succès de ce cloud réside en grande partie dans son API, ses outils, le plus souvent libres, et la puissante communauté qui s'est construite autour. Ce que je vous propose dans cet article, c'est de faire connaissance avec ces outils, et en particulier l'AWS cli **[1]** basée sur boto3 **[2]**, tous les deux sous licence Apache 2 **[3]**.

## 1. NO CREDIT CARD REQUIRED*

*Bon en fait si. Pour accéder au service AWS, même si vous n'utilisez que les services gratuits, il faudra créer un compte et y associer une carte bleue. Pour les besoins de cet article, aucun élément ne devrait vous coûter quoi que ce soit, tout au plus si vous souhaitez installer un serveur sur une instance et vérifier son fonctionnement, il vous en coûtera quelques poussières d'euros.

La première étape de notre périple suppose donc de créer un compte sur la plateforme **Amazon Web Services [4]** (voir figure 1).

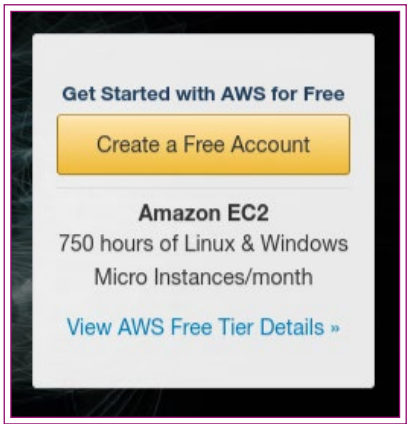



*Fig. 1 : Clique ici pour élargir tes connaissances.*

Deux *best practices* à suivre absolument pour éviter les mauvaises surprises :

1. Le compte que vous allez créer va générer un utilisateur **root**, l'administrateur de votre compte, et comme sur un système **UNIX**, ce compte est à utiliser avec discernement, comprendre que la première action que vous allez mener avec ce compte, et probablement une des seules, c'est de vous créer un utilisateur muni de droits élevés. Pour cela, rendez-vous dans la section IAM **[5]** (*Identity and Access Management*) et dansez le Mia en cliquant sur le menu ***Users*** (voir figure 2).

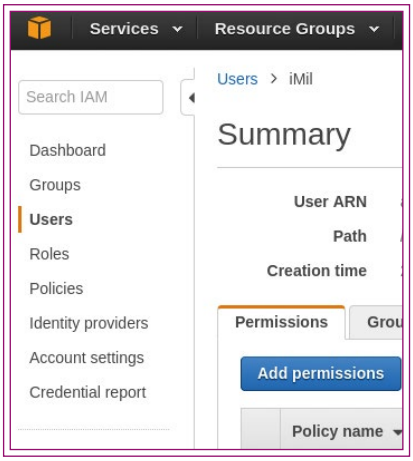



*Fig. 2 : Useradd version AWS.*

Là vous vous assurerez d'avoir bien coché l'accès « programmatique », c'est grâce à cette clé que vous pourrez interagir depuis la CLI ainsi que le SDK **boto3 [2]** (voir figure 3).

Pour les besoins de notre article, vous attribuerez à l'utilisateur créé la *policy Administrator Access* (voir figure 4).

Et veillez bien à sauvegarder votre *access key ID* ainsi que la *secret access key*, nous en aurons besoin pour configurer la CLI.

Pour parfaire la sécurité de vos accès, il est **fortement** conseillé d'activer la MFA (*Multi Factor Authentication*) et d'utiliser ainsi une des applications prévues à cet effet **[6]** sur votre smartphone pour obtenir des clés éphémères.

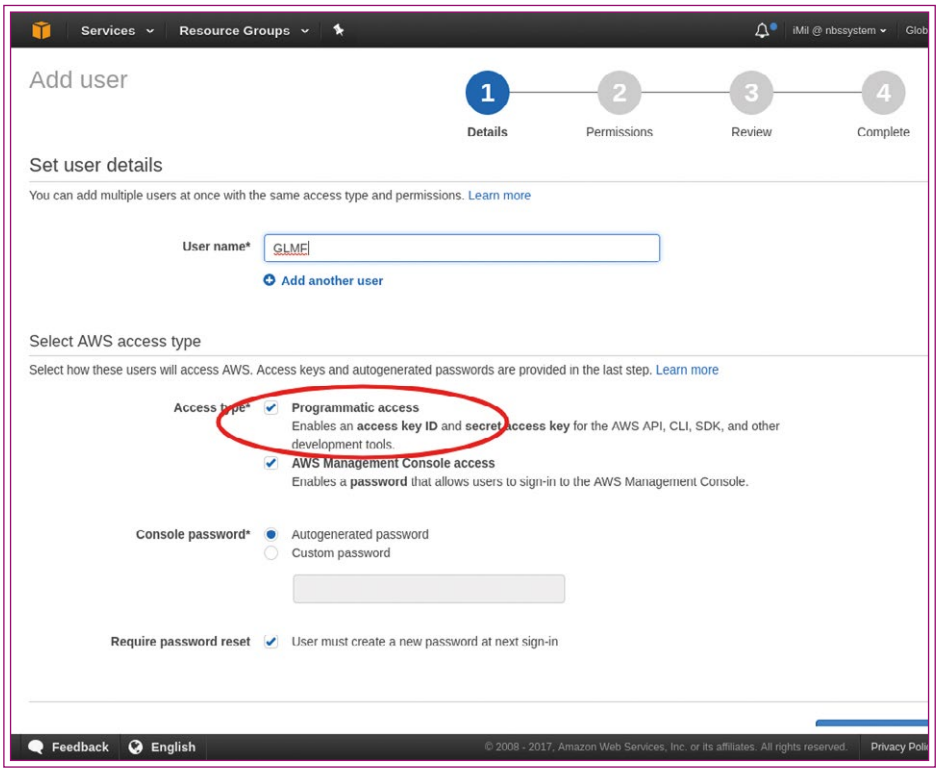



*Fig. 3 : Pour parler à l'oreille du Cloud, clique ici.*

2. Au long de cet article, nous allons créer des *ressources AWS*, et bien que nous allons nous efforcer de ne choisir que des *ressources* éligibles au **Free Tier [7]** (donc gratuites), il sera judicieux de s'assurer que les éléments inutilisés sont bien détruits au fur et à mesure. Nous tâcherons de faire le ménage.

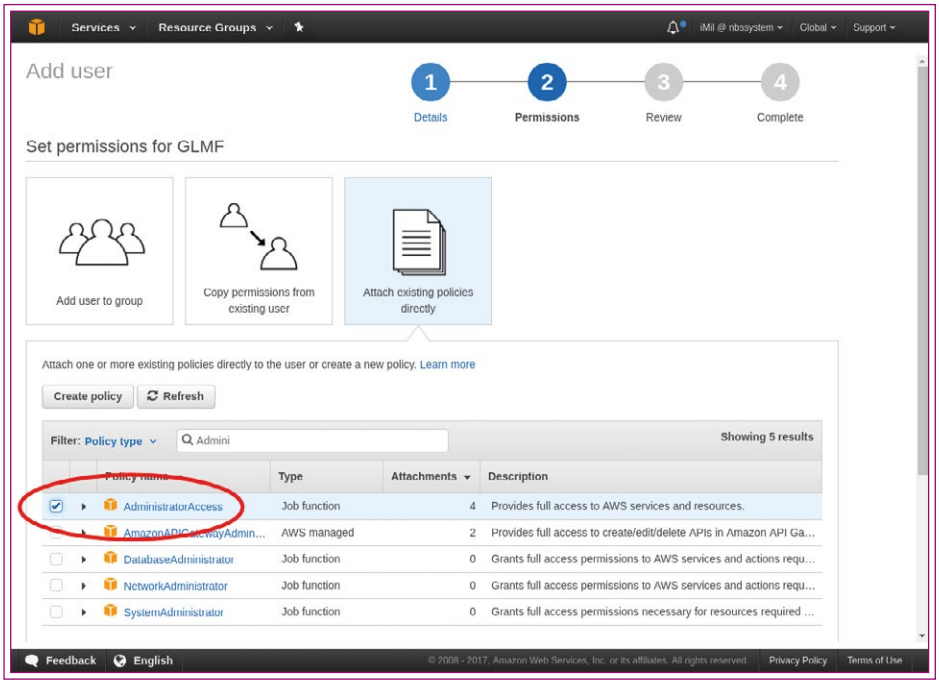



*Fig. 4 : sudo -s.*

## 2. DONNE-MOI TON SHELL BABE

Assez de *weberies*, nous sommes désormais en possession de tous les éléments nécessaires pour interagir avec AWS à l'aide de notre bien aimée ligne de commandes.

La première chose à faire, c'est d'installer **awscli [1]** afin de nous donner accès à l'API Amazon. Afin d'utiliser les versions les plus à jour et ne pas semer le chaos dans vos paquets **Python** déjà installés, nous utiliserons un virtualenv Python :

```
$ sudo apt-get install virtualenv
$ virtualenv -p /usr/bin/python3 ve/glmfaws
Already using interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/imil/ve/glmfaws/bin/python3
Also creating executable in /home/imil/ve/glmfaws/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
```

Et installons la fameuse awscli **[1]** :

```
$ pip install awscli
Collecting awscli
[...]
Installing collected packages: docutils, python-
dateutil, jmespath, botocore, colorama, pyasn1,
rsa, PyYAML, s3transfer, awscli
Successfully installed PyYAML-3.12 awscli-1.11.85
botocore-1.5.48 colorama-0.3.7 docutils-0.13.1
jmespath-0.9.2 pyasn1-0.2.3 python-dateutil-2.6.0
rsa-3.4.2 s3transfer-0.1.10
```

Et voici notre premier contact avec la CLI :

```
$ aws
usage: aws [options] <command> <subcommand>
[<subcommand> ...] [parameters]
To see help text, you can run:

  aws help
  aws <command> help
  aws <command> <subcommand> help
aws: error: the following arguments are
required: command
```

La première et la plus importante chose à savoir, c'est que l'option **help** est disponible sur tous les niveaux d'exécution et présente un format **man** très complet et muni d'exemples concrets. Ensuite, bien comprendre que le premier niveau de commandes correspond au service AWS ciblé. Par exemple, pour interagir avec **EC2 [8]**, on tapera :

```
$ aws ec2 <sous commande>
```

Nous reviendrons sur EC2 plus bas. Comme mentionné, on obtient la totalité des opérations réalisables sur EC2 via la commande **aws ec2 help**.

Mais avant de pouvoir se lancer aveuglément dans la création de 200 machines virtuelles, il est nécessaire d'indiquer à l'AWS CLI nos identifiants afin que cette dernière puisse interagir avec l'API AWS. Ceci s'effectue très simplement à l'aide de la commande :

```
$ aws configure
AWS Access Key ID [****************NVGQ]: l_id_
de_la_clé_notée_plus_haut
AWS Secret Access Key [****************0e98]:
la_clé_secrèt
Default region name [eu-central-1]: la_region_
par_defaut
Default output format [None]: le_format_de_sortie
```

Quelques explications s'imposent. Vous aurez compris que l'*Access Key ID* ainsi que la *Secret Access Key* sont les deux valeurs précédemment sauvegardées. La *Default region name* est la région au sens Amazon du terme ; au moment d'écrire ces lignes, elles sont au nombre de 14 et elles se présentent souvent par un couple code/nom :

| Code | Nom |
|---|---|
| us-east-1 | US East (N. Virginia) |
| us-east-2 | US East (Ohio) |
| us-west-1 | US West (N. California) |
| us-west-2 | US West (Oregon) |
| ca-central-1 | Canada (Central) |
| eu-west-1 | EU (Ireland) |
| eu-central-1 | EU (Frankfurt) |
| eu-west-2 | EU (London) |
| ap-northeast-1 | Asia Pacific (Tokyo) |
| ap-northeast-2 | Asia Pacific (Seoul) |
| ap-southeast-1 | Asia Pacific (Singapore) |
| ap-southeast-2 | Asia Pacific (Sydney) |
| ap-south-1 | Asia Pacific (Mumbai) |
| sa-east-1 | South America (São Paulo) |

La région la plus proche de ma connexion Internet étant Frankfurt, j'ai donc choisi **eu-central-1** pour des raisons pratiques.

Le *Default output format*, assez logiquement, est le format dans lequel vous souhaitez que les résultats de vos commandes soient affichés. Par défaut, ils seront rendus en format **JSON**, mais vous pouvez lui préférer un format texte simple ou encore sous forme de tableau ASCII, plus lisible, mais moins pratique à parcourir.

L'appel à **aws configure** a créé deux fichiers au format ini : **~/.aws/credentials** qui contient la paire *access key*

*id* / *secret key* sous un label **[default]**, et **~/.aws/config** qui contient les régions et formats de sortie, également sous le label **[default]**. Si vous deviez gérer un compte différent, par exemple pour un client, vous pourriez créer un label **[monclient]** dans chacun des fichiers et y ajouter les informations associées.

*Protip* : la CLI est fournie avec tout le nécessaire d'autocomplétion pour votre shell favori, pour bash :

```
$ complete -C '~/ve/glmfaws/bin/aws_completer' aws
```

Pour zsh :

```
% source ~/ve/glmfaws/bin/aws_zsh_completer.sh
```

Un **<Tab><Tab>** plus loin :

```
$ aws ec2 describe-instance[tab][tab]
describe-instance-attribute  describe-instance-status
describe-instances
```

## 3. LE SERVEUR KLEENEX

Et que fait-on avec tout ça ? On manipule de l'infrastructure comme une entité éphémère, des objets qui vivent, qui meurent, mais qui n'ont plus l'attention d'autrefois, les machines (virtuelles) sont un moyen, pas une fin.

Avant d'aller plus loin, il est nécessaire d'ajouter un peu de terminologie à notre bagage, car finalement, AWS, c'est beaucoup de terminologie appliquée à des concepts que vous manipulez depuis des années.

**EC2** : Pour *Elastic Cloud Computing*, cet acronyme représente toute la zone de machines virtuelles, les sous-réseaux dans lesquels elles sont contenues, mais également l'entité virtuelle qui isole votre environnement, le **VPC**, pour *Virtual Private Cloud*.

**Instance EC2** : Ni plus ni moins qu'une machine virtuelle. Ces dernières ont des propriétés : leur type i.e. le nombre de vCPUs et la quantité de RAM qui les compose, leur appartenance à un sous-réseau, les règles de *firewall* qui lui sont associées, aussi appelées *Security Groups* ou encore l'**AMI** (*Amazon Machine Image*), une sorte d'ISO, qui sera utilisée pour instancier cette machine virtuelle. Car oui, jusque dans la terminologie, comme dans du code, on **instancie** une machine virtuelle, c'est un objet qui existe le temps d'un travail.

À l'aide de l'AWS CLI, nous allons donc interroger EC2 sur l'existence et l'état des instances qui existent dans notre compte :

```
$ aws ec2 describe-instances
{
    "Reservations": [
        {
            "ReservationId":
"r-xxxxxxxxxxxxxxxx",
            "OwnerId": "yyyyyyyyyyyyyyyy",
            "Instances": [
                {
                    "InstanceType": "t2.micro",
                    "Hypervisor": "xen",
                    ...
```

Un pavé de JSON nous est renvoyé, décrivant précisément l'ensemble de nos instances. Afin de simplifier la sortie de la CLI, on peut utiliser l'option **query**. Par exemple, la commande suivante affichera l'**id** des instances ainsi que leur état (démarré ou arrêté ici) :

```
$ aws ec2 describe-instances --query
'Reservations[].Instances[].[InstanceId,State.Name]'
[
    [
        "i-0038dcc53ee2c7d51",
        "stopped"
    ],
    [
        "i-8e5f1f32",
        "running"
    ]
]
```

Ou encore leurs noms :

```
$ aws ec2 describe-instances --query
'Reservations[].Instances[].
[InstanceId,Tags[?Key=='Name'].Value[]]'
[
    [
        "i-0037dcc53ee2c7d51",
        [
            "RED_test_instance_slave_2"
        ]
    ],
    [
        "i-0cafbb62d01195e25",
        [
            "RED_test_instance_slave_1"
        ]
    ],
    [
        "i-082946343fd68c1ad",
        [
            "RED_test_instance_master"
        ]
    ],
    [

        "i-8e5f1f32",
        [
            "RED-bastion"
        ]
    ]
]
```

**Pas de panique**, il se trouve que le nom « humain » donné à une instance est catégorisé comme un « Tag », les *Tags* sont un composant très important des objets AWS : ils permettent de mettre un label à à peu près n'importe quel objet et ainsi les discriminer, les chercher, ou par exemple ici, les lister.

La syntaxe **JMESPath [10]** peut paraître inaccessible, mais il est probable que vous n'aurez jamais besoin de plus de complexité que celle visible dans ce filtre, où l'on utilise l'opérateur **?** pour interroger une valeur particulière. Les amateurs de **jq [9]** apprécieront.

Une autre possibilité pour filtrer les résultats est l'utilisation de l'option **filter**, qui n'affichera que les résultats satisfaisant la condition. Cette option est typiquement utilisée pour afficher les informations relatives à un nom d'instance :

```
$ aws ec2 describe-instances --filter
'Name=tag:Name,Values=label-de-plusieurs-instances*'
```

Il est évidemment possible de cumuler les deux. Dans cet exemple, nous afficherons les informations relatives aux adresses IP privées des instances de type **t2.micro** :

```
$ aws ec2 describe-instances --filter
'Name=instance-type,Values=t2.micro'
--query 'Reservations[].Instances[].
NetworkInterfaces[].PrivateIpAddresses[]'
[
    {
        "Primary": true,
        "PrivateIpAddress": "10.1.1.227",
        "PrivateDnsName": "ip-10-1-1-227.
eu-central-1.compute.internal"
    },
    {
        "Primary": true,
        "PrivateIpAddress": "10.1.1.21",
        "PrivateDnsName": "ip-10-1-1-21.
eu-central-1.compute.internal"
    },
    {
        "Primary": true,
        "PrivateIpAddress": "10.1.1.162",
        "PrivateDnsName": "ip-10-1-1-162.
eu-central-1.compute.internal"
    }
]
```

**NOTE**

Au sujet des types d'instances : dans cet article, nous n'utiliserons que des instances de type **t2.micro**, car ces dernières sont éligibles au *free-tier*, comprendre que leur utilisation est gratuite pendant la première année de votre souscription à AWS. Il existe beaucoup de types d'instances pour des usages très divers, puissance CPU, quantité et rapidité de la RAM, puissance GPU... évidemment à des prix très variables et parfois très élevés !

# 4. SHUT UP AND TAKE MY MONEY

Avant d'aller plus loin, nous allons procéder à la création d'une paire de clés RSA de 2048 bits qui servira à nous connecter via SSH sur les instances que nous créerons :

```
$ aws ec2 create-key-pair --key-name my-
eu-central-key
```

Cette commande aura pour effet d'afficher une clé privée au format PEM (la valeur associée à la clé **KeyMaterial**) qu'il vous appartiendra de sauvegarder dans un précieux fichier. Notez qu'une clé est associée à une région, ici nous n'avons pas spécifié l'option **--region** parce que nous avons sélectionné la région **eu-central-1** comme région par défaut lors de la configuration de la CLI.

Muni de ce moyen de connexion, le moment que vous attendez tous, la création d'une nouvelle instance à l'aide d'AWS CLI ; je pose ça là :

```
$ aws ec2 run-instances --count 1
--instance-type t2.micro --key-name my-
eu-central-key --associate-public-ip-
address --image-id ami-5900cc36
```

« Ah ouais, ok ». Non mais je vous promets, c'est pas si terrible. Regardez plutôt :

- On appelle la commande **run-instances**, au pluriel, car nous pourrions en démarrer plusieurs ;
- Justement, nous précisons que nous ne créons qu'une seule instance ;
- Le type d'instance est **t2.micro** ;
- On donne le nom de la clé générée ;
- On souhaite associer à cette instance une IP publique ;
- Et le meilleur pour la fin : cette instance sera construite en utilisant l'*Amazon Machine Image* d'**Id ami-5900cc36**.

Considérez une AMI comme une sorte d'image ISO, ou mieux, une image brute (*raw*) qui sert à instancier la machine virtuelle avec le système d'exploitation et potentiellement quelques spécificités. Les AMI officielles **Debian** par exemple sont générées par le logiciel **bootstrap-vz [11]** puis mises à disposition sur AWS.

Debian justement, possède un identifiant, **379101102735**, qui permet de filtrer la recherche des AMI en ciblant uniquement celles dont ils sont les possesseurs (*Owners*) :

```
$ aws ec2 describe-images --owners 379101102735
```

# Abonnez-vous !




## PARTICULIERS,

➡ Rendez-vous sur :

**www.ed-diamond.com**

pour consulter toutes les offres !

➡ ...ou renvoyez-nous le document au verso complété !

## PROFESSIONNELS,

➡ Rendez-vous sur :

**proboutique.ed-diamond.com**

pour consulter toutes les offres dédiées !

➡ ...ou renvoyez-nous le document au verso complété !

# VOICI LES OFFRES D'ABONNEMENT AVEC GNU/LINUX MAGAZINE !

## CHOISISSEZ VOTRE OFFRE ! Prix TTC en Euros / France Métropolitaine*

| Offre | ABONNEMENT | PAPIER Réf | PAPIER Tarif TTC |
|---|---|---|---|
| LM | 11n° GLMF | ☐ LM1 | 69 € |
| LM+ | 11n° GLMF + 6n° HS | ☐ LM+1 | 125 € |
| | **LES COUPLAGES AVEC NOS AUTRES MAGAZINES** | | |
| A | 11n° GLMF + 6n° LP | ☐ A1 | 105 € |
| A+ | 11n° GLMF + 6n° HS + 6n° LP + 3n° HS | ☐ A+1 | 189 € |
| B | 11n° GLMF + 6n° MISC | ☐ B1 | 109 € |
| B+ | 11n° GLMF + 6n° HS + 6n° MISC + 2n° HS | ☐ B+1 | 185 € |
| C | 11n° GLMF + 6n° LP + 6n° MISC | ☐ C1 | 149 € |
| C+ | 11n° GLMF + 6n° HS + 6n° LP + 3n° HS + 6n° MISC + 2n° HS | ☐ C+1 | 249 € |
| J | 11n° GLMF + 6n° HK* | ☐ J1 | 105 € |
| J+ | 11n° GLMF + 6n° HS + 6n° HK* | ☐ J+1 | 159 € |
| L | 11n° GLMF + 6n° HK* + 6n° LP + 6n° MISC | ☐ L1 | 189 € |
| L+ | 11n° GLMF + 6n° HS + 6n° HK* + 6n° LP + 3n° HS + 6n° MISC + 2n° HS | ☐ L+1 | 289 € |

Les abréviations des offres sont les suivantes : GLMF = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | HK = Hackable

**J'indique l'offre si différente que celles ci-dessus :**

**J'indique la somme due (Total) : €**

**Je choisis de régler par :**

❑ **Chèque bancaire ou postal à l'ordre des Éditions Diamond (uniquement France et DOM TOM)**

❑ **Pour les règlements par virements, veuillez nous contacter via e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20**

**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 : http://**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 :**

.....

.....



## RETROUVEZ TOUTES NOS OFFRES SUR : www.ed-diamond.com !

*Les tarifs hors France Métropolitaine, Europe, Asie, etc. sont disponibles en ligne !

Plus précisément, nous souhaitons connaître l'identifiant d'AMI Debian sur la région **eu-central-1** et de version **debian-jessie** :

```
$ aws --region eu-central-1 ec2 describe-images --owners 379101102735 --filters 'Name=name,Values=debian-jessie*' --query 'Images[].[CreationDate,ImageId]'
[
    [
        "2015-04-26T00:35:24.000Z",
        "ami-02724d1f"
    ],
    [
        "2015-06-07T13:22:32.000Z",
        "ami-02b78e1f"
    ],
    [
        "2016-02-19T14:58:06.000Z",
        "ami-2638224a"
    ],
    [
        "2015-04-26T00:52:08.000Z",
        "ami-28724d35"
    ],
    [
        "2016-09-19T15:39:59.000Z",
        "ami-30e01d5f"
    ],
    [
        "2017-01-15T12:44:34.000Z",
        "ami-5900cc36"
    ],
    [
        "2015-06-07T13:35:52.000Z",
        "ami-5cb78e41"
    ],
    [
        "2016-11-14T14:27:18.000Z",
        "ami-cc8441a3"
    ],
    [
        "2016-04-03T13:18:53.000Z",
        "ami-ccc021a3"
    ]
]
```

On voit que l'image la plus récente porte l'identifiant **ami-5900cc36** : c'est celle que nous utiliserons.

Cet exercice étant passablement fastidieux, vous trouverez les *AMI Ids* à jour dans la section dédiée du wiki Debian **[12]**, mais il est important de savoir manipuler les filtres pour cibler une information rapidement.

La sortie de la commande **aws ec2 run-instances** vous informe sur tous les paramètres relatifs à la création de votre nouvelle instance. En particulier, son **InstanceId** qui va nous permettre de la *tagger* et ainsi ne pas laisser un objet sans nom :

```
$ aws ec2 create-tags --resources i-002d26e659e723086 --tags Key=Name,Value=glmfinstance
```

Notre instance créée, nous allons certainement pouvoir nous connecter dessus via SSH, pour cela, récupérons son IP publique :

```
$ aws ec2 --profile ot describe-instances --filter 'Name=instance-id,Values=i-002d26e659e723086' --query 'Reservations[].Instances[].PublicIpAddress'
[
    "54.93.229.152"
]
```

L'utilisateur par défaut d'une instance basée sur une AMI Debian est **admin**, aussi, en utilisant notre clé privée :

```
$ ssh -i vader-eu-central-1.pem admin@54.93.229.152
ssh: connect to host 54.93.229.152 port 22: Connection timed out
```

Et là... rien ne se passe. Pourquoi ? Parce que par défaut, si rien n'est précisé, les règles de *firewalling* associées à une instance sont extrêmement restrictives : rien ne rentre. Il nous incombe donc de créer une *security rule* que nous allons associer à notre instance afin d'autoriser le port **22**.

En premier lieu, nous allons créer le groupe de sécurité :

```
$ aws ec2 create-security-group --group-name SSHonly --description 'Authorize only the SSH (22/TCP) protocol'
{
    "GroupId": "sg-5e3db735"
}
```

Puis ajoutons la règle au groupe :

```
$ aws ec2 authorize-security-group-ingress --group-name SSHonly --protocol tcp --port 22 --cidr '0.0.0.0/0'
```

Vérifions :

```
$ aws ec2 describe-security-groups --group-name SSHonly
{
    "SecurityGroups": [
        {
            "Description": "Authorize only the SSH (22/TCP) protocol",
            "IpPermissions": [
                {
                    "IpProtocol": "tcp",
```

```
                    "ToPort": 22,
                    "FromPort": 22,
                    "IpRanges": [
                        {
                            "CidrIp": "0.0.0.0/0"
                        }
                    ],
                    "PrefixListIds": [],
                    "Ipv6Ranges": [],
                    "UserIdGroupPairs": []
                }
            ],
            [...]
        }
}
```

Reste à attacher ce nouveau *security group* à notre instance :

```
$ aws ec2 modify-instance-attribute --instance-
id i-002d26e659e723086 --groups sg-5e3db735
```

Et cette fois :

```
$ ssh -i vader-eu-central-1.pem admin@54.93.229.152
The authenticity of host '54.93.229.152 (54.93.229.152)'
can't be established.
ECDSA key fingerprint is SHA256:Z03ASoBWtBYv3KY62Fo75bA1Al3
h5g+Q1gIvO99v+9Q.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.93.229.152' (ECDSA) to the
list of known hosts.
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0644 for 'vader-eu-central-1.pem' are too open.
It is required that your private key files are NOT
accessible by others.
This private key will be ignored.
Load key "vader-eu-central-1.pem": bad permissions
Permission denied (publickey).
```

(Bruit de Millenium Falcon qui démarre pas) Ah ! Oui ! Ne laissez pas votre clé publique à la vue de tous :

```
$ chmod 400 vader-eu-central-1.pem
$ ssh -i vader-eu-central-1.pem admin@54.93.229.152

The programs included with the Debian GNU/Linux system
are free software;
the exact distribution terms for each program are
described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to
the extent
permitted by applicable law.
admin@ip-172-31-31-152:~$
```

*And* voilà !

Avant de poursuivre notre périple, n'oublions pas de nettoyer derrière nous, puisque nous n'avons plus besoin de notre instance, détruisons-la :

```
$ aws ec2 terminate-instances --instance-ids
i-002d26e659e723086
{
    "TerminatingInstances": [
        {
            "CurrentState": {
                "Code": 32,
                "Name": "shutting-down"
            },
            "InstanceId": "i-002d26e659e723086",
            "PreviousState": {
                "Code": 16,
                "Name": "running"
            }
        }
    ]
}
```

## 5. ./DATACENTER.PY

Nous l'avons mentionné au début de cet article, l'AWS CLI est conçue autour du SDK boto **[2]**. Ce dernier permet de manipuler programmatiquement les objets du *cloud* Amazon et ainsi s'approcher du nirvana : l'*infrastructure as code*.

Dans cette partie, nous allons pousser un peu plus loin qu'une simple instance : nous allons mettre en place un serveur web derrière un équilibreur de charge. Pour les besoins de cet exercice, nous utiliserons le fabuleux interpréteur **ipython** et nous aurons évidemment besoin du SDK boto3 :

```
$ pip install boto3
```

boto3 est la dernière implémentation du kit de développement d'Amazon : elle expose au choix une API orientée objet ou un client bas niveau aux différents services tels que EC2, S3, etc. Pour les besoins de cet article, nous utiliserons l'API « client » et non objet afin de rester le plus proche de fonctionnement de **awscli** et ne pas braquer les ennemis de l'orienté objet ;)

Comme on peut s'en douter, boto3 utilise les informations enregistrées lors de la configuration d'**awscli**. De ce fait, sans plus de préparatifs on peut importer le module **boto3** dans la console ipython :

```
In: import boto3
```

Et nous allons nous rendre compte en deux lignes de code de la puissance de cet import. Tout d'abord, nous **instancions**

**EC2** dans une variable, puis nous appelons la méthode **describe_instances()**, pendant de la commande **ec2 describe-instance** de l'**awscli** :

```
In: ec2 = boto3.client('ec2')
In: ec2.describe_instances().keys()
Out: dict_keys(['ResponseMetadata',
'Reservations'])
```

Eh oui, la précédente sortie JSON de la CLI se transforme ici directement en **dict** prêt à l'emploi ! Un petit exemple pour s'en convaincre :

```
In: instances = ec2.describe_instances()
In: instances['Reservations'][0]
['Instances'][0]['Architecture']
Out: 'x86_64'
```

Comme vous pouvez le constater, les noms des méthodes sont sensiblement identiques aux commandes de l'AWS CLI. Cette cohérence des outils, de l'API et des commandes chez Amazon est systématique et très appréciable pour le développeur lors de la recherche de documentation. À ce sujet, notons la remarquable qualité de la documentation de boto3 **[13]**.

À ce stade, voyons ce que propose le SDK pour créer un *Load Balancer*, ou **ELB** pour *Elastic Load Balancer* dans la terminologie Amazon afin de parvenir au but fixé dans ce paragraphe :

**create_load_balancer(**kwargs)** : *Creates a Classic Load Balancer.*

Voici les paramètres nécessaires à l'exécution de cette méthode :

```
response = client.create_load_balancer(
    LoadBalancerName='string',
    Listeners=[
        {
            'Protocol': 'string',
            'LoadBalancerPort': 123,
            'InstanceProtocol': 'string',
            'InstancePort': 123,
            'SSLCertificateId': 'string'
        },
    ],
    AvailabilityZones=[
        'string',
    ],
    Subnets=[
        'string',
    ],
    SecurityGroups=[
        'string',
    ],
    Scheme='string',
    Tags=[
        {
            'Key': 'string',
            'Value': 'string'
        },
    ]
)
```

Il est nécessaire de préciser plusieurs notions ici :

- **Protocol** est le protocole qui sera utilisé par l'ELB, côté Internet.
- **InstanceProtocol** est le protocole qui sera utilisé côté instance.
- **AvailabilityZones**, un concept très important dans une architecture AWS ; une « zone de disponibilité » (**AZ**, ou *Availability Zone*) est un *datacenter* présent dans une région au sens Amazon du terme. Ces zones de disponibilités sont interconnectées comme le seraient des *datacenters* classiques et permettent de mettre en place des services à haute disponibilité. Le nombre d'*Availability Zones* varie selon la région, mais elles sont au minimum au nombre de deux.
- **Scheme** est le type d'ELB. Par défaut, il est **internet-facing** (joignable sur Internet), mais il est également possible de déclarer un ELB privé, par exemple pour mettre en place un service interne hautement disponible (typiquement un cluster LDAP).

Les autres paramètres sont triviaux ou nous les avons vus précédemment.

Pour créer notre ELB programmatiquement, il faudra instancier l'objet **elb** :

```
In: elb = b.client('elb')
```

Et enregistrer quelques informations sur notre environnement :

```
In: azs = [az['ZoneName'] for az in ec2.describe_
availability_zones()['AvailabilityZones']]
In: azs
Out: ['eu-central-1a', 'eu-central-1b']
In: subnets = [sub['SubnetId'] for sub in ec2.
describe_subnets()['Subnets']]
In: subnets
Out: ['subnet-408e2c28', 'subnet-c1d525bb']
```

Afin de pouvoir se connecter sur le port **80** du répartiteur de charge, nous créons un *Security Group* adéquat :

```
In: ec2.create_security_group(GroupName='HTTPonly',
Description='Authorize HTTP traffic only')
Out:
{'GroupId': 'sg-a822b7c3',
 ...}
```

Puis ajoutons la règle elle-même dans ce groupe :

```
In: ec2.authorize_security_group_ingress(
        GroupName = 'HTTPonly',
        IpProtocol = 'tcp',
        FromPort = 80,
        ToPort = 80,
        CidrIp = '0.0.0.0/0'
    )
```

Nous sommes désormais en mesure de créer notre ELB :

```
In: elb.create_load_balancer(
    LoadBalancerName = 'GLMFLB',
    Listeners = [
        {
            'Protocol': 'HTTP',
            'LoadBalancerPort': 80,
            'InstanceProtocol': 'HTTP',
            'InstancePort': 80,
        }
    ],
    Subnets = subnets,
    SecurityGroups = ['sg-a822b7c3'],
    Tags = [{'Key': 'Name', 'Value': 'GLMFLB'}]
)
Out:
{'DNSName': 'GLMFLB-380124310.eu-central-1.elb.
amazonaws.com',
...}
```

Constatons que notre ELB dispose déjà d'un *endpoint* public **GLMFLB-380124310.eu-central-1.elb.amazonaws.com**. On peut même l'interroger :

```
$ curl -I GLMFLB-380124310.eu-central-1.elb.amazonaws.com
HTTP/1.1 503 Service Unavailable: Back-end server is at capacity
Connection: keep-alive
```

Forcement, puisque rien n'y est encore connecté.

Lors de la création de nos deux instances, en plus d'utiliser la méthode **run_instances()**, nous allons faire connaissance avec une fonctionnalité très puissante d'_EC2_: le *UserData*. Il s'agit d'un espace à peu près libre dédié à l'exécution de commandes personnalisées lors de l'instanciation de la machine virtuelle. Ces données peuvent être de deux types :

- Un simple script shell ;
- Un fichier **cloud-init** **[14]**, au format **YAML**, qui permet de spécifier de façon ordonnée des opérations d'initialisation à mener sur l'instance lors de son instanciation.

Par souci de propreté, et pour prendre tout de suite les bonnes habitudes, c'est la dernière méthode que nous emploierons.

**ATTENTION !**

La taille limite du *UserData* est de 16Kio.

Créons un « fichier » **cloud-init** basique :

```
cinit = '#cloud-config\npackage_update:
true\npackages\n  - nginx\n'
```

Un fichier **cloud-init** doit impérativement démarrer par la chaîne de caractères **#cloud-config**. Comprenez que nous allons passer ces informations sous forme de paramètre à **run_instances()** ; dans un scénario plus complet, il eut été plus élégant de créer un vrai fichier...

```
#cloud-config
package_update: true
packages
  - nginx
```

... de le lire et passer son contenu en paramètre. Ici, nous allons simplement invoquer la commande suivante, équivalente à l'appel de **awscli ec2 run-instance** :

```
In: for subnet in subnets: # on boucle sur les
Ids de sous-réseaux
    i = ec2.run_instances(
        MinCount = 1,
        MaxCount = 1,
        ImageId = 'ami-5900cc36',
        KeyName = 'my-eu-central-key',
        InstanceType = 't2.micro',
        NetworkInterfaces = [
            {
                'DeviceIndex': 0,
# avec boto, on doit donner l'index de la "carte"
réseau
                'AssociatePublicIpAddress': True,
                'SubnetId': subnet,
                'Groups': ['sg-5e3db735', 'sg-
a822b7c3'] # SSHonly et HTTPonly
            }
        ],
        UserData = cinit
    )
    ec2.create_tags( # on oublie pas de tagger
les instances !
        Resources = [i['Instances'][0]
['InstanceId']],
        Tags = [{'Key': 'Name', 'Value':
'GLMFinstance'}]
    )
```

Nos instances lancées, nous récupérons les informations les concernant en appliquant un filtre sur les noms d'instances qui commencent par « GLMF » et qui sont en état « running » :

```
In: instances = ec2.describe_instances(
    Filters = [
        {'Name': 'tag:Name', 'Values': ['GLMF*']},
        {'Name': 'instance-state-name', 'Values':
['running']}
    ]
)
```

Et les attachons à l'ELB précédemment créé :

```
In: iids = []
In: for r in instances['Reservations']:
        for i in r['Instances']:
            if i['State']['Name'] == 'running':
                iids.append({'InstanceId':
i['InstanceId']})
In: elb.register_instances_with_load_balancer(
        LoadBalancerName = 'GLMFLB',
        Instances = iids
)
```

On peut s'assurer du bon fonctionnement de l'ensemble grâce à la méthode **describe_instance_health()** :

```
In: elb.describe_instance_health(LoadBalancerName =
'GLMFLB')
Out:
{'InstanceStates': [{'Description': 'N/A',
   'InstanceId': 'i-03d210d44a2e30c43',
   'ReasonCode': 'N/A',
   'State': 'InService'},
  {'Description': 'N/A',
   'InstanceId': 'i-0b9ee518c2728c822',
   'ReasonCode': 'N/A',
   'State': 'InService'}],
  [...]
}
```

Et finalement vérifier que nos serveurs web répondent :

```
$ curl -I GLMFLB-380124310.eu-central-1.elb.
amazonaContent-Length: 867
Content-Type: text/html
Date: Mon, 15 May 2017 14:25:49 GMT
ETag: "5919b7d8-363"ws.com
HTTP/1.1 200 OK
Accept-Ranges: bytes
Server: nginx/1.6.2
Last-Modified: Mon, 15 May 2017 14:14:48 GMT
Connection: keep-alive
$ curl -o- -s GLMFLB-380124310.eu-central-1.
elb.amazonaws.com|grep 'debian and nginx'
<p><em>Thank you for using debian and
nginx.</em></p>
```

*OP delivar* : un serveur web redondé derrière un *Load Balancer*, le tout créé uniquement en Python !

C'est vrai, ça claque. N'oubliez tout de même pas de passer le balai et détruisez consciencieusement toutes les ressources créées pendant cette démonstration sous peine de voir votre carte bleue ponctionnée de quelques centimes d'euros :

```
In: ec2.terminate_instances(InstanceIds =
[i['InstanceId'] for i in iids])
In: elb.delete_load_balancer(LoadBalancerName="
GLMFLB")
In: for sg in ['sg-5e3db735', 'sg-a822b7c3']:
        ec2.delete_security_group(GroupId = sg)
```

Il vous appartient désormais de condenser et classer les méthodes que nous avons vues au long de ce paragraphe pour parfaire votre initiation à boto.

## SALUT LES P'TITS CLOUDS !

(Seuls les vieux peuvent comprendre ce titre). Dans cet article, nous avons touché du doigt l'infinité de possibilités qui s'offrent au développeur-administrateur que l'on appelle aujourd'hui *DevOps*. En effet l'outillage et les contributions sont très nombreux dans l'univers du *Cloud* public, et si vous empruntez ce chemin vous allez faire connaissance avec des produits et projets aussi innovants qu'audacieux. Je vous pose à cet effet ma petite contribution [15], des outils que j'écris et utilise à **${DAYJOB}**, dans lesquels vous retrouverez quelques morceaux des concepts manipulés dans ces colonnes. ■

## RÉFÉRENCES


**[1]** https://github.com/aws/aws-cli

**[2]** https://github.com/boto/boto3

**[3]** https://aws.amazon.com/apache-2-0/

**[4]** https://aws.amazon.com/

**[5]** https://aws.amazon.com/iam/

**[6]** https://aws.amazon.com/iam/details/mfa/

**[7]** https://aws.amazon.com/free/

**[8]** https://aws.amazon.com/ec2/

**[9]** https://stedolan.github.io/jq/

**[10]** http://jmespath.org/

**[11]** https://github.com/andsens/bootstrap-vz/

**[12]** https://wiki.debian.org/Cloud/AmazonEC2Image

**[13]** http://boto3.readthedocs.io/en/latest/

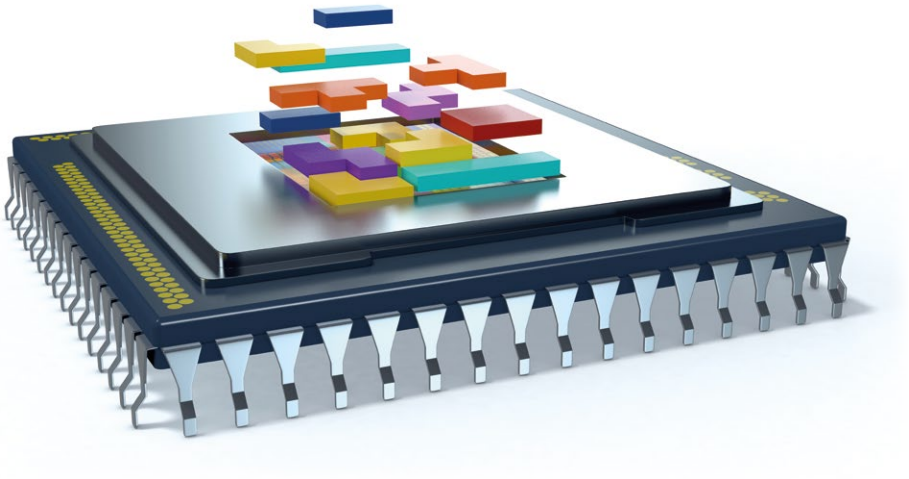
**[14]** http://cloudinit.readthedocs.io/en/latest/

**[15]** https://github.com/iMilnb/awstools

# FREERTOS : APPLICATION À LA RÉALISATION D'UN ANALYSEUR DE RÉSEAU NUMÉRIQUE SUR STM32


**QUENTIN MACÉ ET JEAN-MICHEL FRIEDT**
[Master microsystèmes, instrumentation et robotique (MIR), Univ. de Franche-Comté, Besançon]

**MOTS-CLÉS : FREERTOS, STM32F4, ARM CORTEX M3/M4, ANALOG DEVICES AD9834, LT5537, LIBOPENCM3**



**FreeRTOS est un environnement exécutif à très faible empreinte mémoire (<10 KB) fournissant un ordonnanceur et les mécanismes associés pour le partage de ressources entre tâches. En utilisant une bibliothèque libre supportant les cœurs ARM Cortex, en particulier la gamme STM32, nous rendons FreeRTOS utilisable sur tout microcontrôleur muni d'un tel processeur. Nous appréhendons cet environnement de travail, favorable aux développements collaboratifs, dans le contexte de la réalisation d'un instrument de caractérisation de dispositifs radiofréquences.**

Le développement de systèmes embarqués a pour vocation de minimiser les ressources requises pour atteindre un objectif donné. Dans ce contexte, la programmation de systèmes embarqués numériques s'accommode mal d'environnements inefficaces qui gaspillent les ressources **[1]** mises à disposition par le microcontrôleur : le langage **C** apparaît ainsi comme un compromis optimal entre la qualité du code généré et un niveau d'abstraction suffisamment élevé pour exprimer des algorithmes complexes. En particulier, l'utilisation des pointeurs (`*(type*)adresse=valeur;`) permet de s'approcher au plus près du matériel en manipulant aisément les bus d'adresse, de données et de contrôle qui relient l'unité arithmétique et logique aux périphériques qui l'entourent. Cependant, le `C` se prête relativement mal aux projets complexes nécessitant la cohabitation harmonieuse de plusieurs développeurs : même s'il est envisageable de définir des interfaces entre les fonctions écrites par plusieurs développeurs et garantir la qualité du code par les tests unitaires associés, une séparation du travail sous forme de tâches communiquant entre elles et échangeant des informations semble plus naturelle. La question

porte donc sur les ressources requises pour mettre en place un tel environnement, la garantie de la cohérence des accès concurrents aux ressources induites par la multiplicité des tâches, ainsi que la portabilité du code résultant. Ces arguments sont développés sur www.freertos.org/FAQWhat.html#WhyUseRTOS. **FreeRTOS** (www.freertos.org) propose une hiérarchie de priorités des tâches et donc la garantie que les tâches de priorités les plus élevées sont préemptées avec une latence minimale. Tous les appels bloquants prennent un argument de délai maximum au-delà duquel ils peuvent être débloqués : FreeRTOS respecte donc les préceptes du temps réel de borner les latences et ne jamais bloquer l'exécution d'un programme. Nous verrons cependant que la disponibilité des mécanismes ne garantit pas leur bonne utilisation, et il nous sera possible, par un choix maladroit des priorités de tâches, de bloquer un programme. Nous verrons que le déploiement de FreeRTOS nécessite moins de 8 KB de RAM sur la plateforme cible, respectant nos ambitions de systèmes fortement embarqués à ressources réduites.

Nous appréhendons ce problème, et la solution que nous lui avons trouvé au travers de l'environnement exécutif FreeRTOS **[4]**, dans le contexte de la réalisation d'un instrument de caractérisation de dispositifs radiofréquences : un analyseur de réseau numérique. Cet instrument classique permet de caractériser la réponse d'un système en fonction de la fréquence, en amplitude et en phase. Nous allons nous intéresser aux gammes radiofréquences qui vont nous permettre de caractériser filtres et résonateurs classiquement utilisés dans le traitement du signal radiofréquence, que ce soit pour filtrer une bande de fréquence donnée (e.g. les diverses bandes de fréquences fournissant les divers services accessibles sur un téléphone portable) ou cadencer des dispositifs synchrones (e.g. oscillateur d'un système numérique). Nous nous focalisons sur un objectif de coût minimum, qui va imposer le choix d'un microcontrôleur qui n'est pas supporté par FreeRTOS, fournissant donc une opportunité de comprendre l'architecture de l'environnement de travail : nous découvrirons que seuls 6 fichiers sont nécessaires pour accéder aux fonctions de FreeRTOS, garantie d'une excellente portabilité puisque toutes les fonctions spécifiques à une architecture donnée sont regroupées dans un unique fichier. Cet objectif de coût réduit quelque peu les ambitions de gamme de fréquences accessibles, puisque le synthétiseur de signaux radiofréquences que nous sélectionnons (**Analog Devices AD9834**) ne sera cadencé qu'à 70 MHz, donnant accès aux gammes de fréquences entre le sub-MHz et une vingtaine de MHz (il est classiquement admis qu'une synthèse numérique de fréquence travaille entre DC et le tiers de sa fréquence de cadencement pour éviter la pollution par des raies parasites). Nous verrons qu'avec un peu d'astuce, ce montage nous permettra néanmoins d'émettre dans la bande FM (88-108 MHz).

La présentation porte donc sur trois objectifs :

1. Porter FreeRTOS sur une architecture non supportée en comprenant quels sont les fichiers nécessaires et leurs rôles, en particulier en nous liant à une bibliothèque libre fournissant les accès au matériel que nous avons sélectionné.
2. Démontrer la portabilité du code à diverses plateformes de la même famille en jouant sur le script décrivant l'organisation de la mémoire (*linker script*) et quelques drapeaux de compilation.
3. Démontrer les fonctionnalités apportées par FreeRTOS sur une plateforme matérielle, mais aussi, pour le lecteur désireux de se familiariser avec cet environnement de développement sans avoir accès au matériel, avec l'émulateur **qemu**.

# 1. PRÉSENTATION DE FREERTOS

Notre premier objectif tient donc dans la découverte de FreeRTOS. Rappelons qu'un environnement exécutif a pour vocation de donner au développeur le sentiment de travailler sur un système d'exploitation, ici avec la notion de tâches apparemment exécutées en parallèle et cadencées par un séquenceur. Cependant, l'environnement exécutif n'autorisera pas le chargement dynamique de bibliothèques ou d'applications : le binaire monolithique généré à l'issue de la compilation occupe une quantité de ressources connues à la compilation qui permet de garantir la capacité de la plateforme de travail à exécuter ce code. Nous verrons que FreeRTOS offre une multitude de fonctionnalités pour appréhender des problèmes de gestion de la pile, d'ordonnancement des tâches, et surtout pour garantir la cohérence des accès aux ressources en en protégeant l'accès par les mécanismes classiques de sémaphore et leur version binaire que sont les mutex (*MUtually EXclusive*). Ces mécanismes doivent absolument être implémentés aux côtés de l'ordonnanceur pour garantir l'atomicité de leur manipulation et interdire la préemption de la tâche en train de réserver une ressource, au risque de voir le mécanisme de protection échouer.

Nous choisissons de travailler sur un microcontrôleur de la gamme **STM32** (voir figures 1 et 2), une architecture ARM Cortex M3 ou M4 selon la puissance de calcul requise, par souci de pérennité de l'investissement intellectuel d'appréhender une nouvelle classe de microcontrôleurs. En effet, de nombreux fondeurs choisissent actuellement de décliner ce cœur de processeur, et même si **ST** abandonne sa fabrication, de nombreuses autres solutions resteront disponibles autour de la même architecture (**Atmel SAM3** et **SAM4**, **NXP LPC13xx**,

**TI LM** et **TM**, **EFM32 Gecko**, **Freescale**, etc.). Toujours par souci de portabilité et de liberté, nous désirons ne pas nous lier à une bibliothèque issue d'un fondeur en particulier, même si la disponibilité des codes sources pourrait répondre à nos attentes, mais de nous lier à la bibliothèque libre implémentant un certain nombre de fonctionnalités pour Cortex, **libopencm3** (http://libopencm3.org/).

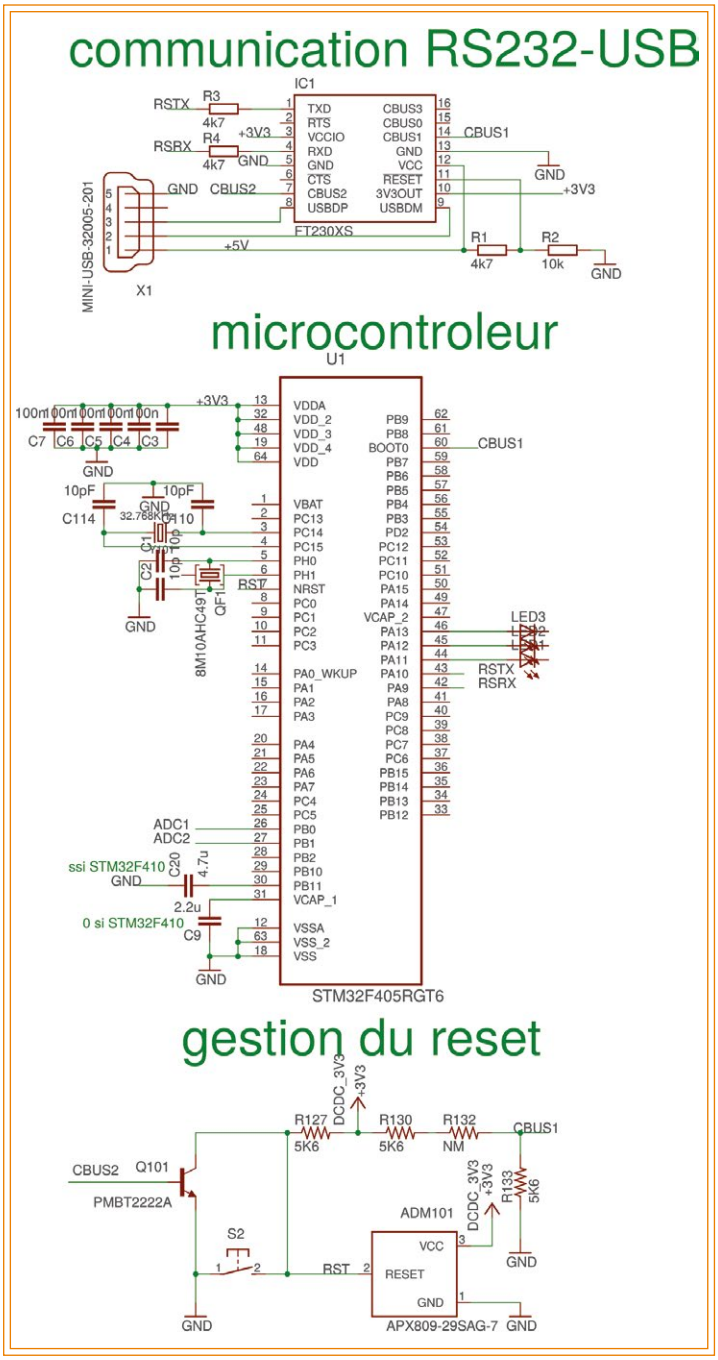



*Fig. 1 : Schéma du circuit.*

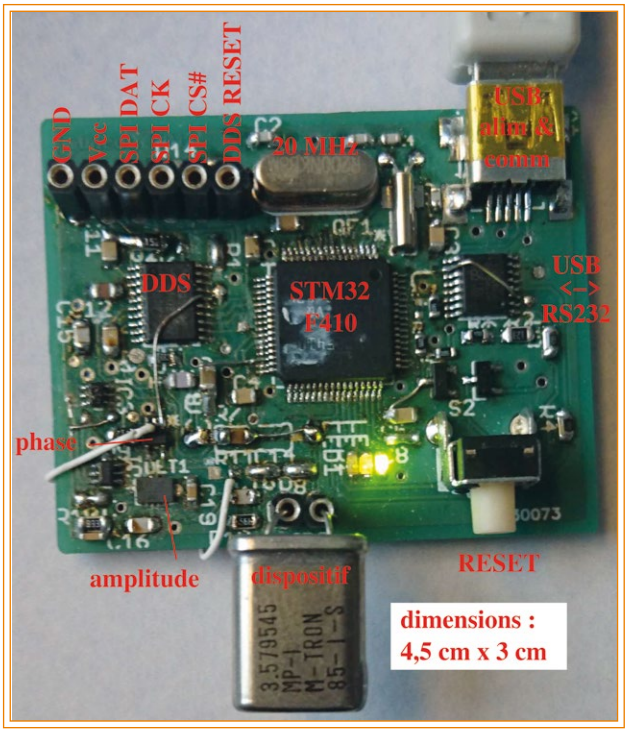



*Fig. 2 : Photographie du circuit assemblé, comprenant un microcontrôleur STM32F410 cadencé à 20 MHz (multiplié en interne à 140 MHz) et les divers périphériques pour réaliser un analyseur de réseau.*

La question est donc : quel est le travail pour porter FreeRTOS à un nouveau microcontrôleur supporté par une nouvelle bibliothèque ?

Après avoir installé la chaîne de compilation pour ARM Cortex M3 et M4, tel que par exemple décrit sur github.com/jmfriedt/summon-arm-toolchain.git qui placera **arm-none-eabi-gcc** et ses dépendances dans **`$HOME/sat`**, nous commençons par télécharger les sources de FreeRTOS, version 9.0.0 à la date de rédaction de cette prose, sur www.freertos.org/a00104.html. Le contenu de cette archive peut paraître impressionnant au premier abord, jusqu'à ce que nous en comprenions l'organisation et que tout FreeRTOS soit contenu dans les 6 fichiers de **`FreeRTOS/Source`** : **`croutine.c`**, **`event_groups.c`**, **`list.c`**, **`queue.c`**, **`tasks.c`**, et **`timers.c`**. Il nous faudra néanmoins un 7ème fichier implémentant la gestion de la mémoire, mécanisme spécifique à chaque architecture de microcontrôleur : nous trouverons notre bonheur dans **`FreeRTOS/Source/portable/GCC/ARM_CM*`**. FreeRTOS ne fournit aucune abstraction du matériel ou fonctionnalité spécifique à une architecture donnée : nous devrons fournir de telles capacités par l'exploitation de libopencm3, et en particulier l'initialisation des périphériques et des horloges, ainsi que les outils de communication.

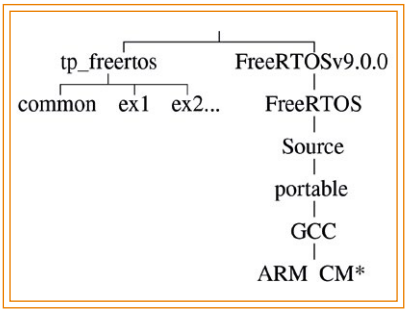



*Fig. 3 : Arborescence du projet.*

Les divers exemples proposés dans ce document sont disponibles sur https://github.com/jmfriedt/tp_freertos/. Nous faisons l'hypothèse que le lecteur a par ailleurs téléchargé les sources de FreeRTOSv9.0.0 et les a placées au même niveau de l'arborescence que les exemples, tel que résumé dans le schéma de l'arborescence de la figure 3. Les divers **Makefile** de nos exemples sont des liens symboliques vers **Makefile.f1_opencm3** (ARM Cortex M3 de type **STM32F1**) ou **Makefile.f4_opencm3** (ARM Cortex M4 de type **STM32F4**). Les fichiers **Makefile.f1_libstm32** sont un résidu de support pour la bibliothèque fournie par ST-Microelectronics **libstm32** qui n'a d'intérêt que pour supporter des périphériques exotiques qui ne sont pas supportés par libopencm3 (par exemple le compteur haute fréquence **HRTIM** qui équipe certains **STM32F3**). Le répertoire **common** contient les fonctions de base qui ne sont pas portables et donc pas implémentées par FreeRTOS : initialisation des horloges, initialisation des périphériques, communication ou acquisition de données (conversion analogique-numérique).

Il devient à cette étape de l'étude nécessaire de définir la déclinaison exacte du microcontrôleur utilisé. Afin de démontrer la compacité de FreeRTOS et sa très faible empreinte mémoire, nous choisissons dans le bas de la gamme de STM32. Par souci de réduction de coût, nous prenons la solution la moins chère disponible chez **Farnell** parmi les STM32F4 : le **STM32F410 [2]** semble un composant de choix, à 3 euros/pièce. Ses 32 KB de RAM et 128 KB de mémoire flash seront plus que suffisants pour l'application que nous envisageons. Le choix de ce microcontrôleur achève de déterminer les fichiers de l'archive de FreeRTOS sur lesquels nous lierons notre application : **FreeRTOSv9.0.0/FreeRTOS/Source/portable/MemMang/heap_2.c** gèrera le tas dans la mémoire volatile et **FreeRTOSv9.0.0/FreeRTOS/Source/portable/GCC/ARM_CM4F/port.c** fournira les fonctions spécifiques au Cortex-M4 (avec unité de calcul en virgule flottante).

## 2. LIBOPENCM3 POUR LE STM32F4

Notre premier effort consiste à valider la capacité à exécuter des fonctions simples sur ce nouveau microcontrôleur afin de vérifier l'initialisation des périphériques et horloges associées. Ces fonctions seront ensuite utilisées depuis FreeRTOS sous forme de bibliothèque : nous prenons donc soin de séparer les fonctions exploitables par la suite de celles permettant ce test rapide. Ainsi, la fonction principale, qui sera ultérieurement remplacée par les tâches de FreeRTOS séquencées par l'ordonnanceur, ressemble à :

```
#include "common.h"

int main(void)
{ int i, c = 0;
  Usart1_Init();
  Led_Init();
  while (1) {
    if (c&0x01) {Led_Hi1();Led_Hi2();} else {Led_Lo1();Led_Lo2();}
    c = (c == 9) ? 0 : c + 1;   // cyclic increment c
    uart_putc(c + '0');         // USART1: send byte
    uart_puts("\r\n\0");
    for (i = 0; i < 800000; i++) __asm__("NOP");
  }
  return 0;
}
```

Tandis que la bibliothèque de fonctions se résume aux bases telles que les initialisations d'horloge, outils de communication (initialisation du port, mise en forme des variables sous forme de trames ASCII) ou manipulation des ports d'entrée-sortie généralistes (GPIO). Ces fonctions sont regroupées dans https://github.com/jmfriedt/tp_freertos/blob/master/common/usart_opencm3.c, avec une utilisation peut-être abusive des conditions de compilation (**#ifdef**) pour tenir compte des diverses architectures supportées. Nous supportons en effet de cette façon, en fonction des drapeaux passés lors de la compilation, trois déclinaisons du STM32 : le **STM32F100 [3]** qui équipe la carte **STM32VL-Discovery** (drapeaux **-DSTM32F1 -DSTM32F10X_LD_VL** pour la carte décrite sur www.st.com/en/evaluation-tools/stm32vldiscovery.html), le **STM32F103** qui équipe par exemple l'**Olimex STM32-P103** décrite sur www.olimex.com/Products/ARM/ST/STM32-P103/ (drapeaux **-DSTM32F1 -DSTM32F10X_MD**) et qui est émulé dans le port d'André Beckus de qemu à cette plateforme (beckus.github.io/qemu_stm32/), et le STM32F410 (drapeaux **-DSTM32F4**) qui va nous intéresser dans la suite de cette présentation (voir figure 1).

En plus des drapeaux (choix de la déclinaison de la bibliothèque libopencm3 et de la nature du processeur, cortex-m3 ou cortex-m4), le script définissant l'organisation de la mémoire achève de définir la cible. Ce script, d'extension **.ld**,

est à ajuster pour chaque processeur : les scripts supportés sont dans le répertoire **ld**, fournissant l'argument de l'option **-T** de l'éditeur de lien au moment de rassembler les objets et les bibliothèques pour former un exécutable. Ces divers drapeaux sont visibles dans l'exemple de **Makefile** proposé sur https://github.com/jmfriedt/tp_freertos/blob/master/0no_freertos/Makefile.common.

La raison pour laquelle est proposé le support du STM32F103 est de permettre au lecteur qui ne possède pas de plateforme matérielle d'exécuter le code sur l'émulateur qemu tel qu'adapté par André Beckus. Dans le cas de cet exemple, nous vérifions le bon fonctionnement de la compilation pour STM32F103 – **-DSTM32F1 -DSTM32F10X_MD** – par l'exécution sous qemu de **arm-softmmu/qemu-system-arm -M stm32-p103 -serial stdio -kernel usart_cm3.bin**.

**NOTE**

Nous avons choisi de cadencer le STM32F410 par un quartz de 20 MHz. Ce choix, arbitraire, rompt avec l'habitude de cadencer les microcontrôleurs de la gamme STM32 avec un quartz de fréquence de résonance multiple de 8 MHz tel que supporté par défaut par libopencm3. La conséquence est de devoir manipuler à la main les paramètres de la boucle à verrouillage de phase (PLL) qui alimente les divers périphériques du microcontrôleur à partir de cet oscillateur de référence. Un ensemble de relations entre coefficients de multiplication et de division de la PLL, compte tenu des fréquences maximales acceptables par les divers périphériques, est résumé dans un document Excel fourni par ST-Microelectronics dans sa note d'application AN3988 et le logiciel associé **STSW-STM32091**. Dans notre cas, cela se résume par :

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/f4/memorymap.h>
#include <libopencm3/stm32/flash.h> // definitions du timer

// APB2 max=84 MHz but when the APB prescaler is NOT 1, the interface
clock is fed
// twice the frequency => Sysclk = 140 MHz, APB2=2 but Timers are
driven at twice that is 140.
const struct rcc_clock_scale rcc_hse_20mhz_3v3 = {
 .pllm = 20,                          // 20/20=1 MHz
 .plln = 280,                         // 1*280/2=140 MHz
 .pllp = 2,                           //      ^
 .pllq = 6,
 .hpre = RCC_CFGR_HPRE_DIV_NONE,
 .ppre1 = RCC_CFGR_PPRE_DIV_4,
 .ppre2 = RCC_CFGR_PPRE_DIV_2,
 .flash_config = FLASH_ACR_ICE | FLASH_ACR_DCE | FLASH_ACR_LATENCY_4WS,
 .ahb_frequency  = 140000000,
 .apb1_frequency = 35000000,
 .apb2_frequency = 70000000,
};

void core_clock_setup(void) {rcc_clock_setup_hse_3v3(&rcc_
hse_20mhz_3v3);} // custom version
```

Le microcontrôleur est ainsi légèrement sur-cadencé, mais nous n'avons pas observé de dysfonctionnement au cours de nos expérimentations, et nous pourrons ainsi générer le signal d'horloge à 70 MHz nécessaire au bon fonctionnement du synthétiseur numérique de fréquence.

En activant GPIOC12 comme broche connectée à la LED pour être en accord avec la configuration de la plateforme émulée par André Beckus (voir **hw/arm/stm32_p103.c** dans ses sources), nous obtenons en sortie :

```
LED On
0
LED Off
1
LED On
2
```

Ceci est en accord avec nos attentes sur les fonctionnalités du programme.

## 3. ORGANISATION D'UN PROGRAMME FREERTOS

Un programme exploitant les fonctionnalités de FreeRTOS doit déclarer les options dont il a besoin. Il est classique de placer ce fichier de configuration, nommé **FreeRTOSConfig.h** – documenté en détail sur www.freertos.org/a00110.html – aux côtés de **main.c** dans le répertoire **src** de travail, en dehors de l'arborescence de FreeRTOS. Ce fichier contient des informations telles que la fréquence du processeur, la fréquence à laquelle l'ordonnanceur considère quelle tâche exécuter, la taille de la pile allouée à l'ordonnanceur (tâche idle), le nombre de niveaux de priorités des tâches et nous verrons plus loin certaines fonctionnalités spécifiques telles que le support des mutex. Bien entendu, augmenter le nombre de fonctionnalités augmente les ressources requises : ce fichier est donc le levier pour adapter FreeRTOS à son application et aux ressources mises à disposition par le microcontrôleur.

La principale difficulté identifiée lors de l'utilisation de la bibliothèque libopencm3 tient au gestionnaire d'interruption *timer* qui cadence l'ordonnanceur.

FreeRTOS s'attend à ce que des fonctions aux noms bien précis soient appelées par le gestionnaire d'interruption. Par exemple, **FreeRTOSv9.0.0/FreeRTOS/Demo/CORTEX_M4F_STM32F407ZG-SK/FreeRTOSConfig.h** propose la solution pour la bibliothèque libstm32 fournie par ST Microelectronics sous forme de :

```
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler
```

Nous adaptons ceci pour libopencm3 selon les conseils de www.jiggerjuice.info/electronics/projects/arm/freertos-stm32f103-port.html pour obtenir :

```
void sv_call_handler(void)   {vPortSVCHandler();}
void pend_sv_handler(void)   {xPortPendSVHandler();}
void sys_tick_handler(void)  {xPortSysTickHandler();}
```

Une fois ces modifications apportées, nous sommes prêts à tester notre premier exemple de programme FreeRTOS :

```
#include "FreeRTOS.h"
#include "task.h"
#include "common.h"

void vLedsFloat(void* dummy)
{while(1)
  {Led_Hi1(); vTaskDelay(120/portTICK_RATE_MS);
   Led_Lo1(); vTaskDelay(120/portTICK_RATE_MS);
  }
}

void vLedsFlash(void* dummy)
{while(1)
  {Led_Hi2(); vTaskDelay(301/portTICK_RATE_MS);
   Led_Lo2(); vTaskDelay(301/portTICK_RATE_MS);
  }
}

void vPrintUart(void* dummy)
{portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
 // while (1) {} /* tester ce qui se passe avec
boucle infinie ! */
 while(1)
   {uart_puts("Hello World\r\n");
    vTaskDelayUntil(&last_wakeup_time, 500/
portTICK_RATE_MS);
   }
}

int main(void){
 Usart1_Init(); // inits clock as well
 Led_Init();
 if (!(pdPASS == xTaskCreate( vLedsFloat, (signed
char*) "LedFloat",64,NULL,1,NULL ))) goto hell;
 if (!(pdPASS == xTaskCreate( vLedsFlash, (signed
char*) "LedFlash",64,NULL,2,NULL ))) goto hell;
 if (!(pdPASS == xTaskCreate( vPrintUart, (signed
char*) "Uart",    64,NULL,3,NULL ))) goto hell;
 vTaskStartScheduler();
 hell:  while(1); // should never be reached
 return 0;
}
```

Ce programme nous informe de l'organisation générale d'un programme FreeRTOS : nos fonctions d'accès bas niveau initialisent les périphériques et les horloges, les tâches sont enregistrées auprès de l'ordonnanceur qui est ensuite exécuté. L'ordonnanceur ne doit jamais arriver en famine d'activité (toujours au moins une tâche en attente) et donc la fin du programme n'est jamais atteinte (équivalent de la boucle infinie principale dans un programme embarqué sans système d'exploitation). Ici encore, ce programme se teste dans qemu par :

```
$ arm-softmmu/qemu-system-arm -M stm32-p103
-serial stdio -kernel output/main.bin
LED Off
LED On
Hello World
LED Off
LED On
LED Off
LED On
Hello World
```

Les tâches sont enregistrées par **xTaskCreate()**, qui prend en argument la fonction à appeler, le nom (libre) qui permet d'identifier la tâche, la taille de la pile allouée à la tâche, les arguments, la priorité de la tâche, et un pointeur de retour de la structure représentant la tâche – le dernier paramètre étant souvent NULL si le descripteur de la tâche n'est pas utilisé dans le programme (par exemple pour influer sur l'ordonnanceur). Sur architecture ARM, la taille de la pile est automatiquement multipliée par quatre pour garantir l'alignement sur les adresses de 32 bits si la valeur est passée sous forme de **STACK_BYTES(val)**. La priorité de tâche est d'autant plus élevée que l'avant-dernier argument est élevé (convention opposée à celle des priorités sous Unix). On pourra s'en convaincre en retirant dans la tâche d'affichage **vPrintUart()** la fonction **vTaskDelayUntil()** qui propose à l'ordonnanceur de préempter la tâche : dans ces conditions, les LEDs ne clignotent plus. Au contraire si, tout en retirant la fonction permettant la préemption, nous inversons la priorité des trois tâches – en passant **vPrintUart()** en 1 et **vLedsFloat** en 3, on constatera que les LEDs se remettent à clignoter. En effet dans ce dernier cas, l'ordonnanceur s'autorise à préempter périodiquement la tâche de priorité la plus

faible pour proposer aux tâches de priorités plus élevées de prendre la main. De la même façon, bloquer par un `while (1) {}` une tâche de faible priorité n'empêchera pas une tâche de forte priorité d'être préemptée, mais une telle boucle infinie dans une tâche de priorité la plus élevée se traduit par un blocage de l'exécution.

**NOTE**

Toutes nos expériences ont exploité la version d'André Beckus de qemu, facile à modifier et à adapter à nos besoins puisque épurée au maximum pour supporter l'ARM Cortex-M3. Cependant, un projet s'est scindé et des développeurs de l'environnement intégré de développement **Eclipse** proposent une version bien plus riche de qemu disponible sur github.com/gnuarmeclipse/qemu/releases/tag/gae-2.8.0-20161227. Cette version reste fonctionnelle en ligne de commandes, tel que décrit sur gnuarmeclipse.github.io/qemu/options/. Pour nos exemples, nous pouvons exécuter sur un exemple qui fait clignoter deux LEDs :

```
$ qemu-system-gnuarmeclipse --verbose --verbose -M STM32-P103 \
-nographic -d unimp,guest_errors -kernel $FREERTOS_HOME/output/main.bin
```

Cela se traduit par :

```
GNU ARM Eclipse 64-bits QEMU v2.8.0 (qemu-system-gnuarmeclipse).
Board: 'STM32-P103' (Olimex Prototype Board for STM32F103RBT6).
Device file: '[...]/qemu/devices/STM32F103xx-qemu.json'.
Device: 'STM32F103RB' (Cortex-M3 r0p1, MPU, 4 NVIC prio bits, 43
IRQs), Flash: 128 kB, RAM: 20 kB.
Image: '[...]/0no_freertos/usart_cm3.elf'.
Command line: (none).
Load   1784 bytes at 0x08000000-0x080006F7.
Load     12 bytes at 0x080006F8-0x08000703.
Cortex-M3 r0p1 core initialised.
'/machine/mcu/stm32/RCC', address: 0x40021000, size: 0x0400
'/machine/mcu/stm32/FLASH', address: 0x40022000, size: 0x0400
'/machine/mcu/stm32/PWR', address: 0x40007000, size: 0x0400
'/machine/mcu/stm32/AFIO', address: 0x40010000, size: 0x0400
'/machine/mcu/stm32/EXTI', address: 0x40010400, size: 0x0400
'/machine/mcu/stm32/GPIOA', address: 0x40010800, size: 0x0400
'/machine/mcu/stm32/GPIOB', address: 0x40010C00, size: 0x0400
'/machine/mcu/stm32/GPIOC', address: 0x40011000, size: 0x0400
'/machine/mcu/stm32/GPIOD', address: 0x40011400, size: 0x0400
'/machine/mcu/stm32/GPIOE', address: 0x40011800, size: 0x0400
'/peripheral/led:red' 12*10 @(331,362) active low '/machine/mcu/
stm32/GPIOC',12
QEMU 2.8.0 monitor - type 'help' for more information^M
(qemu) Cortex-M3 r0p1 core reset.

NVIC: SCR and CCR unimplemented
[led:red off]
[led:red on]
[led:red off]
[led:red on]
```

Ce qui est bien le comportement attendu. Cependant, nous n'avons pas réussi à convaincre le port série d'afficher un message sur la sortie standard ou un client telnet, même en n'utilisant que le port 2, le seul supporté par la carte Olimex STM32-P103.

**NOTE**

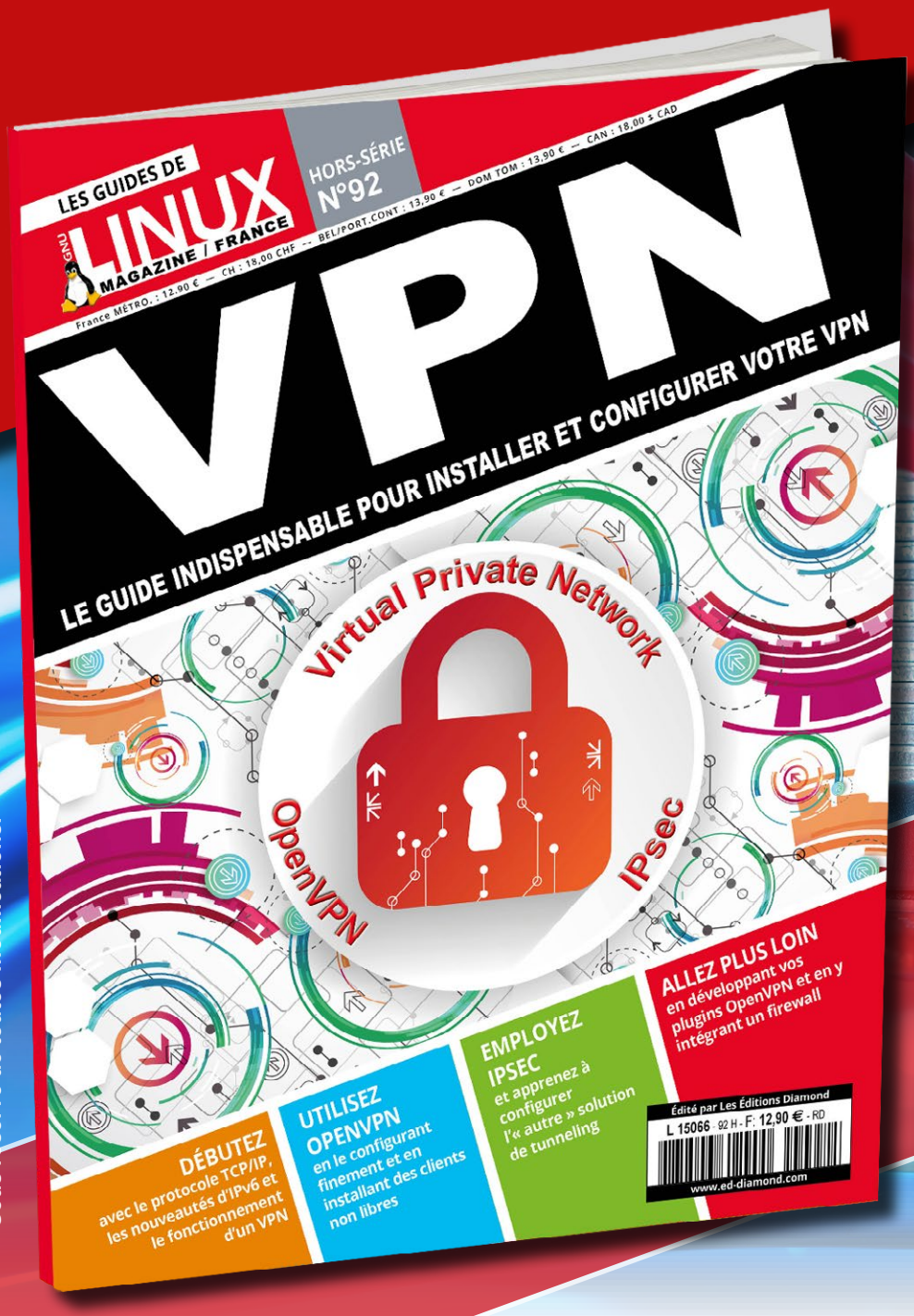
La fréquence du processeur, qui détermine la fréquence du *timer* qui cadence l'ordonnanceur, est indiquée dans `src/FreeRTOSConfig.h`, indépendamment des paramètres que nous avons indiqués dans la configuration des horloges dans `common` (constante `configCPU_CLOCK_HZ`). Le *timer* qui se charge de cadencer l'ordonnanceur sur ARM est `SysTickTimer`. Cette variable détermine aussi la constante utilisée dans le délai implémenté par `portTICK_RATE_MS` en argument de `vTaskDelay()`.

Le lecteur qui désire tenter d'exécuter ce code sous **GNU/Linux** pourra tenter d'exploiter l'émulateur FreeRTOS sur *threads* POSIX : la version 9.0.0 de FreeRTOS est supportée par github.com/megakilo/FreeRTOS-Sim.git. L'exemple ci-dessus finit par fonctionner, mais nécessite d'initialiser un certain nombre de fonctionnalités qui n'ont pas été nécessaires pour faire tourner FreeRTOS sur microcontrôleur. En particulier, toutes les initialisations de `vCreateAbortDelayTasks()` à `vStartTaskNotifyTask()` dans `Project/main.c` doivent être maintenues pour initialiser les structures de données émulant le matériel. Nous obtenons bien dans un terminal à l'issue de la compilation un exécutable fonctionnel sous GNU/Linux qui nous informe du séquencement des tâches :

```
Running as PID: 6651
Led_Hi1
Timer Resolution for Run
TimeStats is 100 ticks
per second.
Hello World
Led_Hi2
Led_Hi1
Led_Lo1
Led_Hi1
Led_Lo2
Led_Lo1
Led_Hi1
Hello World
Led_Lo1
```

# DISPONIBLE DÈS LE 15 SEPTEMBRE

## GNU/LINUX MAGAZINE HORS-SÉRIE N°92 !




L'INDISPENSABLE POUR INSTALLER & CONFIGURER VOTRE VPN

# NE LE MANQUEZ PAS

## CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

## http://www.ed-diamond.com

Ce premier exemple nous permet déjà d'appréhender l'importance des priorités des tâches. Imaginons que nous introduisions maladroitement dans la tâche **vPrintUart()** une boucle infinie vide **while (1) {}** juste avant la boucle proposée dans cet exemple. Dans la configuration qui est proposée, où **vPrintUart()** est de priorité la plus élevée, cette boucle infinie ne peut pas être interrompue par l'ordonnanceur, et le programme est bloqué, aucun message n'apparaît sur le port série et aucune LED ne clignote. Si au contraire nous inversons les priorités de **vPrintUart()** et **vLedsFloat()**, passant le clignotement des LEDs prioritaire devant l'affichage du message, alors la boucle infinie interdit toujours l'affichage d'un message sur le port série (la tâche **vPrintUart()** ne sort jamais de sa boucle infinie), mais au moins les diodes clignotent puisque l'ordonnanceur préempte les tâches de priorité les plus élevées.

## 3.1 Informations sur l'utilisation de la pile et des tâches

Pour le moment, nous n'avons que créé trois tâches qui s'exécutent apparemment simultanément : deux qui font clignoter une LED et la dernière qui communique. Le gain de FreeRTOS est pour le moment douteux. Que peut nous amener l'environnement exécutif ? Le premier gain significatif qui permet d'appréhender un problème classique dans le développement de systèmes embarqués est la gestion de la pile. La pile est la zone temporaire où une tâche stocke les informations qui lui sont relatives au cours de son exécution (variables locales, emplacement du pointeur de programme lors des sauts aux fonctions) : chaque tâche contient sa propre pile, et corrompre la pile est la garantie de crash du programme. La pile (*stack*), généralement placée à la fin de la RAM, puisqu'empiler une variable se traduit par un décrément du pointeur de pile, se distingue du tas (*heap*) qui se trouve généralement en début de RAM, et dont le contenu a la durée de vie de l'exécution du programme (contrairement à la pile qui n'est valable que pendant la durée d'exécution de la tâche). FreeRTOS fournit plusieurs mécanismes pour informer le programmeur du statut de la pile et de son risque de corruption. L'option **#define configCHECK_FOR_STACK_OVERFLOW 2** dans **FreeRTOSConfig.h**, qui nécessite la définition du gestionnaire d'événement **void vApplicationStackOverflowHook(TaskHandle_t xTask, signed char *pTaskName)**, permet d'appeler ledit gestionnaire d'évènements si une corruption de pile survient. Par exemple, si nous définissons le code suivant :

```
void vApplicationStackOverflowHook(TaskHandle_t
xTask, signed char *pTaskName)
{while(1) {uart_puts("\r\nStack: ");uart_
puts(pTaskName);vTaskDelay(301/portTICK_RATE_MS);}
}
```

Nous constatons que réduire la taille de la pile allouée à la tâche **vPrintUart()** à deux mots de 32 bits se traduit par :

```
Stack: Uart
Stack: Uart
```

Ceci indique que la fonction d'affichage d'un message sur le port série a dépassé la taille de pile qui lui a été allouée.

De la même façon, une taille insuffisante par défaut de pile **configMINIMAL_STACK_SIZE**, qui définit en particulier l'allocation de mémoire de la tâche qui ne fait rien (en attente de tâches à activer), se traduit par :

```
Stack: IDLE
Stack: IDLE
```

Ici encore nous avons une tâche qui a dépassé la taille de pile qui lui a été allouée. Bien entendu, un programme fonctionnel peut activer ce mode de déverminage et doit ne jamais voir la fonction **vApplicationStackOverflowHook()** appelée dans la condition nominale de fonctionnement où chaque tâche confine ses allocations en mémoire à la pile qui lui a été allouée.

Dans la même veine, une tâche peut essayer de savoir combien de place il lui reste à allouer sur sa pile. Pour ce faire, l'option de configuration **#define INCLUDE_uxTaskGetStackHighWaterMark 1** est activée et la fonction **uxTaskGetStackHighWaterMark(NULL)** (**NULL** signifiant que la tâche veut connaître son propre état, sinon il faut fournir le *handler* de la tâche consultée) renvoie l'information recherchée.

Enfin, nous pouvons afficher la liste des tâches enregistrées auprès de l'ordonnanceur, leur état et la quantité de pile restante. Ici encore ces options s'activent dans le fichier de configuration, cette fois par :

```
#define configUSE_TRACE_FACILITY     1
#define configUSE_STATS_FORMATTING_FUNCTIONS   1
```

Et en allouant un tableau d'au moins 40 caractères par tâche dont l'état doit être affiché, pointeur qui sera fourni en argument à la fonction **vTaskList** qui remplit le tableau de caractères de la chaîne à afficher :

```
void vPrintUart(void* dummy)
{char c[256];
 portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
 while(1){uart_puts("\nHello World\r\n");
          vTaskList(c); uart_puts(c);
          vTaskDelayUntil(&last_wakeup_
time, 500/portTICK_RATE_MS);
        }
}
```

Ainsi, le code précédant affiche toutes les 500 ms l'état des tâches enregistrées auprès de l'ordonnanceur. Bien entendu, la taille de la pile associée à l'initialisation de **vPrintUart** doit être plus grande que le tableau qui y est créé.

La sortie de ce dernier exemple sera de la forme :

```
Hello World
Uart            R       4       301     3
IDLE            R       0       0       4
LedFlash        B       4       242     2
LedFloat        B       4       242     1
```

Dans les colonnes successives nous voyons le nom que nous avons arbitrairement attribué à la tâche (second argument de **xTaskCreate()**, le statut de la tâche (**R**=*Ready*, **B**=*Blocked*, **D**=*Deleted* et **S**=*Suspended*=*Blocked* sans *timeout*), la priorité de la tâche et la quantité de mémoire encore disponible sur la pile (plus cette valeur se rapproche de **0**, plus le risque de corruption de la pile devient important), puis finalement la position dans la file de l'ordonnanceur de la tâche. Dans tous ces exemples, on pourra s'entraîner à faire varier la taille de la pile allouée lors de la déclaration de la tâche **vPrintUart** et constater la cohérence des informations fournies par FreeRTOS. Notons que seuls quelques KB de mémoire ont été nécessaires pour faire tourner FreeRTOS : les programmes fournis dans ce document fonctionnent tous sur le STM32F100 de la STM32VL-Discovery board (8 KB de RAM) sous réserve de ne pas dilapider les ressources, par exemple en se liant à **newlib**.

## 3.2 Échange de données par files d'attente

Nous savons donc créer des tâches, garantir l'intégrité de leur pile, et créer des variables locales à chaque tâche. Il est courant que plusieurs tâches aient à échanger des données pour y appliquer des traitements successifs. Nous ne savons cependant pas quel est l'état de chaque tâche au moment de la production ou la consommation de données, et il nous faut un mécanisme permettant à deux tâches d'échanger des données sans faire d'hypothèse sur le comportement de l'ordonnanceur. Un tel mécanisme est fourni par les queues, qui sont des FIFO (*First In, First Out*) entre deux tâches.

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "common.h"
#include <stm32/gpio.h>

void vLedsFloat(void* dummy)
[...]

void vLedsFlash(void* dummy)
[...]

void vPrintUart(void* dummy)
[...]

xQueueHandle qh = 0;

void task_tx(void* p)
{int myInt = 0;
  while(1)
    {myInt++;
     if(!xQueueSend(qh, &myInt, 100))
        uart_puts("Failed to send item to queue
within 500ms");
     vTaskDelay(1000);
    }
}

void task_rx(void* p)
{char c[10];
 int myInt = 0;
 while(1)
    {if(!xQueueReceive(qh, &myInt, 1000))
       uart_puts("Failed to receive item within
1000 ms");
     else {c[0]='0'+myInt;c[1]=0;
           uart_puts("Received: ");uart_
puts(c);uart_puts("\r\n");
          }
    }
}

int main()
{Led_Init();
 Usart1_Init();

  qh = xQueueCreate(1, sizeof(int));

// activer ces fonctions fait atteindre le
timeout de transfert de donnees dans la queue
//   xTaskCreate( vLedsFloat, ( signed char * )
"LedFloat", 128, NULL, 2, NULL );
//   xTaskCreate( vLedsFlash, ( signed char * )
"LedFlash", 128, NULL, 2, NULL );
//   xTaskCreate( vPrintUart, ( signed char * )
"Uart",     128, NULL, 2, NULL );
   xTaskCreate(task_tx, (signed char*)"t1",
(128), 0, 2, 0);
   xTaskCreate(task_rx, (signed char*)"t2",
(128), 0, 2, 0);
   vTaskStartScheduler();
hell: while(1) {};
    return 0;
}
```

Dans cet exemple, deux fonctions **task_tx()** et **task_rx()** échangent des données avec une condition de *timeout* qui informe d'un délai excessif entre les transactions. On se convaincra du bon fonctionnement de ce mécanisme en ajoutant les fonctions de commutation des LEDs et de communication – identiques à celles vues auparavant – qui introduisent des latences additionnelles et font atteindre la condition de délai dépassé, en fonction des priorités des diverses tâches.

## 3.3 Protection de l'accès concurrent aux ressources communes à plusieurs tâches

Le danger de partager des structures de données globales – en particulier tableaux servant de tampons lors de l'acquisition et de traitements de données – entre tâches tient en la cohérence des informations. Si une première tâche est en train de manipuler les données requises par une seconde tâche, les valeurs contenues dans les variables peuvent devenir incohérentes. Une méthode pour éviter ce problème est de garantir qu'un seul processus peut accéder à une variable à un instant donné : accès MUTuellement EXclusif ou mutex. Chaque accès à une variable est encadré par un mutex et l'exécution de la tâche n'est possible que si le mutex est débloqué : une tâche qui tente de s'approprier un mutex déjà pris est mise en attente jusqu'à ce que ce mutex soit relâché, et réussir à prendre un mutex garantit qu'aucune autre tâche n'accède aux structures de données protégées jusqu'à ce que ce mutex soit rendu. En pratique, la mise en œuvre peut ressembler à :

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "common.h"
#include "stdlib.h"
// rand

// #define avec_mutex

volatile int global=0;
// la variable incrémentée par les 2 tâches *8
#ifdef avec_mutex
SemaphoreHandle_t xMutex;
// pour FreeRTOSv9
#endif
void task_rx(void* p)
{   char aff[10];
    char *t=(char*)p;
    int myInt = 0;
    volatile int local;
    for (myInt=0;myInt<8;myInt++)
     {
#ifdef avec_mutex
      xSemaphoreTake( xMutex, portMAX_DELAY );
#endif
      local=global;
      local++;
      uart_puts(t);
// cette opération nous fait perdre beaucoup de temps ... il y a
// toutes les chances pour que la seconde tache se lance pendant cet
// intervalle de temps ... mais la seconde tâche va cherche global
// qui n'a pas encore été incrémenté et le résultat sera erroné
// puisque les 16 tâches auront fait une somme de 8
        global=local;

#ifdef avec_mutex
    xSemaphoreGive( xMutex );
#endif
    vTaskDelay( ( rand() & 0x5 ) );  // essayer de déplacer le delay sous cette '}' pour ne pas alterner
    }
    aff[0]=' ';aff[1]=global+'0';aff[2]=' ';aff[3]=0;uart_puts(aff);
    while (1) vTaskDelay( ( rand() & 0x5 ) ); // on n'a jamais le droit de finir toutes les tâches
}

int main()
{ Led_Init();
  Usart1_Init();
  srand( 567 );
#ifdef avec_mutex
  xMutex = xSemaphoreCreateMutex();
#endif
  xTaskCreate(task_rx, (signed char*)"t1", 128, "111111111111111111111111111111111111111111111111\r\n\0", 1, 0);
  xTaskCreate(task_rx, (signed char*)"t2", 128, "222222222222222222222222222222222222222222222222\r\n\0", 1, 0);
  vTaskStartScheduler();
  while(1);
  return 0;
}
```

L'exemple ci-dessus considère une variable globale **global** qui est manipulée par deux tâches. Chacune de ces tâches effectue un calcul long, illustré ici par un transfert sur le port série (une tâche très longue à l'échelle du temps de calcul : un transfert de 45 caractères au débit de 115200 bauds selon un encodage 8N1 sur bus RS232 prend près de 4 ms). Si nous omettons de protéger l'accès à **global**, les deux tâches risquent d'en stocker dans **local** une valeur à un certain temps, de toutes deux effectuer une addition sur cette même valeur initiale, et la première tâche stocke son résultat suivi de la seconde tâche qui stocke le même résultat, induisant une erreur de calcul puisque l'addition aurait dû s'effectuer deux fois au lieu d'une. La conséquence de la préemption d'une tâche par l'ordonnanceur alors qu'un affichage est en cours est donc que d'une part les chaînes de caractères ne sont pas affichées dans leur intégralité, mais sont interlacées, et la somme de **global** n'est pas nécessairement correcte à l'issue du calcul :

# ACTUELLEMENT DISPONIBLE

## LINUX PRATIQUE N°103


# NOS CONSEILS POUR ACCÉLÉRER VOTRE SYSTÈME !

## NE LE MANQUEZ PAS

**CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :**

**http://www.ed-diamond.com**

```
22222222222222222222221111111111111111111111122
22222222222222222
1111111111111111111
22222222222222222222222221111111111111111111111
1111112222222222222222
...
 :  :
```

Les deux symboles ":" à la fin du calcul (code ASCII de "0" plus **12**) indiquent que la somme de **global** n'est pas **16** comme escompté (deux tâches qui incrémentent huit fois), mais moindre à cause du conflit d'utilisation de la ressource commune qu'est la variable globale.

Le palliatif consiste à décommenter l'option de précompilation **#define avec_mutex** qui active la protection de l'accès à **global** puis **local** par une unique tâche à un instant donné. Ce résultat est confirmé par chaque chaîne de caractères passée en argument aux deux instances de la tâche qui s'affichent entièrement et sans être interrompues par l'autre tâche que l'ordonnanceur s'autorise à préempter compte tenu de la durée excessive de la communication par RS232.

```
...
22222222222222222222222222222222222222222
11111111111111111111111111111111111111111
22222222222222222222222222222222222222222
11111111111111111111111111111111111111111
22222222222222222222 ? 22222222222222222222
 @
```

Nous avons ici le "?" et "@" qui sont respectivement le code ASCII de "0" plus **15** et **16**, soit le résultat attendu de la dernière exécution de chacune des tâches qui incrémente chacune **global** en se garantissant la ressource tant que l'affichage n'est pas achevé.

Compiler ces programmes nécessite d'informer FreeRTOS de notre intention d'utiliser les mutex et donc de charger les fonctionnalités associées. Nous devons pour cela ajouter la configuration **#define configUSE_MUTEXES 1** dans **FreeRTOSConfig.h**.

## 3.4 Application à l'analyseur de réseau

Nous avons complété ce rapide tour d'horizon de FreeRTOS, reste maintenant à mettre ces concepts en œuvre dans un cas concret. Nous nous intéressons à la réalisation d'un analyseur de réseau embarqué (voir figure 4), un instrument susceptible de caractériser la fonction de transfert, en amplitude et en phase, d'un dispositif radiofréquence **[5]**. Une description détaillée des choix de conception de l'instrument est proposée sur jmfriedt.free.fr/network_analyzer.pdf et dépasse le cadre de cette présentation, qui se limite à l'implémentation du logiciel sur FreeRTOS. Nous avons donc, en plus du microcontrôleur, besoin de trois composants :

1. Une source radiofréquence. Le synthétiseur numérique direct (*Direct Digital Synthesizer* – DDS) Analog Devices AD9834 fournit les fonctionnalités nécessaires : un composant totalement numérique ne nécessitant que peu de composants passifs annexes, un pas de fréquence fin programmable depuis le microcontrôleur, et une puissance de sortie faible, mais suffisante pour notre mesure. Un DDS est cadencé par une horloge externe, qui dans notre cas sera fournie par une sortie *timer* du microcontrôleur : cadencé à 140 MHz, le STM32F410 peut fournir une horloge à $f_{CK} = 70$ MHz sur une sortie *timer*, dans la plage acceptable par l'AD9834. La fréquence de sortie $f_0$ est liée à $f_{CK}$ et au mot **w** programmé par le microcontrôleur dans le registre adéquat par $f_0 = (w / 2^{28}) \times f_{CK}$ avec $2^{28}$ la taille de l'accumulateur interne au DDS (28 bits). Le pas de fréquence est donc $f_{CK} / 2^{28} \simeq 0{,}3$ Hz, suffisant pour caractériser des dispositifs de fort facteur de qualité (donc de faible encombrement spectral).

2. Un détecteur de puissance. Une puissance – ou plus simplement une tension – s'obtient classiquement par un redressement du signal suivi d'un filtrage passe-bas pour obtenir l'amplitude moyenne du signal. Au lieu de nous battre à polariser convenablement une diode radiofréquence pour concevoir un redresseur fonctionnel dans une large plage de fréquences et de puissances d'entrée, nous nous contentons d'utiliser un détecteur de puissance commercialement disponible chez **Linear Technology**, le **LT5537**. Ce composant, fonctionnel de quelques kHz au GHz et pour des puissances aussi basses que -75 dBm, propose par ailleurs une mise en veille pour une économie d'énergie s'il n'est pas utilisé.

3. Finalement, un détecteur de phase (voir figure 5). Nous choisissons ici la voie du numérique, avec une implémentation classique de mélangeur par porte logique XOR (OU eXclusif). En effet dans une telle porte, deux signaux en phase donnent toujours une sortie nulle (**0 XOR 0 = 1 XOR 1 = 0**) tandis que deux signaux en opposition de phase donnent toujours une sortie égale

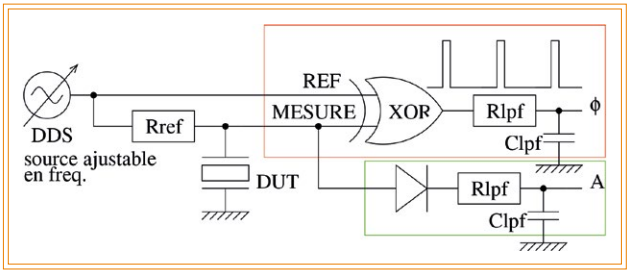



*Fig. 5 : Principe de la mesure d'amplitude (vert) et de phase par porte XOR (rouge).*

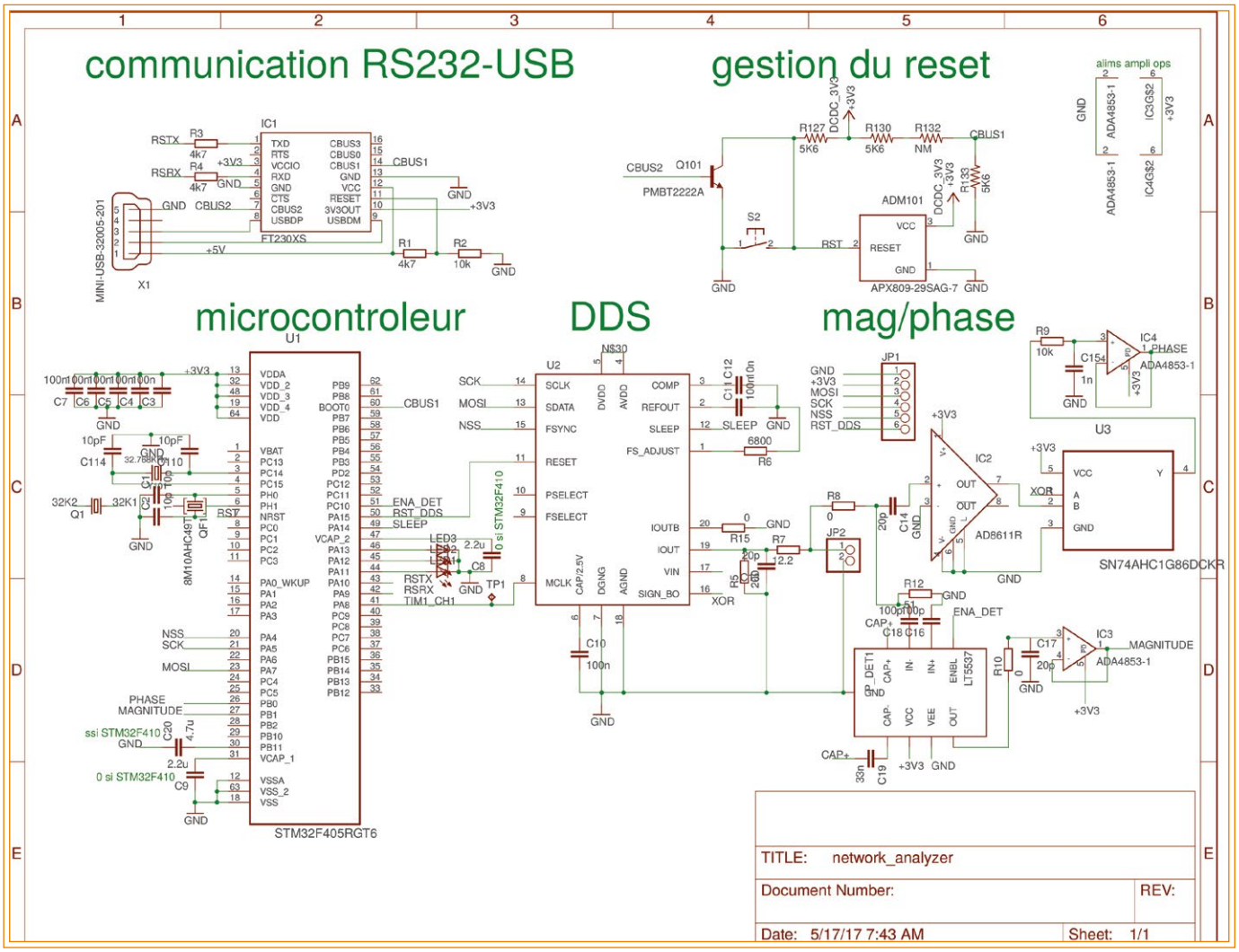



*Fig. 4 : Schéma de l'analyseur de réseau complet. L'ensemble des composants présents sur ce circuit – microcontrôleur, synthèse de fréquence, détecteur de puissance et de phase, coûte moins de 30 euros.*

à la tension d'alimentation (`0 XOR 1 = 1`). Entre les deux, un filtre passe-bas en sortie du XOR donne une tension continue proportionnelle au déphasage entre les deux signaux d'entrée, signal de référence issu du DDS et signal qui a sondé le dispositif en cours de caractérisation (DUT). Alors que le DDS fournit une sortie numérique de fréquence égale à celle générée sur la sortie analogique, le niveau du signal analogique qui a sondé le DUT est insuffisant pour attaquer la seconde entrée du composant numérique qu'est la porte XOR. Un comparateur est donc nécessaire pour saturer le signal en sortie du DUT et permettre une mesure de phase.

Une fois ces concepts mis en œuvre sur un circuit fonctionnel, il nous reste à programmer le microcontrôleur pour balayer séquentiellement les fréquences dans la plage qui nous intéresse, et pour chacune de ces fréquences lire sur deux voies de conversion analogique-numérique du microcontrôleur les valeurs de phase et d'amplitude pour tracer la caractéristique du dispositif analysé. L'utilisateur voulant pouvoir programmer la plage de fréquences et le pas de fréquence du balayage, le programme FreeRTOS se compose de trois tâches :

1. une tâche indiquant que l'instrument est sous tension et en attente d'ordres de l'utilisateur, allumant et éteignant des diodes électroluminescentes (LEDs) dans un motif de chenillard ;
2. une tâche de communication chargée de recevoir les paramètres de l'utilisateur : fréquence de début, fréquence de fin, pas de fréquence et attente entre deux mesures ;
3. une tâche de mesure chargée de programmer le DDS et pour chaque fréquence, de mesurer la phase et l'amplitude et transmettre ces informations à l'utilisateur.

Ces trois tâches doivent donc interagir (voir figure 6, page suivante) : la tâche LED doit cesser de faire clignoter les diodes

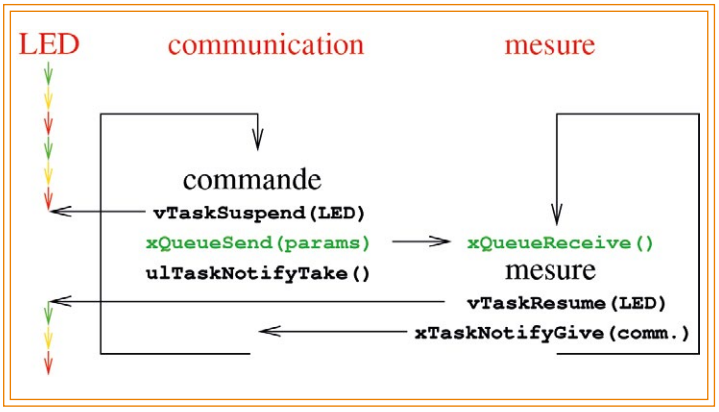



*Fig. 6 : Séquencement des tâches – en rouge en haut du schéma – de l'application : la communication d'une séquence de caractères depuis le PC débloque la tâche communication qui bloque la tâche LED et lance la mesure. Une fois la mesure achevée, les LED se remettent à clignoter et la communication est en attente d'un nouvel ordre de mesure. Le temps s'écoule de haut en bas.*

au cours de la mesure, car l'appel de courant fait varier la tension de référence des convertisseurs analogique-numériques, mais doit se remettre en marche à l'issue de la mesure, la tâche mesure doit être bloquée tant que les paramètres n'ont pas été acquis et la tâche de réception des paramètres doit communiquer les fréquences à la tâche de mesure. L'échange des paramètres se fait par les queues que nous avons vues auparavant. Puisqu'un unique pointeur est fourni comme argument du passage de paramètres par queue, une structure de données comprenant tous les paramètres est créée dont le pointeur sera passé dans la queue entre les tâches de communication et de mesure. Par contre, pour ce qui est de séquencer les opérations en bloquant et débloquant les diverses tâches, nous utilisons des mécanismes optimisés par FreeRTOS au lieu des classiques mutex pour garantir qu'une seule tâche accède à une ressource donnée à un instant donné. Pour ce faire, les mécanismes **`vTaskSuspend()`** et **`vTaskResume()`** permettent à une tâche d'interrompre ou reprendre l'exécution d'une autre tâche dont le descripteur (pointeur fourni comme dernier argument de **`xTaskCreate()`**) est fourni en argument. Par ailleurs, **`xTaskNotifyGive()`** permet à une tâche de débloquer une autre tâche qui est en attente par **`ulTaskNotifyTake()`**. La première fonction prend en argument le descripteur de la tâche à débloquer, tandis que la seconde fonction se contente d'un temps maximum au-delà duquel la tâche se réveillera – éventuellement infini. Enfin, l'échange d'informations par queues, **`xQueueSend()`** et **`xQueueReceive()`**, est aussi utilisé pour séquencer les opérations, toujours avec possibilité de placer un temps maximum (*timeout*) sur la réception si la réception de message ne doit pas être une condition bloquante. Si la réception de message n'a pas abouti avant le *timeout*, **`xQueueReceive()`** renverra **`pdFalse`** (www.freertos.org/a00118.html), permettant de connaître quelle condition a débloqué la tâche.

Une application de ces concepts, dont le code source complet est sur https://github.com/jmfriedt/tp_freertos/tree/master/FreeRTOS-Network-Analyzer-on-STM32, est proposée en figure 7 : un résonateur à onde de volume, tel que classiquement utilisé pour cadencer les microcontrôleurs, est caractérisé par cet instrument. Le paramètre clé d'un résonateur, en plus de sa fréquence de résonance qui détermine la fréquence du signal issu du circuit d'entretien de l'oscillation (oscillateur), est son facteur de qualité qui détermine sa capacité à emmagasiner de l'énergie et donc d'être insensible aux perturbations extérieures, formant ainsi un oscillateur stable. Le facteur de qualité, au-delà de ces considérations énergétiques, détermine la largeur de la raie spectrale caractérisant le résonateur : un résonateur à onde de volume en quartz présente classiquement un facteur de qualité **`Q`** de l'ordre de **$10^5$** pour des fréquences de fonctionnement **`f`** de quelques MHz : la largeur de la résonance est alors **`Δf = f / Q`** de l'ordre de quelques dizaines à quelques centaines de Hz (la valeur choisie ici pour **`Q`** est volontairement surestimée pour choisir **`Δf`** suffisamment petit pour correctement caractériser la réponse du dispositif). Le pas de fréquence lors de la programmation du DDS doit donc être petit devant **`Δf`** : si par exemple la résonance doit se caractériser sur **`N`** points, on choisira un pas de fréquence lors de la programmation du DDS de **`Δf / N`**. Dans notre cas, nous choisissons un pas de 5 Hz sur une plage de **`± 500`** Hz autour de la résonance, soit **`1000 / 5 = 200`** points de mesure qui prennent quelques dizaines de secondes à être transférés par liaison RS232. Le balayage ne doit pas être trop rapide, faute de quoi l'énergie emmagasinée à la résonance dans le résonateur n'est pas totalement dissipée lors de la mesure suivante, déformant la fonction de transfert : nous devons attendre quelques constantes de temps **`Q / (πf) ≃ 6`** ms (à 5 MHz) entre deux mesures pour garantir l'indépendance des mesures successives. Ce temps d'attente pourra être mis à profit pour communiquer les données au PC, la liaison RS232 étant lente.

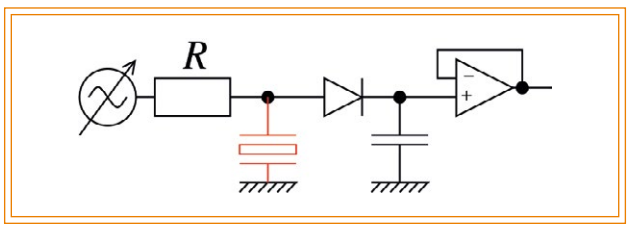
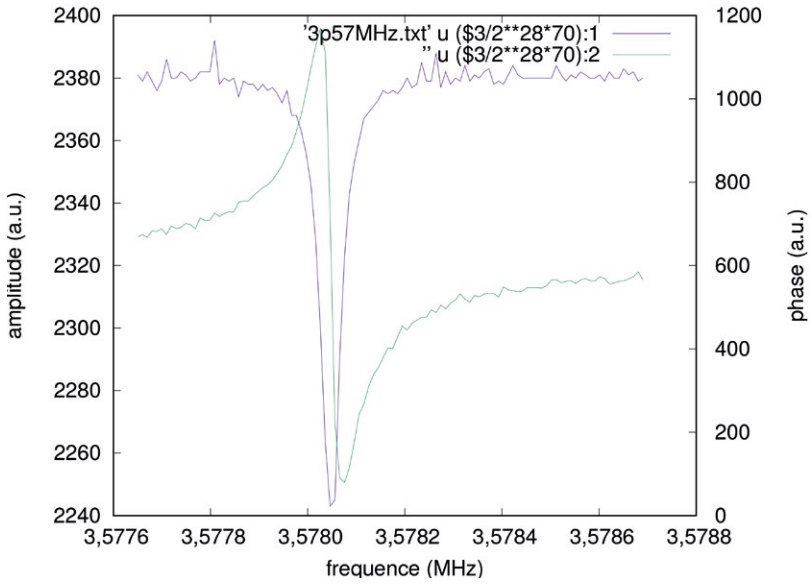



*Fig. 8 : Mesure du coefficient de réflexion du dispositif : la fraction de la puissance qui atteint le détecteur de puissance est celle qui n'est pas transmise par le dispositif vers la masse. L'impédance de référence est R, le DUT est en rouge.*



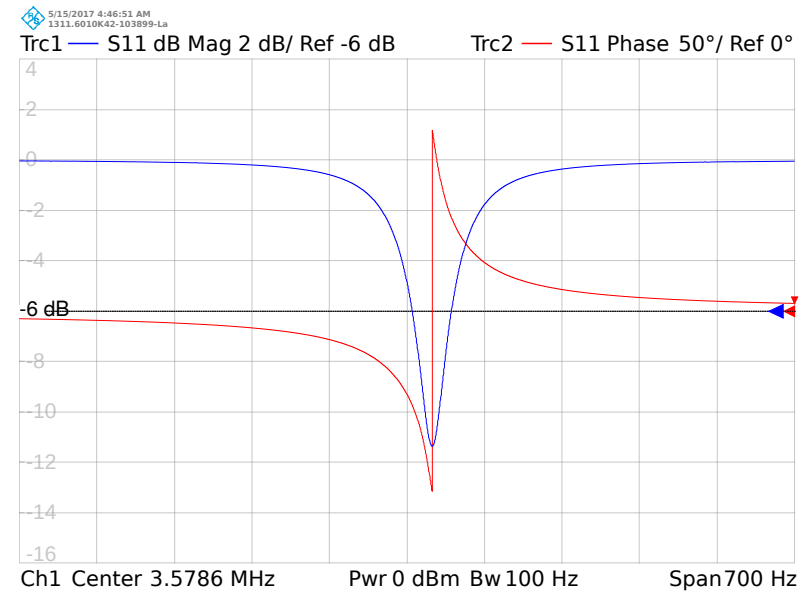




*Fig. 7 : En haut : mesure d'un résonateur à onde de volume par le circuit comprenant un microcontrôleur, un DDS, un détecteur de puissance et un détecteur de phase. La légende en haut à droite du graphique contient les commandes gnuplot permettant l'affichage des courbes avec un axe des abscisses gradué en MHz au lieu des mots représentant la fréquence dans le DDS. En bas : caractérisation du même dispositif avec un analyseur de réseau commercial.*

Dans la configuration de la mesure que nous proposons, dans laquelle le DUT est inséré dans un pont diviseur de tension vers la masse (voir figure 8), la résonance s'observe sous forme d'une chute de puissance arrivant au détecteur : hors résonance l'intégralité du signal fourni par le DDS arrive au détecteur, et une fraction de cette puissance est dirigée vers la masse lorsque la résonance est atteinte. La fraction de la puissance dirigée vers la masse est donnée par la relation classique du pont diviseur de tension entre une impédance de référence – classiquement **50Ω**, ici la résistance en sortie du DDS – et le DUT – et la conservation de l'énergie nous indique la fraction résultante qui est dirigée vers le détecteur de puissance. La figure 7 présente bien une chute de puissance lorsque la résonance est atteinte, associée à une rotation de phase.

Le lecteur désireux de reproduire ces mesures pourra profiter d'une interface graphique qui simplifie la prise en main des paramètres de configuration, tel que présenté sur la figure 9, page suivante.

# CONCLUSION

Nous avons conçu un circuit autour du STM32F410 en vue de réaliser un analyseur de réseau radiofréquence pour la caractérisation des dispositifs fonctionnant dans la gamme de quelques centaines de kHz à quelques dizaines de MHz. Ce développement a été l'opportunité d'explorer les fonctionnalités de FreeRTOS, porté à ce microcontrôleur grâce à l'utilisation de la bibliothèque libopencm3. Nous avons vu que la simplicité de l'architecture de FreeRTOS le rend facilement exploitable sur n'importe quel microcontrôleur muni de quelques KB de mémoire.

Les applications de ce circuit sont multiples, au-delà de la caractérisation des dispositifs radiofréquences : dans le cas particulier des résonateurs à onde de volume décapsulés, leur sensibilité à leur environnement en fait d'excellents capteurs. En particulier, la variation de la fréquence de résonance avec l'épaisseur d'une couche adsorbée en surface du substrat piézoélectrique (chargé de convertir le signal électromagnétique en

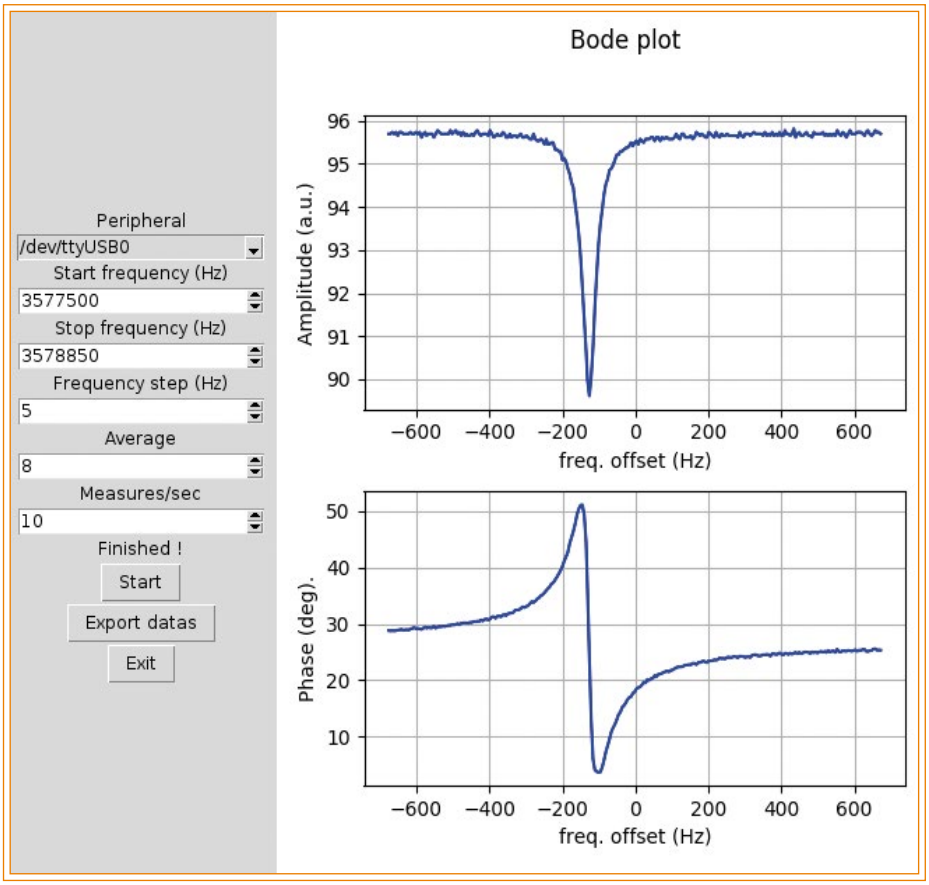



*Fig. 9 : Interface graphique pour la configuration de l'instrument et la restitution des courbes – le code source est disponible sur* github.com/jmfriedt/tp_freertos/blob/master/FreeRTOS-Network-Analyzer-on-STM32/Interface.py *pour l'échange des informations sur port série et l'affichage des informations acquises par les convertisseurs analogique-numériques.*

onde élastique) en fait des capteurs chimiques à la mode, dont l'électronique présentée dans ce document permet de mesurer finement la réponse en observant l'évolution de la phase du résonateur à la résonance, dans une application abusivement nommée de « microbalance à quartz ». De façon générale, les capteurs à sortie de fréquence fournissent une excellente sensibilité de par la capacité à mesurer avec une très grande précision les variations de fréquences, ici uniquement limitée par la stabilité de l'oscillateur qui cadence le microcontrôleur (et donc le DDS au travers de la sortie *timer*).

Le choix d'un synthétiseur numérique de fréquence cadencé à 70 MHz pourrait laisser penser que ce circuit ne peut fonctionner que jusqu'à une vingtaine de MHz dans les conditions nominales de filtrage des raies parasites générées par ce composant numérique. Il n'en est rien : lorsque le composant cadencé à $f_{CK}$ est programmé pour générer un signal à $f_0$, les raies « parasites » sont déterministes et situées à $f_{CK}$, $f_{CK} - f_0$ et $f_{CK} + f_0$. En programmant $f_0 = 25$ MHz, cette dernière raie se trouve à $70 + 25 = 95$ MHz, en plein dans la bande FM commerciale et donc détectable par tout récepteur FM, par exemple ceux équipant les téléphones portables. Le lecteur est donc encouragé à poursuivre l'exploration dans cette voie de transmission d'informations sur porteuse radiofréquence, la majorité des concepts de l'émission radiofréquence étant accessibles par ce circuit, avec la souplesse de la configuration logicielle du DDS par le microcontrôleur. ■

## RÉFÉRENCES

**[1]** CARRY É., FRIEDT J.-M, *« L'environnement Arduino est-il approprié pour enseigner l'électronique embarquée ? »*, conférence CETSIS, 2017, disponible sur jmfriedt.free.fr/cetsis2017_arduino.pdf

**[2]** *« RM0401 – Reference manual, STM32F410 advanced ARM-based 32-bit MCUs »*, octobre 2015, Doc ID 027812 Rev 2

**[3]** STM32F100 ARM Cortex-M3 FreeRTOS Demo sur www.freertos.org/FreeRTOS-for-Cortex-M3-STM32-STM32F100-Discovery.html

**[4]** BARRY R., *« Using the FreeRTOS real time kernel – A practical guide »*, 2009 et sa mise à jour **:** BARRY R., *« Mastering the FreeRTOS Real Time Kernel »*, 2016, sur www.openrtos.net/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

**[5]** RABUS D., FRIEDT J.-M., BALLANDRAS S., MARTIN G., CARRY É., BLONDEAU-PATISSIER V., *« High-sensitivity open-loop electronics for gravimetric acoustic-wave-based sensors »*, IEEE Trans. Ultrason. Ferroelectr. & Freq. Control. 60 (6), 2013, p. 1219 à1226.


# VOYAGE AU CENTRE DU VIRTUEL

## Nantes • Du 14 au 17 novembre 2017

La Cité,
le Centre des Congrès de Nantes

www.jres.org | #JRES2017

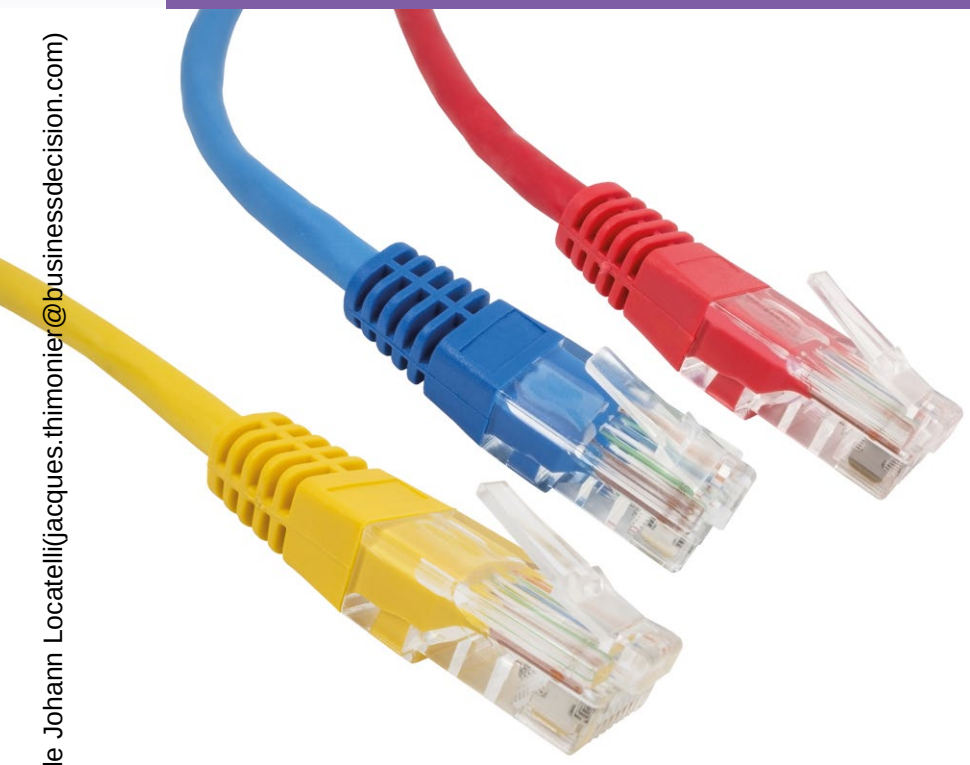


Design Graphique : Nicolas gazut | www.iamcion.com

# INSTALLATION SANS ÉCRAN DE SLACKWARE SUR UN RASPBERRY PI 3

**CHRISTOPHE BORELLY**
[Professeur de l'ENSAM – IUT de Béziers]

**MOTS-CLÉS : RASPBERRY, SLACKWARE, ARM, INSTALLATION**



**Vous venez d'acheter un Raspberry Pi 3 et vous n'avez pas d'écran avec un connecteur HDMI à portée de main pour installer/paramétrer votre petit « jouet ». Qu'à cela ne tienne, un câble Ethernet suffira ! Je vous propose également de voir comment on peut utiliser un autre système Linux que la distribution de référence Raspbian, en installant par exemple une Slackware.**

Comme indiqué sur le site de la fondation **Raspberry Pi [1]**, il existe plusieurs systèmes d'exploitation supportés par ce petit ordinateur de poche à base de processeur ARM, comme **Raspbian**, **Ubuntu Mate**, **OSMC**, **LibrELEC**, **Pinet**, **RISC OS**, etc. La fondation propose même l'installateur **NOOBS** (*New Out Of the Box Software*) qui permet à un débutant d'installer un système **Linux** sur un Raspberry Pi.

Comme j'aime bien sortir des sentiers battus, et que sur mon ordinateur personnel j'utilise le système **Slackware Linux** depuis bien longtemps déjà, j'ai tout de suite voulu essayer d'installer ce système sur mon premier Raspberry. Vous allez donc découvrir dans ce qui suit, comment installer **SARPi [2]** (*Slackware ARM on Raspberry Pi*) sans écran avec uniquement une connexion réseau !

Mais si vous êtes plutôt un inconditionnel de **Debian**, sachez tout de même que l'installation de Raspbian sans écran est bien entendu également possible. Une simple recherche sur Internet devrait vous donner les manipulations à réaliser...

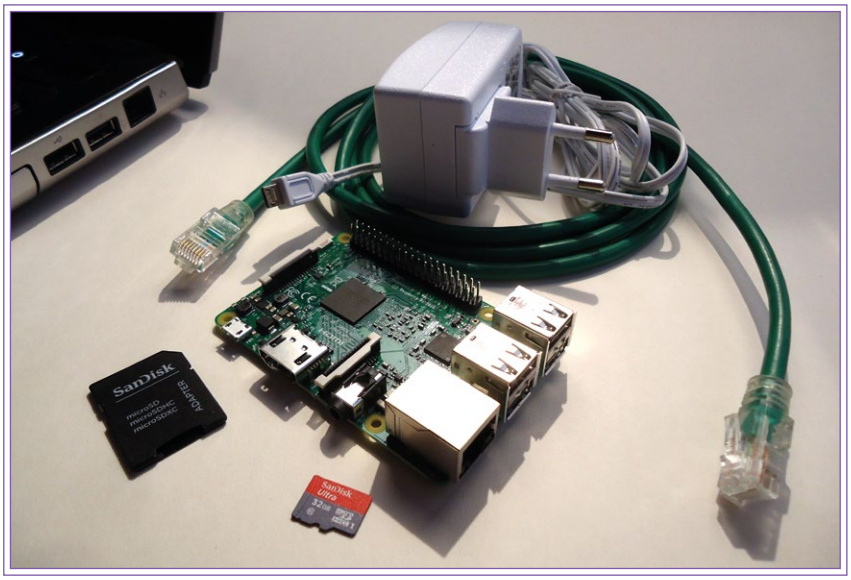



*Fig. 1 : Matériel nécessaire.*

Pour notre installation de SARPi, il vous faudra donc avoir un Raspberry Pi (les versions 1, 2 et 3 sont supportées) et une alimentation USB, ainsi qu'une carte microSD d'au moins 16 Gio avec un adaptateur SD (voir figure 1). Et pour l'installation, un ordinateur sous Linux avec un lecteur de carte SD, une carte réseau et un câble Ethernet.

## 1. TÉLÉCHARGEMENT DE SLACKWARE ARM

La toute première étape consiste à télécharger les paquets officiels de Slackware ARM **[3]**. Il faut savoir qu'ils sont tous regroupés dans le répertoire **slackware** et dans l'une des 16 séries suivantes : **a**, **ap**, **d**, **e**, **f**, **k**, **kde**, **kdei**, **l**, **n**, **t**, **tcl**, **x**, **xap**, **xfce** et **y**.

Avec l'habitude des installations Slackware **[6]** (voir partie « Select »), on se souvient que la série **a** rassemble les paquets de « base » du système, **ap** correspond aux applications en mode texte, **d** tout ce qui sert au développement, **l** toutes les bibliothèques, etc.

Dans cet exemple : pas d'écran, alors je vous propose de faire une installation allégée, sans serveur X ni applications graphiques, donc cela évite de télécharger les séries **e**, **f**, **k**, **kde**, **kdei**, **t**, **tcl**, **x**, **xap**, **xfce** et **y** ! Mais vous pourrez facilement réaliser une installation complète en enlevant les directives **exclude** de la commande suivante.

Nous aurons également besoin en plus, du fichier **PACKAGES.TXT** contenant la description détaillée de chaque paquet et du répertoire **isolinux** contenant les images RAM de l'installateur Slackware classique.

Il y a plusieurs façons de mettre tous ces fichiers à disposition du Raspberry Pi (voir section 4.2.4) : à l'aide d'une clé USB, en créant une partition spécifique sur la carte microSD, ou encore en partageant un répertoire par NFS ou avec Samba. Mais j'ai préféré vous présenter la méthode utilisant un serveur web (elle fonctionne aussi avec un serveur FTP d'ailleurs). Vous pouvez installer par exemple, **apache** ou **ngnix** si vous n'avez pas de serveur web déjà présent sur votre machine.

Comme la racine du serveur web de ma machine se trouve dans **/var/www/htdocs**, je crée un sous-répertoire pour accueillir tous ces fichiers. Il faut prévoir au moins 872 Mio d'espace libre (et 2,4 Gio pour une installation complète) :

```
$ mkdir -p /var/www/htdocs/slackarm-14.2
$ cd /var/www/htdocs/slackarm-14.2
$ SLACK_ARM=ftp.arm.slackware.com::slackwarearm/
slackwarearm-14.2
$ rsync -Pavv --delete $SLACK_ARM/PACKAGES.TXT .
$ rsync -Pavv --delete $SLACK_ARM/isolinux .
$ rsync -Pavv --delete --exclude '*/e/*' \
  --exclude '*/f/*' --exclude '*/k/*' \
  --exclude '*/kde/*' --exclude '*/kdei/*' \
  --exclude '*/t/*' --exclude '*/tcl/*' \
  --exclude '*/x/*' --exclude '*/xap/*' \
  --exclude '*/xfce/*' --exclude '*/y/*' \
  $SLACK_ARM/slackware .
```

Pour vérifier s'il n'y a pas eu de problèmes de téléchargement, on peut se servir du fichier **CHECKSUMS.md5** du répertoire **slackware** (supprimer le **egrep** dans le cas d'une installation complète) :

```
$ cd slackware
$ tail +13 CHECKSUMS.md5 | egrep -v './(e|f|k|t|x|y)' |
md5sum -c
```

## 2. TÉLÉCHARGEMENT DES PAQUETS SPÉCIFIQUES DE SARPI

Il faut ensuite télécharger les 4 paquets « système » préparés par SARPi **[4]**. Cela comprend une nouvelle version du *kernel* et les modules associés, ainsi que le *firmware* de démarrage spécifique au Raspberry Pi 3 et un dernier paquet optionnel contenant 3 « hacks » de paramétrage **[6]**.

Pour bien séparer ces fichiers des précédents, je crée un autre sous-répertoire dans la racine du serveur web :

```
$ SARPI=http://sarpi.fatdog.eu/files/rpi3/142/img/
$ mkdir -p /var/www/htdocs/sarpi
$ cd /var/www/htdocs/sarpi
$ wget $SARPI/kernel-modules-sarpi3-4.4.21-armv7-1_slack14.2_fd0.txz
$ wget $SARPI/kernel_sarpi3-4.4.21-armv7-1_slack14.2_fd0.txz
$ wget $SARPI/sarpi3-boot-firmware-armv7-1_slack14.2_fd0.txz
$ wget $SARPI/sarpi3-hacks-3.0-armv7-1_slack14.2_fd0.txz
```

Ces 4 fichiers seront utiles à la toute fin de l'installation (voir paragraphe 4.2.8).

## 3. PRÉPARATION DE LA CARTE MICROSD

L'installation complète de Slackware 14.2 nécessite environ 9 Gio d'espace disque, il est alors préférable d'avoir une carte microSD d'au moins 16 Gio. Mais pour la version « allégée » que je vous propose dans cet article, vous n'aurez besoin que de seulement 3,7 Gio.

La carte que j'ai prévu d'utiliser fait 32 Gio (formatée d'usine en FAT32), et le système la voit en tant que périphérique **mmcblk0** :

```
root@pccb# dmesg
...
[182937.833859] mmc0: new high speed SDHC card at address aaaa
[182937.842731] mmcblk0: mmc0:aaaa SL32G 29.7 GiB
[182937.844607] mmcblk0: p1

root@pccb# fdisk -l /dev/mmcblk0
Disque /dev/mmcblk0 : 29,7 GiB, 31914983424 octets, 62333952 secteurs
Unités : secteur de 1 × 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 512 octets
taille d'E/S (minimale / optimale) : 512 octets / 512 octets
Type d'étiquette de disque : dos
Identifiant de disque : 0x00000000

Périphérique   Amorçage Début      Fin Secteurs Taille Id Type
/dev/mmcblk0p1           8192 62333951 62325760  29,7G  c W95 FAT32 (LBA)
```

La première étape de la préparation consiste à télécharger une image de l'installateur SARPi **[4]** et à la copier sur la carte microSD. Ici, j'ai choisi la version 14.2 puisque nous avons téléchargé cette version en section 1 :

```
root@pccb# SARPI=http://sarpi.fatdog.eu/files/rpi3/142/img/
root@pccb# wget $SARPI/sarpi3-installer_slack14.2_fd0.img.xz
root@pccb# xz -dc sarpi3-installer_slack14.2_fd0.img.xz | dd of=/dev/
mmcblk0 bs=65536
3+12885 enregistrements lus
3+12885 enregistrements écrits
106496000 bytes (106 MB, 102 MiB) copied, 8,46013 s, 12,6 MB/s
root@pccb# fdisk -l /dev/mmcblk0
...
Périphérique   Amorçage Début    Fin Secteurs Taille Id Type
/dev/mmcblk0p1 *           32 195327   195296  95,4M  c W95 FAT32 (LBA)
```

On peut noter qu'à présent la partition **mmcblk0p1** ne fait plus que 95,4 Mio seulement, qu'elle est maintenant amorçable (la petite *) et qu'elle est toujours au format FAT32. Pour la petite histoire, il faut savoir que seul ce format est supporté pour la partition d'amorce (**/boot**) sur un Raspberry Pi 1, 2 ou 3 !

Voilà à partir de là, on peut normalement insérer la carte microSD dans le Raspberry Pi et le démarrer. Mais par défaut, sa configuration réseau est en DHCP et sans avoir « la main » sur le serveur en question, il n'est pas immédiat de déterminer l'adresse IP qui lui a été attribuée (mais c'est un bon exercice pour les étudiants en réseau).

Alors pour faire simple, je vous propose de fixer celle-ci manuellement afin qu'il soit sur le même réseau que l'ordinateur d'installation. On connectera ensuite directement le Raspberry Pi au PC par un câble Ethernet simple. C'est d'ailleurs pour cela que le téléchargement des paquets vu dans les sections précédentes doit être fait en tout premier (sauf si vous avez un accès Internet par Wifi bien sûr !).

Comme maintenant quasiment toutes les cartes Ethernet sont auto-MDI/MDIX (c'est-à-dire quelles détectent le périphérique utilisé et croisent ou décroisent les fils RX et TX), il n'y a pas besoin d'utiliser un câble croisé pour cette connexion comme la théorie le nécessiterait !

Sur ma machine, je configure par exemple l'adresse IP de la carte **eth0** à 192.168.2.1/24, soit avec le gestionnaire de réseau, soit manuellement avec la commande **ip** (disponible dans le paquet **iproute2**) :

```
root@pccb# ip addr flush dev eth0
root@pccb# ip addr add
192.168.2.1/24 dev eth0
root@pccb# ip addr show dev eth0
2: eth0: <NO-CARRIER,BROADCAST,
MULTICAST,UP> mtu 1500...
```

```
    link/ether 78:2b:cb:c6:f7:42 brd
ff:ff:ff:ff:ff:ff
    inet 192.168.2.1/24 brd 192.168.1.255 scope
global eth0
       valid_lft forever preferred_lft forever
```

Pour paramétrer le démarrage du Rasberry Pi, il faut éditer le fichier **cmdline.txt** de la carte microSD. Il y a 2 possibilités : soit on monte de façon manuelle la partition 1 avec le type **vfat**, soit on éjecte puis on réinsère la carte dans l'ordinateur pour pouvoir accéder aux fichiers (en utilisant le montage automatique des distributions « modernes ») :

```
root@pccb# mount -t vfat /dev/mmcblk0p1 /mnt/tmp
root@pccb# ls /mnt/tmp
COPYING.linux           bootcode.bin  kernel7.img
LICENCE.broadcom        cmdline.txt   overlays
README                  config.txt    start.elf
bcm2708-rpi-b-plus.dtb  fixup.dat     start_cd.elf
...
```

On fixe ensuite l'adresse IP du Raspberry Pi avec le paramètre **nic** et on peut aussi en profiter pour indiquer que le clavier est en français :

```
nic=auto:eth0:static:192.168.2.100:24 kbd=fr
dwc_otg.lpm_enable=0 console=tty1 nofont
root=/dev/mmcblk0p3 rootfstype=ext4 ...
```

Voilà maintenant en insérant la carte microSD dans le Raspberry Pi et en reliant ce dernier directement au PC par le câble Ethernet (voir figure 2), on devrait pouvoir s'y connecter à distance par le réseau en SSH (sans mot de passe).

```
cb@pccb$ ping 192.168.2.100
PING 192.168.2.100 (192.168.2.100) 56(84) bytes of
data.
64 bytes from 192.168.2.100: icmp_seq=1 ttl=64
time=0.618 ms
64 bytes from 192.168.2.100: icmp_seq=2 ttl=64
time=0.315 ms
...
cb@pccb$ ssh root@192.168.2.100
The authenticity of host '192.168.2.100
(192.168.2.100)' can't be established.
RSA key fingerprint is SHA256:/
x2e6Dbq3ptQ9Vr7hV1yjHN50GDFsjVQJgDiwBXXmOQ.

Are you sure you want to continue connecting (yes/
no)? yes
Warning: Permanently added '192.168.2.100' (RSA)
to the list of known hosts.
root@192.168.2.100's password:
```

```
Linux 4.4.21-v7-arm.

If you're upgrading an existing Slackware system,
you might want to
remove old packages before you run 'setup' to
install the new ones. If
you don't, your system will still work but there
might be some old files
left laying around on your drive.

Just mount your Linux partitions under /mnt and
type 'pkgtool'. If you
don't know how to mount your partitions, type
'pkgtool' and it will tell
you how it's done.

To partition your hard drive(s), use 'cfdisk' or
'fdisk'.
To start the main installation (after
partitioning), type 'setup'.
root@slackware:~#
```

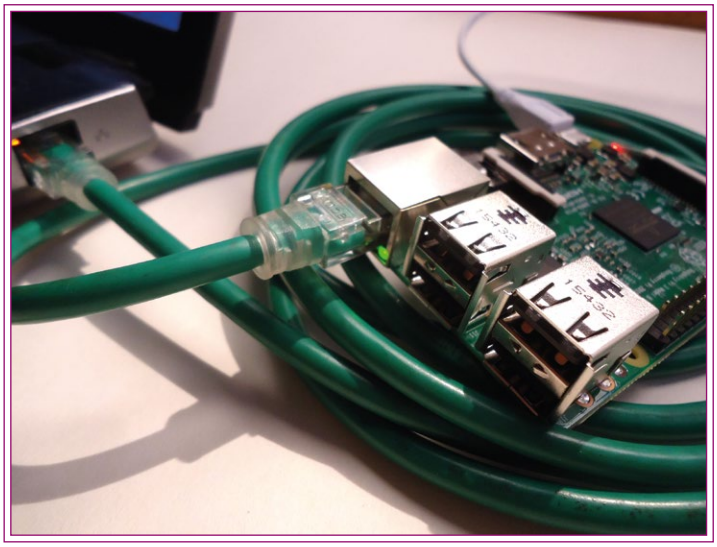


*Fig. 2 : Connexion du PC au Raspberry Pi 3.*

Voilà, le prompt de Slackware apparaît, nous sommes sur la bonne voie !

La première chose à faire est de fixer la date, car le Raspberry Pi redémarre toujours le 1 janvier 1970 :

```
root@slackware:~# date
Thu Jan  1 00:00:44 UTC 1970
```

On peut le faire facilement avec la commande **date** et l'option **-s**. Voici la commande à exécuter sur votre ordinateur pour obtenir la bonne syntaxe :

```
cb@pccb$ echo date -s \"$(date
'+%Y-%m-%d %H:%M:%S')\"
date -s "2016-10-03 14:34:03"
```

On colle ensuite le résultat précédent sur le Raspberry Pi, et on constate que la date est maintenant rétablie :

```
root@slackware:~# date -s "2016-
10-03 14:34:03"
Mon Oct  3 14:34:03 UTC 2016
```

**NOTE**

Si au hasard de vos manipulations, il vous arrive d'avoir à éteindre puis rallumer votre Raspberry Pi après vous être connecté une première fois par SSH, vous aurez probablement un message identique à ceci :

```
$ ssh root@192.168.2.100
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@ WARNING: REMOTE HOST
IDENTIFICATION HAS CHANGED!
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING
SOMETHING NASTY!
Someone could be eavesdropping on you
right now (man-in-the-middle attack)!
It is also possible that a host key
has just been changed.
The fingerprint for the RSA key sent
by the remote host is
SHA256:OctKuqZxYy8wk4tP+gnod43cKAJ1
2G2/O9t2YXnrJcs.
Please contact your system
administrator.
...
Host key verification failed.
```

En mode installation, la clé RSA du serveur SSH est générée pendant le démarrage du Raspberry Pi, elle change alors naturellement à chaque fois. Il faut donc avant de se connecter, supprimer l'ancienne clé dans le fichier **known_hosts** avec la commande suivante :

```
$ ssh-keygen -R 192.168.2.100
# Host 192.168.2.100 found: line 53
/home/cb/.ssh/known_hosts updated.
Original contents retained as /
home/cb/.ssh/known_hosts.old
$ ssh root@192.168.2.100
The authenticity of host
'192.168.2.100 (192.168.2.100)'
can't be established.
...
```

Il ne reste plus qu'à installer le système de façon classique. Enfin presque, car ici pas de CD/DVD, nous allons utiliser un serveur web comme media source. C'est ce que vous allez découvrir en section suivante.

# 4. INSTALLATION DE SLACKWARE

Une installation de Slackware **[5]** commence toujours par le partitionnement du disque (ici, la carte microSD) ensuite, on lance le script **setup** et on suit les instructions...

## 4.1 Création des partitions sur la carte microSD

Pour cet exemple, 2 partitions supplémentaires suffisent : une pour le **swap** (de taille égale à la RAM en général, soit 1 Gio pour un Raspberry Pi 3) et l'autre pour la racine du système de fichiers. Au total, il y aura bien 3 partitions, car la partition 1 a déjà été créée précédemment et correspond à la partition d'amorce du système (**/boot**).

J'ai l'habitude de l'outil **fdisk**, alors vous trouverez en suivant, ce que cela peut donner pour créer la partition de **swap** avec cet outil. Mais sachez tout de même que vous disposez également des commandes **cfdisk**, **sfdisk** et **gdisk** si vous préférez.

```
root@slackware:~# fdisk /dev/mmcblk0
Command (m for help): n
Partition type
   p   primary (1 primary, 0 extended, 3 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (2-4, default 2): 2
First sector (195328-62333951, default 196608): 195328
Last sector, +sectors or +size{K,M,G,T,P} (195328-62333951,
default 62333951): +1G

Created a new partition 2 of type 'Linux' and of size 1023.6 MiB.

Command (m for help): t
Partition number (1,2, default 2): 2
Partition type (type L to list all types): 82

Changed type of partition 'Linux' to 'Linux swap'.
```

Pour créer la partition 3, on fait de même, en tapant **n**, **<Entrée>**, **<Entrée>**, **<Entrée>**, **<Entrée>**. Enfin, on vérifie si tout va bien en affichant la liste des partitions avec **p**, puis on valide avec la commande **w** :

```
Command (m for help): p
...
Device         Boot   Start      End  Sectors    Size Id Type
/dev/mmcblk0p1 *         32   195327   195296   95.4M  c W95 FAT32 (LBA)
/dev/mmcblk0p2       195328  2291711  2096384 1023.6M 82 Linux swap
/dev/mmcblk0p3      2291712 62333951 60042240   28.6G 83 Linux

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks
```

## 4.2 Paramétrage de l'installation

Une fois le disque partitionné, on lance le script **setup** et on obtient la figure 3.

On commence en général par configurer le clavier en choisissant ***KEYMAP***. Mais si vous n'êtes pas trop habitué à l'installation de Slackware, vous trouverez sur le site **[5]**, le détail de chaque étape.

Je vais essayer maintenant de résumer rapidement les principaux points importants.

### 4.2.1 Configuration du clavier (KEYMAP)

Il suffit de choisir **azerty/fr-latin1.map** en général puis de taper **1** sur une ligne vierge dans l'écran suivant pour valider le choix. On passe automatiquement au point suivant (***ADDSWAP***).

### 4.2.2 Activation du swap (ADDSWAP)

L'installeur doit détecter tout seul la partition **/dev/mmcblk0p2**. Il suffit de valider 3 fois en suivant.

### 4.2.3 Sélection de la partition d'installation (TARGET)

Ici, c'est aussi facile, car il ne reste qu'une partition (la numéro 3). On valide et on formate en **ext4** la partition en question. À la fin, un message apparaît indiquant qu'une partition FAT a été détectée (la partition 1 qui contient l'amorce du disque). Il n'est pas nécessaire d'ajouter le montage de cette partition au fichier **/etc/fstab**. On répond donc **No** à la question.

### 4.2.4 Sélection du média (SOURCE)

Dans cette partie, il y a 7 possibilités pour indiquer comment accéder aux paquets téléchargés en section 1. Comme

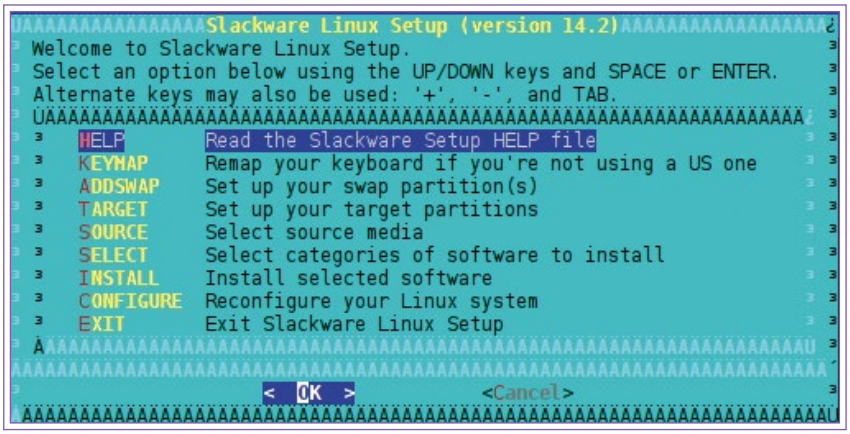



*Fig. 3 : Premier écran du script d'installation de Slackware.*

ma machine possède un service web, je choisis l'option 5 (prenez l'option 2 si vous avez copié les fichiers du tout début de l'article sur une clé USB).

On précise en premier uniquement le protocole et l'adresse IP du serveur. Dans mon cas, j'indique **http://192.168.2.1**. Ensuite, il faut donner le chemin du répertoire **slackware** à partir de la racine du site web. Dans mon cas, j'ai utilisé le répertoire **slackarm-14.2** lors du téléchargement des paquets. Il faut donc indiquer **/slackarm-14.2/slackware** dans la deuxième fenêtre.

Le système va alors tenter de télécharger le fichier **PACKAGES.TXT** pour voir si le paramétrage semble correct. On valide en général cette configuration en cliquant sur **No** (présélectionné par défaut quand tout va bien). Commence ensuite l'analyse des 1322 paquets...

Si on est un peu curieux, on peut consulter les traces du serveur web pour voir ce qu'il s'est passé :

```
root@pccb# tail -f /var/log/httpd/access_log
192.168.2.100 ... "GET /slackarm-14.2/PACKAGES.TXT HTTP/1.1" 200 693943
192.168.2.100 ... "GET /slackarm-14.2/isolinux/setpkg HTTP/1.1" 200 2943
192.168.2.100 ... "GET /slackarm-14.2/slackware/a/tagfile HTTP/1.1" 200 1359
192.168.2.100 ... "GET /slackarm-14.2/slackware/a/maketag.ez HTTP/1.1" 200 9136
192.168.2.100 ... "GET /slackarm-14.2/slackware/a/maketag HTTP/1.1" 200 9136
192.168.2.100 ... "GET /slackarm-14.2/slackware/ap/tagfile HTTP/1.1" 200 881
192.168.2.100 ... "GET /slackarm-14.2/slackware/ap/maketag.ez HTTP/1.1" 200 6642
192.168.2.100 ... "GET /slackarm-14.2/slackware/ap/maketag HTTP/1.1" 200 6642
```

On voit bien le téléchargement du fichier **PACKAGES.TXT** et celui du script **isolinux/setpkg**, puis pour chaque série, on récupère 3 fichiers qui permettent d'automatiser l'installation (**tagfile**, **maketag.ez** et **maketag**).

### 4.2.5 Sélection des séries (SELECT)

Dans la fenêtre suivante, on peut sélectionner les séries que l'on désire installer. Pour cet article, on ne validera que les séries **a**, **ap**, **d**, **l** et **n**.

### 4.2.6 Mode d'installation (INSTALL)

Enfin, on indique une installation complète (**full**) pour installer tous les paquets de chaque série. Je vous laisse regarder la documentation **[5]** si vous souhaitez connaître les subtilités des autres options d'installation.

On peut toujours voir ce qui se passe dans les traces du serveur web. En fait, pour chaque paquet, on télécharge le fichier texte de description (**.txt**) puis l'archive **.txz** :

```
root@pccb# tail -f /var/log/httpd/access_log
192.168.2.100 ... "GET /slackarm-14.2/slackware/a/
aaa_base-14.2-arm-1.txt HTTP/1.1" 200 327
192.168.2.100 ... "GET /slackarm-14.2/slackware/a/
aaa_base-14.2-arm-1.txz HTTP/1.1" 200 6144
192.168.2.100 ... "GET /slackarm-14.2/slackware/a/
aaa_elflibs-14.2-arm-19.txt HTTP/1.1" 200 472
192.168.2.100 ... "GET /slackarm-14.2/slackware/a/
aaa_elflibs-14.2-arm-19.txz HTTP/1.1" 200 3846048
```

### 4.2.7 Configuration (CONFIGURE)

Une fois tous les paquets installés (après une bonne vingtaine de minutes tout de même), on passe à la partie configuration du système.

En premier, j'indique qu'éventuellement une souris USB pourra être connectée au Raspberry Pi, mais sans activer **gpm**.

Ensuite, pour la partie réseau, je précise une adresse IP fixe pour que le Raspberry Pi soit toujours accessible à la même adresse IP. Vous pourrez adapter cela à votre guise.

Puis vient la partie de sélection des services à démarrer. Je ne conserve que **messagebus**, **syslog** et **sshd**.

On peut ensuite répondre **No** pour la sélection de polices de console puis on valide le choix par défaut pour l'horloge et on choisit le fuseau horaire **Europe/Paris**.

Il reste à fixer le mot de passe de l'utilisateur **root**.

ATTENTION, ce n'est pas encore fini ! Dans le menu ***EXIT***, il faut répondre **No** pour revenir au terminal, car il va falloir mettre à jour certains paquets spécifiques à SARPi.

### 4.2.8 Fin du paramétrage

La première des choses à faire consiste à effacer l'image RAM de l'installateur. On monte donc la partition d'amorce puis on efface le fichier **initrd.gz** :

```
$ mount -t vfat /dev/mmcblk0p1 /mnt/boot
$ rm /mnt/boot/initrd.gz
```

Ensuite, il faut enlever les paquets du kernel par défaut de la distribution (version 4.4.14), car SARPi propose la version 4.4.21 :

```
$ ROOT=/mnt removepkg kernel_armv7 kernel-
modules-armv7
```

On télécharge alors manuellement les paquets spécifiques de SARPi sur le Raspberry Pi :

```
$ SARPI=http://192.168.2.1/sarpi
$ cd /root
$ wget $SARPI/kernel-modules-sarpi3-4.4.21-
armv7-1_slack14.2_fd0.txz
$ wget $SARPI/kernel_sarpi3-4.4.21-armv7-1_
slack14.2_fd0.txz
$ wget $SARPI/sarpi3-boot-firmware-armv7-1_
slack14.2_fd0.txz
$ wget $SARPI/sarpi3-hacks-3.0-armv7-1_
slack14.2_fd0.txz
```

Il reste enfin à les installer :

```
$ ROOT=/mnt installpkg /root/kernel*.txz
/root/sarpi*.txz
```

Voilà, c'est terminé !

## CONCLUSION

Vous avez donc maintenant, si tout s'est bien passé, un petit serveur de poche sous Slackware qui pourra vous servir à interfacer des capteurs ou à commander à distance des équipements. Vous pourrez alors directement programmer en **C**, en **Python**, en **Perl**, en **PHP**, en **bash**, etc. puisque la distribution intègre nativement tous ces outils.

« Enjoy! » comme dirait Patrick Volkerding et toute l'équipe de Slackware. ■

## RÉFÉRENCES

**[1]** Site de téléchargement de la fondation Raspberry Pi : https://www.raspberrypi.org/downloads/

**[2]** Site de SARPi (*Slackware ARM on Raspberry Pi*) : http://rpi.fatdog.eu/

**[3]** Site de Slackware ARM : http://arm.slackware.com/

**[4]** Téléchargements de SARPi : http://rpi.fatdog.eu/index.php?p=downloads

**[5]** https://docs.slackware.com/slackware:install

**[6]** http://rpi.fatdog.eu/files/extra/sarpi-hacks.README

# ACTUELLEMENT DISPONIBLE
# MISC N°93 !


## OUTILS, FRAMEWORKS, OBFUSCATION, PROTOCOLES...
## WIKILEAKS ET LES SHADOW BROKERS

# NE LE MANQUEZ PAS
## CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :
## http://www.ed-diamond.com

# RÉALISATION D'UN PAQUET EMACS : UN NAVIGATEUR DE TICKETS GITHUB


**DAMIEN CASSOU**
[Enseignant chercheur et développeur professionnel, auteur et contributeur sur de nombreux projets libres dont Emacs]

**MOTS-CLÉS : EMACS, LISP, HTTP REST, TEST, JSON, GITHUB**

**Emacs montre toute sa puissance à ceux qui le placent au centre de leur quotidien : lecture de courriers électroniques et de flux RSS, gestion des tâches et calendriers, navigation dans des systèmes de fichiers locaux et distants, manipulation de dépôts git, etc. Dans cet article, nous verrons comment quelques lignes d'Emacs Lisp, que nous introduirons, permettent d'interroger facilement un serveur distant (GitHub) et d'en formater la réponse en vue d'une utilisation efficace depuis Emacs. Ce qui suit s'adresse aux débutants en Emacs et en Lisp.**

**Emacs** est bien plus qu'un éditeur de texte : c'est un système permettant de manipuler à peu près tout ce qui peut être représenté par du texte. Je m'en sers pour programmer, gérer mes listes de tâches à faire, lire et écrire mes e-mails, rédiger mes rapports et articles, manipuler mes dépôts git et mon système de fichiers, etc. Dans cet article, nous allons découvrir la programmation sous Emacs au travers de la réalisation d'un navigateur de tickets **GitHub**. Notre paquet aura pour responsabilité d'extraire de GitHub la liste des tickets (*issues* en anglais) ouverts pour un projet donné et d'afficher cette liste sous la forme d'un tableau (voir figure 1).

Par le biais d'un exercice, vous serez amenés à créer un fichier **Emacs Lisp**, à naviguer dans les menus, à trouver la documentation dont vous avez besoin, à gagner du temps avec les raccourcis clavier et à écrire un test unitaire. Dans les pages qui suivent, je me suis attaché à présenter Emacs dans sa version originale : comme le montrent les nombreuses vidéos de manipulation d'Emacs que l'on trouve sur Internet, il est possible de rendre Emacs encore plus puissant et efficace en installant des paquets additionnels.

Les trois encadrés au début de cet article vous permettront de vous familiariser avec quelques points clés d'Emacs pour pouvoir comprendre la suite.

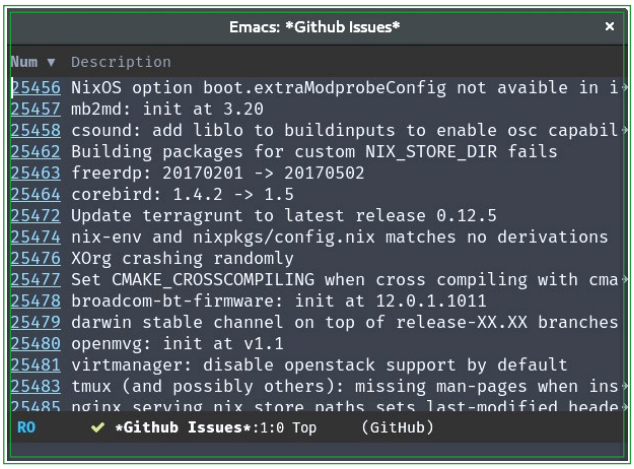



*Fig. 1 : Vue de notre application finale (avec le thème Zerodark).*

# 1. FAIRE UNE REQUÊTE REST

La documentation de l'API GitHub **[1]** nous enseigne que le format de données utilisé est **JSON**, que le serveur est accessible à l'adresse https://api.github.com et que la requête pour obtenir une liste de tickets pour un dépôt donné est : **`GET /repos/:owner/:repo/issues`**. Pour pouvoir récupérer la liste des tickets du dépôt **`nixpkgs`** de l'équipe **NixOS** (un projet parmi tant d'autres), l'URL sera donc https://api.github.com/repos/NixOS/nixpkgs/issues. De son côté, la page de manuel ***URL*** (accessible dans Emacs depuis le menu ***Help > More Manuals > All Other Manuals (Info)*** ou avec le raccourci **`C-h i`**) possède une section nommée ***Retrieving URLs***. Cette section nous enseigne qu'une des fonctions permettant de récupérer un contenu distant est **`url-retrieve-synchronously`**. Cette fonction retourne le *buffer* dans lequel les données récupérées sont écrites. Dans Emacs, tout apparaît dans un *buffer*.

Créez un fichier appelé **`github.el`** en cliquant sur l'icône appropriée de la barre d'outils. L'extension **`.el`** est interprétée comme fichier Emacs Lisp et Emacs se met automatiquement dans ce mode (comme le montre par exemple le menu dédié ***Emacs-Lisp***). Entrez le code suivant dans votre fichier :

```
(pop-to-buffer
 (url-retrieve-synchronously
  "https://api.github.com/repos/NixOS/
nixpkgs/issues"))
```

En mettant le curseur sur **`pop-to-buffer`** et en visitant le menu ***Help > Describe > Describe Function...*** puis en tapant **`RET`** pour accepter la fonction proposée en bas de la fenêtre (ou directement avec le raccourci **`C-h f RET`**), vous pouvez lire la documentation de cette fonction dont le rôle est d'afficher le *buffer* en paramètre, de préférence à côté du *buffer* actif. Alors que le curseur est quelque part au milieu du texte que vous venez de taper (par exemple, toujours sur **`pop-to-buffer`**), utilisez le raccourci **`C-M-x`** pour exécuter ce code (la figure 2 vous donne le résultat). Le *buffer* qui vient de s'afficher commence par les en-têtes HTTP (environ 25 lignes, la plupart masquées si vous avez un petit écran) et se termine par une grande ligne de JSON. Le curseur a été automatiquement placé au début de la ligne de JSON. N'hésitez pas à vous déplacer dans ce *buffer* pour voir ce qu'il contient.

## NOTIONS D'EMACS LISP

**Emacs Lisp est une implémentation de la famille de langages Lisp. La syntaxe de Lisp est un peu déroutante pour qui est seulement habitué aux langages populaires, mais elle est extrêmement simple et toujours une source d'inspiration pour certains langages modernes comme Clojure. Lisp offre des types primitifs comme les nombres (42) et les chaînes de caractères ("bonjour"). Lisp propose la liste comme type d'agrégat principal : `(10 20 30)` est une liste de 3 éléments. Là où Lisp se distingue, c'est qu'il utilise aussi la liste comme unique moyen de former des expressions composées : dans un contexte normal, la liste `(+ 40 2)` sera évaluée par Lisp pour former le nombre 42. Pour empêcher Lisp d'évaluer une liste, il faut la faire précéder d'une apostrophe : `'(+ 40 2)` restera une liste de 3 éléments dont le premier est un symbole. Lorsqu'une liste doit être évaluée, le premier élément, appelé opérateur, indique quel type d'évaluation aura lieu. Si l'opérateur représente une fonction (comme `+` ou `print`), les arguments sont d'abord évalués puis passés en paramètres à la fonction. Dans le cas contraire, l'opérateur doit représenter une forme spéciale (comme `let` et `if`) ou une macro (comme `defun` et `lambda`) et l'opérateur est appelé avec les arguments non évalués. Par exemple, l'expression `(if (= number 42) "you win" (+ 3 0.14))` a pour valeur à la fin de l'évaluation la chaîne de caractères `"you win"` si la variable `number` vaut 42 et le nombre `3.14` si `number` est un autre nombre. Notez que l'expression d'addition n'est évaluée que si `number` n'est pas égal à 42.**

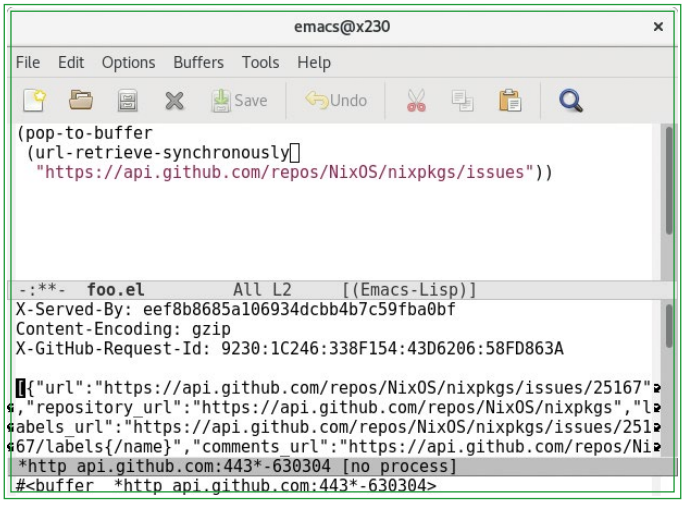



*Fig. 2 : Affichage du résultat d'une requête sur le serveur api.github.com.*

## 2. DÉFINIR UNE PREMIÈRE FONCTION

Pour pouvoir être utilisée au sein d'un programme et être testée, l'expression ci-dessus doit être placée au sein d'une fonction. Le but de cette fonction va être d'envoyer une requête à une URL construite d'après les paramètres et de retourner un résultat basé sur la réponse à cette requête. Avec votre curseur dans le *buffer* de réponse à la requête envoyée à la section précédente, tapez **C-x 0** pour le masquer. Remplacez maintenant le contenu de votre fichier **github.el** par celui-ci :

```
01: ;;; github.el --- Demo for gnu/linux mag
-*- lexical-binding: t; -*-
02: (require 'json)
03: (require 'map)
04: (require 'seq)
05:
```

## INTERAGIR AVEC EMACS

Emacs fournit une barre d'outils et des menus pour interagir avec lui. Lorsque le mode Emacs Lisp est activé (automatiquement à l'ouverture d'un fichier `*.el`), un menu ***Emacs-Lisp*** apparaît. Ce menu permet par exemple d'accéder à une ligne de commandes pour découvrir interactivement le langage au travers du sous-menu ***Interactive Expression Evaluation***. Le sous-menu ***Evaluate Last S-Expression*** permet d'évaluer l'expression avant le curseur. Profitez de vos passages dans les menus pour regarder les raccourcis clavier associés. Par exemple, ***Evaluate Last S-Expression*** est associé au raccourci `C-x C-e`. Ce raccourci signifie que vous devez maintenir la touche <Ctrl> appuyée, appuyer sur <x> et, sans relâcher <Ctrl>, appuyer sur <e>. Un raccourci similaire est `C-M-x` qui nécessite de laisser appuyer <Ctrl> et <Meta> (en général, la touche <Alt> ou <Option>) avant d'appuyer sur <x>. Le raccourci permettant de remplacer le *buffer* actif par un autre (fonction accessible aussi depuis le menu ***Buffers***) est `C-x b`. Ce raccourci nécessite de relâcher la touche <Ctrl> avant de taper sur <b>.

Plusieurs des raccourcis que nous allons utiliser dans cet article commencent par `M-x`. Ce préfixe permet à l'utilisateur de demander à Emacs d'exécuter une *commande* (aussi appelée *fonction interactive*) en fournissant son nom. Après avoir tapé `M-x`, l'utilisateur est invité à taper le nom de la commande qu'il souhaite dans le *mini-buffer* : celui-ci se trouve tout en bas de la fenêtre. Vous pouvez utiliser la touche <Tab> (`TAB`) à cet instant pour compléter votre saisie. Validez le nom de la fonction avec la touche <Return> (`RET` ou entrée). Par exemple, `M-x ema TAB ver TAB RET` exécute la fonction `emacs-version` qui affiche des informations sur la version d'Emacs que vous utilisez. Pour plus de clarté dans cet article, je donnerai le nom des fonctions en entier lorsque je demande de taper un raccourci commençant par `M-x`.

Voici la liste des raccourcis utilisés dans cet article :

| Raccourci | Description |
|---|---|
| `C-h i` | Affiche l'ensemble des manuels du système |
| `C-h f` | Cherche la documentation d'une fonction (celle sous le curseur par défaut) |
| `C-M-x` | Évalue l'expression principale sous le curseur |
| `C-x C-e` | Évalue l'expression précédant le curseur |
| `M-x nom-de-fonction RET` | Évalue la commande (ou fonction interactive) **nom-de-fonction** |
| `C-x 0` | Masque le *buffer* courant |
| `C-x o` | Change le *buffer* actif |
| `C-x C-b` | Affiche la liste des *buffers* |
| `C-x b nom-de-buffer RET` | Remplace le *buffer* actif par **nom-de-buffer**. |
| `C-h m` | Affiche une description du *buffer* en cours |
| `C-h S` | Ouvre le manuel décrivant un symbole (celui sous le curseur par défaut) |

```
06: (defun github-retrieve-issues (owner project)
07:   "Return GitHub issues from OWNER/PROJECT."
08:   (let ((url (format "https://api.github.com/
repos/%s/%s/issues" owner project)))
09:     (with-current-buffer (url-retrieve-
synchronously url)
10:       (json-read))))
11:
12: (github-retrieve-issues "NixOS" "nixpkgs")
```

La ligne 1 est un en-tête standard de fichier Emacs Lisp décrivant le nom du fichier ainsi que son but. La fin de la ligne (**-*- lexical-binding: t; -*-**) active un mode moderne d'évaluation de code souhaitable pour tous les nouveaux fichiers. Enregistrez votre fichier depuis la barre d'outils ou en tapant **C-x C-s** puis, tapez **M-x revert-buffer RET yes RET** pour basculer dans ce mode.

Les trois expressions **require** entre les lignes 2 et 4 demandent à Emacs de charger les bibliothèques **json**, **map** et **seq** dont nous allons nous servir. Placez le curseur à la fin de chacune de ces lignes et tapez **C-x C-e** pour charger la bibliothèque dans l'environnement. Vous pouvez aussi sélectionner les trois lignes et taper **M-x eval-region RET**.

L'expression **defun** entre les lignes 6 et 10 définit une fonction **github-retrieve-issues** prenant **owner** et **project** en paramètres et retournant une liste de tickets récupérés avec l'API github.com. La ligne 7 associe une documentation à la nouvelle fonction. L'appel à la forme spéciale **let** entre les lignes 8 et 10 permet de définir une variable temporaire **url** pendant l'exécution des lignes 9 et 10. La variable **url** prendra la valeur de la chaîne de caractères fournie dans laquelle seront insérés le nom du propriétaire du projet et le nom du projet. Les lignes 9 et 10 exécutent **url-retrieve-synchronously** et passent le résultat (un *buffer*) à la fonction **with-current-buffer** dont le but est de rendre ce *buffer* actif pendant l'exécution de la fonction **json-read**. Cette dernière va analyser le contenu du *buffer* en cours à partir du curseur qui a été automatiquement placé entre les entêtes HTTP et le corps de la réponse JSON par **url-retrieve-synchronously**. La fonction **json-read** retourne une structure de données correspondante au JSON envoyé par GitHub. La forme spéciale **let** et la macro **with-current-buffer** retournant la valeur de leur dernière expression respective, notre fonction retournera la structure de données construite par **json-read**. Placez votre curseur quelque part dans la fonction et tapez **C-M-x** pour la définir. Vous pouvez aussi placer votre curseur à la fin et taper **C-x C-e**.

Enfin, la ligne 12 donne un exemple d'utilisation de la fonction au-dessus. Placez le curseur à la fin de cette ligne et tapez **C-x C-e**. Ceci va temporairement afficher le résultat de l'appel à la fonction en bas de la fenêtre.

## LA DOCUMENTATION D'EMACS

**Emacs est un des logiciels les mieux documentés que je connaisse. Toute sa documentation est accessible depuis le menu *Help*. Les deux premiers éléments de ce menu proposent un tutoriel que je recommande vivement pour bien débuter. Les sous-menus *Search Documentation* et *Describe* vous permettent de trouver la documentation d'un terme particulier. Les manuels interactifs accessibles depuis *Read the Emacs Manual*, *More Manuals > Introduction to Emacs Lisp* et *More Manuals > Emacs Lisp Reference* décrivent l'ensemble des fonctionnalités d'Emacs et du langage Emacs Lisp (les deux premiers peuvent être achetés au format papier).**

## 3. AFFICHAGE EN TABLEAU

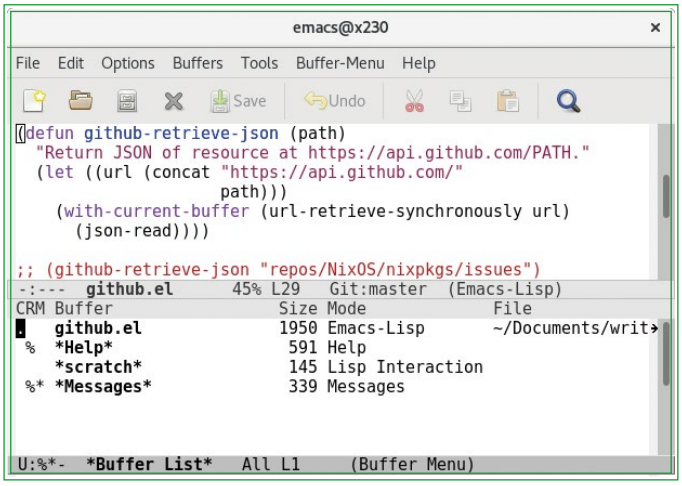



*Fig. 3 : Affichage en tableau de la liste des buffers utilisateurs (tableau généré avec le menu Buffers > List All Buffers).*

Si vous allez dans le menu ***Buffers > List All Buffers*** (ou bien si vous tapez **C-x C-b**), vous obtiendrez la liste des *buffers* utilisateurs actuellement ouverts (comme le montre la figure 3). Les *buffers* sont affichés dans un tableau avec leur nom dans une colonne et d'autres informations dans d'autres colonnes. Certaines colonnes peuvent être triées en cliquant sur leur nom. Nous allons utiliser le même type de vue pour afficher les tickets GitHub. Pour savoir quel mécanisme est utilisé pour afficher la liste des *buffers*, nous pouvons inspecter le code générant cette liste. Pour cela, activez le *buffer* contenant la liste des *buffers* (soit en cliquant dedans, soit en tapant **C-x o**), et tapez **C-h m** pour avoir la description de ce *buffer*. Un *buffer* d'aide vous indique que la liste des *buffers* est gérée par le mode majeur **Buffer Menu** du fichier **buff-menu.el**. Si vous cliquez sur le lien **buff-menu.el**, Emacs vous amènera directement à l'implémentation du mode majeur dont le début est :

```
(define-derived-mode Buffer-menu-mode
tabulated-list-mode "Buffer Menu"
  "Major mode for Buffer Menu buffers.
The Buffer Menu is invoked by the
commands \\[list-buffers],
...
```

Nous apprenons ici que **Buffer-menu-mode** hérite (ou *dérive*) du mode **tabulated-list-mode**. Avec le curseur sur **tabulated-list-mode**, tapez **C-h S RET** (le **S** étant en majuscule, vous devez aussi appuyer sur **<Shift>**) pour qu'Emacs vous amène directement à la page de manuel décrivant ce mode. Vous apprendrez ici que vous allez vous aussi devoir utiliser **define-derived-mode** pour définir votre mode majeur, fonction spécifiant à quoi doit ressembler le *buffer* affichant les tickets, ainsi qu'une "*listing command*", fonction affichant les tickets dans le mode majeur.

Revenez maintenant dans votre fichier en tapant **C-x 0** pour cacher le *buffer* actif puis **C-x b github.el RET** pour afficher votre fichier (le menu ***Buffers*** permet la même chose). Définissons le mode majeur comme étant un tableau ayant une première (petite) colonne pour afficher le numéro du ticket et une autre colonne pour afficher la description. Ajoutez le code suivant à votre fichier :

```
01: (define-derived-mode github-tabulated-mode
tabulated-list-mode "GitHub"
02:   "Displays GitHub issues in a tabulated list."
03:   (setq tabulated-list-format
04:         [("Num" 5 t)
05:          ("Description" 0 t)])
06:   (setq tabulated-list-sort-key (cons "Num" nil))
07:   (tabulated-list-init-header))
```

L'appel à la macro **define-derived-mode** à la ligne 1 permet la définition du mode majeur **github-tabulated-mode** à partir d'un autre (**tabulated-list-mode**). Le nom de ce nouveau mode **"GitHub"**, qui s'affichera dans la barre en bas du *buffer* et dans la documentation, est passé en troisième argument. Le quatrième argument, passé à la ligne 2, est la documentation du mode. Les lignes 3 à 5 donnent une valeur à la variable **tabulated-list-format** grâce à la forme spéciale **setq** qui est l'équivalent d'une affectation et qui prend un nom de variable et une valeur en argument. La page de manuel ouverte ci-dessus spécifie que la valeur de cette variable doit être un vecteur (spécifié entre crochets **[]** aux lignes 4 et 5) dont chaque élément est une liste spécifiant une colonne. La première colonne aura pour titre **"Num"**, contiendra au maximum 5 caractères et pourra être triée (indiqué par le **t**, signifiant « vrai », en fin de ligne) en cliquant sur son nom. La seconde colonne aura pour titre **"Description"**, s'étendra jusqu'au bout de la ligne (ceci est indiqué par le **0**) et pourra être triée elle aussi. La ligne

6 spécifie la colonne à trier par défaut tandis que la ligne 7 prépare la ligne d'en-tête du tableau en fonction des valeurs des variables définies aux lignes 3 et 6. Évaluez cette définition (**C-x C-e** avec le curseur à la fin ou **C-M-x** au milieu).

L'étape suivante consiste à définir une fonction affichant le tableau avec ses données :

```
01: (defun github-display-issues (issues)
02:   "Display ISSUES in a table.
03: ISSUES must be suitable for 'tabulated-
list-entries'."
04:   (let ((buffer (get-buffer-create
"*Github Issues*")))
05:     (with-current-buffer buffer
06:       (setq tabulated-list-entries issues)
07:       (github-tabulated-mode)
08:       (tabulated-list-print))
09:     (pop-to-buffer buffer)))
```

La ligne 4 définit une variable temporaire **buffer** (valide entre les lignes 5 à 9) dont la valeur est le *buffer* nommé **"*Github Issues*"** s'il existe ou, à défaut, un nouveau *buffer* (la fonction **get-buffer-create** se chargeant de ce travail). La ligne 5 active ce *buffer* le temps d'exécuter les lignes 6 à 8. La ligne 6 met la valeur du paramètre **issues** dans la variable **tabulated-list-entries** spécifiant le contenu du tableau à afficher. La ligne 7 active notre mode majeur défini ci-dessus. La ligne 8 affiche le tableau dans le *buffer* actif et la ligne 9 rend le *buffer* visible à l'écran. Évaluez cette définition comme d'habitude.

Nous pouvons maintenant afficher notre premier tableau (voir figure 4) grâce aux lignes suivantes que vous pouvez copier dans le fichier et évaluer :

```
(github-display-issues (list
      (list 1 (vector "1" "An issue"))
      (list 2 (vector "2" "Another one"))))
```

Ces trois lignes appellent la fonction que nous venons de définir en passant une liste en paramètre. Le format de ce paramètre (placé directement dans la variable **tabulated-list-entries** par notre fonction) est décrit par le manuel comme étant une liste dont chaque élément décrit une ligne du tableau. Chaque ligne est définie par une autre liste dont le premier élément est un identifiant pour la ligne (cela peut être n'importe quel objet) et le second est un vecteur contenant une chaîne de caractères par cellule du tableau. Dans l'appel ci-dessus, nous avons utilisé les fonctions **list** et **vector** pour construire les 3 listes et 2 vecteurs. Nous aurions aussi pu écrire le code équivalent :

```
(github-display-issues '((1 ["1" "An
issue"]) (2 ["2" "Another one"])))
```

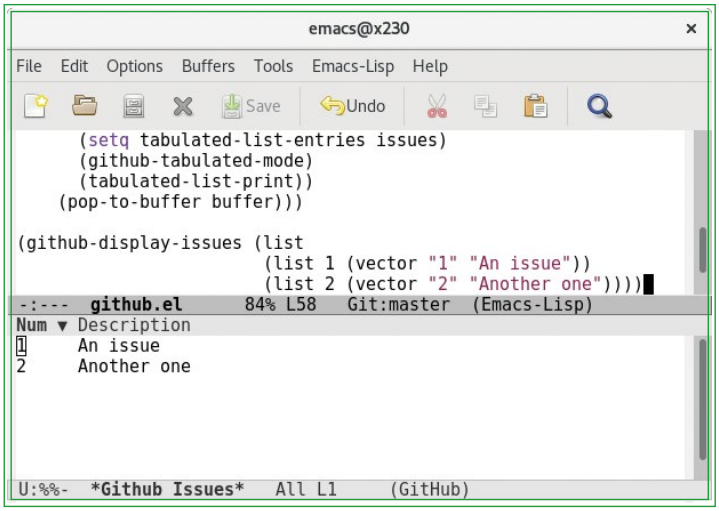



*Fig. 4 : Affichage en tableau de deux tickets créés manuellement.*

## 4. CONVERSION D'UN TICKET EN UNE LIGNE DE TABLEAU

Il nous faut maintenant faire le lien entre notre fonction **github-retrieve-issues** et **github-display-issues**. La première prend un nom de propriétaire et de projet en paramètres, fait une requête et retourne une liste de tickets au format de l'API GitHub. La seconde fonction prend une liste de tickets dans un format très particulier et les affiche dans un tableau. Il nous faut donc maintenant convertir une liste de tickets au format GitHub en une liste acceptable par **github-display-issues**. Pour savoir à quoi ressemble ce que retourne la fonction **github-retrieve-issues**, vous pouvez taper cette ligne :

```
(setq github-issue (seq-elt (github-retrieve-issues
"NixOS" "nixpkgs") 1))
```

Placez-vous à la fin de la ligne et tapez **M-x eval-print-last-sexp RET**. Le fait d'évaluer cette ligne stocke dans la variable **github-issue** le premier ticket de la liste retournée par **github-retrieve-issues** et affiche ce ticket dans le *buffer* courant :

```
((url . "https://api.github.com/repos/NixOS/nixpkgs/
issues/25376") (repository_url . "https://api.
github.com/repos/NixOS/nixpkgs") (comments_url
. "https://api.github.com/repos/NixOS/nixpkgs/
issues/25376/comments") (html_url . "https://github.
com/NixOS/nixpkgs/issues/25376") (id . 225390994)
(number . 25376) (title . "initial config must
include an editor") ...)
```

Nous voyons ici qu'un ticket est représenté par une liste de paires ayant pour clé un symbole (comme **url**, **number** et **title**). La liste de toutes les clés possibles est donnée par la documentation de l'API de tickets **[2]**. Pour récupérer la valeur associée à une clé donnée, nous pouvons utiliser **map-elt** :

```
(map-elt github-issue 'title)
```

Évaluez la ligne précédente pour obtenir le titre du ticket stocké dans **github-issue**. Nous pouvons maintenant écrire une fonction qui prend un ticket au format GitHub en paramètre et le transforme pour être inséré dans une ligne de tableau. Je vous propose de commencer par créer un test unitaire pour exprimer ce que l'on attend de la fonction avant d'implémenter cette dernière :

```
01: (ert-deftest github-convert-issue ()
02:   (let ((issue '((html_url . "url") (number . 25376) (title . "title") (something . 'else))))
03:     (should (equal (github-convert-issue issue)
04:                    (list 25376 (vector "25376" "title"))))))
```

L'appel à la macro **ert-deftest** (ligne 1) définit un nouveau test unitaire dont le nom est **github-convert-issue**. La ligne 2 définit une variable **issue** dont le contenu ressemble à un élément de la liste retournée par **github-retrieve-issues**. L'appel à la macro **should** (ligne 3) fera échouer le test si le paramètre est **nil**, valeur retournée par **equal** en cas de non-égalité. L'appel à la fonction **equal** permet de vérifier que le résultat de la conversion (premier paramètre, ligne 3) est bien égal à la valeur attendue (second paramètre, ligne 4). Nous considérons ici que l'identifiant de la ligne (premier élément de la liste représentant une ligne de tableau) est le numéro de ticket (ligne 4). Pour rappel, les éléments du vecteur (la ligne 4) représentent chacun une cellule du tableau : la première cellule affiche le numéro de ticket et la seconde une description.

Évaluez ce test et lancez-le en tapant **M-x ert RET github-convert-issue RET**. Le test doit échouer avec le message **(void-function github-convert-issue)**, car la fonction testée n'existe pas encore.

La fonction de conversion peut s'implémenter comme ceci :

```
(defun github-convert-issue (issue)
  "Convert ISSUE to a line for 'github-display-issues'.
ISSUE is a map of key/value pairs as returned by 'github-retrieve-issues'."
  (let ((number (map-elt issue 'number)))
    (list number
          (vector (int-to-string number) (map-elt issue 'title)))))
```

Une fois cette fonction évaluée, vous pouvez relancer le test et celui-ci devrait passer. Maintenant que nous savons convertir un ticket, il nous faut être capable de convertir une séquence de tickets (telle que retournée par **github-retrieve-issues**). Ceci se fait simplement grâce à la fonction **seq-map** :

```
(defun github-convert-issues (issues)
  "Convert ISSUES to lines for 'github-display-issues'.
ISSUES is a list of maps as returned by 'github-retrieve-issues'."
  (seq-map #'github-convert-issue issues))
```

La fonction **seq-map** applique la fonction passée en premier paramètre (ici **github-convert-issue**) à chaque élément de la séquence, pris l'un après l'autre, passée en second paramètre (ici **issues**) puis retourne une séquence des résultats. Notre nouvelle fonction **github-convert-issues** retourne donc bien une séquence où chacun des éléments est le résultat de la conversion d'un élément de la séquence **issues**. Évaluez cette fonction.

## 5. ASSEMBLAGE ET FINITIONS DU PROJET

Il nous reste maintenant seulement à rassembler les bouts ci-dessus dans une fonction *interactive* (qui peut être appelée facilement par l'utilisateur) :

```
01: (defun github-display-project-issues (owner project)
02:   "Display a tabulated list of issues in OWNER/PROJECT."
03:   (interactive (list
04:                 (read-from-minibuffer "Owner: ")
05:                 (read-from-minibuffer "Project: ")))
06:   (github-display-issues
07:    (github-convert-issues (github-retrieve-issues owner project))))
```

Mises à part les lignes 3 à 5, le code ne présente aucune nouveauté. Après avoir évalué la fonction ci-dessus, tapez **M-x github-display-project-issues RET**, puis entrez le nom d'un propriétaire de projet (par exemple **NixOS**), validez par la touche **<Entrée>**, puis entrez le nom du projet (par exemple **nixpkgs**) et validez par la touche **<Entrée>**.

Ceci devrait vous donner un affichage assez proche de celui de la figure 1, les liens et le thème sombre en moins. Les lignes 3 à 5 indiquent que l'utilisateur peut appeler directement cette fonction (on dit « *interactively* »), par exemple avec **M-x** (comme nous venons de le faire) ou un menu (voir ci-dessous). Ces lignes indiquent à Emacs que si cette fonction est appelée interactivement, les paramètres **owner** et **project** doivent prendre comme valeur respective les résultats des lignes 4 et 5. Ces lignes demandent à l'utilisateur de rentrer une valeur au clavier.

Vous pouvez facilement ajouter votre fonction à la fin du menu ***Tools*** :

```
(define-key-after
    menu-bar-tools-menu
    [github-display-project-issues]
  '("Display GitHub Issues" . github-display-
project-issues))
```

Pour convertir les numéros de tickets en lien vers la page web du ticket (comme dans la figure 1), vous pouvez adapter la fonction **github-convert-issue** de façon à appeler **(github-convert-issue-number-to-link issue)** au lieu de **(int-to-string number)**. Évaluez à nouveau **github-convert-issue** puis la nouvelle fonction suivante :

```
(defun github-convert-issue-number-to-link
(issue)
  "Convert ISSUE's number to a link to
GitHub's page for ISSUE."
  (list (int-to-string (map-elt issue
'number))
        'action (lambda (_) (browse-url
(map-elt issue 'html_url)))))
```

Cette fonction retourne une liste représentant un bouton sous la forme acceptée par **tabulated-list-mode**. Le bouton doit contenir une action, spécifiée ici par une fonction anonyme grâce à **lambda**. Cette fonction, qui sera appelée quand l'utilisateur cliquera sur le bouton, prend un paramètre (que l'on nomme _ pour signifier que l'on ne s'en sert pas) et appelle la fonction **browse_url** (qui ouvre votre navigateur web) avec l'URL du ticket en paramètre. Je vous laisse l'adaptation du test unitaire en exercice (compliqué).

Vous avez peut-être noté qu'un petit nombre seulement de tickets (30 au maximum) s'affiche, même si le projet en contient plus. GitHub utilise en effet la pagination pour diminuer la taille des réponses à calculer et à renvoyer. Pour demander plus de réponses, la documentation de l'API **[1]** recommande d'utiliser les liens **next** et **last** fournis dans les en-têtes HTTP (donc, avant la liste des tickets au format JSON). Je vous laisse, là encore, le soin d'implémenter cette fonctionnalité par vous-même.

## CONCLUSION

Dans cet article, nous avons vu les bases de l'utilisation d'Emacs ainsi que les bases du langage Emacs Lisp. En très peu de lignes, nous sommes arrivés à faire une requête sur un serveur distant, à transformer la réponse et à l'afficher dans un tableau avec des liens. Nous avons même écrit un test unitaire. Emacs est un outil extrêmement puissant et malléable qui bénéficie de quarante ans d'effort de développement. Sa communauté est de plus en plus active avec de nouveaux paquets tous les jours. ■

## RÉFÉRENCES

**[1]** API de GitHub : https://developer.github.com/v3/

**[2]** API de tickets : https://developer.github.com/v3/issues/#list-issues-for-a-repository

# DJANGO
## PAR LA PRATIQUE

**STÉPHANE TÉLETCHÉA**
[Enseignant-chercheur en bioinformatique structurale, université de Nantes]

**MOTS-CLÉS : PYTHON, INTERNET, APPLICATION, ADMINISTRATION, MTV, BASE DE DONNÉES**

**La mise en place d'un site internet laisse de nos jours peu de choix à la créativité. Il s'agit souvent de faire un compromis entre l'utilisation d'un framework clé en main à la WordPress, ou d'utiliser des services à tout faire fournis par des prestataires tiers. Quand il s'agit de mettre en ligne un service personnalisé, performant et évolutif, il est possible de bénéficier des avantages des deux approches en les combinant, grâce à Django. Nous allons voir dans cet article sa mise en œuvre rapide et un exemple complet d'application.**

Le projet **Django** a été créé à partir d'applications utilisées tous les jours par une équipe de développement dans la ville de Lawrence, Kansas, USA, pour propulser le site du journal de la ville. Tout a commencé à l'automne 2003, quand les programmeurs web Adrian Holovaty et Simon Willison ont commencé à utiliser **Python** à la place de **PHP** pour répondre aux multiples demandes des journalistes et éditeurs du journal. C'est la meilleure approche qu'ils aient trouvée pour répondre « en temps et en heure » aux demandes « urgentes et prioritaires ». À l'été 2005, les outils développés étaient devenus une collection conséquente d'outils. Ils pouvaient maintenant permettre à tous les sites du journal de fonctionner sans problème et être modifiés rapidement pour répondre aux besoins des journalistes. Les développeurs, dans une équipe qui avait accueilli désormais Jacob Kaplan-Moss, ont décidé de libérer le code source sous licence BSD et de le nommer Django, en hommage au fameux guitariste de jazz manouche Django Reinhardt. Depuis, même si les développeurs principaux (Adrian et Jacob) contrôlent l'évolution du *framework*, le projet est développé par une communauté toujours plus importante de développeurs. La première

version stable (1.0.4) est sortie en septembre 2008 et Django a commencé à devenir populaire avec la version 1.4 sortie en 2012, première version LTS (*Long Term Support*) réellement déployée à grande échelle. Le cycle de développement est très rapide avec environ une version mineure par semaine (1.8.16, 1.8.17...) et une version majeure tous les 9 mois (environ) **[1]**. Cet article sera basé sur la version 1.11, dernière version LTS qui supporte encore Python 2.7, maintenue jusqu'en avril 2020.

## 1. MISE EN PLACE RAPIDE

La grande popularité de Django, en parallèle de l'utilisation de plus en plus importante de Python, tient à la simplicité de mise en œuvre initiale. En quelques commandes, nous aurons un site web fonctionnel comprenant une première page d'accueil, une interface d'administration et la gestion très fine des permissions d'accès aux éléments présents dans la base de données. Pour mettre en place ce site, nous allons utiliser la version 1.11, la dernière version qui va supporter Python 2.7 avant le grand saut vers le tout Python 3. C'est le moment de tester nos dernières anciennes lignes de code avant de faire la conversion de tous les éléments et de vous informer sur les changements qu'il faudra apporter **[2]**.

Ayant régulièrement le besoin de créer différents projets, il est utile de cloisonner chacun d'entre eux pour éviter les conflits de versions qui pourraient se présenter sinon. L'écriture de cet article a été réalisée sous **Ubuntu 16.04 LTS** à jour et, sauf indication contraire, les logiciels utilisés sont ceux de la distribution. Nous allons dans un premier temps créer l'environnement virtuel qui va contenir les dépendances Python et y installer la dernière version de Django :

```
$ virtualenv demonstration_django
$ source demonstration_django/bin/activate
$ pip install django==1.11
```

Il faut maintenant commencer le projet qui va nous occuper durant le reste de l'article, à savoir « monSuperSite ». La commande qui permet de créer ce site est **startproject**. Attention, nous allons créer une arborescence à côté ce celle de l'environnement virtuel, pour ainsi séparer l'environnement virtuel du projet, nous verrons l'intérêt plus loin :

```
$ django-admin.py startproject monSuperSite
$ tree -L 2 -d
├── demonstration_django
│   ├── bin
│   ├── include
│   ├── lib
│   ├── local
│   └── share
└── monSuperSite
    └── monSuperSite
```

Le projet « monSuperSite » contient maintenant un sous-répertoire du même nom qui va contenir les informations principales du site. Ce répertoire va servir à régler les accès au site, puis à orienter les requêtes par URL de l'utilisateur. L'environnement minimal nécessaire pour Django est installé, nous allons maintenant personnaliser le site pour lui faire afficher des informations plus intéressantes.

## 2. UTILISATION RAPIDE DU MODÈLE MTV

La gestion des données par Django suit le principe « Model-Template-View », dérivé du très connu « Modèle-Vue-Contrôleur » popularisé par Trygve Reenskaug lors de sa visite au Palo Alto Center **[3]**. Le contrôleur est Django lui-même, le modèle est transformé en tables **SQL** via l'ORM (*Object Relational Mapping*) intégré et la vue est une page HTML enrichie de variables propres à Django, ou qui peuvent être basées sur **Jinja 2 [4]**.

L'accès à une donnée du projet se fait pour le visiteur du site via l'accès à une page selon son URL. Nous allons voir ensemble comment créer une page d'accueil pour notre super site. Le routage des URL se situe tout d'abord dans le fichier **monSuperSite/urls.py**. Les pages à afficher sont déduites à partir d'expressions régulières. Nous allons commencer à explorer son fonctionnement. Il faut lancer Django sur notre nouveau site :

```
(demonstration_django) $ cd monSuperSite
(demonstration_django) $ python manage.py makemigrations
No changes detected
(demonstration_django) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
```

```
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying sessions.0001_initial... OK
(demonstration_django) $ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
April 21, 2017 - 10:30:19
Django version 1.11, using settings 'monSuperSite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Il faut synchroniser les tables préexistantes de Django pour l'authentification des sites et la gestion des permissions avec les commandes **makemigrations** et **migrate**. Nous reviendrons sur leur utilisation plus tard. Le site est maintenant vide, mais fonctionnel (voir figure 1).

**NOTE**

Le prompt de l'environnement virtuel **(demonstration_django)** ne sera plus répété par la suite.

Le fichier **urls.py** permet à Django de faire la correspondance entre une URL et une fonction Python, de Django ou que vous allez écrire. La mise en place rapide du site a prérempli le fichier avec un routage vers l'interface d'administration fournie par défaut :

```
urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```

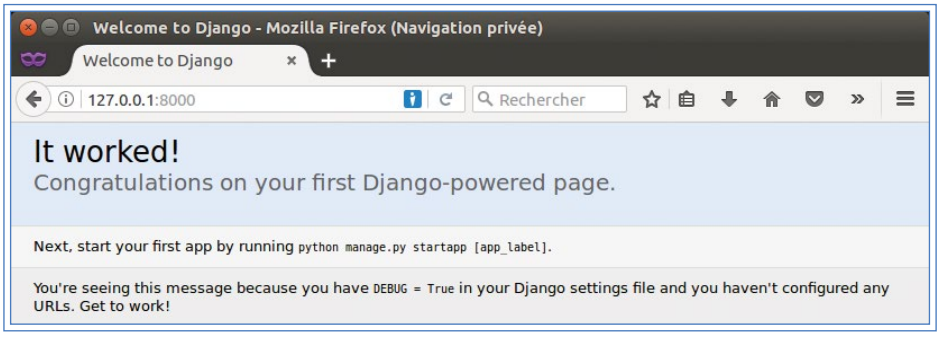



*Fig. 1 : Page initiale de tout projet Django bien commencé, avec les indications pour créer le reste.*

Quand vous vous rendez sur le site http://localhost:8000/admin, l'interface d'administration est présentée. L'objectif de cet article n'est pas de présenter en détail cette puissante interface, pensez simplement à mettre rapidement un mot de passe non trivial avec la commande :

```
$ python manage.py createsuperuser
```

Notez que vous pouvez laisser la commande **python manage.py runserver** en place, Django analysera en temps réel le code que vous êtes en train d'écrire, ce qui vous permet de vérifier que tout fonctionne (ou pas) tout de suite. Nous allons maintenant améliorer la page d'accueil du site, et indiquer à Django où aller la chercher. Nous allons commencer par modifier le fichier **urls.py** pour faire un rendu simple vers un fichier html comme suit (attention à l'indentation en Python) :

```
from django.views.generic.base import
TemplateView

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', TemplateView.as_
view(template_name='index.html'),
name='accueil'),
]
```

Il faut maintenant créer un répertoire **templates** dans le répertoire **monSuperSite** et y créer le fichier **index.html** qui contiendra le minimum « vital » :

```
$ cd monSuperSite
$ mkdir templates
$ cd templates
$ vi index.html
```

Le contenu du fichier **index.html** se compose de balises HTML standards et de définitions spécifiques pour Django. Dans le cas du fichier d'index ci-dessous, ces étiquettes serviront à définir une zone spécifique de contenu avec la commande **{% block content %}**, ce qui va permettre de changer uniquement le contenu du cœur de la page. Ce fichier minimum est le meilleur endroit pour faire les inclusions de code **JavaScript** et **CSS**, afin de modifier très rapidement l'aspect d'un site sans modification intrusive à différents endroits. Dans le code ci-après, le populaire *framework* **Bootstrap** est incorporé en utilisant le code fourni en exemple dans la documentation :

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8"><!-- Latest compiled and minified
CSS -->
<link rel="stylesheet" href="https://maxcdn.
bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3R
Ydg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
<!-- Optional theme -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
com/bootstrap/3.3.7/css/bootstrap-theme.min.css"
integrity="sha384-rHyoN1iRsVXV4nD0JutlnGaslCJuC7uwjduW9SV
rLvRYooPp2bWYgmgJQIXwl/Sp" crossorigin="anonymous">

</head>
<body>
This is the starting page of the project.
   <div id="content">
       {% block content %}{% endblock content %}
   </div>
<!-- Latest compiled and minified JavaScript -->
<script src="https://maxcdn.bootstrapcdn.com/
bootstrap/3.3.7/js/bootstrap.min.js" integrity="sha384-Tc
5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcP
D7Txa" crossorigin="anonymous"></script>

</body>
</html>
```

Pour que Django sache où trouver ce fichier, il faut aussi lui indiquer que **monSuperSite** est une application du projet, il faut donc surcharger dans le fichier **settings.py** la liste **INSTALLED_APPS** comme suit :

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'monSuperSite',
]
```

Ces étapes préliminaires réalisées, notre site est maintenant fonctionnel, nous allons pouvoir le transformer et nous concentrer sur le contenu à présenter à l'utilisateur.

## 3. CONCEPTION DE L'APPLICATION ET DES DÉPENDANCES

L'étape la plus complexe dans la mise en place d'une application, quelle qu'elle soit, est d'établir les relations qui vont relier les différentes tables. L'intérêt pour un utilisateur de Django, c'est qu'il va pouvoir se concentrer principalement sur ce qu'il veut afficher, et non pas sur comment il veut construire les tables de son application selon la base de données qu'il aura besoin d'utiliser. Cette flexibilité est particulièrement déroutante lorsque l'on a l'habitude de concevoir une application de A à Z en établissant une bonne fois pour toutes les relations entre les données. Dans notre cas, nous allons les implémenter pas à pas pour arriver à une application fonctionnelle, riche, sans optimisation préalable.

Nous allons concevoir une petite application de gestion de correspondances entre des protéines, leur structure tridimensionnelle et les ligands connus qui pourraient y être associés. Ce modèle à trois partenaires nous permettra de mettre en place plusieurs types de relations entre les tables et pourrait tout aussi bien être transposé à un modèle d'e-commerce qui relierait produits, clients et fournisseurs.

### RAPPELS DE BIOLOGIE

**Les protéines sont composées de briques élémentaires appelées acides aminés. Ces briques sont reliées entre elles pour former une chaîne polypeptidique qui se replie sur elle-même en trois dimensions pour arriver à une architecture stable. De cette architecture découlent leur fonction et leur activité. Si les informations de structure tridimensionnelle sont disponibles, il est possible de concevoir des inhibiteurs dédiés de leur fonction, ce qui peut amener après un long processus de validation (supérieur à 10 ans...) à la mise sur le marché de nouveaux médicaments. Le site présenté ici permet de mettre rapidement en place un système d'information sur les données publiques disponibles pour chacune des protéines.**

Pour faciliter l'accès aux informations qui seront enregistrées dans les tables, pour permettre leur édition ou leur incorporation, nous allons gérer l'authentification des utilisateurs, et leur définir un rôle spécifique. Nous allons ainsi créer une application dédiée pour l'authentification qui se basera sur le modèle d'utilisateur de Django. Cette pratique assez courante permet de rajouter facilement des champs spécifiques tout en conservant la facilité de gestion de l'authentification fournie par défaut dans Django.

## 3.1 L'application utilisateur améliorée : curators

Un travail important dans ce monde moderne consiste à nettoyer l'information, l'organiser, la rendre intelligible. En anglais, ce travail est nommé la *curation* et jusqu'à ce que l'intelligence artificielle ne les remplace, c'est un travail de fourmi très important. Pour quantifier la qualité de ces curateurs, nous allons créer une application dédiée qui permettra de noter ces curateurs avec un score. Il faut créer une application dédiée et modifier les modèles nécessaires :

```
$ cd ~/monSuperSite
$ python manage.py startapp curators
$ cd curators
$ vi models.py
```

Le fichier **models.py** contient les classes qui seront intégrées dans le projet **monSuperSite** :

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.db import models

# Create your models here.

from django.contrib.auth.models import User
from django.core.validators import MinValueValidator,
MaxValueValidator

class Curator(models.Model):
    user = models.OneToOneField(User, on_delete=models.
CASCADE)
    bio = models.TextField(max_length=500, blank=True)
    score = models.IntegerField(default=0,
                        validators=[MinValueValidator(0),

MaxValueValidator(5)])
    birth_date = models.DateField(null=True, blank=True)
```

Pour que le nouveau modèle créé soit présent dans l'interface d'administration de Django, il faut modifier le fichier **monSuperSite/settings.py** pour ajouter l'application **curators** à la liste **INSTALLED_APPS** comme vu précédemment, mais aussi compléter le fichier **admin.py** de l'application avec a minima les instructions suivantes :

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.contrib import admin

# Register your models here.

from curators.models import *
admin.site.register(Curator)
```

Ces fichiers étant créés, il faut synchroniser les tables pour que les changements soient pris en compte, en faisant appel aux migrations.

```
$ python manage.py makemigrations
Migrations for 'curators':
  curators/migrations/0001_initial.py
    - Create model Curator
(demonstration_django)stephane@maison:
~/monSuperSite$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth,
contenttypes, curators, sessions
Running migrations:
  Applying curators.0001_initial... OK
```

Après ces modifications, l'interface d'administration présente une nouvelle application qui ne contient pour le moment que l'objet **Curator** (voir figure 2).

Dans le cadre de cette présentation rapide de l'authentification, nous nous contenterons de l'interface d'administration principale de Django. Si vous voulez contrôler plus finement la gestion des utilisateurs et leur création en dehors de l'interface d'administration, des explications plus complètes sont disponibles sur les sites donnés en référence **[5]**, avec un code disponible sous **GitHub [6][7]**.

## 3.2 L'application « protéines »

L'application « protéines » va contenir quelques éléments nécessaires à la caractérisation d'une protéine : son nom, sa fonction, sa séquence en acides aminés. Pour la plupart de ces informations, il est possible de récupérer directement les éléments en utilisant une fiche normalisée, mise à jour mensuellement et validée par des scientifiques experts. Nous allons tout d'abord créer l'application « protéines » avant de définir quels sont les champs nécessaires, afin de présenter l'intérêt de l'utilisation de Django. Si vous voulez utiliser une application basée sur du

e-commerce, il vous suffit de changer l'application « protéines » en application « produits ». Pour créer l'application, il faut se situer dans le répertoire **monSuperSite**, le premier créé au début de cette présentation, et créer l'application :

```
$ python manage.py
startapp protein
```

Dans le répertoire **protein** vous trouvez maintenant 5 fichiers Python :

- **admin.py** : sert à définir la présentation des informations de l'application dans l'interface d'administration de Django ;
- **apps.py** : sert à définir des classes génériques pour l'application ;
- **models.py** : fichier qui va contenir les définitions des classes qui seront transformées en tables par la suite ;
- **tests.py** : vous pourrez créer des tests unitaires qui serviront à valider le bon fonctionnement de l'application lors de changements futurs ;
- **views.py** : c'est dans ce fichier que nous pourrons créer les vues spécifiques de l'application.

Nous allons créer dans le fichier **models.py** la classe qui va servir à stocker en base de données les éléments nécessaires pour une protéine donnée. Cette classe sera nommée **Protein**. Pour que l'administrateur puisse y avoir accès dans l'interface d'administration, il suffira de modifier le fichier **admin.py** en référençant la classe **Protein** comme nous l'avons fait pour la classe **Curators**.

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models

# Create your models here.

class Protein(models.Model):
    name = models.CharField(max_length=255)
    description = models.CharField(max_length=1000)
    gene = models.CharField(max_length=100,default='NOGENENAME')
    mw = models.PositiveIntegerField(blank=True,default=0)

    def __unicode__(self):
        return u'%s' % self.name
```

Voici quelques remarques concernant ce fichier **models.py** : tout d'abord, le fichier commence par la déclaration d'encodage. En Python 2.7, les chaînes de caractères ne sont pas par défaut en UTF-8 donc sans cette précaution le moindre accent qui apparaîtrait dans un commentaire entraînerait une erreur.

Notre classe hérite de la classe **Model** de Django, c'est cela qui permet d'avoir la cohérence d'ensemble de l'application. La méthode privée **__unicode__()** n'est utile que pour afficher le nom de l'objet dans l'interface d'admin. Il n'y a pas besoin de spécifier de clé primaire, elle sera générée automatiquement pour nous avec un champ s'auto-incrémentant.

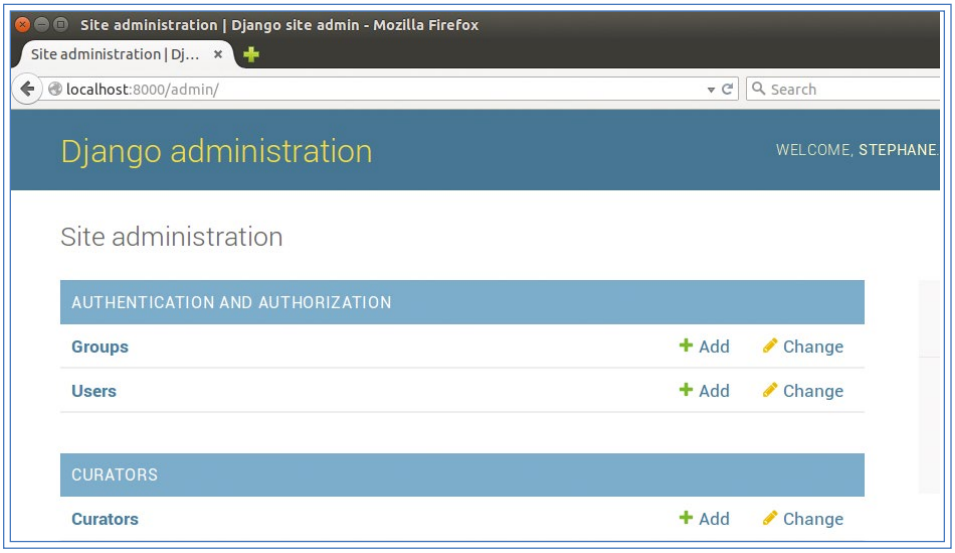



*Fig. 2 : L'interface d'administration contient la nouvelle application qui vient d'être ajoutée au système. Il suffit de cliquer sur le bouton **Add** pour ajouter un nouveau curateur au système, sans intervention additionnelle sur les tables ou la création de pages dédiées.*

Contrairement à la déclaration d'un objet Python classique, il n'y a pas besoin de déclarer une méthode **__init__()** qui servirait à définir les champs de l'objet. Nous pouvons directement créer les champs à partir de définitions explicites (**CharField**, **PositiveIntegerField**, etc.) héritées de la classe **models**. L'ensemble de ces champs est très bien détaillé dans la documentation officielle (en français, et à jour) **[8]**.

Après avoir modifié ces deux fichiers, il faut maintenant indiquer à Django qu'une nouvelle application fait partie du

projet. Comme précédemment, cela veut dire compléter la liste des applications dans le fichier **settings.py**. Une fois cet ajout effectué, il faut effectuer la migration de l'application :

```
$ python manage.py makemigrations
Migrations for 'protein':
  protein/migrations/0001_initial.py
    - Create model Protein
(demonstration_django)stephane@maison:~/monSuperSite$
python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
curators, protein, sessions
Running migrations:
  Applying protein.0001_initial... OK
```

Il nous suffit maintenant de créer une vue associée et de l'appeler à partir du fichier **urls.py**. Comme l'application permet d'isoler les éléments indépendants les uns des autres dans les tables, nous allons continuer ce principe en créant un fichier **urls.py** dans le répertoire **protein**, mais aussi un répertoire **template** qui contiendra les informations spécifiques pour les protéines. Le fichier **urls.py** sera appelé à partir du fichier **urls.py** principal présent dans le sous-répertoire **monSuperSite**, la liste des chemins de recherche sera complétée comme suit :

```
...
    url(r'protein/', include('protein.urls')),
]
```

Le fichier **protein/urls.py** qu'il faut créer contient le code suivant :

```
from django.conf.urls import include,url

from protein.views import *

urlpatterns = [
    url(r'(?P<identifiant>\d+)', one_protein,
name="one_protein")
]
```

Quand un utilisateur fait appel à l'URL **/protein/1**, le premier fichier **urls.py** redirige vers l'application **protein** et le second fichier **urls.py** redirige la requête vers la fonction **one_protein()** présente dans le fichier **views.py**. Le paramètre **1** est récupéré avec une expression régulière nommée, le chiffre sera ainsi passé à la méthode **one_protein()** avec le paramètre **identifiant**. Le fichier **views.py** de l'application contient une fonction unique qui sert à récupérer l'objet dans la base à partir de cet identifiant, il contient :

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.shortcuts import render, get_
object_or_404

from protein.models import Protein

# Create your views here.
def one_protein(request, identifiant):
    my_protein=get_object_or_404(Protein,
pk=identifiant)
    return render(request,"one_protein.
html",{'protein':my_protein})
```

Notez que par rapport au fichier initialement créé, nous importons la fonction **get_object_or_404()**. Cette fonction utilisée en ligne 10 permet de faire une requête dans la base de données et si celle-ci échoue alors une page 404 est présentée. Tant que le paramètre **DEBUG = True** est présent dans le fichier **settings.py**, l'erreur s'affiche dans le navigateur, mais en production il faudra prévoir une vraie page 404 ou une redirection vers une page générique.

Il ne reste plus qu'à créer le *template* **one_protein.html** dans un répertoire **templates** de l'application « protéine ». Un peu de mise en forme est effectué avec Bootstrap, mais il ne s'agit que d'une invitation à faire mieux...

```
{% extends "index.html" %}
{% block content %}
<div class="col-md-12">
<h1>Protein detail</h1>
<dl class="dl-horizontal">
<dt>Protein: </dt><dd>{{ protein.name }}
</dd>
<dt>Description: </dt><dd>{{ protein.
description }}</dd>
<dt>Gene:</dt><dd> {{ protein.gene }}</dd>
</dl>
</div>
{% endblock %}
```

Pour compléter un peu les données, nous allons nous servir de la fiche UNIPROT de la protéine **P53** (http://www.uniprot.org/uniprot/P04637). Il suffit de se rendre dans l'interface d'administration pour remplir la fiche rapidement en copiant-collant les informations. Une fois ces informations entrées, la page faisant appel à l'application que nous venons d'écrire s'affiche avec les informations nécessaires (voir figure 3, page 86).

# ACTUELLEMENT DISPONIBLE

## HACKABLE N°20 !


# CRÉEZ UNE VEILLEUSE QUI MONTRE LES PHASES DE LA LUNE

## NE LE MANQUEZ PAS

**CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :**

**http://www.ed-diamond.com**

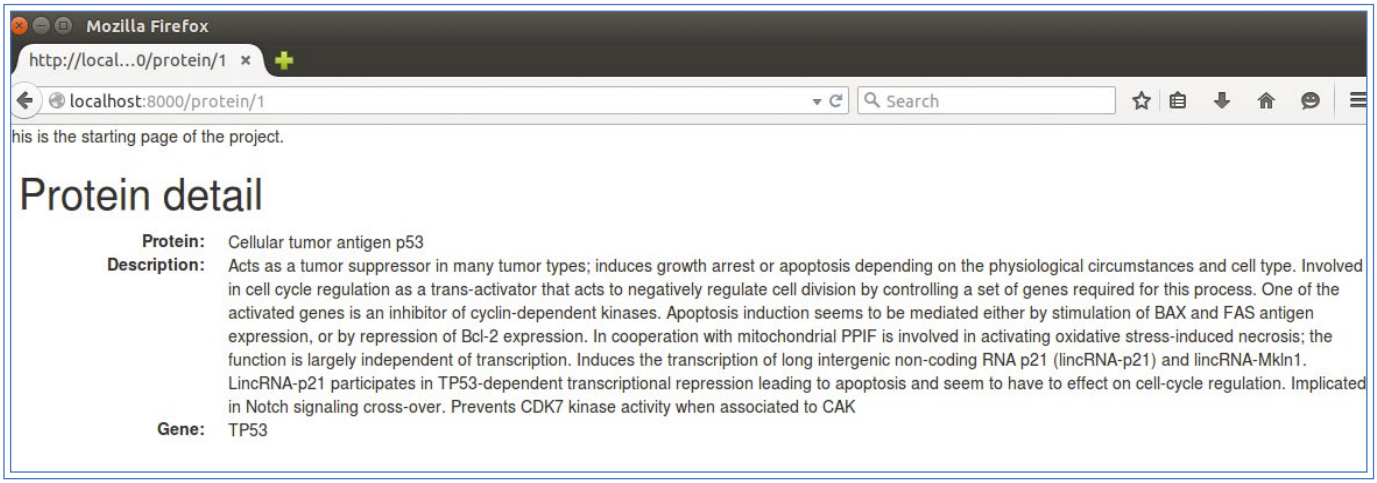



*Fig. 3 : Une première vue complète capable de traiter de manière automatique l'information présente dans la table. L'image présente le résultat de la requête sur la première protéine de la base, P53, et une partie des informations qui sont associées à la fiche.*

## 3.3 Les relations à plusieurs : deux applications supplémentaires

Plutôt que de concentrer nos efforts sur la mise en forme avancée du site (pour le moment), nous allons plutôt ajouter du contenu. L'intérêt d'un site contenant des informations sur des protéines (ou des produits) est qu'il doit être possible de relier ces protéines (ces produits) à d'autres éléments, par exemple à une drogue connue (une molécule pharmaceutique à activité quantifiée), mais aussi aux données tridimensionnelles disponibles sur la protéine. Pour simplifier la gestion des données, mais aussi pour présenter deux cas différents d'utilisation, nous allons supposer que chacune de nos applications aura une relation différente avec la table **Protein**. La première application va concerner le lien entre une structure tridimensionnelle et la protéine, il s'agira d'une relation **n** à **n** (plusieurs protéines peuvent être liées à plusieurs structures), nous l'appellerons « molécule ». L'autre application concerne les ligands, nous allons simplifier la relation avec une protéine à une relation de type **1** à **n**, une protéine pourra posséder plusieurs ligands.

### 3.3.1 L'application « molécule »

L'application est démarrée comme auparavant avec **startapp**, et ajoutée dans le fichier **settings.py**. Le fichier **models.py** est construit en indiquant quelques champs pour le code PDB (à quatre lettres), la chaîne (une lettre) et le nom de la structure. Les informations sont issues du site http:/www.rcsb.org.

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models

# Create your models here.

class Molecule(models.Model):
    pdb_code=models.CharField(max_length=4,blank=True)
    pdb_chain=models.CharField(max_length=1,blank=True)
    description=models.CharField(max_length=255,blank=True)
```

Il faut ensuite synchroniser les données du projet avec **python manage.py makemigrations** et **python manage.py migrate**. Pour l'instant, les applications sont indépendantes et la relation n'est pas établie entre les deux modèles. Comme nous voulons avoir une relation de **n** à **n** entre protéines et molécules, nous allons créer cette relation en y faisant référence à l'aide du mot-clé **models.ManyToManyField()**dans le modèle **molecule** :

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models

# Create your models here.

from protein.views import Protein

class Molecule(models.Model):
    pdb_code=models.CharField(max_length=4,blank=True)
```

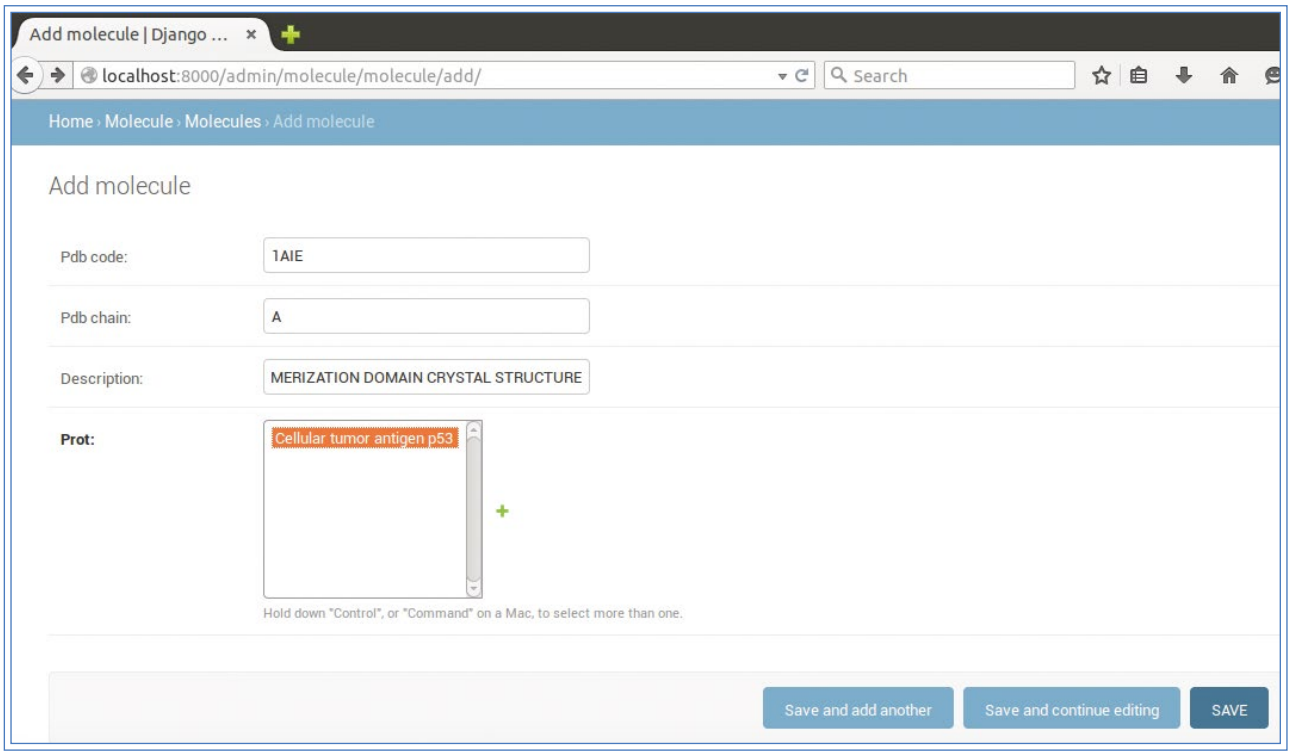



*Fig. 4 : L'ajout de l'entrée 1AIE comme nouvelle molécule se fait dans l'interface d'administration, mais l'établissement d'une relation de plusieurs à plusieurs se voit sous forme de liste d'entrées à choisir ou à ajouter en cliquant sur l'icône « + ».*

```
    pdb_chain=models.CharField(max_length=1,blank=True)
    description=models.CharField(max_
length=255,blank=True)
    prot=models.ManyToManyField(Protein)

    def __unicode__(self):
        return u'%s_%s' % (self.pdb_code,self.pdb_chain)
```

Il faut noter dans ce fichier qu'il faut faire l'import du modèle de l'application « protéine » pour faire la correspondance entre les deux éléments, puis ajouter la relation de plusieurs à plusieurs via l'entrée « prot ». Contrairement à la conception d'une application où les tables doivent être construites une par une, il n'y a pas besoin de se préoccuper de la table intermédiaire qui sert à faire les correspondances, c'est Django qui a créé cette table de manière automatique. Pour enregistrer cette nouvelle relation, il faut procéder aux migrations des tables :

```
$ python manage.py makemigrations
Migrations for 'molecule':
  molecule/migrations/0002_molecule_prot.py
    - Add field prot to molecule
(demonstration_django)stephane@stephane-VirtualBox:~/monSuperSite$
python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, curators,
molecule, protein, sessions
Running migrations:
  Applying molecule.0002_molecule_prot... OK
```

La traduction de cette relation est visible directement dans l'interface d'administration, où il faut maintenant indiquer lors de la création d'une entrée « molecule » la relation qui doit exister avec une entrée « protein » en la sélectionnant par simple clic (voir figure 4).

### 3.3.2 L'application « ligand »

Nous allons cette fois-ci créer une application « ligand » qui doit aussi être référencée dans le fichier **`settings.py`**. La contrainte entre les deux tables sera maintenant réalisée grâce à la méthode **`models.ForeignKey()`**. Le site **pubchem** (https://pubchem.ncbi.nlm.nih.gov/) est une des sources de référence pour les ligands chimiques. Ces molécules peuvent parfois avoir un nom commun qui correspond à des médicaments disponibles en pharmacie, nous utiliserons donc un champ pour le nom chimique, et un autre nom pour le « nom commun » afin de prendre en compte les deux informations :

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models

# Create your models here.

from protein.models import Protein

class Ligand(models.Model):
    pubchem_id=models.PositiveIntegerField(blank=True)
    common_name=models.CharField(max_length=255)
    chemical_name=models.CharField(max_length=255)
    p=models.ForeignKey(Protein)

    def __unicode__(self):
        return u'%s' % self.common_name
```

Si vous avez modifié tous les fichiers nécessaires, la migration se passe sans encombre :

```
Migrations for 'ligand':
  ligand/migrations/0001_initial.py
    - Create model Ligand
(demonstration_django) stephane@maison:~/monSuperSite$
python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
curators, ligand, molecule, protein, sessions
Running migrations:
  Applying ligand.0001_initial... OK
```

Nous avons maintenant mis en place plusieurs tables contenant différents types de relation, il est temps d'exploiter ces relations pour présenter une information enrichie à l'utilisateur de notre site web.

## 4. MISE À JOUR DE LA VUE PROTÉINE

Nous avions commencé par écrire une vue simple des actions disponibles dans la conception de la première application, avec la création d'un *template* dédié à la présentation de l'information. Nous pourrions de même créer un *template* dédié par application pour présenter le détail des objets qui y sont référencés, mais cela ne serait pas utile pour la présentation du service. Nous allons maintenant reprendre la première vue avec une présentation un peu plus complète (un peu plus de code Bootstrap et **jQuery**) du site, l'aspect étant toujours résolument austère.

Notre fichier **proteine/views.py** va maintenant être modifié pour transmettre au *template* les informations concernant les ligands et les molécules associées. Pour avoir l'affichage de quelques résultats, les ligands clofibrate et doxorubicine et la structure PDB 1YC5_B ont été rajoutés via l'interface d'admin. Pour obtenir la liste des éléments reliés à notre protéine, il suffit de trier les éléments qui lui sont associés avec l'attribut **_set.all()**. Cette abstraction permet l'utilisation d'une syntaxe identique pour obtenir les objets liés à l'objet **protein** principal sans qu'il ne soit nécessaire de se soucier du type de relation sous-jacent entre les modèles.

Le fichier **views.py** devient maintenant :

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.shortcuts import render, get_
object_or_404

from protein.models import Protein

# Create your views here.
def one_protein(request, identifiant):
    my_dict={}

    my_protein=get_object_or_404(Protein,
pk=identifiant)
    my_dict['protein']=my_protein

    my_ligands=my_protein.ligand_set.all()
    my_dict['ligands']=my_ligands

    my_molecules=my_protein.molecule_set.all()
    my_dict['molecules']=my_molecules
    return render(request,"one_protein.
html",my_dict)
```

Le fichier *template* **one_protein.html** est enrichi pour prendre en compte les nouveaux éléments :

```
{% extends "index.html" %}
{% block content %}
<div class="col-md-12">
<h1>Protein details</h1>
    <dl class="dl-horizontal">
    <dt>Protein: </dt><dd>{{ protein.name }}
</dd>
    <dt>Description: </dt><dd>{{ protein.
description }}</dd>
    <dt>Gene:</dt><dd> {{ protein.gene }}
</dd>
    </dl>
</div>
<div class="col-md-12">
<h1>Ligands binding to this protein</h1>
    <ul>
    {% for l in ligands %}
```

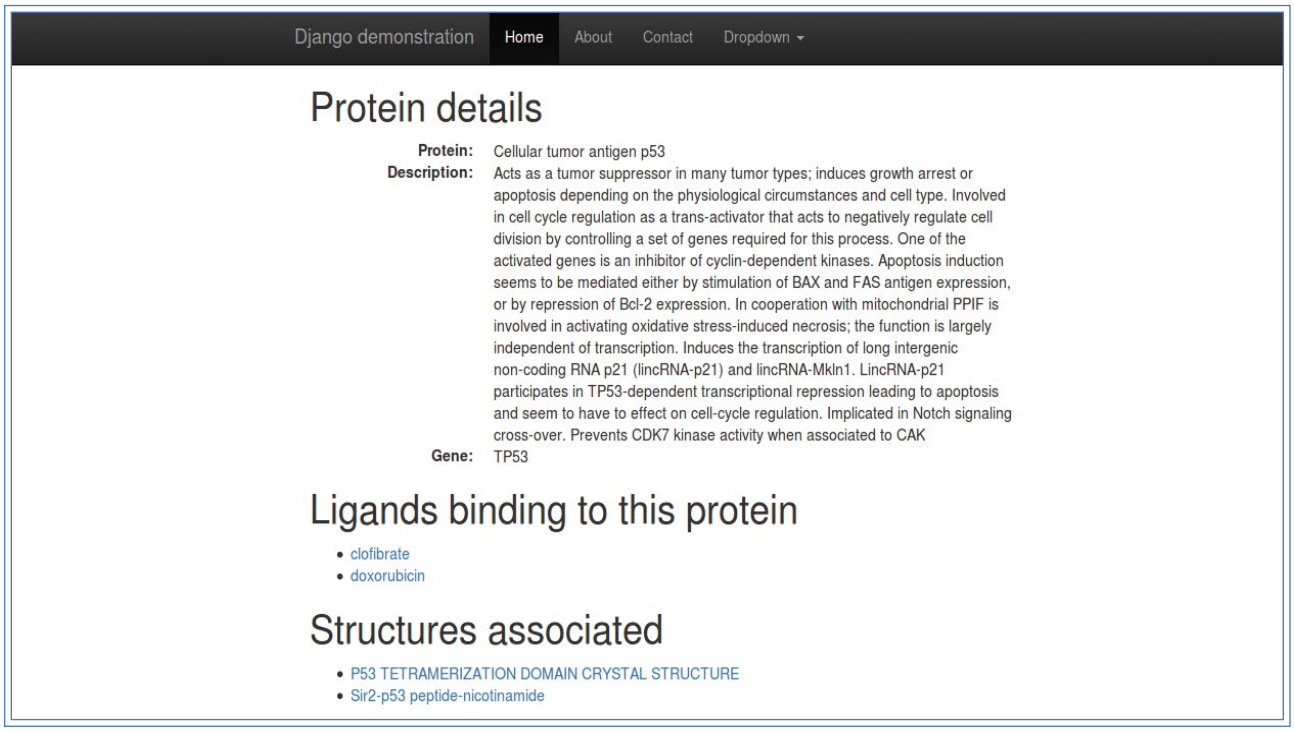



*Fig. 5 : La vue initiale comprenant les informations de la protéine, mais aussi les relations qui ont été ajoutées est affichée dans une page synthétique. L'essentiel du travail de traitement des données est effectué au niveau de la vue, mais il est possible d'accéder à de nombreux attributs dans le template sans requête additionnelle, gage de gain de temps, mais aussi de sécurité.*

```
    <li><a href="https://pubchem.ncbi.
nlm.nih.gov/compound//{{l.pubchem_
id}}">{{l.common_name}}</a></li>
{% endfor %}
    </ul>
</div>
<div class="col-md-12">
<h1>Structures associated</h1>
    <ul>
    {% for m in molecules %}
    <li><a href="http://www.rcsb.org/pdb/
explore/explore.do?structureId=/{{m.pdb_
code}}">{{m.description}}</a></li>
{% endfor %}
    </ul>
</div>

</div>
{% endblock %}
```

Notre première application est fonctionnelle, elle contient les tables contenant différents types de relation (**1** à **1** pour le modèle utilisateur, puis **1** à **n** et **n** à **n**). Le résultat n'est pour l'instant disponible que pour une protéine, mais il est déjà suffisant pour montrer la pertinence de Django pour la mise en forme rapide d'un site web rapide et riche (voir figure 5).

## 5. SAUVEGARDER LES DONNÉES ET LES RESTAURER

Le travail présenté brièvement ne peut être exploitable qu'avec un minimum de données entrées par l'utilisateur. Pour ne pas surcharger l'article, ces données ont été insérées directement via l'interface d'administration, mais il aurait été tout aussi possible de les insérer après authentification en utilisant le modèle **Curator** pour ajouter les données dans la base. Nous n'avons pas effectué de grosses modifications avec les migrations - cela sera pour un autre numéro - mais c'est bien ce que permettent les migrations, en ajoutant par exemple un champ à l'objet **Protein**. Lors des nombreuses manipulations, il est dangereux de ne pas effectuer de copie de ses données, sous peine de perdre le travail en cours. Django nous fournit une interface très pratique pour cela, toujours avec le gestionnaire de projets **manage.py**. Pour sauvegarder les données, il suffit ainsi de faire :

```
$ python manage.py dumpdata > fichier.json
```

Vous pouvez maintenant effectuer les changements nécessaires à l'organisation des tables, ajouter des champs dans chaque modèle, etc. En général les migrations permettent d'éviter les problèmes, mais cela peut arriver suite à une fausse manœuvre... Si cela vous arrivait, il suffit de revenir en arrière en spécifiant la version de migration qui fonctionnait (allez voir dans le répertoire migrations de chacune des applications), et d'importer les données au stade où vous en étiez à ce moment.

```
$ python manage.py loaddata fichier.json
```

## 6. PASSAGE EN PRODUCTION

Cet élément est rarement traité, car si vous développez l'application, vous n'êtes probablement pas l'administrateur système qui va s'occuper de cette étape. Comme vu en début d'article, nous avons cherché à cloisonner les versions de Django et de ses dépendances dans un environnement virtuel dédié. Le plus simple est de se baser sur celui-ci pour gérer les dépendances à installer sur le serveur. Pour connaître la liste des paquets installés, il faut exécuter la commande :

```
$ pip freeze > requirements.txt
```

Sur l'exemple proposé dans cet article, le résultat est :

```
appdirs==1.4.3
Django==1.11
packaging==16.8
pkg-resources==0.0.0
pyparsing==2.2.0
pytz==2017.2
six==1.10.0
```

Vous pouvez maintenant copier le code sur le serveur qui va héberger le site, ou mieux, cloner le dépôt du projet, il n'y aura en effet que quelques étapes à réaliser :

- adapter le fichier **settings.py** pour définir l'URL pour les fichiers statiques et media ;
- ajouter le nom de la machine dans la liste **ALLOWED_HOSTS** ;
- passer le drapeau **DEBUG = True** à **False** ;
- changer la base de données (passer de **Sqlite** à **MySQL** par exemple). Le plus simple est de faire le *dump* de la base avec Django, de changer les réglages dans le fichier **settings.py**, et de charger les données ;
- déplacer les fichiers statiques au bon endroit avec la commande **python manage.py collectstatic** ;
- utiliser le fichier de configuration pour le serveur web, qui traitera les fichiers statiques et utilisera l'environnement virtuel que vous allez recréer.

Pour recréer l'environnement virtuel, il vous suffit sur la seconde machine de lancer à nouveau un **virtualenv** (dans un espace dédié, indépendant du projet de site) et d'installer les dépendances avec la commande :

```
$ pip install -r requirements.txt
```

Pour Apache 2.4, le fichier de configuration suivant présente un exemple directement utilisable dans un hôte virtuel dédié :

```
<VirtualHost *:80>

   ServerName localhost
   ServerAlias localhost

    # https://docs.djangoproject.com/en/
dev/howto/deployment/wsgi/modwsgi/
    # Pour monSuperSite

    WSGIDaemonProcess monSuperSite
python-path=/home/stephane/monSuperSite:
/home/stephane/monSuperSite/tools/
django1.8/lib/python2.7/site-packages
processes=2 threads=4 display-
name=monSuperSite
    WSGIProcessGroup monSuperSite
```

```
    WSGIScriptAlias /monSuperSite '/home/
stephane/monSuperSite/monSuperSite/wsgi.py'

    # Pour monSuperSite

    <Directory /home/stephane/monSuperSite/
monSuperSite/>
        WSGIProcessGroup monSuperSite
        <Files wsgi.py>
            Options +ExecCGI -MultiViews
+FollowSymLinks
            AllowOverride None
            Require all granted
        </Files>
    </Directory>

    Alias /monSuperSite/static /home/
stephane/monSuperSite/monSuperSite/static/
   <Location /monSuperSite/static/>
        WSGIProcessGroup monSuperSite
        Options -Indexes +FollowSymLinks
        Require all granted
    </Location>

    Alias /monSuperSite/media /home/
stephane/monSuperSite/monSuperSite/media/

    <Location /monSuperSite/media/>
        WSGIProcessGroup monSuperSite
        Options -Indexes +FollowSymLinks
        Require all granted
    </Location>

    # Pour séparer les bons logs de
l'ivraie

    ErrorLog ${APACHE_LOG_DIR}/
monSuperSite-error.log
    CustomLog ${APACHE_LOG_DIR}/
monSuperSite-access.log combined
</VirtualHost>
```

Vous pourrez trouver plus de détails sur l'intégration complète d'un site tournant sous Django sur mon site **[9]**.

## CONCLUSION

Nous avons pu voir dans cet article le fonctionnement opérationnel de Django, et vu les différences qui existent dans son implémentation lorsqu'il s'agit de mettre en place un site fonctionnel. Au lieu d'utiliser un *framework* clé en main comme il en existe tant, Django vous permet de vous libérer des vicissitudes de l'utilisation d'une base de données, en l'ajustant à votre problème si nécessaire. Nous n'avons fait que survoler les fonctionnalités, en particulier la conception des tables et leur évolution, à travers un cas simple. Dans des cas plus complexes, Django nous apporte une réelle flexibilité dans le déploiement d'applications en peu de temps, et ce pour un investissement raisonnable. De par la complexité de Django, sa puissance, et son rayonnement, c'est assurément un *framework* à considérer pour des projets pérennes. Le prototypage d'un site est rapide, surtout en le couplant avec d'autres technologies éprouvées en JavaScript et en CSS. C'est cependant la vitalité de la communauté, la diversité des applications et la richesse des briques logicielles qui en font un des acteurs majeurs du Web de demain. ■

## RÉFÉRENCES


**[1]** Roadmap pour les prochaines versions de *Django* : https://www.djangoproject.com/download/

**[2]** Différences entre python2 et python3 : https://wiki.python.org/moin/Python2orPython3

**[3]** L'origine du MVC : https://fr.wikipedia.org/wiki/Modèle-vue-contrôleur

**[4]** Le langage de templates : https://docs.djangoproject.com/fr/1.11/topics/templates/

**[5]** Description complète de l'authentification dans Django : https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication

**[6]** Une explication rapide pour étendre le modèle d'authentification : https://simpleisbetterthancomplex.com/tutorial/2016/07/22/how-to-extend-django-user-model.html

**[7]** Code et explications prêts à l'emploi pour les templates de gestion de l'authentification : http://blog.narenarya.in/right-way-django-authentication.html , https://www.codementor.io/jadianes/build-data-product-django-user-management-du107uyuh

**[8]** Détail des instances et des options disponibles dans la classe models : https://docs.djangoproject.com/fr/1.11/ref/models/fields/

**[9]** Introduction au déploiement d'une application de bioinformatique propulsée par Django : http://www.steletch.org/spip.php?article91

# CHEZ LES BARBUS – JAVA & SÉCURITÉ : AUTHENTIFICATION À DEUX ÉTAPES

**ROMAIN PELISSE**
[Sustain Developer @Red Hat]

**FRANÇOIS LE DROFF**
[Achitecte @Adobe]

**MOTS-CLÉS : JAVA, SÉCURITÉ, SAML, OAUTH, 2FA, CLOUD, INTÉGRATION CONTINUE**

**« Chez les Barbus - Java & Sécurité », c'est le titre de la conférence donnée par François Le Droff et Romain Pelisse à l'occasion de Devoxx France 2015. Cet article propose d'en reprendre le contenu de manière plus didactique et plus adaptée à ce nouveau support. Si la première partie a été publiée dans le GNU/Linux Magazine [1], voici maintenant sa seconde partie, qui discute des vertus de l'authentification à deux étapes, mais aussi des dangers liés à l'intégration continue sans politique de sécurité, et aussi au « Cloud ».**

Dans le premier article, François et Romain ont longuement discuté des problématiques de sécurité des applicatifs (**Java/JEE** ou autres), évoquant la méthode du « threat modeling » proposée par **Microsoft** et permettant d'évaluer sa sécurité applicative, et même d'identifier les zones à risque.

L'article a continué par une brève présentation du cas d'étude, une petite application interne faite par François, mais qui touche à beaucoup de données sensibles. Après avoir évoqué la relative inutilité des « firewalls », et discuté longuement des possibilités beaucoup plus pertinentes des « reverse proxy », la difficile question du chiffrement des données, de bout en bout, a elle aussi été abordée.

Avant de conclure, François et Romain ont aussi abordé la problématique de l'authentification et de sa mise en place au sein de l'applicatif utilisé pour le cas d'étude. C'est par ce point que nous commençons ici : voyons comment assurer une authentification forte – tout en justifiant aussi son absolue nécessité.

# 1. AUTHENTIFICATION À DEUX ÉTAPES (2FA)

**Romain :** Du coup, j'imagine que, une fois arrivé là, tu as demandé à ton fournisseur d'identité (IDP) d'utiliser une authentification forte, une authentification à deux facteurs (souvent nommée 2FA pour « *2 Factors Authentication* »).

> **François :** Oui, en effet. Désormais, l'utilisateur devra non seulement me démontrer qu'il sait quelque chose, mais aussi qu'il a quelque chose que mon IDP pourra valider : l'utilisateur fournira un secret (son mot de passe), mais aussi un jeton (« token »). Pour obtenir ce jeton, il devra utiliser un générateur installé sur un appareil référencé par mon IDP (une clé USB, une **Yubi key** ou une application sur son téléphone).

**Romain :** C'est parfait en terme de sécurité, mais en terme d'expérience utilisateur, c'est assez rébarbatif. À chaque nouvelle session, l'utilisateur devra saisir un mot de passe et en plus fournir un jeton.

> **François :** De manière générale, c'est vrai que l'équilibre entre l'expérience utilisateur, le confort et la sécurité est difficile à trouver...

**Romain :** Et du coup, mettre en place ce genre de mécanisme, pourtant nécessaire, devient difficile à vendre.

> **François :** Pour convaincre tes utilisateurs ou tes sponsors, il suffit d'un peu de pédagogie, raconte quelques histoires. Comme la comique, mais authentique histoire de Paris Hilton : l'actrice/héritière était alors cliente de l'opérateur téléphonique **T-Mobile**. Comme beaucoup de fournisseurs de services en ligne, TMobile.com exigeait alors que ses utilisateurs répondent à une « question secrète » s'ils oublient leur mot de passe. Pour le compte de Hilton, la question secrète était très classique : "Quel est le nom de votre animal de compagnie préféré ?" En fournissant correctement la réponse, tout internaute pourrait donc modifier le mot de passe de Hilton et accéder librement à son compte. Or il suffisait d'une simple requête sur un moteur de recherche pour connaître le nom de ce chien de star...

**Romain :** Et ce qui devait arriver arriva **[2]**...

> **François :** C'est déjà un argument très fort pour le 2FA, et si ça ne suffit pas, fais circuler l'article de Mat Honan **[3]**, journaliste nouvelle technologie chez **Wired** qui s'est fait vandaliser l'ensemble de ses comptes sociaux en ligne et effacer l'ensemble de ses archives digitales.

**Romain :** Oui, après avoir vu cette histoire, j'ai moi-même immédiatement basculé la plupart de mes comptes sur du 2FA, qu'il s'agisse de **Twitter**, **Facebook**, **Google**, **Yahoo**, **GitHub**, **PayPal**, etc.

> **François :** C'est effrayant de découvrir comment ce journaliste a pu voir toute sa vie numérique « détruite » aussi facilement. Il témoigne ainsi dans un second article **[4]** : « Je n'ai pas activé l'authentification à deux facteurs sur Google ou Facebook. Je n'ai jamais mis en place de comptes de messagerie dédiés (et secrets) pour la gestion des mots de passe. Je prends ces mesures maintenant. Mais je sais aussi que, peu importe les mesures de sécurité que je prends, elles peuvent toutes être défectueuses par des facteurs indépendants de ma volonté.»

**Romain :** Note que ceci nous amène à un point important dont nos collègues et nos pairs dans notre industrie ne sont pas forcément conscients : en tant que développeurs, nous formons une cible privilégiée pour les pirates, et il est donc de notre responsabilité de protéger nos secrets.

> **François :** En effet, nous avons nos profils sur **LinkedIn**, où nous nous sommes qualifiés « full stack engineer » ou « expert sécurité ». Et pour peu que l'on travaille chez **BNP** (où 1 ms valent 100 millions d'euros), les accès aux secrets associés à notre emploi forment une cible de choix pour de nombreux pirates mal intentionnés.

**Romain :** Bref, si vous êtes convaincus, allez faire un tour sur le site de https://twofactorauth.org/, vous verrez quels services en ligne supportent le 2FA.

> **François :** Vous y constaterez qu'à l'heure où nous écrivons cet article, certains acteurs majeurs dans nos vies de développeurs (comme **Docker** et **JetBrains**) ne supportent pas encore l'authentification forte. Vous noterez aussi, au passage, que le monde émergeant de l'« Internet of Things » (IoT) et les applicatifs santé font eux aussi globalement fi de la sécurité !

**Romain :** Et de manière surprenante (et peu rassurante) il en est de même pour la plupart des sites de micropaiement. Et en tant que « particulier », si vous n'avez peur de rien, vous pouvez aussi utiliser les services bancaires de **CitiBank** ou de la **Banque Postale** qui ne supportent pas le 2FA. Les sites les plus sécurisés de ce point de vue sont, en fait, les réseaux sociaux ! Ce qui signifie que si on peut accéder à vos données médicales, utiliser vos fonds pour jouer en bourse à votre place ou faire des micropaiements en votre nom, il est beaucoup, beaucoup plus difficile de voler la photo de votre chat sur Facebook !

**François :** Ceci nous ramène directement à la problématique d'expérience utilisateur évoquée plus tôt, on notera que justement, ces sites de réseaux sociaux font tout pour offrir le meilleur confort d'utilisation à leurs utilisateurs. Du coup, ils sont les premiers à chercher (et trouver) un bon équilibre entre l'expérience utilisateur et la sécurité ; et ceci en proposant néanmoins l'authentification à deux facteurs.

**Romain :** Preuve donc qu'on peut parfaitement y arriver. Et l'une des clés de cet équilibre est, par exemple, de permettre aux utilisateurs de définir leurs téléphones et navigateurs habituels comme étant « dignes de confiance », assurant ainsi que la session y soit maintenue, sans demander sans cesse de s'authentifier.

**François :** Figure-toi que c'est ce que nous avons mis en place pour notre application ! Voyons en détail comment nous avons choisi de l'implémenter.

## 2. SAML ET OAUTH

**Romain :** La dernière fois, tu nous avais montré comment tu avais mis en place l'authentification **SAML**. J'imagine qu'il va falloir repartir de là.

**François :** Tout à fait : une fois authentifié (avec SAML donc et en mode 2FA), nous provisionnons un client

**OAuth2** – uniquement valable pour notre utilisateur, et associé à son secret.

**François :** Va alors démarrer une petite « danse » OAuth2 : une série de pas (de redirections) que vont effectuer « client » et « serveur » avec un minimum d'intervention humaine. *In fine*, notre client récupère un jeton (« token ») qui lui permet de s'authentifier, sans saisie de mot de passe et de secret.

**Romain :** OK, super, mais ce jeton, pour des raisons évidentes de sécurité, doit bien avoir une date de péremption, n'est-ce pas ? Et une fois atteinte, le jeton n'est plus valable.

**François :** Oui, bien sûr, mais c'est pour ça qu'on utilise aussi un jeton de rafraîchissement (« refresh token »). Ce jeton ne peut être utilisé qu'une seule fois et est remis avec le jeton d'authentification. Une fois ce dernier périmé, le client peut simplement retourner cet autre jeton pour renouveler son authentification (et récupérer un nouveau jeton de rafraîchissement au passage).

**Romain :** Mais, attends, du coup, si ton client – téléphone ou navigateur, est compromis, que se passe-t-il ?

**François :** Pas de problème, on peut alors révoquer, côté serveur, tous ces jetons, pour être sûr que les appareils ou clients compromis ne puissent pas être utilisés pour attaquer notre système. Mais avant de conclure sur cette partie, il est important de noter que SAML et OAuth2 n'étaient pas les seules solutions à notre disposition.

**Romain :** Oui, parce que même si OAuth2 colle parfaitement à tes besoins, il ne fournit pas de standard pour la signature de la requête.

**François :** C'est une bonne raison pour opter pour l'utilisation de **JWT token** qui supporte cette fonctionnalité.

**Romain :** Oui, mais au fait, pourquoi ne pas simplement utiliser **OAuth1** ? C'est standard, non ? Et, ça comprend un standard de signature de requête...

**François :** Je crois oui, mais je t'avoue que quand j'ai regardé la spécification... Je n'ai rien compris !

**Romain :** OK, ce genre de chose arrive, mais du coup, tu n'as pas jeté un œil au code pour y voir plus clair ?

**François :** Si.

**Romain :** Et alors ?

**François :** Rien compris non plus.

**Romain :** OK, je pense que là, tout est dit sur ce sujet :) !

## 3. EXTENSION DU PÉRIMÈTRE DE LA LUTTE - CI & CLOUD

**Romain :** Jusqu'à maintenant, quand on pense sécurité des applicatifs, on a naturellement tendance à penser aux plateformes de production, et à délaisser, voire à être laxiste avec les environnements de développement et les plateformes d'intégration continue.

**François :** C'est mal.

**Romain :** Et si en plus tu déploies dans le « Cloud », ta chaîne de déploiement continue doit être irréprochable, ou tout un nouvel univers de failles et d'exploitation s'ouvre aux attaquants, dans un domaine où les experts de la sécurité sont très loin d'avoir évangélisé et diffusé les bonnes pratiques associées. Voyons un peu toutes ces problématiques en détail.

# DISPONIBLE DÈS LE 29 SEPTEMBRE

## MISC HORS-SÉRIE N°16 !


MAÎTRISEZ LES OUTILS MATÉRIELS & LOGICIELS POUR LES AUDITER

## NE LE MANQUEZ PAS

### CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

### http://www.ed-diamond.com

## 3.1 Gestion des secrets

**François :** Commençons par regarder notre code, contient-il des secrets ?

**Romain :** C'est un excellent point de départ. Tu as bien raison de vérifier ceci.

**François :** Oui, on y a pensé, en effet, mais est-ce que c'est vraiment important ? À part pour la production bien sûr.

**Romain :** Méfie-toi, car même si les données ne sont pas sensibles, les secrets peuvent quand même donner accès à de nombreux systèmes. Ainsi, nombreux sont les utilisateurs d'**Amazon** qui poussent leur clé sur **GitHub**. Immédiatement repérées, leurs clés peuvent être utilisées par des tiers pour générer des instances (et miner du bitcoin par exemple), et avec une facture de plusieurs milliers de dollars **[5]** !

**François :** Clairement il faut faire attention à ne pas exposer ses secrets, mais il faut aussi pouvoir gérer leur cycle de vie, assurer qu'ils soient proprement déployés, mais aussi accessibles aux personnes autorisées.

**Romain :** Et tout ceci n'est pas franchement simple, surtout avec l'exigence d'aujourd'hui en terme d'automatisation des déploiements d'applicatifs ou de systèmes. Une solution, François ?

**François :** Dans le cadre de notre cas d'étude, j'avais utilisé **Chef** pour l'automatisation de mon infrastructure. Chef est un *framework* open source écrit en **Ruby**. Il permet de déployer facilement des serveurs et des applications à grande échelle sur des infrastructures d'entreprise. Il existe un bel écosystème de recettes et bibliothèques autour de cette solution, l'une d'elles s'appelle **Chef-vault**.

**Romain :** Chef-vault ? C'est intéressant, dis-m'en un peu plus.

**François :** Chef-vault **[6]** est un outil de gestion des secrets open source et communautaire. Il s'appuie sur le système de chiffrement du *framework* Chef, mais le pousse plus loin : il permet d'éviter l'utilisation d'une clé partagée par tous les utilisateurs et les infrastructures **[7]**.

**Romain :** Et si j'ai bien compris, Chef-vault s'appuie sur une liste blanche (« white list ») ? Mais comment tu montes en charge avec un mécanisme pareil ?

**François :** Bien vu, Chef-vault est assez bien limité lorsqu'il s'agit d'*auto-scaling* ou d'auto-guérison. Noah Kantrowitz le résume bien dans son blog **[8]** : il y propose d'ailleurs quelques alternatives. Mais cette limitation de Chef n'en était pas une pour moi, puisque mon application interne avait une charge déterministe.

**Romain :** Rappelons enfin également que : que nous utilisions Chef et Chef-vault ou pas, il faudra bien sécuriser notre serveur de secrets.

**François :** Comme tous les autres éléments de la machinerie qui construisent et déploient les applicatifs qui tournent sur nos serveurs.

## 3.2 Jenkins

**François :** Donc là, on est bon sur les secrets applicatifs, mais *quid* de la sécurité du serveur d'intégration continue ?

**Romain :** C'est aussi une source de beaucoup de failles, car peu de gens se soucient de ces serveurs qui pourtant offrent d'utiles, mais aussi très dangereuses fonctionnalités.

**François :** Par exemple, sur **Jenkins**, tu peux créer des tâches qui exécutent des scripts Shell, directement sur le système, ou sur un nœud esclave !

**Romain :** Et comme souvent, on ne prend pas soin de mettre en place une authentification sur le serveur, on ne sait même pas qui a créé le script, ni comment !

**François :** Il est donc crucial de s'assurer que le serveur suive les bonnes pratiques de sécurité, car il peut non seulement être détourné, mais aussi offrir un accès à beaucoup de points sensibles de notre application.

**Romain :** Comme son code source, ou même simplement les secrets que nous venons d'évoquer. Si on a accès au serveur d'intégration continue, on a accès à tout ça ! Il faut donc mettre en place, comme l'avons déjà évoqué **[1]** une authentification forte, un chiffrement des flux de données, et une journalisation appropriée des événements pour permettre un audit de sécurité.

**François :** Même constat pour nos serveurs d'artefacts.

**Romain :** Justement, parlons un peu d'eux.

## 3.3 Nexus, Satellite et les gestionnaires d'artefacts

**François :** Depuis nos serveurs d'artefacts, nous collectons des bibliothèques/gem Ruby, des fichiers **JavaScript**, des paquets/npm **Node.js**, des archives/jar Java, des paquets/rpms **Linux** et d'autres archives propriétaires : la collection complète des binaires et fichiers qui seront déployés sur nos serveurs. Chacun de ces binaires, chacune de ces archives, chacun de ces

builds peuvent être un vecteur de « malwares » et autre cheval de Troie. Leurs intégrités doivent être contrôlées de façon continue.

**Romain :** Une solution est de mettre en place des serveurs mandataires (audités et sécurisés), des « proxies » tels que **Nexus** ou **Artifactory**, pour les dépendances Java. Node ou Ruby, ou même **Red Hat Satellite**, pour les dépendances systèmes (paquets RPMs). Les équipes de production et de sécurité peuvent ensuite vérifier les binaires utilisés dans l'assemblage des logiciels et ceci sans interférer avec les développements.

**François :** Comment peuvent-ils vérifier qu'un fichier binaire n'a pas été corrompu ?

**Romain :** En vérifiant que les signatures correspondent à celles publiées. C'est fastidieux à faire à la main, mais ce genre d'outil s'en charge automatiquement et rapporte les éventuelles infractions détectées.

## 3.4 Le grand méchant Cloud

**Romain :** Ton application finie et maintenant à la pointe de la sécurité, j'imagine que tu vas maintenant « voir si tu n'es pas un homme du XXIème siècle », comme dirait Audiard, et la déployer sur le « Cloud », non ?

**François :** Avec mes problématiques de sécurité, je ne sais pas si c'est une bonne idée.

**Romain :** Non, au contraire ! L'architecture d'un « Cloud » (comme Amazon, **Azure**, **CleverCloud**, etc.) te force en effet à réfléchir à la sécurité dans ce contexte, mais t'apporte aussi beaucoup dans ce domaine ! Par exemple, combien de systèmes, pourtant bien sécurisés lors de la mise en place, ne sont par la suite plus maintenus ni mis à jour ? Combien d'applications « oubliées » sont ainsi exposées au monde extérieur et offrent des failles connues pour le pirate informatique ?

**François :** Chez nous ? Aucune bien sûr !

**Romain :** Bien sûr ! Quoi qu'il en soit, sur le « Cloud », les systèmes sont souvent automatiquement rafraîchis, « patchés », mis à jour. Et si on fait bien les choses, les machines virtuelles ne sont utilisées que quelques jours, voire quelques heures, avant d'être décommissionnées et reconstruites avec une version mise à jour du système contenant ainsi tous les éventuels correctifs de sécurité publiés depuis la précédente version. Comme le fait **Netflix** !

**François :** C'est tout l'intérêt de l'approche décrite comme « pet versus cattle » (animaux domestiques versus troupeaux), qui peut se résumer ainsi : « Ne traitez plus vos serveurs à la main, cessez de passer votre temps à les cajoler, traitez-les plutôt comme du bétail : au moindre signe de fatigue ou de maladie, tuez-les et remplacez-les. » En effet, il faut une famille entière pour éduquer un animal domestique et juste quelques cowboys pour gérer un troupeau (cette métaphore ne plaira pas aux végétariens et autres vegans, désolé).

**Romain :** Tu peux aller encore plus loin. Je viens de mentionner Netflix ; chez Netflix ils ne laissent jamais tourner leurs instances, sur Amazon, pas plus de quelques jours. Si tu as déjà travaillé avec l'infrastructure d'Amazon, tu sais que : bien que le service soit d'excellente qualité, il n'est pas fiable à 100%. De manière générale, on n'est jamais à l'abri d'une défaillance de logiciel même s'il s'agit de son propre logiciel.

**François :** Surtout que pour gérer les défaillances, il faut s'y prendre en amont, dans le code. Être sûr d'avoir géré toutes les pannes et avaries possibles.

**Romain :** Et c'est pour ça que Netflix a développé un outil nommé **Chaos Monkey [9]**, qui sert à tester que le système produit a bien géré tous les cas possibles, en... causant des avaries !

**François :** C'est super pour les phases de qualification !

**Romain :** Mais, dans le cas de Netflix, il le laisse même s'exécuter en production !

**François :** Impressionnant en effet... Quand j'y pense, en fait, il serait génial d'avoir exactement la même chose pour la sécurité. Une sorte de « hacky monkey » qui forcerait les développeurs à penser, dès le début, à la sécurité.

## 3.5 Jouer au pompier

**Romain :** Penser à la sécurité dès le début c'est important, mais ce n'est pas tout. En fait, aujourd'hui, quand on voit les attaques, de type déni de services ou autres, ou encore le vol de données, on réalise qu'il est tout aussi important de savoir comment réagir.

**François :** C'est vrai. La question n'est plus de savoir si on sera attaqué, mais comment le détecter, réagir et assurer que si l'attaque réussit, le système ou les données ne peuvent pas être compromis outre mesure.

**Romain :** Et c'est possible ! Regarde comment GitHub a superbement survécu à plusieurs attaques de type déni de service distribué (DDoS), tout en communiquant dessus, sans complexe.

**François :** À l'inverse, le cas de la base de données piratée de (site de rencontres adultères), dont seuls les mots de passe étaient chiffrés, (comme nous l'avons évoqué

lors du premier article) est un parfait contre-exemple. Ils auraient mieux fait, avec le recul, de chiffrer les données de l'utilisateur, pas seulement son mot de passe.

**Romain :** On peut même aller plus loin dans ce domaine, par exemple en utilisant un module matériel de sécurité, un HSM (« *Hardware Security Module* » **[10]**).

**François :** En fait, en conclusion, et pour faire une métaphore facile à retenir, quand on pense à la sécurité, il faut penser comme un pompier. Les pompiers ne font pas que faire briller leur beau camion, ils se préparent aussi à affronter et arrêter des incendies, dans notre cas, il faut donc être prêt à faire face à une attaque.

**Romain :** Nous donnerons toutes leurs chances aux pompiers, en équipant notre système de détecteurs de fumée (comme des systèmes de détection d'intrusion), mais aussi de portes coupe-feu (avec des outils comme **SELinux** ou le gestionnaire de sécurité de la JVM (« *security manager* »), pour s'assurer que l'incendie ne pourra pas se propager...

**François :** Comment marchent ces derniers ?

**Romain :** Les deux outils suivent le même principe. Ils vérifient que les activités du système, dans le cas de SELinux, et de la machine virtuelle Java, dans le cas du « security manager », soient conforme aux attentes. C'est-à-dire conformes à une politique qui définit quelles utilisations des ressources peuvent être faites par tel ou tel processus ou programme. Ainsi, si ton serveur HTTP – **Nginx** ou **Apache** – se met soudainement à vouloir écrire dans le fichier **`/etc/password`**, SELinux va lui interdire.

**François :** Et pareil avec Java, j'imagine ? Si mon applicatif « web » veut soudainement ouvrir une connexion directe vers mon serveur LDAP ou ma base de données, le gestionnaire de sécurité de la JVM le bloque ?

**Romain :** Exactement. Du coup, le pirate a certes compromis ton système, mais sa marge de manœuvre est très réduite.

**François :** Oui, mais si l'exploit se base sur une « activité » normale de l'applicatif. Par exemple, si l'on modifie le contenu d'un fichier de données légitime de l'application, là, on n'est plus protégé.

**Romain :** Non, en effet, et c'est là encore qu'il est essentiel de penser à la sécurité en développant. Dans ce cas précis, il est nécessaire de développer un mécanisme de sécurité supplémentaire.

**François :** Par exemple, faire un contrôle de cohérence (« checksum ») sur le fichier ?

**Romain :** Bonne idée, en effet ! Je te félicite, après être devenu un vrai « DevOp », tu viens de devenir un « DevSec » !

# CONCLUSION

Nous avons démontré dans cet article l'absolue nécessité de l'utilisation systématique d'une authentification forte et sa relative facilité de mise en place. Ne vous en privez pas ! Votre compagnie, et peut-être vous-même, pourriez être amenés à le regretter.

Nous avons vu ensuite comment le périmètre de la lutte en sécurité s'est largement élargi avec l'arrivée de l'intégration continue et du « cloud ». Il ne suffit plus aujourd'hui d'assurer la sécurité du seul applicatif en production, mais aussi de s'assurer que toute la chaîne de production liée à ce dernier n'est pas compromise, tout autant que sa plateforme d'exécution...

Bref, comme nous l'avons évoqué tout au long de ces deux articles, il est essentiel de penser à la sécurité, de la conception à la mise en production de l'applicatif, mais aussi au-delà car, rappelez-vous, un bon pompier s'entraîne aussi à affronter des incendies... ■

# RÉFÉRENCES


**[1]** PELISSE R. et LE DROFF F., « *Chez les Barbus – Java & Sécurité* », *GNU/Linux Magazine n°196*, septembre 2016, p.58 : http://www.gnulinuxmag.com/creez-votre-premiere-intelligence-artificielle/

**[2]** How Paris (Hilton) Got Hacked ? http://archive.oreilly.com/pub/a/mac/2005/01/01/paris.html

**[3]** Mat Honan Hacked (1) : https://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/

**[4]** Mat Honan Hacked (2) : https://www.wired.com/2012/08/mat-honan-data-recovery/

**[5]** My $500 Cloud Security Screwup : https://securosis.com/blog/my-500-cloud-security-screwup

**[6]** Chef Vault : https://github.com/chef/chef-vault/

**[7]** What can Chef Vault do for you : https://blog.chef.io/2016/01/21/chef-vault-what-is-it-and-what-can-it-do-for-you/

**[8]** Chef et la gestion des secrets : https://coderanger.net/chef-secrets

**[9]** Le « Chaos Monkey » de Netflix : https://medium.com/netflix-techblog

**[10]** Hardware Security Module (HSM) : https://en.wikipedia.org/wiki/Hardware_security_module

# SERVEURS DÉDIÉS Synology®

AVEC ikoula HÉBERGEUR CLOUD

Votre serveur dédié de stockage (NAS) hébergé dans nos Data Centers français.

POUR LES LECTEURS DE **LINUX MAG***

**OFFRE SPÉCIALE -60 %**

À PARTIR DE

**5,99 €** HT/MOIS

~~14,99 €~~

CODE PROMO **SYLIM17**


- Bande passante **100 Mbit/s**
- Station de **surveillance**
- Support technique **en 24/7**
- Trafic réseau **illimité**
- Système d'exploitation **DSM 6.0**
- Hébergement dans **nos Data Centers**

*Offre spéciale -60 % valable sur la première période de souscription avec un engagement de 1 ou 3 mois. Offre valable jusqu'au 31 décembre 2017 23h59 pour une seule personne physique ou morale, et non cumulable avec d'autres remises. Prix TTC 7,19 €. Par défaut les prix TTC affichés incluent la TVA française en vigueur.

**CHOISISSEZ VOTRE NAS**

https://express.ikoula.com/promosyno-lim

ikoula HÉBERGEUR CLOUD — /ikoula — @ikoula — sales@ikoula.com — 01 84 01 02 50

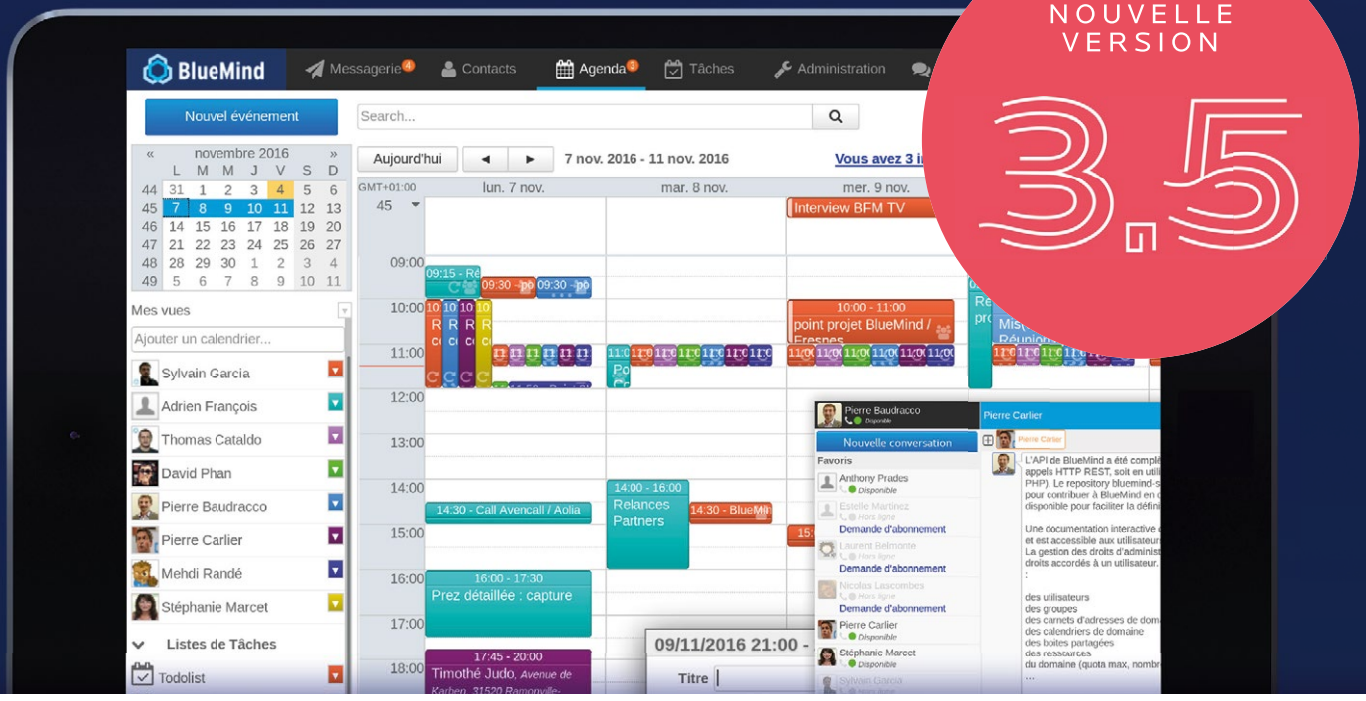
NOM DE DOMAINE | HÉBERGEMENT WEB | SERVEUR VPS | SERVEUR DÉDIÉ | CLOUD PUBLIC | MESSAGERIE | STOCKAGE | CERTIFICATS SSL

# BlueMind

SOLUTION OPENSOURCE
PROFESSIONNELLE DE MESSAGERIE
COLLABORATIVE

# LIBÉREZ VOTRE MESSAGERIE

EMAILS
AGENDAS
CONTACTS
TÂCHES
CHAT
MOBILES
DOCUMENTS & PIÈCES JOINTES
ENVOI FICHIERS VOLUMINEUX
...
...




NOUVELLE VERSION
3.5

*FRANCAIS / NOMBREUSES RÉFÉRENCES / ERGONOMIQUE / ÉVOLUTIF / ÉCONOMIQUE*

Découvrez l'écosystème BlueMind et toutes les fonctionnalités sur
**www.bluemind.net**