



Pushlet et Pilot : pour un nouveau modèle de développement Web

Mehdi Soua mehdi.soua@laposte.net
Dominique Sauquet dominique.sauquet@ecp.fr

Contact : dominique.sauquet@ecp.fr

Introduction

Depuis quelques temps nous assistons à une effervescence du Web et beaucoup pensent que nous sommes à l'aube d'une nouvelle révolution Internet, celle du Web 2.0 [01]. Aujourd'hui, les sites Web ne sont plus seulement composés de simples listes de liens hypertexte sur lesquels on doit cliquer. Les nouvelles applications Web se sont en effet dotées de fonctionnalités riches et d'une ergonomie qui n'a rien à envier aux applications « client lourd ». Tout ceci se base sur un fonctionnement simple et vieux de sept ans qui est le transfert asynchrone.

Le transfert asynchrone consiste à limiter les échanges entre le client (le navigateur) et le serveur aux seules données utiles. Ainsi, au lieu de recharger des pages entières, seuls les paquets de données nécessaires sont récupérés.

On retrouve aujourd'hui cette technologie dans de nouvelles applications comme Google Maps [02] qui se base sur la technologie Ajax (*Asynchronous JavaScript And XML*) [03].

Ajax semble promis à un bel avenir et il est vrai qu'il est déjà adopté par les développeurs défricheurs de nouvelles technologies et qu'il intéresse les grands éditeurs de logiciels. Néanmoins, il peut aujourd'hui apparaître comme une plateforme jeune qui n'offre pas encore une grande maturité et pour laquelle le développement reste complexe et requiert un haut niveau d'expertise.

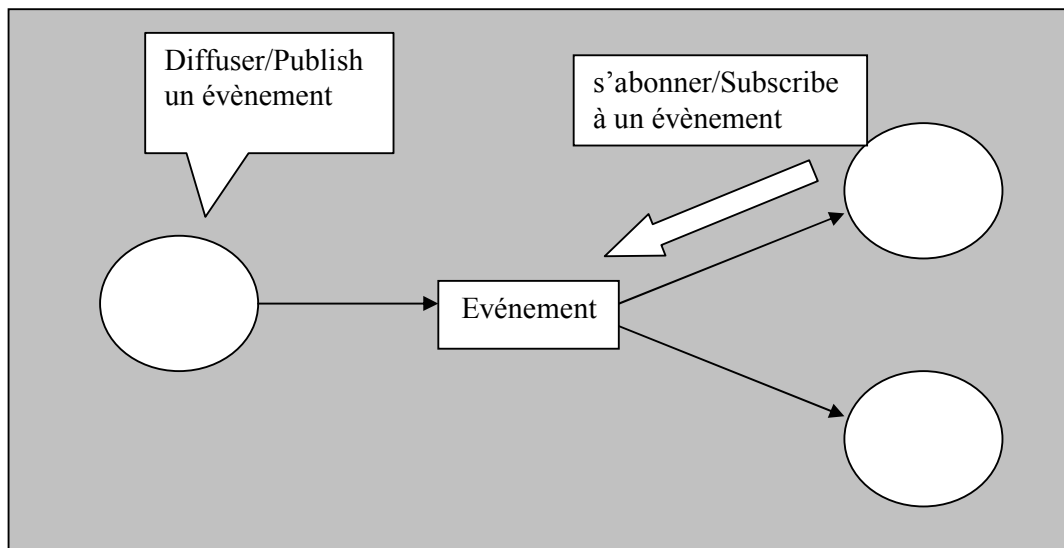
Ce que nous allons présenter ici se base sur la technologie **Pushlets** [04]. Cette technologie, plus mûre de 5 ans qu'Ajax, permet le transfert asynchrone et connaît un succès certain dans les applications Web qui mettent en œuvre des fonctions de Push.

Notre contribution vient de l'intégration de **Pushlets** avec un framework de développement d'applications, du nom de **Pilot**. Comme on le montrera, cette intégration apporte une véritable plus-value dans le développement d'applications complexes qui accèdent à des composants hétérogènes (bases de données, systèmes d'aide à la décision, composants logiciels divers...).

Qu'est ce que les *Pushlets* ?

Pushlets est un mécanisme qui permet de réaliser des applications selon le paradigme **subscribe/publish** (contrairement à Ajax qui se base sur un fonctionnement PeerToPeer). Ceci concerne de nombreuses applications comme celles de suivi de cours de bourse, de vente aux enchères en ligne, de monitoring, de surveillance météo, de travail collaboratif, ... Ces applications ont en commun que le client ou navigateur va être mis à jour (modification de la page affichée) lors de l'occurrence de certains évènements.

On parle donc de **subscribe** puisque l'application va « souscrire » aux évènements qu'elle veut surveiller et de **publish** puisque l'occurrence d'un évènement va déclencher une publication vers les clients ayant souscrit à l'évènement en question. Un évènement peut être déclenché par un client donné (exemple d'un chat où un navigateur envoie un message au serveur qui le redistribue vers les autres navigateurs) ou bien par un composant externe qui communique avec le serveur (exemple d'un trigger dans une base de données qui se déclenche lors d'un changement de valeur).

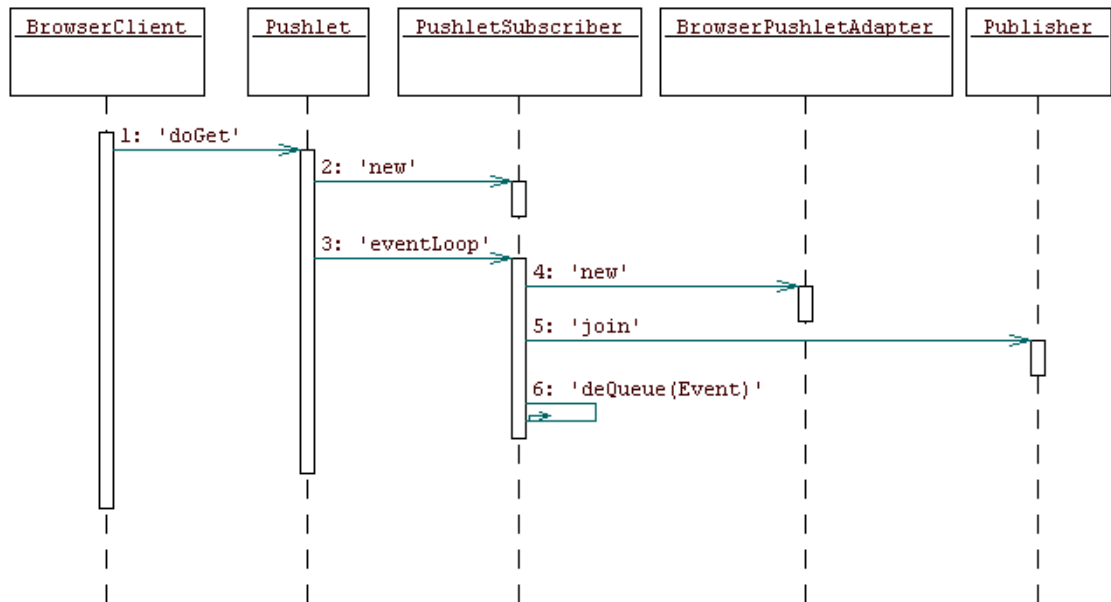


Pour réaliser ces fonctionnalités, *Pushlets* se base sur la technologie servlet. Le mécanisme qu'il implémente est basé sur une connexion *http servlet* et sur l'envoi de code JavaScript au navigateur. Le navigateur reçoit donc, sous la forme de code JavaScript, une information liée à l'évènement qui vient d'arriver, évènement faisant partie de la liste de ceux auxquels il a souscrit. C'est l'interprétation automatique du code JavaScript qui provoque le rafraîchissement de la page affichée par le navigateur.

Pour illustrer le fonctionnement de Pushlet, le diagramme de séquence UML ci-dessous décrit l'abonnement d'un client à un évènement du *Publisher*.

L'objet `Pushlet` est l'instance de Servlet qui traite les demandes en provenance du navigateur. Cet objet, pouvant être sollicité par différents clients, crée, pour chaque

demande de souscription, un objet `PushletSubscriber` auquel il délègue le traitement de la souscription (méthode `eventLoop()`).



www.pushlets.com

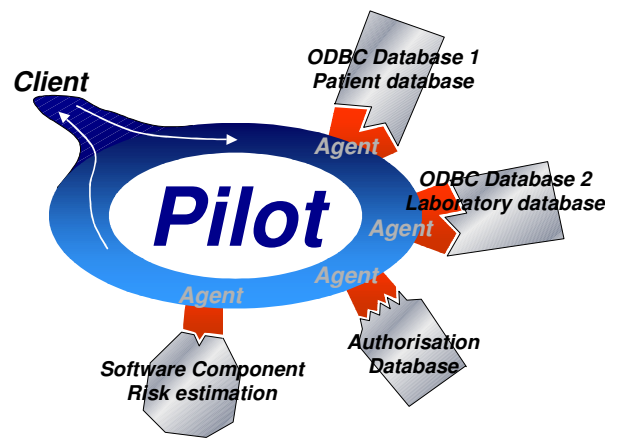
Le **Pilot** : middleware et framework

Le middleware

Le **Pilot** est un middleware issu de la recherche européenne (projet Synex, programme AIM)[05]. C'est un composant destiné à s'intégrer au sein d'une architecture distribuée *un client / n serveurs*. Il se place entre le client et les différents serveurs et sert d'interlocuteur unique pour le client. Son rôle est de permettre l'exécution de services complexes en les décomposant en services élémentaires ou étapes et gérant la synchronisation de l'exécution de ces étapes.

Le **Pilot** a accès à tous les composants distants de la plateforme et sait communiquer avec eux grâce à des modules appelés agents. Dans cette architecture, le concepteur de systèmes complexes a un accès unifié aux modules logiciels (bases de données, serveurs d'authentification, modules d'aide à la décision, ...). C'est une véritable plateforme d'intermédiation entre modules logiciels parlant des langages différents et utilisant des protocoles de communication différents.

Le **Pilot** est un outil de communication et d'intégration qui se présente sous la forme d'une API (Application Programming Interface) Java.



Il repose sur les notions de *Services*, d'étapes ou *Steps* et d'*Agents* :

- Un *Service* correspond à une tâche complexe à réaliser pouvant mettre en jeu plusieurs composants extérieurs. Le service est décomposé en étapes élémentaires et sa description se fait en XML. Il permet de décrire les conditions d'exécution des étapes, les conversions de paramètres ou encore des tâches de synchronisation.
- Les étapes ou *Steps* vont exécuter une action élémentaire comme l'accès en lecture ou en écriture à une base de données. L'exécution d'une étape peut être soumise à la validation d'une condition. Les étapes peuvent se déclencher à la suite les unes des autres ou en parallèle. Le modèle permet aussi de réaliser des boucles sur les étapes.
- Les *Agents* sont les modules qui permettent l'intégration des composants distants. Ce sont en fait des classes Java jouant le rôle de connecteurs. Si l'intégration d'un nouveau système comme un legacy système nécessite de créer sa propre classe Agent, il existe en revanche un certain nombre d'Agents « prêts à l'emploi » pour des cas courants comme l'intégration de bases de données relationnelles ou l'envoi de mails. La description XML des services fait alors référence aux classes Agents appelables par le Pilot pour l'application en question.

Ainsi, une intégration complexe de composants distribués sera facilitée car elle se fera simplement par la description en XML des services (plus simplement et plus souplesment que par l'écriture de code Java ou C++) et par la création des classes Agents en Java, classes dont la structure reste simple.

La description des différents services traités par le **Pilot** se fait au moyen d'un fichier au format XML. Le fichier commence toujours par la balise <servicelist>. La description de chaque service s'effectue ensuite à l'intérieur d'une balise <service>. Voici un exemple d'un service en XML.

```
<servicelist>
  <service name="Nom_Court"
    inputdtd="Nom_fichier"
    outputdtd="Nom_fichier">
    <longname> Nom_Long </longname>
    <step>
      Description de l'étape 1
    </step>

    <step>
      Description de l'étape 2
    </step>
  </service>
  <service name="Nom_Court"
    inputdtd="Nom_fichier"
    outputdtd="Nom_fichier">
    <longname> Nom_Long </longname>

    Description des étapes
  </service>
</servicelist>
```

De même que des Conditions d'exécution d'une étape peuvent être décrites à l'intérieur de la description d'un step, un mécanisme particulier, du nom de Valuator permet de décrire les paramètres d'entrée d'un step. Ces paramètres proviennent des paramètres d'entrée du service ou des résultats d'exécution des steps précédents. La description XML d'un step est donc :

```
<step name="nom-du-step">
  Description de l'appel à un agent
  <condition>
    Description de la condition
  </condition>
  <valuator>
    Description de la construction des paramètres
  </valuator>
</step>
```

Le framework pour applications Web

Le **Pilot** a aussi été intégré à un serveur http par le biais de la technologie servlet. Le mécanisme se base sur une servlet générique qui traite l'exécution des services. Cette servlet, représentée par la classe `GenericPilotServlet` :

1. reçoit une requête du navigateur dans laquelle figure en paramètre le nom du service à exécuter
2. construit les données d'entrée nécessaires à l'exécution du service en question
3. lance l'exécution du service (par l'API Java du **Pilot**)
4. renvoie au navigateur les données résultant de l'exécution du service :
 - a. soit après conversion « manuelle » des données (les données de l'exécution sont sous la forme d'un arbre DOM à partir duquel doit être généré de l'HTML)
 - b. soit (et c'est ce qui est recommandé) en redirigeant le résultat vers le moteur velocity [06] et en lui passant aussi le nom d'une *template*¹.

Un fichier XML de configuration permet de décrire comment doivent être construits les paramètres d'entrée de l'exécution de chaque service. Cette description part du simple constat que lorsque du code doit être exécuté sur le serveur http (une servlet dans l'architecture J2EE classique, un service dans notre framework) les données d'entrée de ce code sont soit des paramètres de la requête envoyée au serveur, des attributs de la même requête ou des objets conservés en session.

Dans le même fichier de configuration, il est possible de décrire des traitements en sortie d'exécution d'un service, par exemple mettre en session des éléments retournés par le service, pour un usage ultérieur en paramètre d'entrée d'un autre service.

¹ Fonctionnement de velocity dans le framework Pilot :

Ce moteur utilise un template d'HTML contenant des instructions de manipulation où les variables sont celles retournées par l'exécution du Pilot. L'exécution d'un service terminée, **Pilot** passe la main au moteur velocity qui génère la version HTML du template avec les valeurs particulières retournées par **Pilot**.

Intégration de *Pushlets* et *Pilot*

Le besoin

La cible du framework Pushlets est la réalisation d'applications complexes. Nous pouvons gager que ces applications auront très vite besoin d'intégrer des modules d'un Système d'Information (SI). Donnons ici quelques exemples :

- un évènement *join-listen* qui modélise l'entrée sur la plateforme peut nécessiter l'accès à une base de données pour authentification, l'émission d'un ticket de paiement, la vérification de statut ...
- un évènement *subscribe* peut nécessiter de vérifier des droits d'accès
- un évènement *publish* peut générer une sauvegarde d'une trace sur le serveur, un envoi de mails, ...
- un évènement *leave* peut générer un archivage, un envoi de mails, la clôture d'un ticket de paiement ...

L'implémentation de ces fonctions nécessite la compréhension du code source du framework Pushlets. En effet, le développeur devra « décortiquer » le code de la servlet Pushlet qui traite les évènements et intégrer les traitements propres à son SI à l'intérieur du code Pushlet. Ce mode de développement est tout sauf fiable et simple. Il ne permet pas de facilement faire évoluer l'application et aussi il doit être porté à chaque livraison d'une nouvelle version de Pushlets.

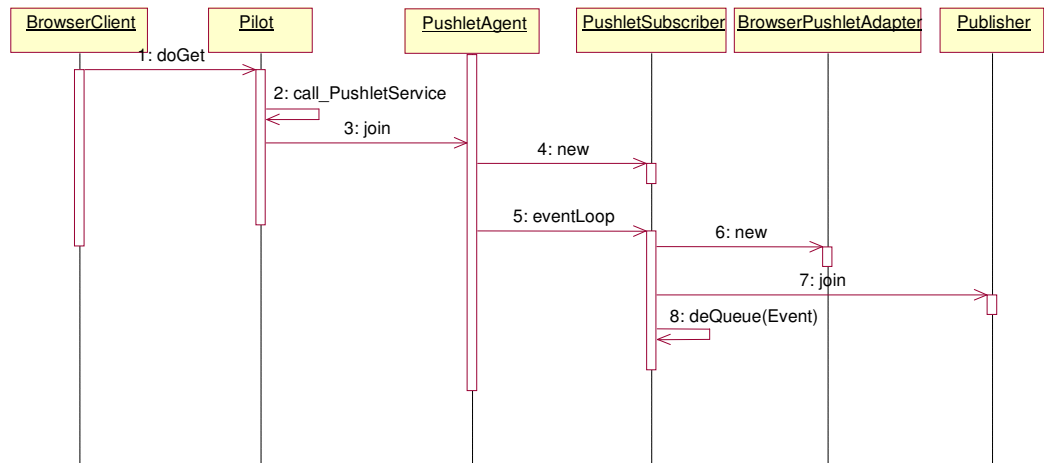
Pour pallier ces inconvénients, nous proposons d'intégrer *Pushlets* à *Pilot* afin de construire une plateforme de développement bénéficiant des apports des deux technologies.

La méthode

Le fonctionnement standard de *Pushlets* passe par l'utilisation d'une servlet `Pushlet` qui traite les évènements envoyés par le navigateur. Dans notre architecture, cette servlet `Pushlet` est remplacée par la servlet `GenericPilotServlet` qui récupère donc l'évènement `Pushlet`. Cet évènement est ensuite traité grâce à un service XML du *Pilot*, service que nous appelons *Pushlet*. Ce service exécute une série de tâches ou étapes correspondant à l'évènement envoyé.

La connexion aux fonctionnalités standards du framework *Pushlets* est réalisée par un Agent `PushletAgent`. La classe `PushletAgent` possède différentes méthodes qui correspondent aux évènements de *Pushlets*. Ces méthodes que l'on appelle « méthodes Agent » sont appelées à partir du service *Pushlet*. Elles contiennent du code Java permettant la liaison avec *Pushlet*. A chaque évènement de client vers serveur (*join*, *subscribe*, *listen*, ...) correspond une méthode Agent. Cette méthode peut être appelée dans un step d'un service et intercalée avec les autres étapes « métier » comme celles qui consistent à accéder au SI.

Nous allons revoir maintenant le scénario d'un *subscribe* où on ne passe plus par la servlet de *Pushlets* mais où c'est le « `PushletAgent` » qui réalise le traitement du « *join* ».



En déléguant à **Pilot** le traitement des évènements de **Pushlets** en entrée, ceci nous permet de réaliser toute une série d'actions. Le service qui prend en charge le traitement du *join* peut, par exemple, vérifier avant de lancer la souscription, que l'utilisateur qui se connecte est inscrit dans la base. Ceci peut s'écrire ainsi :

```

<step name="getUserId">
  <agent name="DataBaseAgent" method="selectData"/>
  <condition>
    <simple stepname="service" tagname="p_event" value="join"/>
  </condition>
  <valuator>
    <copier stepname="service" tagname="login"/>
    <copier stepname="service" tagname="password"/>
  </valuator>
</step>

<step name="execJoin" final="true">
  <agent name="agent.PushletAgent" method="join"/>
  <condition>
    <simple stepname="getUserId" tagname="id"/>
  </condition>
</step>
  
```

L'ordonnancement des traitements réalisés par les services de **Pilot** offre une multitude de possibilités. De cette manière, on contourne la rigidité et la complexité de **Pushlets**.

Mieux comprendre avec un exemple

Considérons un site de vente en ligne avec enchères dont la logique métier fait que l'entrée sur le site doit d'abord vérifier l'inscription du client dans une base de données puis créer un panier automatiquement affecté au client.

La traduction informatique de cette logique métier est décrite par un service XML du Pilot du nom de `myMarket`. L'exécution de ce service sera déclenchée à la réception d'une demande de connexion. Le serveur reçoit alors l'évènement `join/subscribe` avec comme sujet « BuyOnLine ».

Le service `myMarket` traite cet évènement en vérifiant le référencement du client dans la base (step 1), puis en exécutant un `join` (step 2) et un `subscribe` (step 3) pour finalement créer le panier.

Sont décrites ci-dessous les *steps* (étapes) qui réalisent cet enchaînement. Le premier *step* traite l'authentification du client. Par la suite, les *steps* `execJoin` et `execSub...` font appel au "PushletAgent" qui procède au traitement des évènements. Enfin le dernier *step* `list` affecte au client un panier.

```
<service name="myMarket">
  <step name="getUserId">
    <agent name="DataBaseAgent" method="selectData"/>
    <condition>
      <simple stepname="service" tagname="p_event" value="join">
    </condition>
    <valuator>
      <copier stepname="service" tagname="login"/>
      <copier stepname="service" tagname="password"/>
    </valuator>
  </step>

  <step name="execJoin">
    <agent name="agent.PushletAgent" method="join"/>
    <condition>
      <simple stepname="getUserId" tagname="id"/>
    </condition>
  </step>

  <step name="execSubscribe">
    <agent name="agent.PushletAgent" method="subscribe"/>
    <valuator>
      <copier stepname="service" tagname="p_id"/>
      <copier stepname="service" tagname="p_subject"/>
    </valuator>
  </step>

  <step name="list" final="true">
    <agent name="agent.ListAgent" method="setList"/>
    <valuator>
      <copier stepname="service" tagname="p_id"/>
    </valuator>
  </step>
</service>
```

p_subject = BuyOnLine

p_id = le login du client

A la toute fin de l'opération, lorsque le client confirme la commande, un évènement *leave* est envoyé au serveur. Le service qui prend en charge cet évènement réalise différents traitements comme le lancement du module de facturation et l'envoi d'un mail de confirmation au client.

```
<step name="invoicing">
  <agent name="agent.InvokingAgent" method="setInvoice"/>
  <condition>
    <simple stepname="service" tagname="p_event" value="leave"/>
  </condition>
  <valuator>
    <copier stepname="service" tagname="p_id"/>
    <copier stepname="service" tagname="ProductList"/>
  </valuator>
</step>

<step name="send">
  <agent name="col.agent.MailAgent" method="send"/>
  <valuator>
    <constant tagname="host" value="smtp.network.net"/>
    <constant tagname="from" value="admin@network.net"/>
    <constant tagname="subject" value="Invoice"/>
    <constant tagname="text" value="An Invoice has been sent"/>
    <copier stepname="service" tagname="to"/>
  </valuator>
</step>

<step name="execLeave" final="true">
  <agent name="agent.PushletAgent" method="leave"/>
  <valuator>
    <copier stepname="service" tagname="p_id"/>
    <copier stepname="service" tagname="p_subject"/>
  </valuator>
</step>
```

Ces deux exemples montrent la puissance et la modularité de l'intégration d'un SI dans une application réalisée avec notre framework.

Un des autres aspects de la plateforme est celui qui concerne les flux asynchrones.

Dans notre site de vente en ligne, les ventes peuvent se faire aux enchères. Une telle fonctionnalité est d'une grande complexité et les interfaces des sites de vente aux enchères ne sont pas toujours efficaces et ne permettent pas d'avoir les informations sur les enchères en temps réel. L'un des sites les plus connus est ebay[07]. Dans ce site à chaque fois que l'on veut enchérir, on doit passer par une autre page pour saisir la surenchère. Ce mode de fonctionnement n'est pas très efficace puisque entre temps une autre personne pourra enchérir sans que l'on soit au courant du nouveau prix.

À l'aide de notre framework, nous offrons une interface réactive grâce à un mécanisme de flux asynchrones qui nous permet de recevoir les informations en temps réel sans avoir à rafraîchir la page. Ce mécanisme est basé sur l'utilisation de **Pushlets**.

Un client pourra ainsi avoir un tableau de bord où figure les informations des produits qu'il a ajoutés à son panier. Les nouvelles enchères sont directement affichées sur ce tableau.

Comment cela fonctionne-t-il ?

Lorsqu'un client ajoute un produit dans son panier, il souscrit automatiquement à ce dernier grâce à l'évènement *subscribe*.

```
<step name="execSubscribe">
  <agent name="agent.PushletAgent" method="subscribe"/>
  <valuator>
    <copier stepname="service" tagname="p_id"/>
    <copier stepname="service" tagname="p_subject"/>
  </valuator>
</step>
```

Par la suite à chaque fois qu'un client enchérit sur ce produit, un évènement *publish* est envoyé au serveur avec les informations correspondantes au produit (identifiant du produit, identifiant du client, surenchère). Une fois l'évènement reçu, le mécanisme de **Pushlet** renvoie aux clients qui ont souscrit à ce produit la nouvelle surenchère.

```
<step name="execPublish">
  <agent name="agent.PushletAgent" method="publish"/>
  <valuator>
    <copier stepname="service" tagname="p_id"/>
    <copier stepname="service" tagname="p_subject"/>
    <copier stepname="service" tagname="newprice"/>
  </valuator>
</step>
```

Les informations renvoyées par **Pushlet** étant en javascript, le navigateur du client traite ce code à la volée et affiche directement le nouveau prix sans que le client n'ait à rafraîchir sa page.

Un tel fonctionnement permet au client, à travers un simple tableau de bord, de suivre en même temps plusieurs ventes et de pouvoir poser des surenchères sur un produit sans quitter les autres des yeux.

Photo	Prix actuel	vendeur	Temps restant	Enchérir
	CITROEN - C8 - 2.0 HDi110 Pack 22.000,00 EUR	Vendeur1	<1m	surenchère <input type="text"/> Confirmer
	MOTOROLA A925 NEUF ! + MENU EN FRANCAIS + ! DÉBLOQUÉ ! 269,90 EUR	Vendeur2	2h 44m	surenchère <input type="text"/> Confirmer
	STAR WARS EPISODE 3 - LA REVANCHE DES SITHS - NEUF SOUS 18,69 EUR	Vendeur3	5h 29m	surenchère <input type="text"/> Confirmer

construit à partir d'ebay

En utilisant une technologie standard, la réalisation d'un tel enchaînement de traitements nécessite la création de différentes pages rechargeant à chaque fois le contenu en entier et nous n'aurons pas d'informations mises à jour automatiquement sans l'intervention de l'utilisateur. Mais en utilisant **Pilot** et **Pushlets** tout ceci est fait en une seule page et en récupérant à chaque fois juste les paquets de données utiles.

Conclusion

Grâce à son architecture, **Pilot** intègre facilement un Système d'Information (SI). Les agents, qu'il s'agisse d'agents fournis avec **Pilot** ou d'agents créés pour des besoins spécifiques, permettent d'intégrer des éléments extérieurs. Ainsi, il est facile de connecter au **Pilot** plusieurs bases de données, différents modules qui constituent le SI de même que l'on peut accéder, à l'aide d'un autre agent, à un service Web.

De son côté, **Pushlets** offre des fonctionnalités pour réaliser un client riche mais l'intégration d'un SI et/ou de règles métier y est vraiment complexe.

Donc en intégrant le framework **Pushlets à Pilot**, nous disposons d'un composant qui exploite les ressources d'un SI tout en permettant de développer un client Web riche ayant des fonctionnalités de Push. L'assemblage de ces deux technologies offre une facilité de développement d'applications Web (gain de temps, facilité d'évolution ou de customisation) et qui permet une utilisation simple et souple de **Pushlets** dans des applications complexes.

En plus des avantages que nous avons cités, il nous semble important de conclure en disant qu'avec une telle plateforme, les développeurs s'abstraient des détails d'implémentation pour se concentrer sur les règles métiers qui vont être modélisées en XML. Tout cela s'inscrit dans la démarche SOA, nouvelle approche de développement des systèmes complexes.

Références

[01] : http://fr.wikipedia.org/wiki/Web_2.0

[02] : <http://maps.google.com/>

[03] : <http://www.activemq.org/Ajax>

[04] : <http://www.pushlets.com>

[05] : Xu Y, Sauquet D, Degoulet P, Jaulent M-C Component-based mediation services for the integration of medical applications. *Artif Intell Med* 2003; 27(3):283-304.

[06] : <http://jakarta.apache.org/velocity/index.html>

[07] : Site de vente en ligne <http://www.ebay.fr/>