Compatible One: Livrable 28.1
Spécifications du service de stockage et de caching
Version 0.2

Stefane Fermigier (Nuxeo), Fabio Mancinelli (XWiki)

April 21, 2011

Abstract

Ce document présente les spécifications du service de stockage et de caching.

Contents

1 Besoins

L'objectif de ce service est de fournir trois types de stockage et un service de caching, accessibles sous forme d'API depuis le ou les runtime(s) de la plate-forme, et sous forme de service web ou d'un protocole ad-hoc (memcached, thrift, ProtocolBuffer) :

- Stockage relationnel (inspiré par RDS d'Amazon), acessible depuis une API native (ex: MySQL), ou par du REST.
- Stockage de blobs (inspiré par S3 d'Amazon), i.e. espace de stockage d'objets binaires accessibles depuis une API REST.
- Stockage post-relationnel (i.e NoSQL), sous une ou plusieurs formes à déterminer (type SimpleDB d'Amazon, BigTable de Google) comme CouchDB, MongoDB, Cassandra, etc.
- Composant de caching distribué pour permettre aux applications de pouvoir mémoriser de façon temporaire les résultats des computations afin d' augmenter les performances. Le composant de caching utilisera une API standard comme l'API JCache, utilisée aussi par Google AppEngine, et sera exposé aussi au travers d'une interface REST.

Il est à noter que loin d'être exclusifs, ces services sont la plupart du temps complémentaires ("NoSQL" signifiant dans ce contexte "Not Only SQL") et utilisés conjointements par les sites les plus exigeants du moment (Facebok, Twitter, etc.).

Le travail prévu dans cette sous-tache concernera plutôt l'intégration de solutions de caching existants (comme memcached) et du développement pour les intégrer dans la plate-forme Compatible One.

Chacun de ces stockages devra être autant que possible scalable et distribué, si possible sans nécessité une implication forte des développeurs (séparation des responsabilités).

Les compromis à effectuer et les choix des technologies afférentes, seront réalisés en fonction d'une collecte initiale de besoins auprès des membres du consortium, particulièrement ceux impliqués dans le SP4.

2 Architecture générale

NB: Cette partie devrait probablement être bougée dans un document commun aux SP du SP2.

Pour bien comprendre les enjeux de ce SP, comme d'ailleurs pour la plupart des autres liés au PaaS, il convient de bien distinguer des considérations qui devraient en principes être orthogonales:

- *La gestion des services*: le fait, pour un utilisateur du PaaS, de provisionners un service donné, selon des caractéristiques (resources allouées, SLAs, etc.) qu'il choisit en fonction des besoins de son application; de supprimer ce services (ou de s'en retirer la possibilité de l'utiliser à l'avenir); d'en modifier les caractéristiques à la volée, si cela est possible.

- *L'utilisation des services*: le fait pour une application d'accéder à un ou des services PaaS: CRUD sur un service de persistence, envoi de messages sur un bus de messages, etc.

Ces deux considérations sont liées à deux rôles différents (qui peuvent être remplis par deux personnes distinctes ou la même personne):

- *Le développeur* choisit les différents types de service qui convient à l'application qu'il est en train de développer (ex: une base MySQL, une base Redis, une base Neo4j, un stockage de blobs et un cache distribué). Grâce à l'outil en ligne de commande de Compatible One, il peut facilement déployer ces services sur son portable, ou à défaut sur un petit serveur dans son bureau ou dans le cloud, ou encore sur un serveur de qualification ou de préproduction dans le cloud.

- *L'utilisateur* (aussi appelé *administrateur* ou *devops*[1]) du PaaS qui provisionne les même services pour une utilisation en production dans le cloud, avec le niveau de performance, de redondance, et d'autoscalabilité (lorsque cela est possible)

Dans le même ordre d'idée, il faut distinguer:

- *Les protocoles* d'accès aux services: à l'exception du runtime qui a un statut à part, les services du PaaS sont accessible selon un protocole client-serveur qui leur est propre, et qu peut être implémenté, côté client, dans n'importe quel langage de programmation.

- *Les APIs* d'accès aux services: selon le langage utilisé (ex: Java, Python...), une ou plusieurs API privilégiées peuvent être fournies aux développeurs, notamment dans le cadre du runtime PaaS, sans que cela constitue une obligation de les utiliser.

Détaillons à présent un peu plus ces différents point.

[1] `http://en.wikipedia.org/wiki/DevOps`

2.1 La gestion des services

La proposition de valeur du cloud pour les développeurs, c'est, en théorie, de leur permettre de s'abstraire totalement de la gestion des services nécessaires au fonctionnement de leurs applications (SGBD, bus de messages, cache, etc.).

L'expérience (car c'est bien ainsi qu'il faut l'appeler) Google AppEngine montre qu'un PaaS entièrement fondé sur ces principes impose des restrictions considérables aux développeurs — pas de thread, temps que réponse limité pour chaque requète, modèle de persistence éloigné du modèle relationnel, pas d'accès au filesystème, etc. — qui rendent l'expérience très contraignante, et pour tout dire insatisfaisante pour la plupart des développeurs.

Force est donc de constater que le terme "PaaS" englobe deux types de services:

- Des services véritablement "internet scale", élastiques, où les développeurs peuvent puiser dans un pool de resources virtuellement infini et multitenant, et sont facturés en fonction uniquement de leur usage effectif de ces resources.
  Des exemples de tels services sont Amazon S3, Google BigTable et les autres services accessible à travers le runtime de Google AppEngine (e.g., Memcache, Blob store, etc.).

- Des services à la scalabilité plus limitée, qui impliquent lors de leur instanciation la réservation de resources dédiés (VM, stockage, etc.).
  Des exemples de tels services sont Amazon RDS ou EC2.

En pratique, on ne peut pas, pour des raisons fondamentales, pour un certain nombre de services, faire une abstraction totale de la façon dont ceux-ci sont configurés et provisionnés. Il apparaît donc pertinent de proposer à l'utilisateur du PaaS la possibilité, via une API simple, de provisionner de tels services, selon les caractéristiques qui lui conviennent.

Ce provisionnement devra néanmoins s'abstraire suffisamment des détails techniques de bas niveau nécessaires à la mise en oeuvre de services correspondants. Par exemple, dans le cas d'un service de stockage relationnel, l'API devra permettre de gérer les paramètres comme la taille de la base, mais sera opaque par rapport à l'emplacement physique des fichiers ou la configuration des utilisateurs.

2.2 API pour la gestion des services

2.2.1 État de l'art

Sun Cloud API: `http://kenai.com/projects/suncloudapis/pages/HelloCloud`.

Rackspace: http://www.rackspace.com/cloud/cloud_hosting_products/servers/api/, comparaison avec AWS: http://www.rackspace.com/cloud/blog/2010/06/15/a-close-look-at-the-rackspace-cloud-servers-api-and-how-it-compares-

DMTF: http://dmtf.org/standards/cloud

DeltaCloud: http://incubator.apache.org/deltacloud/api.html

Red Hat Entreprise VM API: http://markmc.fedorapeople.org/rhevm-api/en-US/html/

WebSphere: http://publib.boulder.ibm.com/infocenter/wscloudb/v1r1/index.jsp?topic=/com.ibm.websphere.cloudburst.doc/ra/rer_clouds.html

OCCI: http://ns.ggf.org/Public_Comment_Docs/Documents/2011-01/draft-occi_http_rendering.pdf

Commentaires: http://stage.vambenepe.com/archives/863, http://stage.vambenepe.com/archives/894, http://stage.vambenepe.com/archives/1161.

2.2.2 Proposition

- Compatible One proposera une API REST de gestion des services.
- Cette API sera générique: elle permettra de provisionner aussi bien des ressources IaaS (VM, stockage, réseau) que des services de niveau PaaS (SGBD, bus de message, container d'applications, services OSGi dans un container OSGi, services WSGI dans un serveur WSGI, jobs MapReduce, etc.).
- Les services sont provisionnés à l'aide d'un POST sur une URL donnée pour un client donnée (ex: http://api.compatibleone.com/nuxeo), de documents JSON qui contiennent les caractéristiques souhaitées du service PaaS: type de service, nombre de serveurs, mémoire, disque, redondance, etc.). Le POST redirige sur une URL qui représente ensuite le service en tant que resource.
- Lorsque l'on fait un GET sur l'URL d'un service, on récupère un document JSON qui représente le service: son état (démarré, en cours de démarrage, arrêté, en cours d'arrêt, en erreur, etc.), ses caractéristiques, des informations de monitoring et de metering, l'URL d'accès au service (e.g., http://api.compatibleone.com/nuxeo/services/rdb/mydb1). Le format et la structure des documents JSON dépendra du service en question et fournira la possibilité de découvrir les paramètres de configuration disponibles afin de pouvoir les modifier successivement. Ceci implique que ces documents devront fournir un descriptif des paramètres (embedded ou dans un document à part) pour que les clients puissent construire des interfaces utilisateurs de façon automatique et dynamique.

- On peut modifier un service (typiquement, le démarrer / l'arrêter) à l'aide d'un PUT sur la même URL. Cette modification sera effectuée en utilisant également des documents de type JSON ayant la même structure de ceux récupérés précédemment avec des GETs sur l'URL des services. Le descriptif des paramètres guidera le client dans la construction de documents bien formés (i.e., qui contiennent les paramètres nécessaires avec des valeurs correctes)
- Pour détruire un service, il suffit d'un DELETE sur la resource.
- Il est possible de lister les services actifs à l'aide d'un GET sur l'URL d'entrée (e.g., http://api.compatibleone.com/nuxeo/services/running).
- Il est aussi possible d'avoir la liste de tous les services disponibles à l'aide d'une autre URL (e.g., http://api.compatibleone.com/nuxeo/services).
- Pour accéder à un service, on utilise une URL retournée contenue dans le document retourne par un GET sur la resource qui représente le service (cf. infra).

TODO: Lien avec les API existantes? Couche de compatibilité AWS / autres?

API client Comme l'API proposée est basée sur HTTP, un simple client capable de communiquer en utilisant ce protocole peut être utilisé pour interagir avec les services du PaaS. Ce type de clients sont disponibles dans presque la totalité des langages, et aussi en ligne de commande.

Cependant, afin de faciliter l'interaction entre les clients et les services du PaaS, il serait souhaitable de leur fournir des composants qui encapsulent le dialogue avec les API REST décrites dans la section précédente. Ces composants devront être disponible dans plusieurs langages.

En considérant que les use case prévus sont centrés surtout sur la plateforme Java, l'objectif c'est de fournir au moins une API client Java.

Il est interessant de noter qu'ils existent déjà des projets qui fournissent une API abstraite pour l'allocation et l'utilisation de services PaaS et des instantiation de cette API vers des fournisseur de service IaaS et PaaS. Un de ces projet particulièrement actif est JCloud.

L'idée serait, donc, de concevoir l'API client comme une extension de l'API JCloud:

- Comme JCloud supporte seulement de composants de type Compute et Blobstore, des extension à l'API abstraite de JCloud seraient envisagées pour supporter des services comme le stockage relationnel.
- Une extension aux implémentations déjà disponibles pour pouvoir effectivement utiliser les services mis à disposition par la couche PaaS de Compatible One.

Ces extensions seront proposée comme contributions au projet JCloud afin de faire avancer une API prometteuse afin qu'elle aie plus de chances de s'imposer comme standard de facto.

Le choix d'utiliser des API REST pour exposer les services PaaS permets aussi de fournir un client à ligne de commande. Ceci est l'approche suivi par Cloud Foundry qui expose ses fonctionnalités PaaS à travers un tel client.

TODO: exemples.

2.3 L'utilisation des services

Une fois les services provisionnés, pour utiliser un des services proposés par le PaaS, le développeur peut, grâce à un simple appel de méthode, obtenir une clef d'accès au service qu'il souhaite (typiquement une URL), et utiliser ensuite cette clef pour se connecter au service.

Exemples:

- URL pour un stockage S3: http://s3.compatibleone.com/
- URL pour un SGBDR: mysql://server/database
- URL pour un cache memcached: memcached://server/...

2.4 Challenges / questions ouvertes

- Sécurité. Cf. par exemple `http://www.thebuzzmedia.com/designing-a-secure-rest-api-wi`
- Modèle REST suffisamment générique pour se mapper sur les différents services de cloud existants.
- Lien aves les standards (ex: OCCI, DMTF Cloud).
- Adresser plusieurs cas d'usage, depuis l'accès à des services hautement standardisés jusqu'à des services custom.
- Séparation des rôles.
- Ligne de commande vs. API.

3 Stockage relationnel

Le stockage relationnel est le fondement la majorité des applications d'entreprises depuis une trentaine d'années, et plus récemment, depuis les années 2000 et l'avênement des architecture LAMP (Linux + Apache + MySQL + Perl/Python/PHP) ainsi que de ses variantes (e.g. Mysql + Ruby on Rails) pour les applications web.

L'augmentation du traffic des sites web depuis 10 ans, mais aussi l'apparition de nouveaux modes d'interaction plus participatifs ("Web 2.0") ont entraîné

une explosion des besoins sur les SGBDR utilisés dans ce contexte, avec pour conséquence:

- Le développement de mécanismes permettant d'augmenter le point à partir duquel les SGBDR ne peuvent plus suivre (ex: réplication master-slave et master-master) ou de techniques de programmation qui permettent de contourner ces contraintes (ex: décomposition fonctionnelle, sharding).
- Le constat que le modèle relationnel ne permet pas une scalabilité infinie, et qu'il n'est pas non plus adapté à tous les besoins actuels[2], et l'apparition subséquente de nouveaux modèles, popularisés à présent sous le nom "NoSQL".

3.1 Modèle

Le modèle relationnel est un modèle bien ancré dans un formalisme mathématique [Codd...] et dans des specifications établies de longue date [SQL 98, etc.].

Néanmoins force est de constater que deux SGBDR ne sont jamais interchangeables, et que certains systèmes sont plus adaptés à certains usages, et d'autres à d'autres. Par exemple, Nuxeo préfère PostgreSQL alors que ERP5 et XWiki semblent préférer MySQL.

Le PaaS Compatible One ne cherchera donc pas à fournir une abstraction du modèle relationnel et du langage SQL, commune aux différentes implémentations, mais offrira un accès à ces SGBD selon leur protocole natif et leur dialect SQL propre.

3.2 Implémentations

Deux SBGDR open source sont considérés comme les leaders: MySQL[3] et Postgresql.

Compte-tenu de ceci, et du fait que les différents SGBDR open source (ou non) du marché présentent des caractéristiques qui les rendent plus ou moins compétitifs selon les applications[4], il ne paraît pas pertinent d'en choisir un plutôt qu'un autre, mais il convient plutôt à notre sens de donner la possibilité d'utiliser l'un ou l'autre, et plus généralement aux fournisseurs ou au gestionnaire de cloud d'instancier facilement d'autres SGBDR en utilisant un modèle et des API de provisionning similaires.

[2]Michael Stonebraker, *The End of a DBMS Era (Might be Upon Us)*, `http://cacm.acm.org/blogs/blog-cacm/32212-the-end-of-a-dbms-era-might-be-upon-us/fulltext`

[3]La situation est rendue un peu plus compliquée suite au rachat de Sun par Oracle et aux nombreux forks qui ont suivi: Drizzle, Skysql, MariaDB, etc.

[4]Par exemple, le cas d'utilisation SP4.1 (Nuxeo) exprime une préférence nette pour Postgresql, alors que le cas SP4.2 (XWiki) semble plutôt privilégier MySQL.

3.3 Protocoles

Chaque SGBDR implémente en général son propre protocole, en général binaire pour des raisons d'optimisation.

Il serait illusoire pour le projet Compatible One de proposer un protocole unique indépendant du SGBDR utilisé, d'autant que ça ne résoudrait pas la question des spécificités sous-jacentes.

Il peut cependant être intéressant d'explorer la possibilité d'encapsuler l'accès à ces SGBD dans des appels JSON en HTTP, comme suggéré dans *HTTP JSON AlsoSQL interface to Drizzle*[5].

3.4 API d'utilisation

Paradoxalement, alors que les protocoles et les dialectes SQL peuvent varier d'un SGBD à l'autre, il n'est pas déraisonnable d'utiliser une API

Deux exemples en Java:

- Le standard JPA (implémenté par exemple par Hibernate)
- DataNucleus, qui est mis en avant par Google dans GAE.

En Python, il existe plusieurs ORM, le plus abouti semble être à ce jour SQLAlchemy, et le plus populaire celui de Django.

3.5 API de gestion

Cf. proposition générique + détailler les paramêtres spécifiques à chaque SGBD.

3.6 Scalabilité

TODO: auto-scaling, sharding, réplication, etc.

3.7 Facturation

TODO.

4 Stockage de blobs

Amazon S3 a été, avec EC2, le premier service cloud lancé par Amazon en mars 2006. Ce service

4.1 Modèle et protocole d'accès

TODO: Description succinte du protocole S3.

Cf. `http://en.wikipedia.org/wiki/Amazon_S3`

[5] `http://www.flamingspork.com/blog/2011/04/21/http-json-alsosql-interface-to-drizzle/`.

4.2 Implémentations

TODO: lister les services équivalent (cf. jclouds.org).

En choisir une implémentation. Par exemple Swift d'OpenStack ?

4.3 API d'utilisation

Il existe une librairie qui fournit une couche d'abstraction au dessus d'un stockage de blobs type S3: jclouds[6]

4.4 API de gestion

S3 étant "infiniement scalable", l'utilisateur n'a pas besoin de provisionner son propre service S3 (même si cela reste une possibilité).

Il lui suffit de se connecter au service S3 proposé par le PaaS, en utilisant comme point d'accès une URL codée en dur.

4.5 Facturation

Amazon facture son service S3 d'une part en fonction du volume stocké, d'autre part en fonction du volume transféré.

5 Stockage post-relationnel (NoSQL)

Les limitations, pour le stockage de volume de données massifs "internet scale", du modèle relationnel ont amené depuis 5 ans un certain nombre de développeurs et de startups à créer de nouveau modèles qui présentent des caractéristiques de scalabilité différentes, ou une richesse fonctionnelle plus grande.

Il s'agit d'un écosystème très riche et en évolution permanente (cf. `http://www.mynosql.com` pour un fil d'actualité sur le sujet et `http://nosql-database.org/` pour une liste extrêmement complète des différents systèmes disponibles aujourd'hui).

TODO: parler du théorème CAP.

5.1 Modèles

On peut distinguer dans l'écosystème NoSQL quatre grands types de bases:

- Les bases clefs-valeurs. Ex: Tokyo Cabinet, etc.
- Les bases orientées colonnes. Ex: HBase, Cassandra.
- Les bases "orientés documents". Ex: CouchDB, MongoDB, etc.
- Les bases orientées graphes. Ex: Neo4j.

[6] `http://jclouds.org`

5.2 Implémentations

Encore plus que pour les SGBDR, les SGBD NoSQL présentent des caractéristiques fonctionnelles et techniques extrêmement différentes les un des autres. Il est donc illusoire de proposer un "service générique NoSQL". C'est au développeur de choisir le service qu'il veut, et à l'utlisateur de pro

5.3 API de gestion

L'API REST de gestion est similaire à ce qui a été décrit précédemment.

Il faut noter une différence cependant: certaines SGBD NoSQL permettant l'autoscalabilité par simple ajout de serveurs, il doit être possible de passer en paramêtre (dans le document JSON décrivant le service) des options relatives à ces considérations, par exemple:

- ajoute un nouveau serveur automatiquement si la charge, ou la taille des données gérées, dépasse un certain seuil,
- passe sur un serveur avec plus de RAM si l'emprunte mémoire du serveur dépasse un certain seuil,
- etc.

Il est à noter que ces options sont extrêmement dépendantes du service requis.

TODO: exemples.

5.4 API d'usage

En Java, il est intéressant de constater que l'API DataNucleus permet d'adresser aussi bien le stockage SQL qu'un certain nombre de SGBD NoSQL: actuellement, GoogleStorage, HBase, MongoDB + extensibilité apr plugins OSQGi.

5.5 Références

MongoDB on EC2, `http://nosql.mypopescu.com/post/5010094347/mongodb-on-ec2`.

MongoDB in the Amazon Cloud, `http://nosql.mypopescu.com/post/4233617312/mongodb-in-the-amazon-cloud`.

6 Service de cache

6.1 Modèle

memcached, ehcache ?

`http://en.wikipedia.org/wiki/Memcache` `http://en.wikipedia.org/wiki/Ehcache`

6.2 Implémentations

Idem.

6.3 Protocole d'accès

Protocole memcache.

6.4 API d'usage

Java: jcache ? (cf. google)

6.5 API de gestion

Cf. proposition générique.