

# Services Web : Choix Architecturaux

Philippe Mougin

Avril 2007



---

Cabinet d'Architectes en Systèmes d'Information

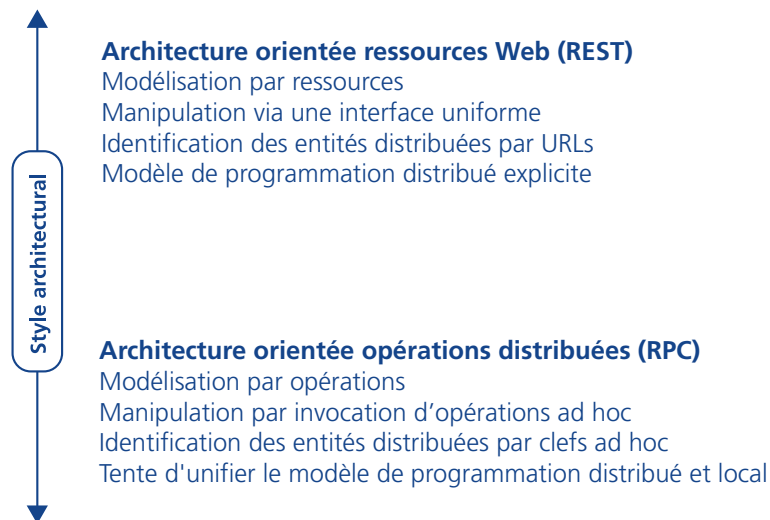


## Services Web : Choix Architecturaux

Afin de réutiliser au mieux l'infrastructure existante et de favoriser l'interopérabilité, vous concevez une architecture distribuée en utilisant des services Web sur HTTP(S)... Bien ! Mais vous êtes-vous posé la question du style architectural, c'est-à-dire du choix entre une approche orientée appel de procédure à distance ou une approche orientée manipulation de ressources ? Et celle du choix entre une pile de protocoles lourde ou une pile légère pour les échanges avec les services Web ? Vos décisions auront un impact important sur les propriétés de votre système : complexité, facilité d'évolution, obsolescence technique, capacité à monter en charge, stabilité, ouverture. Nous proposons ici un tour d'horizon des grandes options architecturales... et quelques précieux conseils.

# 1. Styles Architecturaux

REST (Representational State Transfert) et RPC (Remote Procedure Call) sont les deux styles architecturaux les plus couramment utilisés pour la mise en œuvre des services Web sur HTTP. Ils sont très différents dans leurs concepts et dans les propriétés qu'ils induisent sur les systèmes. Ainsi, on peut positionner une architecture sur l'axe ci-dessous selon qu'elle met plus ou moins en œuvre les principes de chacun des styles.



## 1.1. RPC

L'idée du style RPC (et de son adaptation à la programmation objet : le style objet distribué) est astucieuse. Elle consiste à étendre à la construction de systèmes distribués les concepts et techniques de programmation qui ont fait leurs preuves : découpage en procédures (ou méthodes), appel de procédure, etc. L'objectif est ici d'offrir en distribué un modèle de programmation le plus proche possible du modèle utilisé en local. Ainsi, il devient possible d'invoquer depuis une application une procédure ou une méthode exposée par un programme distant. On retrouve cette approche dans des technologies telles que DCE, CORBA, DCOM, .NET Remoting ou encore RMI. En ce qui concerne les services Web, le style architectural RPC se retrouve dans des technologies comme XML-RPC et dans de nombreuses utilisations de SOAP (via, par exemple, des frameworks comme JAX-RPC).

Penchons nous sur un exemple fictif de services Web de gestion des contrats chez France Télécom. Dans notre exemple, nous disposons d'un service Web permettant d'obtenir le contrat associé à un numéro de téléphone donné. Voyons à quoi ressemble, en architecture RPC, le pseudo code d'une application cliente, c'est-à-dire utilisant le service. Nous supposons ici l'utilisation d'un

langage de programmation statiquement typé (comme le sont Java et C#). Notre application récupère et affiche le nom du titulaire du contrat associé au numéro de téléphone 01 58 56 10 00.

```
// Définition de la classe Contrat
class Contrat
{
    Date    echeance
    String  referenceClient
}

// Récupération d'un proxy vers le service de gestion de contrat
Service contratService = WebServiceRegistry.lookup("FT_contrat_service")

// Récupération du contrat via un appel RPC au service Web
Contrat contrat = contratService.rechercherContratPourNumeroTel("01 58 56 10 00")

// Impression de la date d'échéance après récupération via un appel de méthode
print contrat.getEcheance()
```

## 1.2. REST

Plus de notions de procédure et d'invocation à distance avec REST<sup>1</sup>. À la place, des ressources distribuées (nos entités «métier» distribuées), désignées par des URL et dotées de représentations concrètes (XML, image, etc.) Dans notre exemple, chaque contrat, de même que chaque client, sera modélisé par une ressource Web, et désigné par un URL particulier. La représentation d'une ressource «contrat» pourrait être un fragment XML ressemblant à ceci :

```
<contrat>
  <echeance>
    2007-12-29
  </echeance>
  <client>
    http://www.francetelecom.com/clients/2340027
  </client>
</contrat>
```

Avec REST, les ressources sont manipulées à distance via un ensemble prédéfini d'opérations, principalement GET, PUT, DELETE et POST, que le programmeur utilise explicitement dans son code. Si vous reconnaissez là les principes de base du Web, vous avez vu juste ! L'idée consiste ici à construire des services Web en utilisant directement ces principes. On ne cherche plus à masquer le réseau au développeur, mais l'on adopte au contraire une sémantique explicitement distribuée, très différente de l'appel de procédure. On utilise ainsi directement la sémantique définie par HTTP, au lieu de la recouvrir d'une couche orientée RPC.

---

<sup>1</sup> On pourra se former sur REST en consultant les travaux de Roy Fielding [2], l'architecte principal de HTTP 1.1, le RESTwiki [3] et l'ouvrage RESTful Web Services [4].

Reprenons notre exemple d'utilisation de service Web depuis un client, en architecture REST cette fois-ci :

```
// Récupération du contrat via une interaction REST avec le service Web
XML contrat = GET http://www.francetelecom.com/contrats/telephonie/0158561000

// Impression de la date d'échéance après récupération via XPath
print contrat/echeance
```

Lorsque, comme c'est souvent le cas, il est possible de le mettre en œuvre, REST peut doter un système distribué de propriétés intéressantes : simplicité, forte capacité à monter en charge, temps de réponse réduits, couplage faible, synergie avec les autres technologies du Web (Web sémantique par exemple). Propriétés que l'on ne retrouve pas avec RPC. L'expérience montre notamment que ce dernier induit un fort couplage entre composants et rend difficiles les évolutions du système distribué. Or, dans une architecture d'entreprise ou inter-entreprise, il est important de pouvoir faire évoluer les composants le plus indépendamment possible les uns des autres, une capacité qui caractérise les architectures dites *faiblement couplées*<sup>2</sup>.

Lorsqu'un service évolue, par exemple pour s'adapter à des évolutions métier, on souhaite minimiser les interventions à effectuer sur les clients existants (modification, recompilation, redéploiement).

Notre exemple va servir à illustrer certaines différences importantes entre RPC et REST du point de vue du couplage.

### 1.3. Couplage

#### Couplage sur la structure des données

En RPC, la structure des données échangées est projetée sur le système de type de notre langage de programmation. Elle est par conséquent fixée dès la compilation. Cela n'est pas imposé par REST qui permet donc de mieux supporter les évolutions de structure des données.

Par exemple, dans le cas de notre service de gestion de contrats, nous souhaitons associer une nouvelle donnée aux contrats : un nombre de points de fidélité, qui augmente lors de l'utilisation de la ligne et permet ensuite au titulaire de bénéficier de réductions ou de cadeaux merveilleux.

Dans le cas RPC, la plupart des outils de service Web requièrent d'intervenir non seulement côté serveur, ce qui est normal puisque nous modifions notre modèle de données, mais aussi au niveau des clients existants, **même lorsqu'ils n'ont pas vocation à utiliser cette nouvelle donnée ou ne sont pas encore prêts à le faire**. Ainsi, il faudra modifier la définition de la classe `Contrat` dans chaque application cliente, et régénérer puis redéploier sur chaque client le code des stubs qui assurent l'acheminement des requêtes et des réponses.

---

<sup>2</sup> Le couplage s'analyse également sur d'autres dimensions. L'une des plus connues est la dimension *temporelle* (chaînes de traitement synchrones vs. queues de messages).

```
// Définition de la nouvelle classe Contrat
class Contrat
{
    Date    echeance
    String  referenceClient
    Number  points
}
```

Contrairement à RPC, REST n'impose pas de projection de la structure des données échangées sur le système de typage statique de notre langage et donc pas d'intervention à mener sur nos clients existants, qui peuvent continuer à fonctionner tels quels. Bien sûr REST n'est pas magique et il pourra être nécessaire de modifier les clients qui doivent manipuler la nouvelle donnée. Mais à la différence de RPC, ce sont uniquement ces clients-là qu'il faudra modifier. Il est également possible que les données soient suffisamment auto-descriptives et le code de certains clients suffisamment *data-driven* pour qu'ils puissent utiliser la nouvelle donnée sans avoir à être modifiés et redéployés (imaginez, par exemple, un client chargé d'imprimer le contrat renvoyé par le service Web et programmé de manière à itérer sur l'ensemble des champs du contrat, quels qu'ils soient, et à les imprimer).

Face à cette situation les fournisseurs de technologies RPC commencent à réagir et certaines technologies récentes de service Web, telles JAX-WS, permettent en architecture RPC certaines modifications de structure des données échangées (comme dans notre exemple) sans requérir une modification des clients. Mais ces technologies sont encore peu déployées et produisent des systèmes plus couplés que ce qu'il est possible d'obtenir en architecture REST. Par exemple, avec la version actuelle de JAX-WS (v. 2.0), le changement du type d'un paramètre de `double` en `int` requiert la régénération des stubs et le redéploiement sur les clients, même si ces derniers ne passaient que des valeurs entières. De plus, la structure des données ne peut pas être elle-même définie dynamiquement, c'est-à-dire dépendre de l'état du système à l'exécution, alors que ce niveau poussé de dynamisme est possible en REST (à nouveau, pensez *data-driven programming*).

En n'imposant pas, sur les données, de couplage plus fort que ce qui est requis par les besoins intrinsèques aux traitements, REST nous laisse libre de mettre en œuvre des *patterns* de programmation permettant de maximiser, autant que possible, la compatibilité ascendante et descendante des clients et des services (*patterns* tels que *default-value-for-missing-data*, *default-value-for-unknown-value*, etc.). L'adaptabilité du système en est renforcée.

### Couplage sur le nom des opérations

En RPC, le nom des opérations (ex: `rechercherContratPourNumeroTel`) dépend de chaque service distant et est utilisé dans le code des clients pour effectuer les appels. Lorsqu'une évolution d'un service RPC requiert l'évolution du nom de certaines opérations, par exemple pour refléter une évolution de la sémantique des opérations, il est nécessaire d'intervenir au niveau des clients pour introduire ces changements. En REST, les opérations (GET, PUT, DELETE, POST, etc.) sont fixées par le protocole (celui-ci pouvant néanmoins être extensible, comme c'est le cas de

HTTP). Leur sémantique étant très générale, le code côté client est, dans un certain sens, moins «auto-documenté» qu'en RPC, mais plus résistant aux évolutions de sémantique des services : le couplage est plus faible.

### Couplage sur les références aux entités

En RPC, les références vers d'autres entités sont modélisées de manière spécifique à chaque application ou groupe d'application. Dans notre exemple, le contrat renvoyé par le service contient une référence client spécifique, par exemple «2340027». En général, cette référence sera uniquement interprétable par les applications partageant le même référentiel «Clients» ou ayant accès à des services pouvant interpréter cette référence.

Au contraire, en REST, les entités distribuées sont des ressources Web et sont donc référencées par des URL. Dans notre exemple, notre contrat est un document XML dans lequel la référence au client est simplement l'URL de la ressource correspondante – par exemple <http://www.francetelecom.com/clients/2340027>. En REST, les références deviennent donc des identifiants universels, dont une partie de la sémantique n'est pas liée (couplée) à un groupe d'applications spécifiques mais définie de manière universelle pour toutes les ressources par la spécification HTTP et les spécifications attenantes. La preuve : je peux entrer cet URL dans la barre d'adresse de mon navigateur Web et obtenir le document XML décrivant le client correspondant.

De plus, il devient possible de migrer les entités vers d'autres SI, ou d'autres parties du SI, sans impacter les applications qui font usage des références. Dans notre exemple, si France Telecom décide de sous-traiter la gestion de certains dossiers client à une autre structure, il lui suffira de renvoyer, dans les données décrivant le contrat, le nouvel URL de la ressource décrivant le client, par exemple: <http://www.orange.fr/telephonie/clients/C-X2-98764>. Et pour effectuer cette transition complètement en douceur, une redirection HTTP sera mise en place de l'ancienne ressource vers la nouvelle. Les applications clientes ne seront pas impactées : couplage faible !

## 1.4. Transparence de localisation

RPC tente d'offrir au développeur un modèle de programmation unifié, que les composants manipulés soient locaux ou distants. L'idée d'offrir cette transparence de localisation est séduisante, mais s'avère souvent délétère. La raison principale en est que cette unification est, jusqu'à preuve du contraire, impossible à vraiment réaliser, sauf en dégradant et en complexifiant massivement le modèle de programmation en local, comme s'y sont essayées les premières versions de la technologie EJB ou encore COM/DCOM. La fameuse « Note sur l'informatique distribuée » de Jim Waldo & al. [1], que l'on consultera pour plus d'informations sur ce point, tirait la sonnette d'alarme dès les années 90. Au final, il semble que les abstractions bien adaptées à la programmation en local (procédures ou méthodes, arguments, appels de procédures...) ne

soient pas bien adaptées à la programmation distribuée<sup>3</sup>... Et encore moins à la programmation distribuée sur le Web et aux systèmes faiblement couplés. Occupées à résoudre des problèmes très complexes liés à la vision unifiée, s'alourdissant avec le temps et les versions, les technologies de service Web RPC passent à côté des problèmes importants, comme ceux du couplage, ou ne les traitent que partiellement.

### 1.5. HTTP & REST

Dernier élément clef : HTTP a été conçu pour fabriquer des systèmes utilisant l'approche REST. Ainsi, REST est le style architectural nativement supporté par HTTP. Alors que, avec RPC, la sémantique des échanges est opaque pour l'infrastructure HTTP, elle devient connue avec REST. Ainsi, plus vos services Web respectent les principes de REST plus ils bénéficient des capacités pouvant être offertes par votre infrastructure HTTP, par exemple :

- Réémission des requêtes
- Politique de mise en cache des réponses
- Utilisations des clients HTTP existants
- Système de redirection
- Négociation de contenu
- Codes d'erreurs standardisés
- etc.

Pour prendre une image, mettre en œuvre RPC avec HTTP c'est un peu l'équivalent d'utiliser un avion pour rouler sur une route : il faut vraiment avoir une très bonne raison pour le faire, et ce n'est pas, en tout état de cause, la solution à envisager en premier.

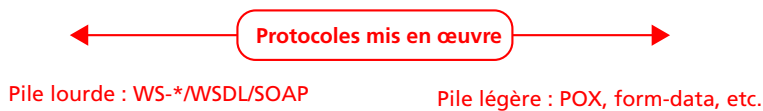
---

<sup>3</sup> Néanmoins, la messe n'est pas dite. D'intéressants travaux sur les objets distribués sont menés. Voir par exemple le modèle TeaTime, utilisé par le projet Croquet [7].

## 2. Piles de Protocoles

Examinons maintenant les piles de protocoles permettant la description des services Web et la mise en œuvre des échanges sur le réseau. D'un côté on trouve l'ensemble SOAP/WSDL/WS-\* (prononcez «WS star»). De l'autre, on trouve les classiques POX (Plain Old XML), form-data, et diverses notations compactes et légères, comme YAML ou encore JSON qui permet notamment des échanges simplifiés en environnement AJAX.

On peut positionner une architecture sur l'axe ci-dessous selon qu'elle met plus ou moins en œuvre les composants de ces piles techniques.



Par exemple, si une architecture met en œuvre des spécifications WS-\* (WS-Reliable Messaging, WS-Atomic Transaction, etc.) elle se positionne à l'extrême gauche de l'axe. Si elle ne met en œuvre que SOAP et WSDL, elle se positionne sur la partie gauche de l'axe, alors que si elle n'utilise que SOAP ou que WSDL, elle se positionne au centre gauche. Sur la droite de l'axe, on positionnera les architectures utilisant la pile légère. Plus les protocoles seront légers (XML minimaliste, voir uniquement form-data, texte simple ou query string) plus l'architecture sera positionnée vers la droite de l'axe.

L'expérience montre que la pile légère est la mieux adaptée dans la majorité des cas. Quatre raisons principales à cela :

- La plupart des applications n'ont pas besoin, pour les services Web, des fonctionnalités avancées offertes par la pile lourde (sécurité au niveau de fragments de message, commit à deux phases, etc.). Les fonctionnalités requises par la majorité des applications sont accessibles avec la pile légère. En particulier, on notera que cette dernière permet de réaliser des systèmes bien sécurisés et offrant des modèles fiables de transfert sur HTTP.
- L'utilisation de la pile lourde entraîne un fort couplage technique entre clients et services. En effet des implémentations de cette pile doivent être déployées côté client et serveur. Or, plus la pile est lourde (située sur la gauche de notre axe), plus ces implémentations sont rares et cantonnées aux grandes plateformes (Java et .NET). La pile légère, en revanche, utilise des technologies d'ores et déjà quasi-universellement déployées. Les services Web utilisant la pile légère sont donc accessibles depuis un nombre beaucoup plus grand de consommateurs. De même, il est possible d'exposer des services basés sur la pile légère depuis un plus large spectre de plateformes.

- Un des objectifs de la pile lourde est d'être indépendante du protocole sous-jacent (dans le contexte de cet article, il s'agit de HTTP). Une grande partie de la pile est donc dédiée à fournir des abstractions et des fonctions déjà présentes dans HTTP. Utiliser la pile lourde au-dessus de HTTP, c'est donc empiler inutilement des couches redondantes et faire le choix d'utiliser HTTP essentiellement comme un simple protocole de transport (couche ISO 4) alors qu'il s'agit en fait d'un protocole de niveau application (couche ISO 7).
- Les composants de la pile lourde souffrent d'un grand nombre de problèmes, parmi lesquels une trop grande complexité, des problèmes majeurs d'interopérabilité et une grande immaturité.

De plus, la pérennité des technologies de la pile lourde est très douteuse. Au-delà du « hype », même les fondations de la pile lourde que sont SOAP et WSDL, minés par des problèmes structurels, ont peu de chance de s'imposer largement et durablement, malgré le support de fournisseurs majeurs tels que Microsoft et IBM.

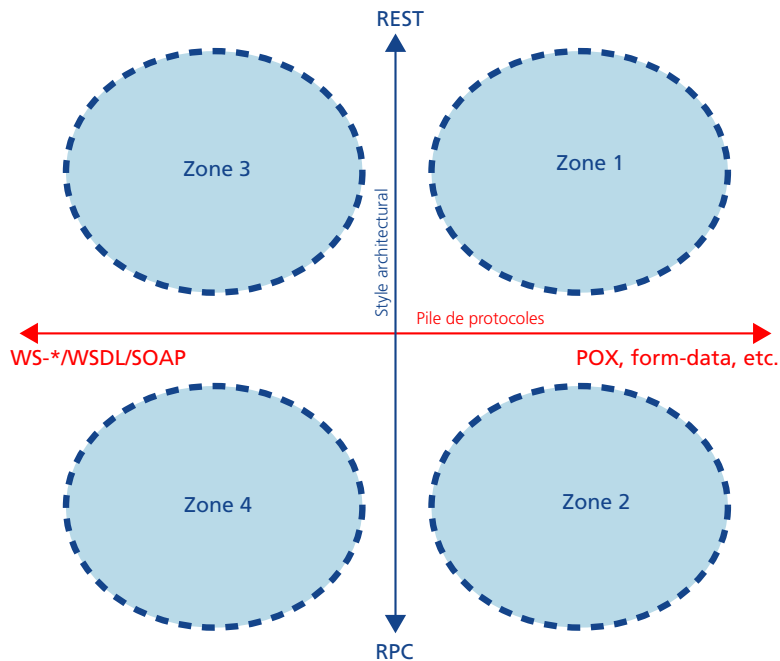
Au vu de ces éléments, nous conseillons de rejeter le raisonnement *qui peut le plus peut le moins*, qui préside souvent à l'adoption de la pile lourde. Au contraire, on utilisera une approche minimaliste dans le choix des protocoles et technologies utilisés : toute introduction de complexité devra être justifiée et ne se faire qu'en dernier ressort. On adoptera donc la pile la plus légère possible. Même le choix de XML ne s'imposera pas sans justification : parfois, un simple form-data ou un autre type MIME sera plus adapté.

Cela dit, certains systèmes peuvent requérir l'utilisation de la pile lourde, motivée en général par la nécessité d'interagir avec des services existants utilisant cette pile, ou des besoins particuliers (transactionnel distribué, sécurité à granularité fine, etc.)

### 3. Espace architectural

L'utilisation d'une pile particulière de protocoles n'induit pas automatiquement le choix du style architectural (REST ou RPC). Ainsi, il est possible de créer une architecture REST avec les composants de la pile légère, mais aussi avec ceux de la pile lourde (au moins à partir des versions 1.2 de SOAP et 2.0 de WSDL, cette dernière étant encore en développement). Il en va de même pour le style architectural RPC.

On peut donc réunir nos deux axes d'analyse pour former le référentiel suivant. Il nous permet d'identifier quatre grandes zones dans l'espace des choix architecturaux qui nous intéressent ici.



Selon les systèmes distribués à mettre en place, l'une ou l'autre de ces zones s'avèrera la plus adaptée. En général, pour les systèmes basés sur les services Web HTTP, ce sera la zone 1 (REST + pile légère).

On considèrera donc la **zone 1** de notre espace architectural comme **la zone de confort maximal pour les services Web sur HTTP**, et l'on tentera dans la mesure du possible de construire des architectures se situant dans cette zone. L'outillage requis est habituellement déjà déployé. En effet, pour accéder aux services Web un client devra simplement disposer d'une API permettant d'émettre des requêtes HTTP, ce que l'on trouve aujourd'hui dans quasiment toutes les plateformes. Côté serveur, on pourra directement utiliser, pour exposer les services, les

technologies destinées aux applications Web. Par exemple, en Java les Servlets feront parfaitement l'affaire. Il existe même un framework open-source facilitant les développements, le framework Restlet [5]. Un effort de définition d'un framework standard est en cours au travers de la JSR 311 : Java API for RESTful Web Services [6].

La **zone 2** sera envisagée si l'on peut justifier que RPC est mieux adapté que REST au système à construire. Par exemple, il peut être nécessaire d'interagir avec des services ou des clients existants développés dans le style RPC, ou bien de traiter un domaine modélisable de manière plus pertinente sous forme d'opérations distribuées que de ressources distribuées. On conserve dans cette zone les avantages de la pile légère.

La **zone 3** permet de bénéficier des avantages de REST, mais subit les inconvénients de la pile lourde. Son choix devra donc être justifié par des besoins particuliers requérant l'utilisation de la pile lourde. Devant la popularité grandissante de REST, cette zone est de plus en plus mise en avant par les fournisseurs d'infrastructures basées sur la pile lourde. Elle reste cependant assez peu usitée car les protocoles de la pile lourde, bien que permettant dans leurs versions les plus récentes la mise en œuvre de REST, ne sont pas très bien adaptés à ce style architectural et induisent souvent un REST «dégradé».

La **zone 4** sera à envisager en dernier ressort car elle combine les inconvénients de la pile lourde et du style RPC (en particulier le couplage fort).

Enfin, on sera vigilant sur les normes de développement et les outils. En effet, beaucoup d'ateliers de développement et de technologies de services Web produisent des architectures se situant dans la zone 4.

## Références citées dans cet article :

- [1] A Note on Distributed Computing. Samuel C. Kendall, Jim Waldo, Ann Wollrath et Geoff Wyant. <http://research.sun.com/techrep/1994/abstract-29.html>
- [2] Architectural Styles and the Design of Network-based Software Architectures. Roy Thomas Fielding. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- [3] RESTwiki. <http://rest.blueoxen.net/cgi-bin/wiki.pl>
- [4] RESTful Web Services. Leonard Richardson et Sam Ruby, éditions O'Reilly. Publication prévue en mai 2007.
- [5] Restlet : <http://www.restlet.org>
- [6] JSR 311 – Java API for RESTful Web Services : <http://jcp.org/en/jsr/detail?id=311>
- [7] TeaTime Architecture : [http://www.opencroquet.org/index.php/TeaTime\\_Architecture](http://www.opencroquet.org/index.php/TeaTime_Architecture)

## A propos d'OCTO Technology

Fondé en 1998, OCTO Technology est le premier Cabinet d'Architectes des Systèmes d'Information. Consulté sur tous les chantiers stratégiques de refonte ou de rénovation, OCTO, dimensionné pour traiter les plus grands projets est devenu en quelques années la référence des grands comptes.

Ses fondateurs sont très attachés au respect des valeurs du métier d'Architecte : l'écoute et le devoir de conseil. L'organisation du Cabinet permet aux clients de profiter de la valeur de la communauté d'Architectes sur chaque mission réalisée, privilégiant l'intelligence collective à la pensée unique.

Pour en savoir plus : [www.octo.com](http://www.octo.com)

## Bibliographie OCTO Technology

Dans la collection : « Une Politique pour le Système d'Information »



Une Politique pour le Système d'Information

**Descartes - Wittgenstein - (XML)**



Une Politique pour le Système d'Information

**Gestion des identités**

OCTO Technology publie régulièrement le fruit de ses expériences et de ses recherches sous forme de Livres Blancs :

- [Architecture Orientée Services \(SOA\)](#), Une politique de l'interopérabilité (2005)
- [Architecture de Systèmes d'Information](#) - Gouvernance de la Donnée (2004)
- [Architecture d'Applications](#) - la solution .NET (2003)
- [Architectures de Systèmes d'Information](#) (2002)
- [Le Livre Blanc de la Sécurité](#) (2001)
- [IRM, Gestion de la Relation Client sur internet](#) (2000)
- [EAI, Intégration des Applications d'Entreprise](#) (1999)
- [Les Serveurs d'Applications](#) (1999)

OCTO Technology est également l'auteur de trois ouvrages parus aux Editions Eyrolles :



« Le projet e-CRM - Relation client et Internet » (2002)



« Les Serveurs d'Applications » (2000)



« Intégration d'Applications - l'EAI au cœur du e-business » (2000)





---

Cabinet d'Architectes en Systèmes d'Information