Component Framework 2.0

Ce document est un guide de création de Composants JBI à l'aide du Component Framework (CF)

PEtALS Team

FABRE Olivier <olivier.fabre@ebmwebsourcing.com>

- June 2007 -

(CC) EBM WebSourcing - This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/

Table des matières

Component Framework 5
1. Comment créer un ServiceEngine (ex : composant XSLT) ? 6
  1.1. Les classes à implémenter 6
    1.1.1. La classe principale du composant 6
    1.1.2. La classe de traitement des messages JBI 6
  1.2. La configuration du composant 7
  1.3. L'activation de services 8
2. Comment créer un BindingComponent (ex : composant Mail) ? 9
  2.1. Les classes à implémenter 9
    2.1.1. La classe principale du composant 9
    2.1.2. La classe de traitement des messages JBI 9
    2.1.3. La classe de traitement des message externes 11
  2.2. La configuration du composant 12
  2.3. L'activation de services 13
  2.4. La consommation de services 13
3. Pour aller plus loin... 15
  3.1. Les extensions des descripteurs de déploiement (fichier jbi.xml) 15
    3.1.1. Les extensions du composant 15
    3.1.2. Les extensions des champs "provides" ou "consumes" des ServiceUnits 16
  3.2. Paramètres avancés de configuration du Composant 17
  3.3. Paramètres avancés de configuration des Service Units 17
  3.4. Les intercepteurs d'Exchanges 18
    3.4.1. Définition 18
    3.4.2. Implémenter un intercepteur 19
  3.5. Configuration 19
    3.5.1. Configurer un intercepteur au niveau composant 19
    3.5.2. Configurer un intercepteur au niveau service-unit 20
  3.6. Notification de performance 21
    3.6.1. Introduction 21
    3.6.2. Cas d'usage 21
    3.6.3. D'un point de vue client 22
    3.6.4. Coté développement 24
  3.7. L'envoi d'Exchange dans le conteneur JBI 25
    3.7.1. Le send synchrone avec timeout 25
    3.7.2. Le send synchrone sans timeout 25
    3.7.3. Le send asynchrone sans synchronisation des réponses 25
    3.7.4. Le send asynchrone avec synchronisation des réponses 26

Liste des illustrations

3.1. Performance notification RECEIVING_FROM_OUTSIDE/SENDING_TO_NMR 21
3.2. Performance notification RECEIVING_FROM_NMR/SENDING_TO_OUTSIDE 22
3.3. MBean de notification de performance du BC Soap 23
3.4. MBean de notification de performance du BC Soap 23
3.5. Réception des notifications de performance via la JConsole 24
3.6. Réception des notifications de performance via le plugin Eclipse Petals 24

Liste des tableaux

3.1. Configuration avancée du composant 17
3.2. Configuration avancée des Services Units (champs provides) 17
3.3. Configuration avancée des Services Units (champs consumes) 18
3.4. Configuration des intercepteurs au niveau du composant 20
3.5. Configuration des intercepteurs au niveau de la service unit 20
3.6. Performance notification format 22

Component Framework

Le Component Framework (CF) est un framework de developpement de composants JBI. Il se décline en un ensemble de classes facilitant l'interfacage avec le container JBI, notamment en prenant en charge l'utilisation des differents aspects décrits dans le chapitre 7 de la spécification JBI (Component Framework).

Il fournit des classes abstraites à étendre en fonction du type de composant (Binding Component ou Service Engine). Il fournit de plus des classes utilitaires facilitant la manipulation des MessagesExchanges ou des extensions presentes dans les descripteurs de déploiement des ServiceUnits. Enfin, il ajoute une notion d'"intercepteurs" qui permettent au composant d'effectuer des pré-traitements ou post-traitements sur les MessagesExchanges.

Chapitre 1. Comment créer un ServiceEngine (ex : composant XSLT) ?

Le composant XSLT est un bon exemple de Service Engine. Il ecoute les messages en provenance du conteneur JBI et applique une transformation XSL (préalablement définie à l'aide d'une Service Unit) sur leur contenu.

1.1. Les classes à implémenter

1.1.1. La classe principale du composant

La première classe à implémenter est la classe du corp du composant. L'exemple choisi propose de créer un Service Engine. La classe doit étendre la classe abstraite **org.objectweb.petals.component.framework.se.AbstractServiceEngine**.

Cette classe est utilisée par le conteneur durant les phases d'initialisation (init), de démarrage (start), d'arret (stop) ou de désactivation (shutdown) du composant. Le développeur peut agir sur chacune de ces phases en implémentant respectivement les méthodes **doInit()**, **doStart()**, **doStop()** ou **doShutdown()**.

Le développeur a accès aux éléments essentiels du composant comme le ComponentContext (contexte d'exécution du composant), le DeliveryChannel (API de communication avec le bus JBI), le Logger ou les ComponentProperties (toutes les propriétés du composant présentes dans les extensions du descripteur de déploiement jbi.xml). Ces différents éléments sont accessibles par les méthodes suivantes : **getContext()**, **getChannel()**, **getLogger()** et **getComponentConfiguration()**.

Le composant XSLT n'exige aucune initialisation particulière, l'implémentation de la classe **AbstractServiceEngine** est donc très simple :

```
package org.objectweb.petals.engine.xslt;

import org.objectweb.petals.component.framework.se.AbstractServiceEngine;

public class XsltEngine extends AbstractServiceEngine {
}
```

1.1.2. La classe de traitement des messages JBI

La deuxième classe à implémenter est la classe qui permet au composant de traiter les messages provenant du conteneur JBI. La classe doit étendre la classe abstraite **org.objectweb.petals.component.framework.listener.AbstractJBIListener**.

A chaque messages reçu du conteneur JBI, la méthode **onJBIMessage(Exchange exchange)** est appelée. C'est dans cette méthode que le developpeur peut effectuer les traitements sur le message exchange (la transformation xsl pour le composant d'exemple). Voici un exemple d'implémentation de cette classe pour le composant XSLT (il s'agit d'une version allégée pour les besoins de la documentation) :

```
protected void onJBIMessage(Exchange exchange) {
    try {
        // Get in Message content
        Source inContent = exchange.getInMessageContentAsSource();

        // Retrieve CF extensions
        Extensions extensions = new Extensions(getProvides()
                .getExtensions());

        // Retrieve XLS style sheet path
        String xslPath = retrieveXSLPath(exchange, extensions);

        // Apply transformation
        Source outContent = xslt.transform(inContent, xslPath);
```

```
        // Set the result to return
        exchange.setOutMessageContent(outContent);
    } catch (BusinessException e1) {
        try {
            exchange.setFault(e1);
        } catch (MessagingException e) {
            getLogger().log(Level.SEVERE, "Can't return fault to consumer",
                    e);
        }
    } catch (MessagingException e) {
        exchange.setError(e);
    }
}
```

Cette methode fait usage de classes et méthodes utilitaires proposées par le CF. Un exemple de classe utilitaire **org.objectweb.petals.component.framework.util.ExchangeImpl.** Cette classe implémente une interface similaire à **javax.jbi.messaging.MessageExchange**, mais lui ajoute des méthodes utilitaires et des controles supplémentaires permettant de manipuler facilement les principaux constituants d'un MessageExchange comme le contenu du message d'entrée (**getInMessageContentAsSource()**), le contenu du message de sortie (**setOutMessageContent(outContent)**), les Faults (**setFault(exception)**), les Errors (**setError(exception)**)...

Le JBIListener propose des méthodes permettant d'accéder : au ComponentContext (**getComponentContext()**), au DeliveryChannel (**getdeliveryChannel()**), au Logger (**getLogger()**) et au Composant (**getComponent()**). Il donne aussi accès au bloc Provides (représentation Java du champ provides d'une service unit) correspondant au endpoint cible du message exchange en cours de traitement. La méthode permettant d'accéder à cet objet Provides est **getProvides()**.

1.2. La configuration du composant

La configuration du composant se fait dans le descripteur de déploiement de ce dernier (fichier META-INF/jbi.xml dans l'archive du composant), selon la spécification. Dans ce fichier de configuration sont décrits : la classe principale du composant (balise "component-class-name"), la classe d'installation du composant (balise "bootstrap-class-name"), le classpath à utiliser pour l'execution du composant (balise "component-class-path") et le classpath à utiliser pour l'installation du composant (balise "bootstrap-class-path").

La partie spécifique à la configuration du CF se présente sous la forme d'extensions dans la description du composant (balise "extensions:extensions"). Voici un exemple de descripteur de déploiement pour le composant XSLT :

```
<jbi version="1.0" xmlns='http://java.sun.com/xml/ns/jbi'
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:extensions="http://petals.objectweb.org/extensions/">
 <component type="service-engine">
  <identification>
   <name>petals-engine-xslt</name>
   <description>A Xslt Service Engine</description>
  </identification>
  <component-class-name description="Xslt Component class">
   org.objectweb.petals.engine.xslt.XsltEngine
  </component-class-name>
  <component-class-path>
   <path-element>petals-se-xslt-1.4-SNAPSHOT.jar</path-element>
   ...
  </component-class-path>
  <bootstrap-class-name>
   org.objectweb.petals.component.framework.DefaultBootstrap
  </bootstrap-class-name>
  <bootstrap-class-path>
   <path-element>petals-se-xslt-1.4-SNAPSHOT.jar</path-element>
   ...
  </bootstrap-class-path>
  <extensions:extensions>
   <jbi-listener-class-name>
    org.objectweb.petals.engine.xslt.listener.JBIListener
   </jbi-listener-class-name>
  </extensions:extensions>
 </component>
</jbi>
```

Dans le cas d'un Service Engine, une seule extension est requise : **jbi-listener-class-name**. Il s'agit du nom complet de la classe de traitement des messages JBI comme défini dans les chapitres précédents.

1.3. L'activation de services

Un ServiceEngine n'est accessible que lorsqu'il expose des services sous forme de ServiceEndpoints. Un ServiceEndpoint permet d'exposer dans le bus JBI un service proposé par le ServiceEngine. Un exemple de service fournit par le composant XSLT est la transformation d'un message xml entrant en un message xml sortant en suivant les règles décrites dans une feuille de transformation XSL.

L'activation de endpoints intervient lors du démarrage de ServiceUnit (appel de la méthode **start(String suName)** du ServiceUnitManager du composant) contenant des champs "provides". Voici un exemple de ServiceUnit (SU) activant un endpoint pour le composant XSLT :

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:petals="http://petals.objectweb.org/"
 xmlns:extensions="http://petals.objectweb.org/extensions/">

 <services binding-component="false">
  <provides interface-name="petals:XsltInterface"
   service-name="petals:XsltService" endpoint-name="XsltEndpoint">
   <extensions:extensions>
    <xsl>test.xsl</xsl>
   </extensions:extensions>
  </provides>
 </services>
</jbi>
```

A l'issu du démarrage de cette SU, un endpoint est activé dans le conteneur pour le composant XSLT. Ce endpoint a pour nom d'interface, de service et de endpoint les valeurs definies respectivement dans les attributs **interface-name**, **service-name** et **endpoint-name** de la balise "provides". La balise "provides" inclu une balise "extensions" qui correspond aux extensions Petals. Les extensions Petals permettent de spécifier des informations de configuration du composant pour un endpoint donné. Dans le cas du composant XSLT, une extensions est utilisée pour spécifier le chemin relatif de la feuille de transformation XSL dans l'archive de la SU.

Chapitre 2. Comment créer un BindingComponent (ex : composant Mail) ?

Le composant Mail sera notre example d'implémentation d'un Binding Component sur la base du CF. Il permet l'echange de courriels entre un serveur de courriel et le container JBI.

2.1. Les classes à implémenter

2.1.1. La classe principale du composant

De même que pour un ServiceEngine, la première classe à implémenter est la classe du Composant. L'exemple choisi propose de créer un Binding Component. Dans ce cas, la classe à étendre est la classe **org.objectweb.petals.component.framework.se.AbstractBindingComponent**.

Cette classe hérite d'une classe commune à la classe AbstractServiceEngine. Elle est donc utilisée pour chaque phase du cycle de vie du composant et donne accès au ComponentContext, au DeliveryChannel, au Logger et aux paramètres de configuration du composant.

Le composant Mail nécessite trois classes utilitaires pour le traitement des courriels entrants et sortants ainsi que pour la gestion des connections au serveur de courriel. Ces trois classes sont instanciées à l'initialisation du composant et sont utilisées durant l'execution du composant. Voici l'implémentation de la classe du Composant Mail :

```
public class MailBCImpl extends AbstractBindingComponent {

    @Override
    protected void doInit() throws JBIException {
        this.sessionDescriptorBuilder = new SessionDescriptorBuilder(
                getLogger());
        this.mimeMessageManager = new MimeMessageManager(getLogger(), this);
        this.mailSessionManager = new MailSessionManager(getLogger());
    }

    public MailSessionManager getMailSessionManager() {
        return mailSessionManager;
    }

    public MimeMessageManager getMimeMessageManager() {
        return mimeMessageManager;
    }

    public SessionDescriptorBuilder getSessionDescriptorBuilder() {
        return sessionDescriptorBuilder;
    }
}
```

La méthode **doInit**(), surchargée ici pour instancier les trois classes utilitaires, est appelée en fin d'initialisation du composant pour permettre au développeur d'effectuer des tâches d'initialisation spécifiques. De même que pour un ServiceEngine, le développeur peut agir sur les phases de démarrage (**doStart**()), d'arrêt (**doStop**()) ou de désactivation (**doShutdown**()).

2.1.2. La classe de traitement des messages JBI

De même que pour les ServiceEngines, la deuxième classe à implémenter est la classe qui permet au composant de traiter les messages provenant du conteneur JBI. Il s'agit de la classe abstraite **org.objectweb.petals.component.framework.listener.AbstractJBIListener**.

Dans le cas d'un BindingComponent, le role de cette classe est de transférer des messages JBI vers des services externes au conteneur JBI. Dans le cas du composant Mail, le JBIListener doit transformer les messages exchange JBI en mails et les envoyer, via le protocole SMTP, à un server de courriel. Voici l'implémentation simplifiée du JBIListener dans le cas du composant de mail :

```
public class MailBCJBIListener extends AbstractJBIListener {

    protected MailSessionManager mailSessionManager;

    protected MimeMessageManager mimeMessageManager;

    protected SessionDescriptorBuilder sessionDescriptorBuilder;

    @Override
    protected void init() {
        this.mimeMessageManager = ((MailBCImpl) getComponent())
                .getMimeMessageManager();
        this.mailSessionManager = ((MailBCImpl) getComponent())
                .getMailSessionManager();
        this.sessionDescriptorBuilder = ((MailBCImpl) getComponent())
                .getSessionDescriptorBuilder();
    }

    public void onJBIMessage(Exchange exchange) {

        if (!exchange.isInOnlyPattern()) {
            try {
                exchange.setFault(new PEtALSCFException(
                        "MailBC can only handle InOnly message exchanges"));
            } catch (MessagingException e) {
                getLogger().log(Level.SEVERE, "Can't return fault to consumer",
                        e);
            }
        } else {
            try {
                Extensions extensions = new Extensions(getProvides()
                        .getExtensions());

                // Build a mail session descriptor from the given address (URI)
                SessionDescriptor sessionDescriptor = sessionDescriptorBuilder
                        .build(exchange, extensions);

                // Build the destination address
                InternetAddress internetAddress = new InternetAddress(
                        sessionDescriptor.getToAddress());

                // Build the mail session
                Session session = mailSessionManager
                        .createSessionPropertiesFromDescriptor(sessionDescriptor);

                // Create a mail message with the incoming jbi message
                MimeMessage mimeMessage = mimeMessageManager
                        .mapNormalizedMessageToMimeMessage(session,
                                internetAddress, exchange.getInMessage());

                // Send this mail
                mailSessionManager.sendMail(mimeMessage, sessionDescriptor,
                        session);

            } catch (Throwable e) {
                try {
                    exchange.setFault(new Exception(e));
                } catch (MessagingException e1) {
                    getLogger().log(Level.SEVERE,
                            "Can't return fault to consumer", e);
                }
            }
        }
    }

}
```

En plus de la méthode **onJBIMessage** (obligatoire), le JBIListener propose une methode **init()** qui permet au développeur de mettre en place un contexte initial de fonctionnement pour son JBIListener. Cette méthode est appelée immédiatement après l'instanciation du JBIListener. Les JBIListener sont instanciés par le CF lors du démarrage du composant (appel de la méthode **start()** sur la classe principale du composant).

2.1.3. La classe de traitement des message externes

Le troisième type de classe à implémenter est **org.objectweb.petals.component.framework.listener.AbstractExternalListener**. Ce type de classe est spécifique aux BindingComponents. Elle est utilisée pour traiter les messages en provenance de consommateurs de services externes au conteneur JBI. Grâce à ce type de classes, le Binding Component donne accès au service du conteneur via différents protocoles (HTTP, JMS...).

Lors du démarrage d'une ServiceUnit contenant des champs "consumes" : le CF instancie un ExternalListener par element "consumes" défini.

Dans le cas du composant de mail, l'ExternalListener démarre un scheduler qui va intéroger, à intervalle de temps régulier, un serveur de courriel externe dont les coordonées (adresse, port, protocole...) ont été spécifiées dans les extensions d'un "consumes". Si il trouve de nouveaux messages, il les "aspire", les transforme en Exchange et les envoie au service JBI ciblé.Voici le code de cet ExternalListener :

```
public class MailBCExternalListener extends AbstractExternalListener {

    private Timer t;

    protected MailSessionManager mailSessionManager;

    protected SessionDescriptorBuilder sessionDescriptorBuilder;

    protected MimeMessageManager mimeMessageManager;

    @Override
    protected void init() {
        this.mailSessionManager = ((MailBCImpl) getComponent())
                .getMailSessionManager();
        this.sessionDescriptorBuilder = ((MailBCImpl) getComponent())
                .getSessionDescriptorBuilder();
        this.mimeMessageManager = ((MailBCImpl) getComponent())
                .getMimeMessageManager();
    }

    @Override
    protected void start() throws PEtALSCFException {
        // Extract session information from SU extensions
        Extensions extensions = new Extensions(getConsumes()
                .getExtensions());
        SessionDescriptor sessionDescriptor = sessionDescriptorBuilder.build(
                null, extensions);

        // Create and register the listener
        MailBCExternalScheduler listener = null;
        t = new Timer();
        listener = new MailBCExternalScheduler(getLogger(), this,
                sessionDescriptor, mailSessionManager, mimeMessageManager);

        // Start listening for new mails
        t
                .schedule(listener, 0, Integer.parseInt(sessionDescriptor
                        .getPeriod()));

    }

    @Override
    protected void stop() throws PEtALSCFException {
        t.cancel();
    }
```

```
}
```

De même que pour le JBIListener, une methode **init()** permet au développeur d'initialiser l'ExternalListener. La méthode **getConsumes()** permet d'accéder au champs "consumes" (sous forme d'objet Java) qui est à l'origine de la création du Listener. L'implémetation des méthodes **start()** et **stop()** est obligatoire. Ces méthode sont appelées respectivement lors du démarrage ou de l'arrêt d'une ServiceUnit.

L'ExternalListener propose des méthodes permettant d'accéder : au ComponentContext (**getComponentContext()**), au DeliveryChannel (**getdeliveryChannel()**), au Logger (**getLogger()**) et au Composant (**getComponent()**).

Le Scheduler (MailBCExternalScheduler), instancié durant le "start()" du Listener, envoie au conteneur JBI les courriels qu'il a transformés en Exchanges. Pour cela, il utilise des méthodes utilitaires proposées par les Listeners. Ces méthodes sendXXX permettent d'envoyer des messages de façon synchrone ou asynchrone, ces diverses méthodes sont détaillées dans le chapitre "Pour aller plus loin..." ci-dessous). Voici un exemple de code utilisant une méthode sendSync sans timeout :

```
protected void process(Message message) {
  try {
      // Transform mail in MessageExchange
      MessageExchange exchange = processor.process(message);
      if (exchange != null) {
        // Send exchange to the JBI Container
        externalListener.sendSync(new ExchangeImpl(exchange));
      }
  } catch (JBIException e) {
      String msg = "Error processing a mail : "
          + message.getMessageNumber();
      logError(msg, e);
  } finally {
      try {
        message.setFlag(Flag.DELETED, true);
      } catch (MessagingException e) {
        logWarning("Mail message can't be marked as deleted. "
            + "It will not be deleted from the store", e);
      }
  }
}
```

2.2. La configuration du composant

La configuration d'un Binding Component est similaire à celle d'un ServiceEngine, à l'exception d'un paramètre additionnel : **external-listener-class-name**. Il s'agit du nom qualifié de la classe de traitement des messages externes comme défini dans les chapitres précédents. Voici un exemple de descripteur de déploiement pour le composant Mail :

```
<jbi xmlns="http://java.sun.com/xml/ns/jbi"
 xmlns:extensions="http://petals.objectweb.org/extensions/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
 <component component-class-loader-delegation="parent-first" type="binding-component">
  <identification>
   <name>petals-binding-mail</name>
   <description>Mail binding component</description>
  </identification>
  <component-class-name description="Mail Binding Component class">
   org.objectweb.petals.binding.mail.MailBCImpl
  </component-class-name>
  <component-class-path>
   <path-element>petals-bc-mail-1.3-SNAPSHOT.jar</path-element>
   ...
  </component-class-path>
  <bootstrap-class-name>
   org.objectweb.petals.component.framework.DefaultBootstrap
  </bootstrap-class-name>
  <bootstrap-class-path>
   <path-element>petals-bc-mail-1.3-SNAPSHOT.jar</path-element>
   ...
  </bootstrap-class-path>
  <extensions:extensions>
```

```
   <external-listener-class-name>
    org.objectweb.petals.binding.mail.listeners.MailBCExternalListener
   </external-listener-class-name>
   <jbi-listener-class-name>
    org.objectweb.petals.binding.mail.listeners.MailBCJBIListener
   </jbi-listener-class-name>
  </extensions:extensions>
 </component>
</jbi>
```

2.3. L'activation de services

L'activation de services pour un Binding Component est semblable à celle des ServiceEngines. Un exemple de service offert par le composant Mail est l'envoi de courriel à une adresse spécifiée. Voici un exemple de Service Unit "provides" pour le composant de Mail :

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:petals="http://petals.objectweb.org/"
 xmlns:extensions="http://petals.objectweb.org/extensions/">

 <services binding-component="true">
  <provides interface-name="petals:MailInterface"
   service-name="petals:MailService" endpoint-name="MailEndpoint">
   <extensions:extensions>
    <scheme>smtp</scheme>
    <hostname>www.ebmwebsourcing.org</hostname>
    <username>ofabre</username>
    <from>olivier.fabre@enstimac.fr</from>
    <to>olivier.fabre@ebmwebsourcing.com</to>
   </extensions:extensions>
  </provides>
 </services>
</jbi>
```

Cette SU active un endpoint qui expose le service d'envoie de courriels à l'adresse spécifiée dans les extensions (champ "to"), en utilisant le serveur de courriel spécifié (champ "hostname").

2.4. La consommation de services

De même que pour l'activation de services, la consommation de services se configure par le déploiement de Service Unit. Similairement au champ "provides" précédemment decrits, le SU contient un champ "consumes".

Lors du démarrage d'une SU contenant un champ "consumes", un ExternalListener est démarré. Il écoute les requêtes en provenance de consommateurs externes et les envoie au service cible, défini par les attributs de la balise "consumes".

Voici un exemple de Service Unit "consumes" pour le composant de Mail :

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:petals="http://petals.objectweb.org/"
 xmlns:extensions="http://petals.objectweb.org/extensions/">

 <services binding-component="true">
  <consumes interface-name="petals:HelloworldInterface"
   service-name="petals:HelloworldService" endpoint-name="HelloworldEndpoint">
   <extensions:extensions>
    <scheme>imap</scheme>
    <hostname>www.ebmwebsourcing.org</hostname>
    <username>mailbc</username>
   </extensions:extensions>
  </consumes>
 </services>
</jbi>
```

Dans le cas du composant de Mail, l'ExternalListener scrute un dossier de courriel (par défaut le dossier INBOX) sur le serveur spécifié dans les extensions (champ "hostname") pour l'utilisateur spécifié (champ "username"). Les courriels présents dans le dossier sont transformés en Exchanges et sont envoyés au endpoint spécifié dans les attibuts du champ "consumes" (ici un endpoint du composant Helloworld).

Chapitre 3. Pour aller plus loin...

Les précédents chapitres décrivent les bases de la création de composants basés sur le CF. Le CF propose de nombreuses autres possibilités de configuration et d'optimisation des composants comme la gestion des threads, des intercepteurs de message exchanges...

3.1. Les extensions des descripteurs de déploiement (fichier jbi.xml)

La spécification JBI autorise le développeur à ajouter des extensions (balises xml) dans les descripteurs de déploiement des SharedLibraries, ServicesUnits, ServiceAssemblies et Components. Voici un exemple d'extensions dans une SU :

```
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:petals="http://petals.objectweb.org/"
 xmlns:extensions="http://petals.objectweb.org/extensions/">
 <services binding-component="true">
  <consumes interface-name="petals:HelloworldInterface"
   service-name="petals:HelloworldService" endpoint-name="HelloworldEndpoint">
   <extensions:extensions>
    <scheme>imap</scheme>
    <hostname>www.ebmwebsourcing.org</hostname>
    <username>mailbc</username>
   </extensions:extensions>
  </consumes>
 </services>
</jbi>
```

Petals propose, au travers du CF, une classe utilitaire facilitant la manipulation de ces extensions dans les composants. Les extensions reconnues par Petals sont des extensions de type clé-valeur, la clé étant le nom de la balise et la valeur étant le texte contenu entre les balises ouvrantes et fermantes. Par exemple :

```
<username>mailbc</username> ---> clé = username et valeur = mailbc
```

Pour que Petals puisse reconnaitre ces extensions, il faut qu'elles soient placées entre deux balises ouvrante et fermante ayant pour namespace :

```
"http://petals.objectweb.org/extensions/"
```

3.1.1. Les extensions du composant

3.1.1.1. Ou les placer dans le descripteur de deploiement ?

Elle doivent être placées juste avant la balise fermante **</component>**. Voici un exemple d'extensions dans un descripteur de déploiement de composant :

```
<jbi version="1.0" xmlns='http://java.sun.com/xml/ns/jbi'
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:extensions="http://petals.objectweb.org/extensions/">
 <component type="service-engine">
  <identification>
   <name>petals-engine-xslt</name>
   <description>A Xslt Service Engine</description>
  </identification>
  <component-class-name description="Xslt Component class">
   org.objectweb.petals.engine.xslt.XsltEngine
  </component-class-name>
  <component-class-path>
   <path-element>petals-se-xslt-1.4-SNAPSHOT.jar</path-element>
```

```
  ...
 </component-class-path>
 <bootstrap-class-name>
  org.objectweb.petals.component.framework.DefaultBootstrap
 </bootstrap-class-name>
 <bootstrap-class-path>
  <path-element>petals-se-xslt-1.4-SNAPSHOT.jar</path-element>
  ...
 </bootstrap-class-path>
 <extensions:extensions>
  <jbi-listener-class-name>
   org.objectweb.petals.engine.xslt.listener.JBIListener
  </jbi-listener-class-name>
  <pool-size>15</pool-size>
 </extensions:extensions>
</component>
</jbi>
```

3.1.1.2. Comment y accéder dans le composant ?

Les extensions du descripteur de déploiement d'un composant sont accessibles à partir de la classe principale du composant, grâce à la méthode **getComponentConfiguration()**. Cette méthode retourne un objet ComponentConfiguration disposant de getter pour toutes les propriétés standards du composant (pool-size, ignored-status...). Les propriétés spécifiques au composant (extensions propre au composant) sont accessibles à partir de l'objet ComponentConfiguration, à l'aide de la méthode **getProperties()**. Elles se présentent sous la forme d'un objet java.util.Properties.

3.1.2. Les extensions des champs "provides" ou "consumes" des ServiceUnits

3.1.2.1. Ou les placer dans le descripteur de deploiement ?

Elle doivent être placées juste avant la balise fermante **</consumes> ou </provides>**. Voici un exemple d'extensions dans un descripteur de déploiement d'une ServiceUnit (pour un champ "consumes") :

```
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:petals="http://petals.objectweb.org/"
 xmlns:extensions="http://petals.objectweb.org/extensions/">
 <services binding-component="true">
  <consumes interface-name="petals:HelloworldInterface"
   service-name="petals:HelloworldService" endpoint-name="HelloworldEndpoint">
   <extensions:extensions>
    <scheme>imap</scheme>
    <hostname>www.ebmwebsourcing.org</hostname>
    <username>mailbc</username>
   </extensions:extensions>
  </consumes>
 </services>
</jbi>
```

3.1.2.2. Comment y accéder dans le composant ?

Les extensions des champs "provides" et "consumes" sont accessibles dans les deux types de Listeners (JBIListener ou ExternalListener), grâce à la méthode **getExtensions()**.

Chaque Listener est lié au runtime à un champs "provides" ou "consumes". Un ExternalListener n'est lié qu'à un champ "consumes". Par contre un JBIListener peut être lié à un "provides" dans le cas du traitement d'une demande de service (provider) ou à un champ "consumes" dans le cas du traitement d'une réponse à l'appel d'un service (consumer).

La méthode **getExtensions()** permet au développeur d'accéder directement aux extensions en rapport avec le message à traiter sans avoir à différencier les cas consumer ou provider. Les extensions se présentent sous la forme d'un

objet Extensions. Il propose une méthode **getPetalsExtensions()** qui retourne un objet java.util.Properties qui contient l'ensemble des clés-valeurs. Les valeurs des extensions Petals sont aussi accessibles individuellement à partir des methodes **getValue(String key, String defaultValue)** et **getValue(String key)**. Enfin, l'objet Extensions propose une méthode **getDocumentFragment()** qui retourne un objet org.w3c.dom.DocumentFragment contenant l'ensemble des extensions (Petals ou non) sous forme d'arbre DOM.

3.2. Paramètres avancés de configuration du Composant

Le CF propose des paramètres de configuration communs à tous les composants. Ces paramètres sont à spécifier dans les extensions présentes dans son descripteur de déploiement (fichier jbi.xml) :

Tableau 3.1. Configuration avancée du composant

Parametre	Description	Valeur par defaut	Obligatoire
pool-size	Nombre de threads écoutant des messages en provenance du conteneur JBI (JBIListeners). Entier supérieur à 1	10	Non
ignored-status	Status des messages exchanges ne devant pas être traités par le composant. Les valeurs acceptées sont : **DONE_AND_ERROR_IGNORED**, **DONE_IGNORED**, **ERROR_IGNORED** or **NOTHING_IGNORED**	DONE_AND_ERROR_IGNORED	Non
jbi-listener-class-name	Nom qualifié de la classe du composant qui étend la classe **AbstractJBIListener**		Oui
external-listener-class-name	Nom qualifié de la classe du composant qui étend la classe **AbstractExternalListener**		Non

3.3. Paramètres avancés de configuration des Service Units

Le CF propose des paramètres de configuration communs à toutes les Services Units. Ces paramètres sont à spécifier dans les extensions des champs provides ou consumes présents dans le descripteur de déploiement de la SU (fichier jbi.xml) :

Tableau 3.2. Configuration avancée des Services Units (champs provides)

Parametre	Description	Valeur par defaut	Obligatoire
wsdl	Adresse d'un fichier wsdl décrivant les services et opérations proposées par un endpoint activé par la service unit. **Cette extension n'est utilisable qu'avec les champs provides**. L'adresse peut être une url de type "http" ou "file" ou un chemin relatif par rapport à la racine de l'archive de la SU. Ex : "file:///user/ofabre/test.wsdl" ou "/WSDL/test.wsdl". Si aucun wsdl n'est spécifié, une description simplifiée est crée automatiquement par le CF		Non

Tableau 3.3. Configuration avancée des Services Units (champs consumes)

Parametre	Description	Valeur par defaut	Obligatoire
pattern	Message exchange pattern abréviation. Ce paramètre peut être utilisé en conjonction avec une méthode utilitaire des Listeners : **createExchange(MEPConstants mep)**. Cette méthode renvoie alors un Exchange correspondant au type de pattern spécifié. Les valeurs admises sont : **in-only**, **robust-in-only**, **in-opt-out** et **in-out**.		Non
operation	Opération à appeler sur un service. Ce paramètre peut être utilisé en conjonction avec les méthodes sendXXX des Listeners. Si l'opération n'est pas spécifiée dans l' Exchange à envoyer, c'est la valeur de ce paramètre qui est utilisée.		Non
timeout	Timeout en millisecondes lors d'un envoie synchrone. Ce paramètre peut être utilisé en conjonction avec la méthode **sendSync(Exchange exchange)** des Listeners. Un envoi synchrone est alors effectué avec un timeout correspondant à cette valeur. Entier supérieur ou égal à 0. 0 pour aucun timeout.	0	Non
org.objectweb.petals.routing.strategy	Cette propriété définie la stratégie de routage. Deux types de stratégies peuvent etre définies: highest ou random. Les autres parametres représentent respectivement la ponderation locale, la ponderation des endpoints actifs et la ponderation des endpoints inactifs. La strategy "random" choisit un endpoint en fonction des pondérations définies. L'endpoint qui a la plus forte pondération sera plus facilement selectionné par rapport aux autres. La strategy "highest" choisit le premier endpoint dans la liste qui a la plus forte ponderation.		Non
org.objectweb.petals.transport.compress	La charge utile du MessageExchange est un texte XML. Il peut être intéressant de la compresser avant que les messages ne soient échangés entre deux noeuds Petals. Les valeurs admises sont **true** ou **false**. Positionner la valeur à **true** permet de compresser le contenu du message.		Non
org.objectweb.petals.messaging.noack	Tous les échanges JBI finissent par un message contenant un status "DONE" ou "ERROR". Le consommateur doit accepter ces messages, sinon ils sont accumulés dans le NMR. De plus, ces messages génèrent du traffic souvent inutile. Les valeurs admises sont **true** ou **false**. Positionner la valeur à **true** permet de ne pas envoyer les messages de type "DONE" ou "ERROR".		Non
org.objectweb.petals.transport.qos	Cette propriété définie la règle de qualité de service supportés par les transporters PEtALS. Les valeurs possibles sont: reliability ou fast. Si ce n'est pas spécifié, la règle "reliability" est selectionnée par défaut.		Non

Les paramètres **org.objectweb.petals.routing.strategy**, **org.objectweb.petals.transport.compress**, **org.objectweb.petals.messaging.noack** et **org.objectweb.petals.transport.qos** sont positionnés automatiquement par le CF lors de l'envoi de messages avec les methodes sendXXX des Listeners.

3.4. Les intercepteurs d'Exchanges

3.4.1. Définition

Un intercepteur permet de traiter un MessageExchange sous la forme d'un Exchange:

- A l'envoi de l'Exchange au provider. C'est la dernière opération effectuée avant l'envoi effectif du message.

- A la réception d'un Exchange. C'est la première opération effectuée à la réception du message.

3.4.2. Implémenter un intercepteur

Un intercepteur doit étendre la classe abstraite **org.objectweb.petals.component.framework.interceptor.Interceptor**. Un exemple d'intercepteur qui affiche l'ID du message exchange serait :

```
public class MessageInfoPrintInterceptor extends Interceptor {

    @Override
    public void handleMessageAccept(Exchange exchange,
            ServiceUnitExtensibleElement consumeProvide,
            Map<String, String> parameters) {
        System.out.println(exchange.getExchangeId());
    }

    @Override
    public void handleMessageSent(Exchange exchange,
            ServiceUnitExtensibleElement consumeProvide,
            Map<String, String> parameters) {
        System.out.println(exchange.getExchangeId());
    }
```

Les deux méthodes implémentés ci-dessus permettent l'interception d'Exchange:

- Lors d'une reception de message, la méthode **handleMessageAccept est invoquée.**
- Lors de l'émission d'un message, la méthode **handleMessageSent** est invoquée.

Ces deux méthodes ont la même signature, on peut alors accéder:

- à l'Exchange.
- a l'objet parent Consumes/Provides.
- aux paramètres additionnels définis pour chaque bloc d'extension (voir chapitre suivant).

La classe abstraite Interceptor fournit un ensemble d'attributs définit à son initialisation. Lors d'une interception, on peut ainsi accéder:

- Au composant (AbstractComponent).
- A une map de propriétés, sous forme de clé-valeur. Ces propriétés ont été extraites du descripteur jbi.xml du composant (voir chapitre suivant).

3.5. Configuration

3.5.1. Configurer un intercepteur au niveau composant

Les intercepteurs à utiliser dans le composant sont définis au niveau du bloc d'extensions du descripteur jbi.xml du composant. La syntaxe est la suivante:

```
<extensions:extensions>
  <interceptors>
    <interceptor
      class="org.objectweb.petals.component.framework.interceptor.impl.EchoInterceptor"
      name="echo"
      active="true">
        <param name="name1">value1</param>
        <param name="name2">value2</param>
    </interceptor>

    <interceptor
      class="org.objectweb.petals.component.framework.interceptor.impl.MessageInfoPrintInterceptor"
```

```
      name="info"
      active="false">
        <param name="name11">value11</param>
        <param name="name22">value22</param>
    </interceptor>

  </interceptors>
</extensions:extensions>
```

Tableau 3.4. Configuration des intercepteurs au niveau du composant

Parametre	Description	Valeur par defaut	Obligatoire
class	Le nom de la classe d'intercepteur. Cette class doit étendre la classe abstraite org.objectweb.petals.component.common.interceptor.Interceptor. Cette classe doit être présente dans le classloader, soit au niveau du composant, du CF ou dans une shared library.		Oui
name	Nom de l'intercepteur. Ce nom sera utilisé pour la configuration additionelle au niveau des service units.	Le nom de la classe	Non
active	L'intercepteur est actif pour toutes les service units.	true	Non

Les elements `<param name="xxx">yyy</param>` sont des parametres de configuration que le developpeur peut définir au niveau du composant. Au runtime, ces attributs sont disponibles au niveau de la classe intercepteur.

3.5.2. Configurer un intercepteur au niveau service-unit

Les intercepteurs définis au niveau du composant peuvent être utilisés au niveau des service units. Pour utiliser un intercepteur, il faut les configurer au niveau du bloc extensions d'une service unit de la facon suivante:

```
<extensions:extensions>
  <interceptors>
    <accept>
      <interceptor name="echo">
        <param name="aa">A</param>
        <param name="bb">B</param>
      </interceptor>
      <interceptor name="intercept"/>
    </accept>
    <send>
      <interceptor name="info"/>
    </send>
  </interceptors>
<extensions:extensions>
```

Tableau 3.5. Configuration des intercepteurs au niveau de la service unit

Parametre	Description	Valeur par defaut	Obligatoire
name	Le nom de l'intercepteur a utiliser. C'est le nom qui est definit au niveau du composant.		Oui

Les elements `<param name="xxx">yyy</param>` sont des paramètres de configuration que le développeur peut définir au niveau de la service unit. Au runtime ces attributs sont accessibles dans la map **parameters** de la méthode d'interception. Ce mécanisme permet une configuration plus fine des intercepteurs.

Il est important de noter que les parametres niveau service unit sont prioritaires sur ceux du composant. Si le developpeur du composant définit des attributs au niveau de la service-unit et portant le meme nom que ceux du composant, la map passée en paramètre contiendra les paramètres mixés (composant + service-unit).

On définit les intercepteurs à utiliser en émission et en réception de **Echange** respectivement dans les éléments **send** et **accept**.

L'ordre d'appel des intercepteurs est l'ordre dans lequel ils sont définis dans les blocs receive et send.

Les intercepteurs qui sont définis au niveau des composants avec un parametre **active** à **true** sont dans tous les cas prioritaires sur les intercepteurs définis au niveau des service units et mis en tête de liste des intercepteurs ordonnés. Ils sont aussi configurables de la même façon avec les paramètres additionels.

3.6. Notification de performance

3.6.1. Introduction

Le CF intégre un MBean JMX permettant la publication de notification mesurant le temps de traitement des messages.

Quatre types de notifications sont envoyées :

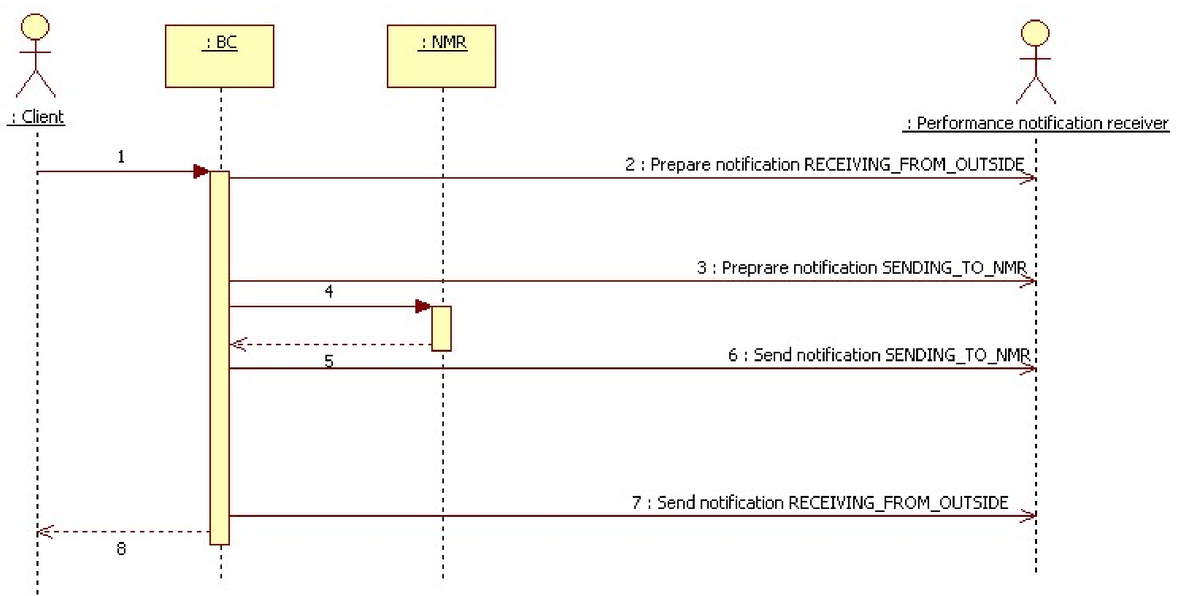
- RECEIVING_FROM_NMR : temps de traitement global, par le composant émetteur, d'un message en provenance du NMR,
- SENDING_TO_NMR : temps de traitement global d'un message émis par le composant émetteur sur le NMR. Ce temps intègre toute les durées de traitement des composants entrant en jeu, aussi bien des composants du bus `petals` que des composants externes,
- RECEIVING_FROM_OUTSIDE : temps de traitement global, par le composant émetteur, d'une requête venant de l'extérieur,
- SENDING_TO_OUTSIDE : temps de traitement global par une chaine de liaison externe au bus et dont le composant émetteur est en attente de la réponse.

3.6.2. Cas d'usage

Ces notifications se situent au niveau composant, et vont toujours par paires :

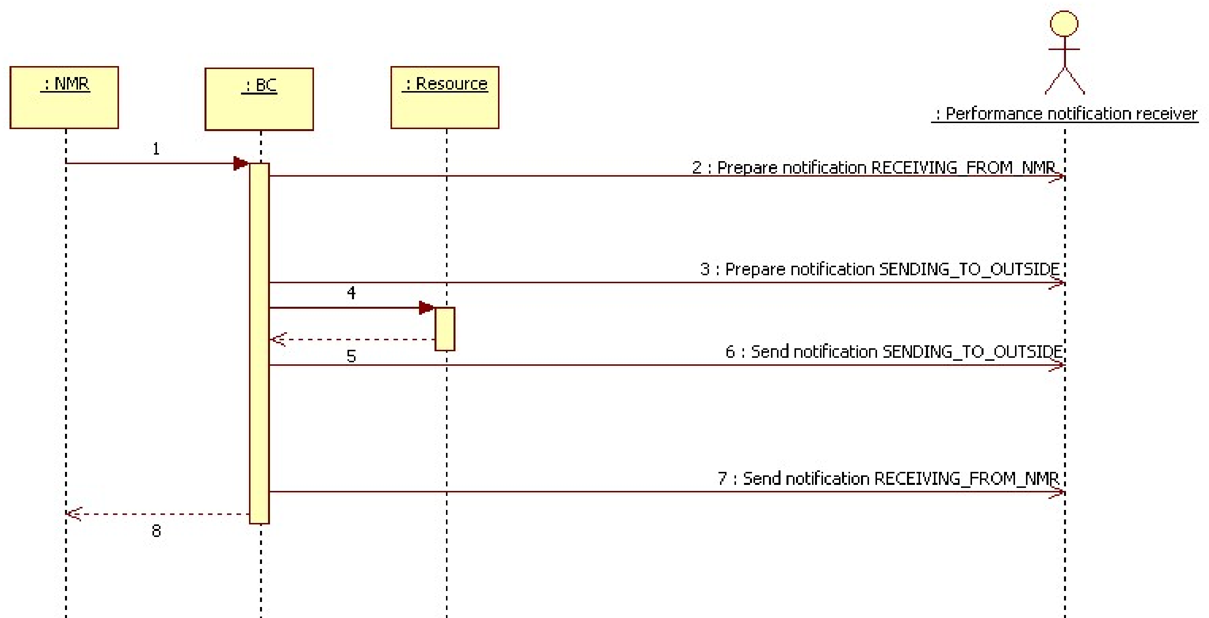
- dans le cas d'un binding-component permettant l'accès au bus depuis l'extérieur, chaque solicitation externe engendre les notifications de performance de type RECEIVING_FROM_OUTSIDE et SENDING_TO_NMR,

Figure 3.1. Performance notification RECEIVING_FROM_OUTSIDE/SENDING_TO_NMR



- dans le cas d'un binding-component permettant l'accès à une resource depuis le bus, chaque solicitation du NMR engendre les notifications de performance de type RECEIVING_FROM_NMR et SENDING_TO_OUTSIDE.

Figure 3.2. Performance notification RECEIVING_FROM_NMR/SENDING_TO_OUTSIDE



- dans le cas d'un service engine, chaque solicitation du NMR engendre une notification de type RECEIVING_FROM_NMR. Si de plus d'autre service sont invoqués par ce service engine, des notifications de type SENDING_TO_NMR sont aussi envoyées.

3.6.3. D'un point de vue client

Les notifications de performance sont des notifications JMX envoyés par un MBean embarqué dans chaque composant, via le CF. Le MBean est un MBean de type custom qui se trouve dans le domain org.objectweb.petals et se nomme performance_notifier_<component-name> où <component-name> est le nom du composant.

Le contenu d'une notification est un table de String respectant :

Tableau 3.6. Performance notification format

index	description
0	UID identifiant un ensemble lié de notifications.
1	timestamp, correspond aussi au début de la prise de mesure.
2	composant émetteur
3	type de la notification, voir ci-dessus pour plus d'information sur les types de notification.
4	durée en milliseconde du traitement.
5	donnée utilisateur, le contenu est fonction du composant. Se référer au guide utilisateur du composant pour avoir plus d'information sur le contenu exact de ce champ.

Dans la console MX4J de petals, les MBeans de notification performance se trouve dans l'onglet Server View, exemple :

Figure 3.3. MBean de notification de performance du BC Soap

Domain: org.objectweb.petals

org.objectweb.petals:type=binding,name=petals-binding-soap

org.objectweb.petals:type=custom,name=performance_notifier_petals-binding-soap

org.objectweb.petals:type=installer,name=petals-binding-soap

Pour recevoir, les notifications de performance d'un composant, il faut au préalable les avoir activées au niveau du MBean du composant. Ceci se fait, dans la console MX4J de Petals, après avoir cliqué sur le MBean du composant souhaité, en positionnant l'attribut notificationsEnabled à true. Il est aussi possible de définir un pas d'envoi des notifications via l'attribut notificationsStep.

Figure 3.4. MBean de notification de performance du BC Soap

Petals view | Server view | MBean view

MBean org.objectweb.petals:type=custom,name=performance_notifier_petals-binding-soap
Description JMX Performance Notifier

Attributes

Name	Description	Type	Value	New Value
notificationsEnabled	Notification activation	java.lang.Boolean	false	true / false set
notificationsStep	Notification step	java.lang.Integer	1	1 set

Set all

Operations

Name	Return type	Description

Constructors

La réception des notifications de performance nécessite un client JMX qui doit souscrire aux notifications de performance. Il est possible d'utiliser la JConsole pour recevoir les notifications. Le plugin Eclipse Petals propose aussi une vue "Notifications Performance", plus riche que ce que propose la JConsole.

Figure 3.5. Réception des notifications de performance via la JConsole

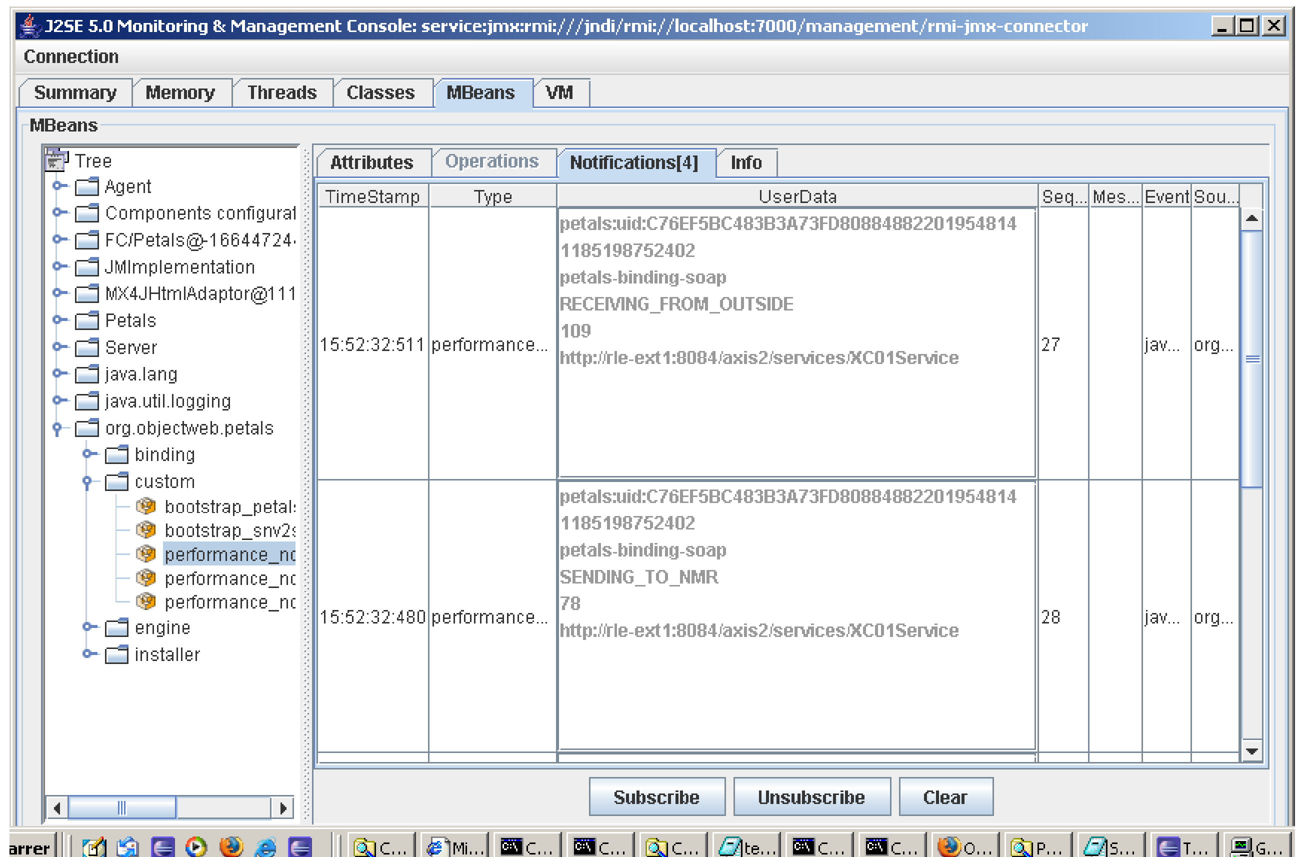
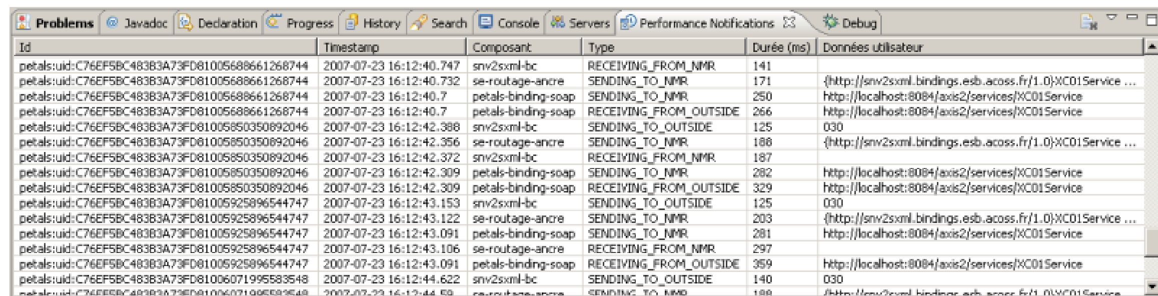


Figure 3.6. Réception des notifications de performance via le plugin Eclipse Petals



3.6.4. Coté développement

Les classes du CF embarquent nativement la gestion de quelques unes de ces notifications. Dans les autres cas, les notifications doivent être implémentées au niveau du composant lui-même.

Le CF embarque les cas suivants:

- notifications de type RECEIVING_FROM_NMR, lors de la réception d'un message NMR dans un service-engine ou un binding-component,
- notifications de type SENDING_TO_NMR, dans un service-engine ou un binding-component, à condition que celui-ci **utilise les méthodes du CF** pour faire un appel NMR.

Dans les autres cas, les notifications doivent être gérées au niveau du code des composants en utilisant les méthodes :

- AbstractComponent.createPerformanceNotificationSendingToOutside() : pour créer une notification de type SENDING_TO_OUTSIDE,

- AbstractComponent.createPerformanceNotificationReceivingFromOutside() : pour créer une notification de type RECEIVING_FROM_OUTSIDE,
- AbstractComponent.createPerformanceNotificationSendingToNMR() : pour créer une notification de type SENDING_TO_NMR,
- AbstractComponent.createPerformanceNotificationReceivingFromNMR() : pour créer une notification de type RECEIVING_FROM_NMR,
- AbstractComponent.sendPerformanceNotification(...) : pour publier une notification.

3.7. L'envoi d'Exchange dans le conteneur JBI

Le CF propose des méthodes utilitaires permettant d'envoyer des MessageExchanges sous la forme d'Exchange à travers le bus JBI. Ces méthodes sont accessibles dans les deux types de listeners (le JBIListener et l'ExternalListener). Elles sont au nombre de quatres : deux synchrones et deux asynchrones. Elles sont identiques pour les deux types de listeners à l'exception près que les methodes de l'ExternalListener ajoute les informations de destination du message (interface-name, service-name, endpoint-name) à partir des attributs du champs "consumes" d'une SU associé à ce Listener. Ces informations de destination ne sont ajoutées que si elles ne sont pas spécifiées dans le MessagesExchange passé en paramètre de la méthode.

3.7.1. Le send synchrone avec timeout

La signature de cette méthode est la suivante : boolean **sendSync**(Exchange exchange, long timeoutMS).

Elle envoie le message passé en paramètre dans le bus JBI, de façon synchrone, avec le timeout spécifié en paramètre.

Si aucune opération n'est spécifiée dans l'Exchange, celle ci est recherchée dans les extensions du champ "provides" ou "consumes" associé au listener.

Les propriétés "routing.strategy", "transport.compress", "messaging.noack" and "transport.qos", éventuellement spécifiées dans les extensions du champ "provides" ou "consumes" associé au listener, sont automatiquement configurées dans l'Exchange.

Les intercepteurs associées au "provides" ou "consumes" de ce listener sont automatiquement exécutés (voir chapitre "Les intercepteurs de messages exchanges").

Si les notifications de performance du composant sont activées, une notification est automatiquement envoyée à chaque appel de la méthode. Il est possible de customiser le champ userData des notifications en surchargeant la méthode getPerformanceNotificationUserData du JBIListener.

3.7.2. Le send synchrone sans timeout

La signature de cette méthode est la suivante : boolean **sendSync**(Exchange exchange).

Cette méthode est semblable à la précédente mais recherche la valeur de timeout à utiliser dans les extensions du champ "provides" ou "consumes" associé au listener. Si aucune valeur n'est trouvée, la valeur 0 est utilisée par défaut (cad aucun timeout).

3.7.3. Le send asynchrone sans synchronisation des réponses

La signature de cette méthode est la suivante : void **send**(Exchange exchange).

Cette méthode est semblable au send synchrone sans timeout à l'exception près que l'envoi est asynchrone et qu'aucune notification de performance n'est crée.

3.7.4. Le send asynchrone avec synchronisation des réponses

La signature de cette méthode est la suivante : void **sendAsync**(Exchange exchange).

Cette méthode est semblable à la précédente mais ajoute un mécanisme de synchronisation des réponses. En effet, une méthode accept bloquante est proposées par les listeners. Cette méthode reste bloquée jusqu'à ce que la réponse à l'Exchange passé en paramètre revienne au consommateur de service. La méthode **accept**(Exchange exchange) retourne alors l'Exchange contenant la reponse à l'appel asynchrone.

Voici un exemple de code illustrant l'utilisation de cette méthode send asynchrone et de la méthode accept associée :

```
Exchange m1 = createExchange(extensions);
Exchange m2 = createExchange(extensions);
sendAsync(m1);
... do some stuff
sendAsync(m2);
... do some stuff
Exchange reponse1 = accept(m1) --bloquant--
... do some stuff
Exchange reponse2 = accept(m2) --bloquant--
... do some stuff
```